EINDHOVEN UNIVERSITY OF TECHNOLOGY
Department of Mathematics and Computing Science

MASTER'S THESIS

Cryptography as a service in a cloud computing environment

Hugo A.W. Ideler

Eindhoven, December 2012

Supervisors:	Prof. dr.	Milan Petković*
	Prof. dr.-ing.	Ahmad-Reza Sadeghi†
Instructors:	MSc.	Sven Bugiel†
	MSc.	Stefan Nürnberger†

*Eindhoven University of technology
†Center for Advanced Security Research Darmstadt (CASED)

Abstract

Nowadays, a serious concern about cloud computing is the protection of clients' data and computations against various attacks from the cloud provider's side as well as outsiders. Moreover, cloud consumers are rather limited in implementing, deploying and controlling their own security solutions in the cloud.

In this thesis, we present a cloud architecture which enables cloud consumers to securely deploy and run virtual machines in the cloud, in which the complete deployment lifecycle is considered, and with a special focus on both the malicious insider as well as the external attacker. Our implementation, which is based on the Xen hypervisor, revolves around three core modifications to the status quo.

First, the clients' cryptographic operations run in an isolated client-specific secure execution domain, protected from cloud administrators and outside attackers having gained access to a VM's data. Second, we present a design that guards the confidentiality and integrity of a user's VM as a whole against the cloud administrator. This is achieved by enforcing stronger access control and by noninvasive modifications to the Xen architecture. Third, we extend Xen with a design for secure credentials provisioning by leveraging standard trusted computing technology, enabling cloud consumers to always be in control of the access to, and usage of, their cryptographic credentials in the cloud.

We evaluate our implementation in a lab setup. In addition, we perform rudimentary experiments in the EC2 public cloud to the extend possible with the constraints imposed by the cloud provider.

Acknowledgements

This thesis is the result of my graduation project at the Center for Advanced Security Research Darmstadt (CASED) in cooperation with the Eindhoven University of Technology.

I wish to thank my supervisor from Darmstadt, prof. Ahmad-Reza Sadeghi, under whose supervision and guidance I was able to work on a highly interesting and intriguing topic. Furthermore, I am deeply grateful to my instructors, Sven Bugiel and Stefan Nürnberger, on whom I could always rely during my months in Darmstadt to help me out with feedback and fruitful discussions. It was a great pleasure for me to work with them.

Finally, I expressly thank my Eindhoven supervisor, prof. Milan Petković, for giving me the opportunity to do my master's thesis at a different university and for his continuous support during the whole project.

Hugo A.W. Ideler
Darmstadt, Germany
December 2012

Contents

List of Figures

List of Tables

1. Introduction

Trustworthy cloud computing — fact or fiction? This is a question that is not easy to answer, though it is certain that this topic has attracted significant momentum and attention in both academia and industry [**VZ09**]. For some, cloud computing heralds the fulfillment of the long-held dream of "computing as a utility," while for others cloud computing is nothing more than a rehash of established ideas [**Arm+10**].

Defining cloud computing is not simple. This is due, in part, to the variety of different technologies involved, as well as the hype surrounding the topic [**Vaq+08**]. Nevertheless, the oft-cited definition as put forward by the NIST [**MG11**] strikes a good balance. It defines cloud computing as, "*a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.*"

However, in spite of the many potential blessings of cloud computing, this new paradigm is also associated with additional risks which *cloud consumers*, people or organizations leasing cloud resources, need to be aware of [**CSA10**]. First of all, the *cloud service provider* (CSP), who offers and controls the cloud resources, has full access to any data or computations brought into the cloud. Second, because CSPs multiplex their limited resources to multiple consumers for efficiency reasons, there is the threat of other cloud consumers, with whom cloud resources are shared, breaking the isolations imposed by the CSP. Third, while moving to the cloud *adds* attack channels, the 'old' channels, misused by external attackers, still exist. In fact, as a recent report by Alert Logic shows [**AL12**], most of the security incidents in cloud-hosted software still occur through old attack vectors, e.g. web services exposed to the Internet.

While protection against external attackers is essentially a business of cloud consumers and hard to address at infrastructure level, the CSP can play a role in offering security services for enabling security in depth. In non-cloud settings, security in depth has been comprehensively explored — for instance, hardware security modules (HSMs), physical devices that use and manage cryptographic credentials, are an established methodology to protect high value keys. By requiring an out-of-band channel for the extraction of keys, the high value keys

remain protected in spite of a breach of a connected computer system. While cloud consumers are typically not allowed to bring their own HSMs to the data center, the CSPs can alleviate this constraint by starting to provide virtual HSMs in the cloud.

The conclusion that cloud consumers, by the very nature of cloud computing, will need to invest a degree of trust in the CSP seems inevitable. The challenge for cloud computing is, therefore, to reduce this degree of trust to acceptable levels. One promising direction that has been proposed, is by leveraging *trusted computing* technologies [**SGR09**]. The benefit for cloud consumers of using trusted computing is that it allows them to determine whether the CSP is indeed running the trusted software that it claims. Unfortunately, while the trusted computing underpinnings are well known, actual implementations which make use of these principles are scarce.

The goal of this thesis is to design and implement a cloud security architecture which enables cloud consumers to securely deploy and run their virtual machines in the cloud, protecting their high-value cryptographic credentials against external as well as internal attackers. Special consideration is given to providing this protection during the entire lifecycle of virtual machines, i.e., from booting up encrypted images, to securely isolating the virtual machine at runtime.

To realize our architecture, we employ well-established trusted computing techniques and the concept of privilege separation. Our prototypical implementation and evaluation is based on the popular Xen hypervisor, commonly applied by today's largest cloud service providers such as Amazon or Rackspace.

Thesis outline

In chapter 2, we give an introduction to the technologies which underly today's cloud computing, and we introduce technologies which play a fundamental role in our proposed design. Once the background information is explained, we are able to discuss the problem description in chapter 3 in sufficient technical detail. We cover the related work to this thesis in chapter 4 in which we discuss the state of the art in this field.

We present our architecture in chapter 5, where we unfold our *cryptography as a service* (CaaS) design. A technical discussion of our implementation follows in chapter 6. After an evaluation in chapter 7, we discuss future work in chapter 8 and draw conclusions in chapter 9.

Typesetting and layout

Notational formatting

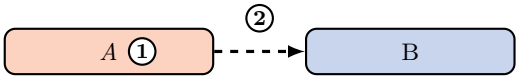
The following typographical convention will be applied.

- Acronyms are typeset in smallcaps (e.g., TPM).
- Functions are typeset in sans-serif and namespaces are prepended in brackets (e.g., ⟨ABC_⟩Example).
- Literals are typeset in teletype font (e.g., `/etc`).

Figure formatting

The figures presented in this thesis share a common layout. In the main text, we refer to components shown in the figure using labeled references. In addition, when a distinction between trusted and untrusted components is relevant, we highlight trusted parts in red with italic font, while untrusted parts are shown in blue with normal font.

Example.



① A trusted component.
② Communication initiated from a trusted to an untrusted component.

2. Background information

In this chapter, we provide an introduction to selected topics which are necessary for understanding the solutions proposed in this thesis. The topics we will discuss are virtualization in general, virtualization using the Xen hypervisor in specific, and the trusted computing technology. Readers who are sufficiently familiar with these topics can safely skip one or more sections.

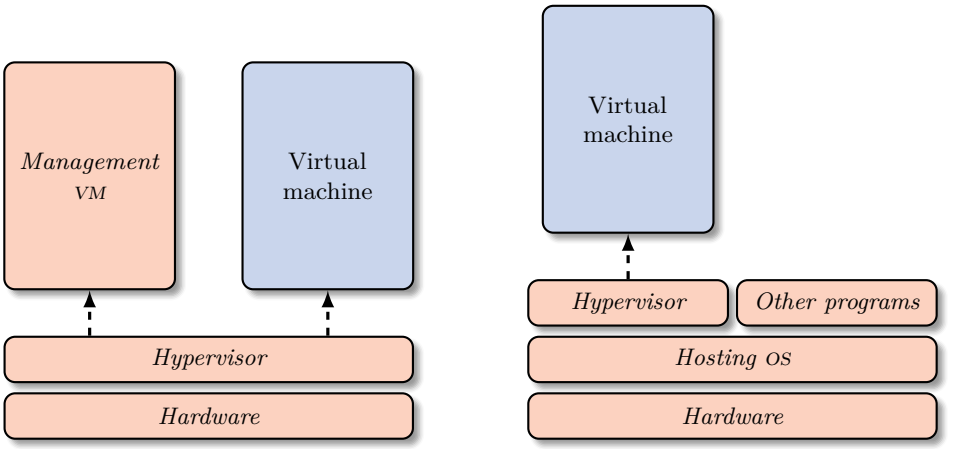
2.1. Introduction to virtualization

Virtualization is an ubiquitous architectural pattern in the field of computer science. It is found in many components of modern computers at both software and hardware levels. The virtualization pattern is related to the concepts of abstraction and emulation in that a higher level layer imposes obligations on a lower level layer, regardless of how it is implemented.

A typical example is memory virtualization in which the illusion of nearly unlimited, contiguous memory is presented to a program running on an operating system. By removing the burden of complex memory management from the application, the software becomes less error prone and also enables the operating system to use the limited RAM resources more efficiently.

Other examples that are commonplace are virtualization of I/O devices (e.g., mounting an image of a CD-ROM without actually writing it to a CD) and virtualization of entire platforms. Already long before the advent of present day cloud computing, virtualization of platforms has been used in large mainframes in the 1950s. Only in the last decade has large-scale virtualization taken flight, forming a cornerstone of the concept of cloud computing.

In literature, authors generally discern between three levels of cloud computing, namely *software as a service* (SaaS), *platform as a service* (PaaS), and *infrastructure as a service* (IaaS). These three degrees deal with cloud computing at different abstraction levels, and each degree has its own particular use cases. However, only the lowest level of these, IaaS, deals with virtualization directly. Considering that this thesis focuses on infrastructure solutions, we will not discuss PaaS or SaaS any further, though we note that these concepts can be utilized on top of the IaaS foundations provided in this thesis.



(a) Type-I hypervisor: the hypervisor is truly a distinct entity and is the highest privileged code running on the machine.

(b) Type-II hypervisor: the hypervisor is an application running inside an OS and is not isolated.

Figure 2.1: Schematic virtualization overview. In both types a hypervisor provides for the virtualization operations, but how the hypervisor itself is positioned differs.

2.1.1. Virtualization technologies

A hypervisor[1] is a software layer that multiplexes hardware resources among one or more guest operating systems, in analogy to how a supervisor multiplexes resources among applications inside an operating system (OS).

Although there exist a variety of different virtualization technologies, usually a distinction is made between type-I and type-II hypervisors (Fig. 2.1). This distinction between hypervisor categories goes back to Goldberg, who already discussed this subject in times long before the present day virtualization techniques [**Gol73**].

1. The **type-I** hypervisors are also called *bare-metal* hypervisors, since they run directly on the hardware without an intervening layer of an OS. Examples of this category are Xen, Hyper-V, and Vmware ESX [**Bar+03**; **VV09**; **VM**]. Some of the type-I designs, most notably Xen, make use of a "management VM" to perform tasks such as device emulation inside a special VM instead of doing these in the hypervisor. An oft-cited argument

[1] The term virtual machine monitor (VMM) can be used interchangeably with hypervisor. For consistency, we will use the term hypervisor throughout this thesis.

in favor of type-I hypervisors, is that a small, dedicated hypervisor benefits isolation between VMs, improving security.

2. The **type-II** hypervisors are a somewhat broader category because the more flexible architecture allows for a wider variety of virtualization solutions. This category includes virtualization software such as Virtualbox, KVM, or Vmware Player [**VB**; **Kiv+07**; **VM**] which run inside the host OS as an application. Therefore, a type-II hypervisor is also referred to as a *hosted* hypervisor.

   One the one hand, this means that type-II virtualization software can optionally benefit from a convenient integration with other applications running in the host. On the other hand, the hypervisor is not isolated but is exposed to a large environment, making it more difficult to reason about the security of the hypervisor.

Nevertheless, we remark that the type distinction between hypervisors is only an indicator and that the reader must be careful in making too strong generalizations. For example, while type-II hypervisors usually have some performance overhead by not running on the hardware directly, this is not always the case. Consider for instance the type-II KVM hypervisor which runs as a Linux kernel module. By being part of the kernel, KVM has complete access to the hardware and can achieve performance in the range of contemporary type-I hypervisors [**Des+08**].

2.1.2. Virtualization techniques

The success of the IBM PC during the last decades means that its CPU instruction set, i.e., the IA-32 (or $x86$) instruction set, is by far the most commonly used architecture in the desktop and server environments. Due to the importance that the IA-32 chipset makers placed on backwards compatibility, the instruction set has evolved over time to include many redundant features. Moreover, the instruction set was never designed with virtualization in mind.

Hence, without special CPU instructions, the ubiquitous $x86$ processor architecture cannot be easily virtualized. Popek and Goldberg identified in 1974 that an architecture is virtualizable if the set of *sensitive* instructions, e.g. those that change the configuration of resources on the system, is a subset of the set of *privileged* instructions, e.g., those that require the processor to run in kernel mode and will trap (i.e., switch to kernel mode and jump to an exception handler) if this is not the case [**PG74**]. This means that if an instruction set is virtualizable according to this definition, virtualization is essentially completed

with handling (by emulation) all traps in the hypervisor and jumping back to the originating code. However, the $x86$ architecture contains 17 instructions which do not possess this property [**Chi07**]. These instructions do not trap to a higher privilege level but instead silently fail, making it impossible for the hypervisor to catch this exception. Hence, these instructions break the virtualization. Three methods have been developed to circumvent these problems on the $x86$ architecture.

Dynamic rewriting. The solution that requires the least changes to the guest OS (i.e., the OS being virtualized) and can be achieved on both modern and legacy hardware, is dynamic (binary) rewriting. In the dynamic rewriting approach, the virtualization environment scans the stream of CPU instructions and identifies privileged instructions which are then rewritten to emulated versions [**Chi07**]. The performance of dynamic rewriting has never been truly satisfactory, however.

Hardware assisted virtualization. Another approach which requires no changes to the underlying OS but achieves better performance, is using CPU virtualization extensions. These extensions are privileged CPU instructions added by Intel and AMD as a response to the slow performance of binary rewriting. These virtualization instruction sets — respectively VT-X and AMD-V — are sophisticated extensions to the standard IA-32 instruction set. These instructions add the ability to "enter a VM" with its own set of CPU registers. For instance, this makes it much easier to set a distinct CR3 pointer[2] for a guest VM so that the guest has its own page tables distinct from the host, without the requirement to interpose each CR3 update made by the guest. All sensitive instructions can trap out of the VM mode, allowing the host to safely handle the instruction before reentering the guest, who remains unaware to what happened behind the scenes. All modern hypervisors support the hardware assisted virtualization mode which can achieve near-native performance.

Paravirtualization. The other virtualization approach which achieves near-native performance, is the *paravirtualized* (PV) approach. This mode was developed as an answer to the poor performance of binary rewriting when hardware virtualization extensions did not exist yet. Nonetheless, this mode has not yet lost its relevance, owing to the wide deployment and hardware features

[2]The CR3 register contains a memory address pointing to the location in RAM where the page tables reside.

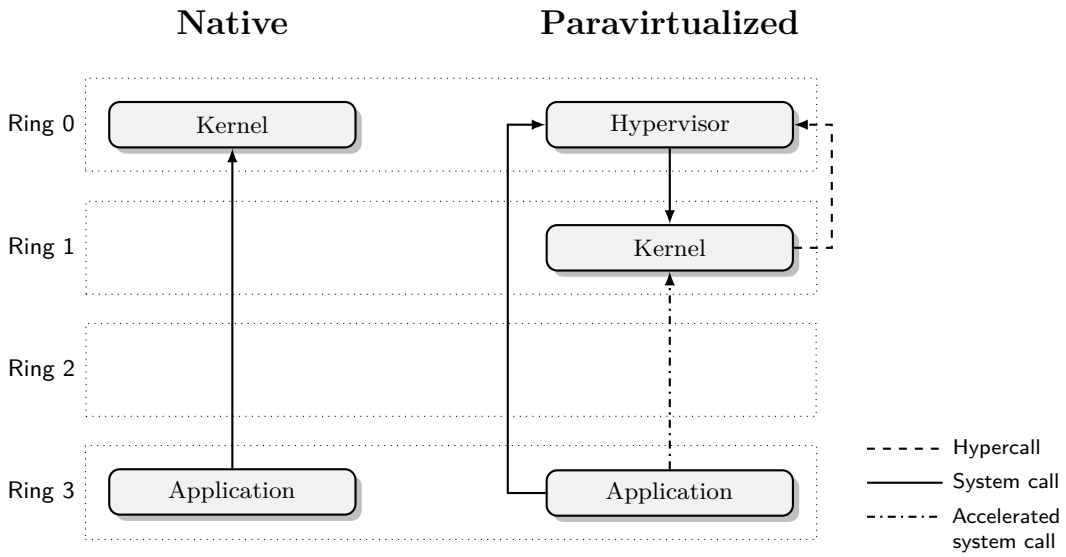


Figure 2.2: Overview of protection rings in native and paravirtualized environments (on IA-32). Adapted from Chisnall [**Chi07**].

independence. Before we continue, we first give a brief introduction to privilege rings in the *x*86 architecture.

On the *x*86 architecture, there are four protection rings available. Typical operating systems such as Linux and Windows utilize only two of these rings, referring to these as *kernel mode* and *user mode*. As seen in the left side of Fig. 2.2, this leaves two rings unused.

On the other hand, in a paravirtualized setting the kernel is lowered in privilege to *ring 1*, with a hypervisor taking its old place.[3] This means that the kernel has to be modified in order to relay certain tasks to the hypervisor. In other words, the kernel needs to be "PV-aware" to function in a paravirtualized environment, otherwise it will crash (since, as noted, not all instructions can be virtualized). Furthermore, there are considerable performance gains to be made if the VM kernel is PV-aware, for instance, by packing multiple page table updates into one request.

However, since now the hypervisor sits at the place of the kernel, all user space code that executes system calls (which works via raising an interrupt)

[3]The situation is more complicated on the *x*86-64 architecture due to removal of rings. Even though the general principles stay the same, additional workarounds are necessary which will not be discussed here.

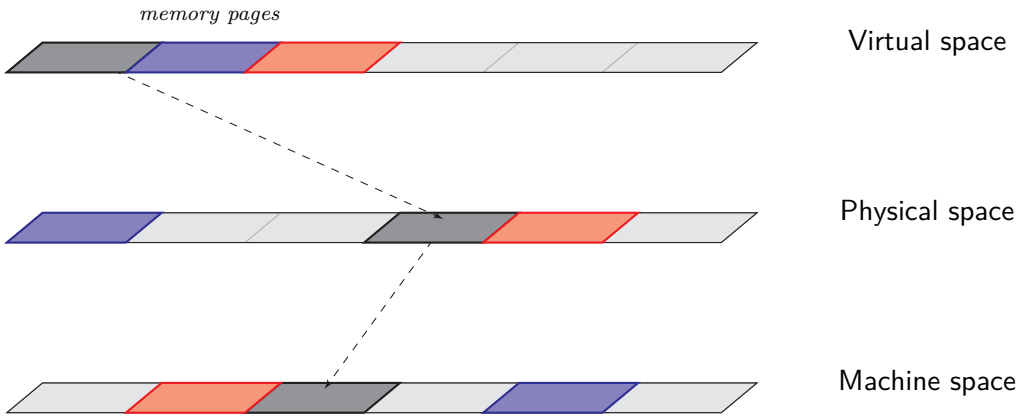


Figure 2.3: With OS virtualization, memory access goes through three layers of indirection. An application perceives the memory as the *virtual* address space, the OS perceives the memory as the *physical* space and the hypervisor sees the memory as the real *machine* space. Based on Chisnall [Chi07].

end up at the hypervisor and not the kernel. As depicted in Fig. 2.2, this means that the hypervisor has to forward these requests to the kernel. This extra ring switch is costly, therefore, *accelerated system calls* have been introduced which let the guest kernel install handlers for these via a hypercall, after which they operate at practically the same speed as ordinary system calls because the hypervisor registers them as exception handlers to the hardware.

2.1.3. Memory virtualization

In Fig. 2.3, we give an abstract view of how virtualization affects the memory.[4] This figure shows how seemingly contiguous virtual and physical pages can be randomly backed at lower levels. At the top layer, it is shown how the memory might appear to a running program. Beneath this layer, the OS backs this memory arbitrarily using pages from its memory pool, in order to multiplex the limited memory among different applications. Note that which virtual memory pages are backed, and by which physical page precisely, is up to the OS to decide. The hypervisor does the same steps at a level lower, in order to multiplex memory among multiple VMs.

The translation of memory addresses between layers in Fig. 2.3 is done

[4]While the terminology used in this figure is the Xen nomenclature, the principles are the same for all hypervisors.

in hardware for efficiency reasons.[5] However, while Fig. 2.3 is conceptually a good model to work with, the hardware support exists only for only *one* address translation. The fact that the hardware provides only one layer of abstraction, while virtualization needs two layers, has long been the Achilles' heel of virtualization on $x86$. The least intrusive solution is to use *shadow paging*, in which the hypervisor keeps a shadow table for use by the hardware address translation.

A more efficient approach — which requires paravirtualization and has helped spur the growth of paravirtualization — is the approach where a PV-aware guest OS asks the hypervisor to update the page tables on behalf of the guest OS. In both these approaches the result is that the hardware address translation is aware of only two of the three memory layers.

The most recent solution, as introduced by the aforementioned Intel and AMD hardware assisted virtualization modes, is the hardware assisted paging support. Using these new hardware extensions, it is now possible to efficiently have three levels of indirection without modifying the guest OS, greatly simplifying the virtualization effort. Nonetheless, the paravirtualized mode remains highly popular due to its extensive deployment at popular cloud providers and hardware support independence.

2.2. Introduction to the Xen hypervisor

Xen, first released in 2003 by a Cambridge research group [**Bar+03**], is the archetypal type-I paravirtualized[6] hypervisor and used extensively in public cloud deployments. Designed as a paravirtualized solution in the time when no hardware virtualization extensions existed, it could initially only run modified Linux guests. Nowadays, by making use of the QEMU processor emulator [**Bel05**], Xen can also run Microsoft Windows and unmodified Linux (or BSD) guests using either binary rewriting or a hardware assisted mode.

The Xen hypervisor has been developed with the philosophy that the hypervisor should be a thin and minimal layer. Due to this design decision, certain virtualization work is necessarily delegated to a level above the hypervisor. Therefore, one specific VM is imbued with higher privileges than a regular VM (cf. Fig. 2.1a) and is assigned to performing device emulation for the other VMs. This management VM is called the domain zero (Dom0) in Xen, where in Xen

[5] The hardware component that takes care of this is called the memory management unit (MMU) and it works closely together with the translation lookaside buffer (TLB), which caches results, to provide these translations.

[6] Although it also supports running in hardware assisted mode called *hardware virtual machine* (HVM) in Xen.

terminology VMs are called domains. In this thesis, we will often use the term domain when talking about VMs at a Xen architectural level; nevertheless, the terms VM and domain are interchangeable. Regular VMs are called unprivileged domains in Xen and referred to as DomUs.

The Dom0 is not meant as an ordinary VM in which to run workloads. On the contrary, the use of Dom0 must be restricted, since a compromise of Dom0 can affect all VMs on the machine. The first responsibility of Dom0 is to communicate with devices such as disks and network interface cards (NICs) and offer simple virtual devices to the other VMs (see Fig. 2.4). The second responsibility of Dom0 is to perform the complex operation of building domains, because in a thin hypervisor design such as Xen, this is not done by the hypervisor. The third responsibility of Dom0 is to provide a platform via which it is possible to give commands to the hypervisor, e.g., commands for starting or stopping a VMs.

In advanced deployment scenarios, Dom0 is an anchor via which the server can be made part of a larger cloud infrastructure. By leveraging additional cloud software in Dom0, launching a VM can then be achieved from a central location without directly logging into the Dom0.

2.2.1. Virtual devices

Domains in Xen cannot be allowed to talk to the hardware directly. In the first place this is a practical consideration. Since hardware and firmware tend to be stateful, interfacing them asynchronously by different domains will lead to undefined and unexpected behavior. Secondly, the interactions of one domain with these devices could affect other domains — violating the virtualization principle of isolation.

Therefore, Xen multiplexes hardware devices in Dom0, as exhibited in Fig. 2.4. In Xen, the DomU kernel is PV-aware and talks to Dom0 instead of directly to the hardware. Applications in DomU do not notice a difference since their kernel abstracts from this. In Dom0, the kernel reroutes input and output requests by the guest domains to its disks abstractions or filesystem abstractions, depending on the configuration by the cloud administrator. For instance, a DomU disk could be contained in a file container on a disk accessible by Dom0, either locally or network mounted.

Grant tables. The actual communication between Dom0 and DomUs takes place without intervention of the hypervisor. This is achieved by the use of shared memory which is implemented in Xen via so-called *grant tables*. Using grant tables, domains have discretionary access control over the memory pages

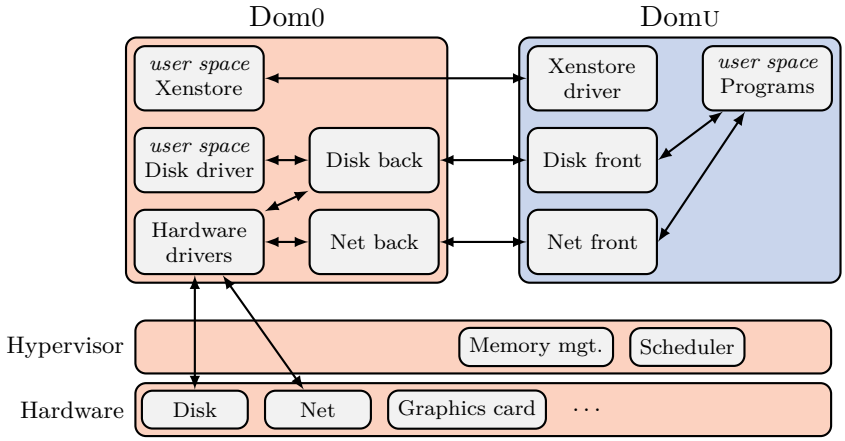


Figure 2.4: A typical Xen PV setup. Although details have been omitted, it can be seen how DomU relies on Dom0 for its I/O. Furthermore, in Dom0 not all logic needs to be implemented in kernel drivers. For instance, the Xenstore lives in user space and certain disk back-ends can live in user space, too.

grant ref	*MFN*	*dom id*	*flags*
⋮			
1375	42348	12	PERMIT
1376	42347	12	PERMIT | READONLY
⋮			

Table 2.1: An example of Alice's grant table with fabricated values. In this example, Bob's domain ID is the value 12. Additionally, one page is shared read-only.

they possess. A domain can share a regular memory page with another domain by adding an entry to its grant table indicating which memory page to share with whom.[7]

For example, domain Alice decides to share some of her RAM with domain Bob. Because she knows the domain identifier of Bob, she can put an entry in her grant table for a memory page she wishes to share. In case Alice wishes to share multiple memory pages with Bob, she will have to add multiple rows in the table, one for each machine frame number (MFN).[8] An example of Alice's grant table is shown in Table 2.1. After this, Alice tells Bob which grant references he should use when mapping the shared memory.

On the other side, Bob makes a hypercall to Xen, giving the domain identifier of Alice and the offset in her grant table indicated by the grant reference number. Xen then looks in Alice's grant table at the offset determined by the grant reference and checks if Alice has indeed listed Bob as the domain she wants to share the memory with. Xen does some sanity checking, such as checking whether Alice actually owns the memory page she wants to share and checks the flags. After these checks have been passed, both Alice and Bob can write simultaneously to the same piece of physical memory. With Alice and Bob writing to the same part of memory, this will require synchronization techniques, especially on symmetric multiprocessor architectures (SMP) architectures. The Xen solution to this problem is to use *ring buffers*. These are standard solutions in memory in which both sides can read and write simultaneously. The buffer is augmented with a set of running counters, accessible by both, that ensure that no data is corrupted as long as both parties behave as agreed.

One helpful feature Xen offers for inter-VM communication over shared memory are *event channels*. By using these event channels, domains can notify each other when there is data waiting to be read in a shared memory page. Setting up an event channel between two domains requires an action on both sides, similar to the use of grant tables. One domain opens an event channel on a randomly allocated port number, and the other domain connects to this port after which the channel can be used in both directions to send notifications (without a payload).

Xenstore. The above scenario made assumptions — it assumed Alice and Bob know each other's domain identifier, and that Bob knows the offset in Alice's

[7] For completeness, we note that grant tables also supports an operation to permanently *transfer* a page to another domain instead of only sharing. Because we do not use this operation in this thesis, we do not discuss this further.

[8] The typical page size used on $x86$ is 4096 bytes, so sharing multiple pages is often a necessity.

grant table (i.e., the grant reference) as well as the event channel port number. All this information needs to be communicated out-of-band between Alice and Bob. The solution used in Xen is the Xenstore: a key-value store organized in a tree (i.e. like a filesystem), accessible by both Alice and Bob via which domains in Xen can share small pieces of information (most commonly, information about virtual devices).

The Xenstore is served by Dom0 and it communicates with all the other domains via shared memory communication channels. For each of these domains, an initial communication channel with the Xenstore is set up when the domain is being created. Hence, the Xenstore serves, amongst other things, as a bootstrapping system; only this communication channel needs to be preliminarily setup, all other communication channels can be built with information derived from the Xenstore. All the Xen PV drivers, including disk and network, use this technique of discovery via the Xenstore.

2.2.2. Tools

Besides the components discussed so far, which had a direct relation to device virtualization, there are more Xen components which play a role in this thesis.

Domain building. Because Xen is a thin hypervisor, the responsibility of domain building (Xen terminology for VM creation) lies with Dom0.[9] The process of domain building is complex, but it can be summarized in the following description:

i. Dom0 parses the to-be-booted kernel and checks any flags inside;
ii. the hypervisor is asked to set up a new memory range for the new domain;
iii. the kernel and ramdisk images for the new VM are placed in this range;
iv. the virtual devices are attached to real devices inside Dom0;
v. the hypervisor is requested to schedule the new domain.

Toolstack. Xen comes with a Dom0 tool with which the cloud administrator can interact with the Xen system. This tool, `xl`, is the "Swiss army knife" with which the administrator can perform operations varying from starting or stopping a VM, to setting quotas or migrating away VMs. As an example, an administrator can create a VM with the following command, which will read the VM settings from the specified configuration file.

[9] This was not always so: Xen developers moved the domain building functionality out of the hypervisor and into Dom0 between Xen version one and version two [Chi07, p. 16].

```
# xl create /etc/xen/guest.cfg
```

The program logic of the `xl` tool resides in a separate library for which `xl` is merely a front-end. Hence, the VM can be managed by not only the `xl` tool but also by API calls to the Libxl library. We defer an in depth discussion of the toolstack to the implementation chapter, where we discuss our modifications.

Mini-OS. Xen comes with a minimal paravirtualized operating system specifically made for Xen, called Mini-OS, which makes many simplifying assumptions about the environment in order to achieve a very minimal trusted computing base (TCB). Mini-OS is well suited for creating various helper VMs with a much smaller TCB than approaches using a commodity OS would be able to achieve. We make extensive use of the Mini-OS system in our proposed design.

At a later point in this thesis, during the implementation chapter, we discuss Mini-OS in more detail.

2.2.3. Xen security

In Xen, a domain is *privileged* if it has certain rights which are not normally granted to regular VMs as used by cloud consumers. By default the only privileged domain is Dom0. Dom0 uses these powers via several privileged hypercalls.[10] An example of such a privileged hypercall is the hypercall with which Dom0 maps (at any moment) memory of other domains into its own page tables, i.e. introspection. This privileged hypercall is used at several places in Dom0 to perform necessary duties, and is by itself a legitimate hypercall.

Introspection is not only powerful because it can read any memory, it is also the case that a DomU cannot detect if it is being introspected. Introspection can actually enhance a DomU's security, e.g., through allowing the Dom0 to detect attacks and malware living in DomUs [**NBH08**; **GR03**]. However, exactly this introspection power can be abused by an attacker with access to Dom0 because there is no way to discern whether this power is being used for good or for malign purposes.

We identified three causes why Dom0 is privileged and, conversely, why it is difficult to deprivilege Dom0.

1. *Domain building.* As mentioned earlier, the Xen design advocates a thin hypervisor. Since, as explained in previous subsection, the domain

[10] The so-called *domain control* series of hypercalls. For details, see the hypercall Table F.1 on p. 124.

building process is complex it has (in the Xen philosophy) not a place in the hypervisor itself.

A specific case involves suspension and resumption. In a suspend operation, the state of a domain — essentially, its memory pages — is flushed to a file (and vice versa for resumption). During this serialization Dom0 reads the target domain's page tables and rewrites all page references to machine-independent values. This is a crucial step, and to accomplish it Dom0 needs privileges to access other VMs' memory.

2. *Lifecycle management.* The Dom0 performs administrative duties on behalf of the CSP. Lifecycle management involves launching, pausing, resuming, and migrating VMs. In addition, it also involves querying and controlling resources such as the scheduler used, processor and memory usage, or the system clock.

3. *Virtual devices.* Regular VMs cannot be allowed to talk to the *real* hardware directly, since this affects all other VMs. Therefore, Dom0 needs to have a privileged status to directly talk to the hardware. However, if ignoring direct memory access (DMA) attacks for the moment, this privileged status is not as powerful as in the first case. Namely, talking to hardware does not require Dom0 to introspect other VMs.[11] In addition, offering virtual devices to other VMs does not require a privileged status. When using a virtual device, a domain *offers* a memory page to Dom0 in which their communication takes place — it is not necessary for Dom0 to unilaterally map such memory.

Finally, another liability for a cloud consumer's data resides with the VM image. Since the virtual disks are emulated in Dom0 and not in the hypervisor, this ought to be taken in account in any security design based on Xen. Moreover, when a VM is not (yet) running, it is stored as a virtual disk in a file container. Such a file container can reside anywhere in the cloud, which makes it particularly challenging to assert that it is stored at a trusted location.

[11] In fact, in the current Xen architecture the permission of communicating with hardware is indicated using an I/O capability in the hypervisor on a per-domain basis and does not exclusively require a privileged status.

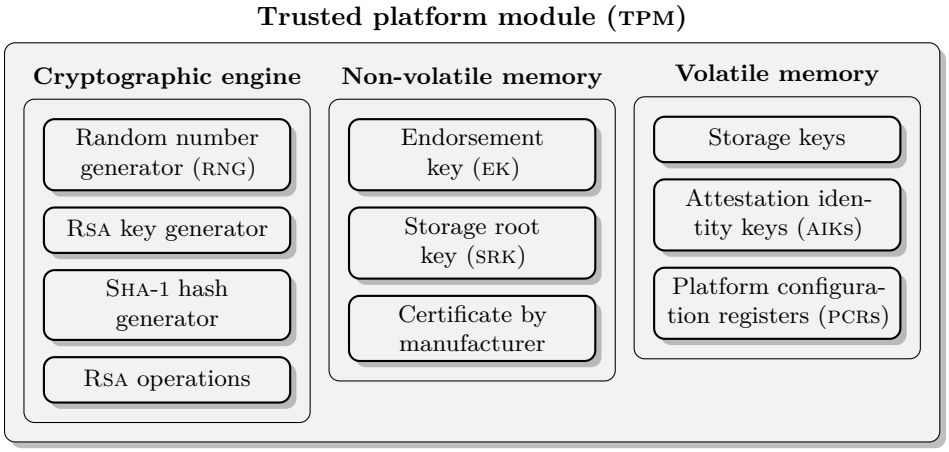


Figure 2.5: Schematic overview of a TPM. Based on TCG documentation [**TCGb**].

2.3. Introduction to trusted computing

In this section, a brief introduction to trusted computing is given. Trusted computing is an important building block for the *cryptography as a service* (CaaS) design proposed in this thesis. This section is organized in two subsections, *core concepts* and *operations*.

2.3.1. Core concepts

Terminology. The concept of "trust" can have many different interpretations. In this thesis, it is defined as follows.

- A *trusted* system or component is one whose failure can break the security policy, while a *trustworthy* system or component is one that won't fail [**And01**].
- The trusted computing base (TCB) is defined as the set of components (hardware, software, human, etc.) whose correct functioning is sufficient to ensure that the security policy is enforced, or, more vividly, whose failure could cause a breach of the security policy [**And01**].

  In other words, the TCB of a system is the set of its *trusted* components.

Background. The Trusted Computing Group (TCG) is an industrial initiative with the goal of establishing *trusted computing* (TC) technology. The TCG is backed by companies such as Intel, AMD, IBM and Microsoft [**TCG**]. The core of

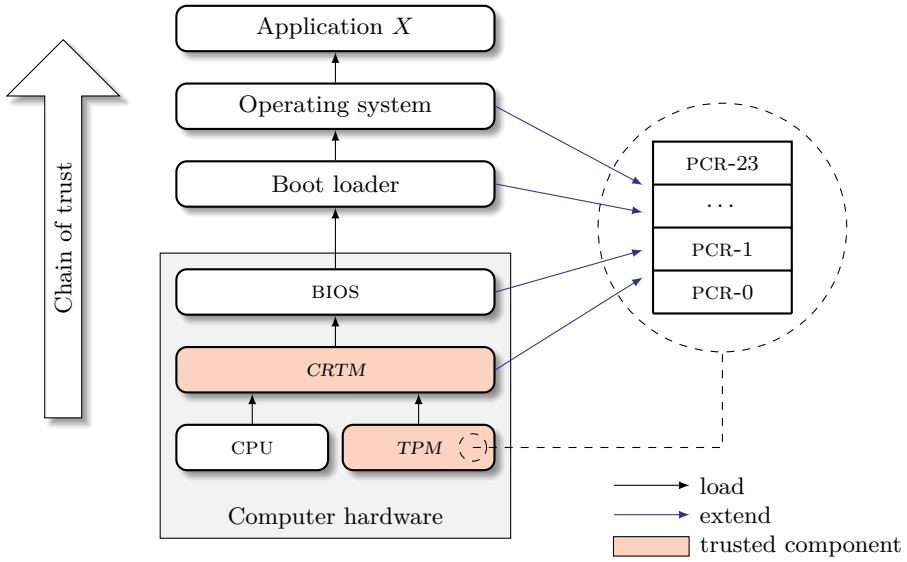


Figure 2.6: The chain of trust in an authenticated boot. Through the consistent use of the *extend* operation, the trust can be extended from the CRTM to applications running on the OS. Adapted from Sadeghi [**Sad11**].

their trusted computing technology is formed by a hardware security module soldered to the motherboard: the *trusted platform module* (TPM). The TPM is not a cryptographic accelerator (in fact, it is orders of magnitudes slower than a CPU) but provides, barring physical attacks, a tamper-proof environment for cryptographic operations. The TCG has published the design of a TPM over several specifications which describe the commands and data structures a TPM must adhere to [**TCG03**].

The TPM provides several security features, discussed in the following paragraphs. The information in this section is from Sadeghi [**Sad11**] as well as the TPM specifications [**TCG03**].

Cryptographic engine. In Fig. 2.5 a schematic overview of a TPM is shown. Depicted are the various cryptographic operations and the non-volatile and volatile memory components.[12] The TPM offers several cryptographic operations (encrypting, decrypting, signing and hashing) to the running OS. In particular, the TPM provides a secure generation, storage and usage of asymmetric

[12]Note that symmetric key operations are not natively supported. However, the TPM can be used to store such keys in a secure environment.

cryptographic keys. Unless explicitly specified during key creation, the private key-part of keys created by the TPM can never leave the hardware chip in unencrypted form.

Root of trust. The TPM enables the *measurement* of the software configuration which runs on the machine. Measuring means calculating a hash of a program binary and storing this hash digest in the TPM. In TPM terminology, a register which can hold a complete hash is called a *platform configuration register* (PCR), and storing a hash in a register is called the *extend* operation (defined more precisely below). The TPM comes with 24 PCRs, each of 160-bit length where a hash can be stored.[13]

This extend operation and these PCRs are invaluable tools for determining if a trusted configuration is running. Consider, for instance, if we wish to know whether program X is trusted (see Fig. 2.6). When communicating and interacting with this program, we might be led to believe that this program is trusted (or not), but to be certain, we would need to know the source code of X — or in the very least, we would need to know the hash digest of what a trusted version of X looks like.

But to determine what version of X is running, we need to ask this to the entity who loaded X: the OS. The statement that the OS will give, however, is only reliable to the degree that the OS itself is trusted. Hence, this chain of trust trickles all the way down to the very start of booting up the machine: a small piece of immutable firmware, called the *core root of trust for measurements* (CRTM).

The example shown in Fig. 2.6 is an instance of the so-called *static root of trust for measurements* (SRTM), also known as authenticated boot. Each step in the loading process involves a trusted program loading a new trusted program and extending the trust in the TPM. It is static in the sense that it not possible to create a root of trust at any later point — the extension steps must have been taken from the very beginning of booting the machine.

Authenticity of TPM. The TPM comes embedded with a unique master keypair of which the private part never leaves the chip: the *endorsement key* (EK). During the creation of the TPM chip, the manufacturer embeds the private and public parts of the EK into the chip. Furthermore, it also embeds a certificate (signed by the manufacturer) on the public key-part of the EK which vouches for the authenticity of the chip: the *endorsement credential*.

[13] We assume in this thesis exclusively version v1.2 of the TPM specification, which has more PCRs than the preceding version.

In principle, this certificate allows a third party to verify that messages signed with this EK come from a genuine TPM. Moreover, it allows a third party to communicate over a trusted channel with the TPM. However, due to privacy concerns of traceability, usually the EK is not used directly but an intermediary *attestation identity key* (AIK) is used which gets signed by an externally trusted certificate authority (CA).[14]

Since the TPM itself has only very limited non-volatile storage capacity, most keys are stored outside the TPM, typically on the hard disk. However, this does not compromise confidentiality or integrity of keys because these are encrypted using the *storage root key* (SRK) of which the private key-part never leaves the TPM.

Dynamic root of trust. The TCG recognized that there are scenarios in which a user might wish to begin a root of trust *after* booting. Therefore, Intel and AMD took steps to simplify the chain of trust. Both chip manufacturers introduced CPU instructions which allow the measured environment to be started at any arbitrary time after booting. Intel brands this technology under trusted execution technology (TXT) while AMD brands it as part of AMD-V [**TXT**; **AMDV**].[15] Such a chain is referred to as a *dynamic root of trust for measurements* (DRTM) because all the measurements (or rather, the lack thereof) up to invoking this operation do not play a role.

An instantiation of this technology, which will play a role in our implementation, is TBoot (trusted boot) [**TB**]. This technology is a trusted bootloader written by Intel which makes use of the TPM and relies on the TXT processor instructions found on most Intel chipsets. By using TBoot, one can boot a trusted configuration without worrying about the measurements related to previous components, such as the BIOS.

2.3.2. Operations

We briefly discuss the TPM operations that play a role in this thesis. The reader may choose to skim this subsection initially, and refer back to it when TPM operations are referenced.

Extend operation. Storing hashes in a PCR is achieved in chained fashion: instead of storing all hashes in a linked list, a new hash simply hashes an old

[14] Strictly speaking, this approach has now been replaced with direct anonymous attestation (DAA), which removes the dependency on a trusted third party (TTP) using a zero-knowledge protocol. The underlying idea remains the same, so we will not discuss DAA here.

[15] Formerly known as respectively Intel LaGrande and AMD-SVM.

hash as part of its input. This makes effective use of the limited memory in a TPM, and does not compromise security in any way.

The TPM stores these measurements in volatile memory which is cleared at boot time. To place a measurement via *extending*, the OS sends to the TPM:

command:	⟨*TPM_*⟩Extend(i, m)
result:	PCR i ⟵SHA1(PCR i, m)

Clearly, the value in a PCR i depends not only on the *last* but on *all* values that have been extended to that position and their ordering. When considering the chain of measurements in a specific PCR slot, then the trust in a certain hash after j extend operations, i.e. h_j, depends on the trust of the preceding extend operation h_{j-1}. The first hash in this chain, h_0, is extended by the CRTM. Because the TPM specification does not cover advanced physical attacks, this hardware measurement is axiomatically assumed to be trusted.

Attestation. An important aspect of having a measured environment, is to prove to third parties that a trusted configuration is running. This step, called *attestation* in TC terminology, utilizes the TPM to generate an authentic report for the *appraiser* party — who has to judge whether he or she trusts the TPM, and whether the stated PCRs correspond to a trusted configuration.

Attestation is achieved through the ⟨*TPM_*⟩Quote command. This command takes a nonce from the appraiser (in order to ensure freshness), and outputs a signed structure which contains the nonce and a list of PCRs. Typically, an AIK will be used for this purpose, with a trusted CA certifying that this AIK comes from a valid TPM.

Key creation and usage. The TPM can create several kinds of keys, including migratable and non-migratable. Migratable keys are keys which can be exported to another TPM (by encrypting it for that TPM's public key), while for non-migratable keys, the private key-part is strictly bound to this platform.

Keys are created using the ⟨*TPM_*⟩CreateWrapKey command. The command creates a new key and *wraps* it inside a parent key (hence the name), which must be loaded at that moment. In this way, the TPM stores its keys in a tree.

As mentioned earlier, most keys are not loaded into the TPM initially. Hence, before using an operation with keys, they first need to be loaded using ⟨*TPM_*⟩LoadKey2.

Binding. Binding keys are keys whose private key-part lives in the TPM, and for which the decryption, using ⟨*TPM_*⟩Unbind, is performed inside the TPM. For

migratable keys, this is nothing else than normal asymmetric encryption. For non-migratable keys, however, a useful scenario exists, which is to bind the key to the current platform.

This act of binding to the platform can furthermore restrict the use of the key exclusively to a trusted configuration. The TPM can also generate a certificate over this key which can be used as a proof for third parties, confirming that whatever they encrypt using this public key can only be decrypted if the system is in a trusted state. This kind of non-migratable binding key we also refer to as a *certified binding key*.

Sealing. Binding to a trusted state is a very useful TPM feature and can also be achieved using non-migratable storage keys, where it is known as *sealing*. The command ⟨*TPM_*⟩Seal takes a blob of data, the handle of a non-migratable storage key loaded in the TPM, and a list of PCRs to restrict to.[16] It will then encrypt this blob and return this to the caller; i.e., the encrypted blob is stored on external storage, but only the TPM will ever be able to decrypt it — which it will only do if the configuration is in the state specified during sealing.

An example use of this function is a trusted OS which saves its state to disk before rebooting. By sealing it first, it can ensure that in case the machine is booted into a rogue OS, the TPM will not disclose the data.

Localities. The TPM is a memory-mapped I/O device, meaning that one can talk to the TPM by merely addressing a fixed memory address. Furthermore, the TPM has the concept of *localities*, which are essentially privilege levels. These localities can be used in binding to the platform (i.e., restrict the use of a key to a certain locality) but also have effects on which PCRs can be used by which locality.

The TPM has a very simple way of discerning localities: communication with the TPM can take place at different memory addresses, and each different memory address corresponds to a different locality. The implicit assumption is that the first OS to boot has the highest locality possible, therefore, only this entity decides to which TPM locality any subsequently spawned programs are allowed to write.

[16] Due to the nature of the TPM as a secure but slow module, it makes sense to only use these operations to seal and unseal symmetric keys. These keys can then be used by the caller to encrypt/decrypt a larger data blob in a faster manner. This is also known as hybrid encryption.

Monotonic counters. The TPM comes with four independent monotonic counters. A monotonic counter is an integer which supports only two operations, a ⟨*TPM_*⟩ReadCounter command, which returns the current value, and a ⟨*TPM_*⟩IncrementCounter.

A trusted monotonic counter is a highly useful tool for situations where replay attacks need to be prevented. For example, if a trusted OS stores a virtual wallet to disk, it may decide to sign the data before writing it away.[17] Then, when the data is read back in a later phase, the signature on the data may convince the trusted OS that the data has not been tampered with. However, while this might hold true, it gives no guarantee that an adversary did not overwrite the data with an older copy.

The solution for this problem is to let the trusted OS (i) increase the value of the monotonic counter (invalidating all previous copies possibly stored on disk) and (ii) placing the new value of the monotonic counter in the blob to sign and write to disk. The next time the trusted OS reads the data from disk, it will reject the data if the value of the counter does not equal the saved counter value in the data blob.

While the TPM is a fine device for storing such a trusted monotonic counter, it comes with a limitation, namely, only one counter can be used at a time; a reboot is needed to switch between one of the four counters. While this might appear as a limitation, in fact, a single trusted monotonic counter suffices [**Sar+06**]. A trusted OS can offer an unlimited set of virtual counters to programs while relying on the TPM trusted monotonic counter for protection against replay attacks on its own virtual counters when these are stored to disk.

Sarmenta et al. identified that a monotonic counter should satisfy three requirements [**Sar+06**]:

1. The value must be non-volatile (i.e., must not be lost and must only change when explicitly incremented).
2. The value must be irreversible (i.e., no decrease is permitted).
3. The commands must be atomic (i.e., the behavior of two parallel read and increment operations must be defined).

Monotonic counters will play a role in our design to protect against rollbacks of state data by an adversary.

[17] Or alternatively, using encryption or with an hash-based message authentication code (HMAC).

Ownership. The TPM supports the concept of *ownership* of the TPM. Before a TPM can be used, it first must be taken ownership of. This ownership operation installs a new SRK in the TPM by overriding any old one, effectively invalidating any keys belonging to the previous owner.

All the keys that the TPM creates can be protected by the so-called *authentication data*, which basically is a password. Such a protection measure can be useful if the system is used by multiple persons besides the platform owner. Clearing the platform owner is typically done from the BIOS and will erase all the keys from the TPM. For this clearing operation, no additional password is needed; the fact that the user can enter the BIOS is sufficient proof for the TPM that this user is the (new) platform owner.

3. Problem description

As stated during the introduction chapter, the goal of this thesis involves enabling cloud consumers to securely deploy and run their virtual machines in the cloud and to protect their high-value cryptographic credentials against external as well as internal attackers. In this chapter, this problem is defined more precisely and security goals for our architecture are derived. First, we discuss the attack channels, our adversary model, and assumptions that we make. Second, we specify the exact requirements as set out in this thesis.

3.1. Attacker model

For analyzing our attacker model we look at the channels through which a VM can be threatened, and we discuss which adversaries use these channels.

3.1.1. Attack channels

Figure 3.1 exhibits the following channels which can be used to attack a VM.[1]

C1. The hypervisor has wide ranging capabilities to disrupt and subvert a VM. In a type-II hypervisor, this inherently also includes programs running in the hosting OS.

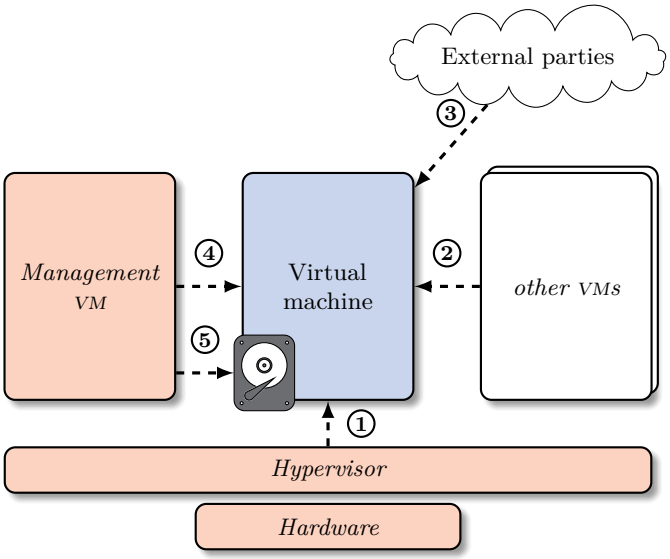
C2. The isolation between VMs is fallible and therefore a channel.

C3. Each virtual machine is typically connected to the network just as an ordinary physical machine would be.

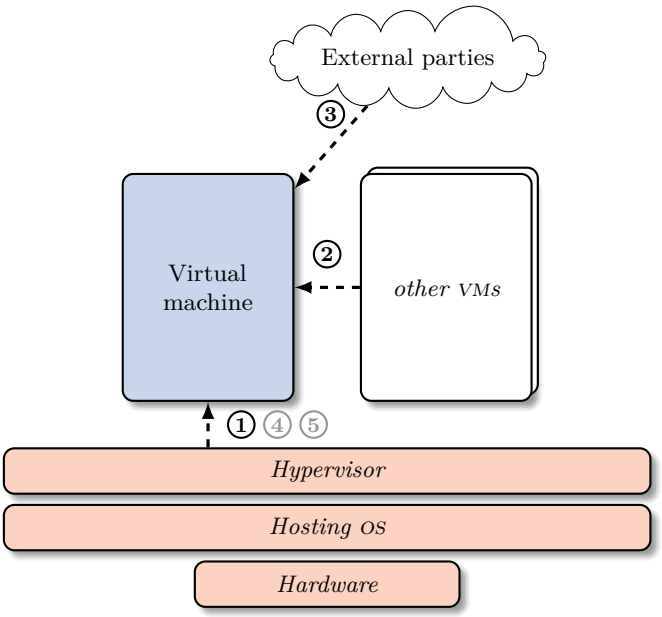
C4. The management VM can start or stop VMs and perform other maintenance. In some hypervisor designs, e.g. vanilla Xen, this channel has access to the memory of the VM.[2]

[1] Observe that channels C4 and C5 are exclusive to type-I; in type-II these cannot be discerned from channel channel C1. We remark that although the Xen hypervisor has both these type-I channels, this might not be the case for all type-I designs.

[2] We are primarily referring to accessing other VM's memory via hypercalls. However, we also consider DMA as part of this channel.



(a) Channels for the **type-I** hypervisor design.



(b) Channels for the **type-II** hypervisor design.

Figure 3.1: Overview of the channels through which attacks on a virtual machine can take place. The harddisk symbol is an abstraction for all virtual devices, including network traffic. Observe that channels which were shown distinct in type-I, are mangled into a single channel in type-II.

C5. The management VM can access the virtual devices belonging to a VM (i.e., the virtual I/O). This channel is a subset of channel C4, since when the security of the VM memory is broken, the virtual devices can be considered compromised as well. However, we will see cases where this distinction is relevant.

3.1.2. Assumptions

We have made assumptions regarding the channels. These are now briefly described here.

1. *No physical channel.* One kind of channel that we did not list, is the *physical* attack channel. There is hardly a limit to what a physical attacker can do against a system. For this thesis, we axiomatically rule out such attacks and assume that the data centers where the machines are located provide adequate physical security.

2. *Hypervisor assumed trusted.* We will ignore the hypervisor (channel C1) as an attack channel. While there is related work specifically addressing this problem (e.g., by reducing the hypervisor TCB [**SK10**; **Zha+11a**] or by removing the hypervisor altogether [**Kel+10**]), we consider it out of scope for this thesis.

3. *Hypervisor designs of type-II are not considered.* In a type-II hypervisor, which has no distinct management domain, we cannot consider the cloud administrator an adversary while on same time assuming the hypervisor is trusted (see the previous item).

   In other words, in a type-II hypervisor, we would be restricting the cloud administrator to only channel C3 instead of the much more interesting channels C4 and C5. Therefore, since countermeasures against the cloud administrator play a large role in this thesis, we decide to focus exclusively on the type-I category.

4. *No side channels.* In this thesis, we ignore the inter-VM attack channel (channel C2). For example, recent research has achieved interaction between co-resident VMs without abusing any hypervisor vulnerabilities, but merely by using hardware properties [**Ris+09**; **Zha+11b**; **Zha+12**].

   However, though some of these attacks will be discussed in the related work chapter, we will not attempt to protect against these attacks since the scope of this thesis does not deal with such advanced low-level attacks.

3.1.3. Adversaries

We consider two adversaries in our attacker model. We discuss these briefly and explain their relation to the aforementioned interaction channels.

1. *Malicious insider.* The first adversary is the malicious insider, also known as the cloud administrator (or operator).[3] This adversary has access through channels C4 and C5, originating from the management VM.[4] We remark that it is not strictly always a CSP employee who plays the role of malicious insider. If the management domain is compromised by a third party, then for all intents and purposes, this attacking third party plays the role of malicious insider, even though this party has no affiliation with the CSP at all.

   Moreover, the malicious insider has full control over the networking at the CSP.[5] Notwithstanding that how the malicious insider intercepts traffic on the network is an out of scope topic, for Xen, this interception risk is true in a more simple way. In Xen, all VMs will access the network through the management Dom0 (i.e., channel C5), and are therefore exposed even if the adversary does not intercept on the network itself.

2. *External adversary.* As second adversary, we consider the external adversary. This adversary finds its way through channel C3, the user's VM itself, by exploiting services which are accessible from the Internet.

   As a typical example, an external adversary compromises a webserver on a VM in the cloud. (Which is still very common, as highlighted by the earlier cited cloud survey [AL12].)

We have not incorporated the *co-resident adversary* in our attacker model. While this entity is definitely a factor to consider in cloud computing, it is not the focus of this thesis. By assuming that channel C2 is not relevant for this thesis, and likewise for channel C1 (we include attacks on the *management domain* by co-residents as part of this channel, ignoring the subtle differences in attacks), we effectively rendered this adversary without any channels of its own. Hence, we do not consider the co-resident adversary as part of the attacker model. A summary of all the adversaries and their attack channels is given in Table 3.1.

[3] The malicious insider is identified by the Cloud Security Alliance as top threat number three to cloud computing security [CSA10].

[4] See the work by Rocha and Correia for examples of how a cloud administrator can mount such attacks [RC11].

[5] More precisely, this network environment can be described in the Dolev-Yao model [DY83]. However, we will not pursue a formal protocol verification in this thesis.

adversary	*channels*
Malicious insider	C4, C5
External adversary	C3

Table 3.1: Summary of the attacker model, listing adversaries with respect to attack channels.

3.2. Requirements

First, we determine the security objectives of this thesis. Second, these are mapped to a list of requirements.

3.2.1. Security objectives

Recall the stated goal of this thesis as the protection of *high value keys* (HVKs) in the cloud. When considering this goal in the classical *confidentiality, integrity and availability* (CIA) security attributes,[6] the goal can be phrased as the protection of confidentiality and integrity of HVKs in the cloud. However, the third component of the CIA triad, availability, does not appear. Availability with respect to cloud computing is a wholly different topic, and less a subject of technological solutions rather than a topic of service level agreements and trusting the CSP that a single adversary is not able to bring down the whole network. For this thesis, we will consider availability out of scope.

Clearly, the protection of HVKs against the malicious insider will involve technical measures to curtail the power of the cloud administrator. However, protection measures that segregate the HVKs away from a malicious insider could also benefit the VM *as a whole* at little additional costs. While the protection of a VM as a whole has not been precisely defined at the outset of this thesis, the architecture proposed in this thesis lends itself very well to improving the whole VM security — if the HVKs are guaranteed secure, then these might be leveraged to extend security to the VM as a whole. Hence, we consider the confidentiality and integrity of the *VM as whole* as a requirement, too. (Though with a weaker attacker model than for the HVKs, since, as stated earlier, securing the VM as a whole against external attackers is not realistically achievable at the infrastructure level.)

A summary of the core security objectives in this thesis is given in Table 3.2, where the confidentiality and integrity objectives are listed with respect to

[6] The CIA triad represent the three most ubiquitous security requirements and forms a useful methodology to analyze the security of systems. (See for example Bishop [**Bis04**].)

	Malicious insider	*External attacker*
Confidentiality	VM	HVKs
Integrity	VM	HVKs

Table 3.2: Summary of security objectives based on the objectives (vertical) and the adversaries (horizontal). Per field we indicate whether we protect only the HVKs or the complete VM.

adversaries from the attacker model. In addition, there are more security objectives to consider which are not easily expressed in the CIA terminology. For example, consider protection against rollbacks. In the following subsection, we map all objectives to a list of requirements.

3.2.2. List of requirements

The following requirements are identified based on the problem description.

R1. *Confidentiality and integrity of cloud consumer's* ***high value keys***: these must remain protected with respect to the attacker model, i.e., the malicious insider and external adversaries.

R2. *Confidentiality and integrity of cloud consumer's* ***virtual machine***: these must remain protected with respect to the attacker model, i.e., the malicious insider.

R3. *Strong coupling of cloud consumer's HVKs and VM*: a cloud consumer's HVKs may only be made available to the correct VM.

R4. *Freshness*: it must be prevented that the malicious insider rolls back to a previous version of the HVK states which are made available to a VM. (To prevent, for instance, loading of revoked HVKs keys.)

R5. *Migration*: the cloud consumer's VM *with* HVKs must be migratable to a different trusted node.

4. Related work

Through the years, a large body of academic literature has been published related to the security of virtualization and cloud computing. However, no universal solution for this major topic has been forthcoming yet, although improvements are being made in several areas. Hopes have been placed in the area of cryptography, in particular such that decryption keys with the CSP is not necessary.

In the area of cryptography, the recent development which has generated a lot of attention in the security community is *fully homomorphic encryption* (FHE) as put forward by Gentry [**Gen09**]. Using this technique, it is theoretically possible to outsource computations to a third party, without actually revealing the data or computation itself. Hypothetically, this could be the panacea that allows consumers to utilize the cloud without disclosing their secrets. However, in its current form this technology is far from maturity. For the foreseeable future, more practical and efficient approaches are necessary. Moreover, Van Dijk and Juels proved that FHE can only satisfy a very limited set of use cases, and cannot provide complete privacy when dealing with applications which take input from multiple clients [**VDJ10**].

Therefore, pragmatic solutions, that make less strong assumptions, are still highly relevant for the foreseeable future. In this chapter we will review several of the papers which are most related to the work in this thesis, discussing for each paper the applicability for the requirements from section 3.2.

4.1. Virtualizing the TPM

Berger et al. proposed virtualizing the trusted platform module [**Ber+06**]. To achieve this, they proposed the general architecture as seen in Fig. 4.1, in which the management domain takes care of the TPM emulation by spawning a new virtual TPM (vTPM) for each new DomU. A vTPM is (in principle) a complete TPM; though it is implemented in software form instead of the traditional form as a dedicated hardware module.

However, a vTPM suffers from a marked disadvantage: a virtualized TPM does not possess a manufacturer certificate on its EK like the real TPM does. Unfortunately, reusing the EK from the TPM is not an option since a TPM never

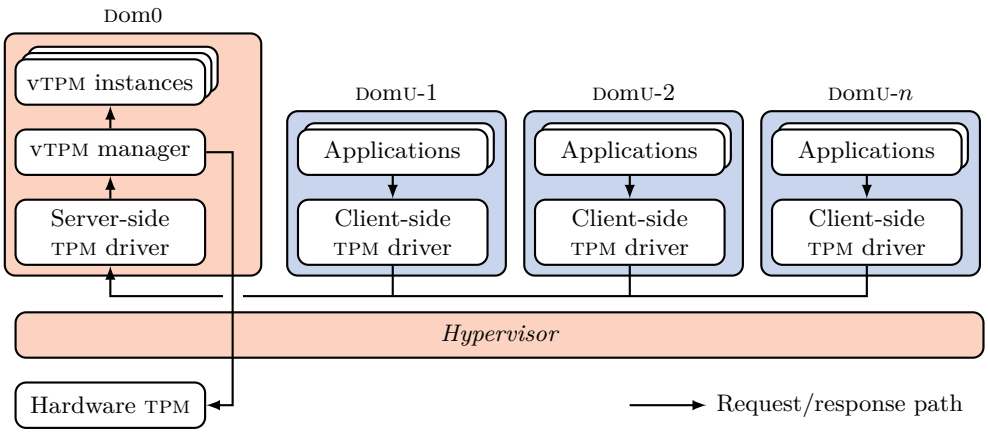


Figure 4.1: Standard virtualized TPM architecture. Adapted from Berger et al. [**Ber+06**].

discloses the private key part of its EK. One might try forwarding EK related requests from the vTPM to the hardware TPM, but this will cause inconsistencies. For example, there are operations in which a TPM creates and signs a certificate over non-migratable keys that it has created. The TPM will, however, never certify a key that it has not created itself (such as those created in the vTPM), nor is there any point in bringing a vTPM key into the TPM — the TPM will not certify migratable keys, for it cannot make guarantees over keys which have left the platform.

Therefore, Berger et al. propose three approaches on how to bind the vTPM to the physical hardware TPM. The three proposed approaches range from signing the vTPM EK/AIK using the *physical* TPM counterparts, to having a local authority provide the vTPM AIKs. A detailed discussion is out of scope, though for this thesis it is relevant that each approach comes with a trade-off, and all approaches require awareness at the challenger side.

Applicability. As seen in Fig. 4.1, the vTPMs live in the management node (Dom0 in Xen). Hence, even though the vTPM provides an effective way to harbor our HVK secrets in a isolated location (req. R1), it only guarantees this versus the external adversary and not against the cloud administrator. Likewise, the VM is not protected against the cloud administrator in their design (req. R2). As long as the vTPMs live in Dom0, these limitations cannot be alleviated.

Furthermore, the authors claim the vTPM measures domains once they are started [**Ber+06**, p. 11]. However, Murray et al. claim that this implementation is

not waterproof [MMH08, p. 6]. Supposedly, it risks a time of check, time of use (TOCTOU) attack by not correctly making the domain builder code vTPM aware. When looking at the code currently in the Xen source tree, this is indeed seems to be the case.

Providing vTPM functionality in our architecture is desirable and should be supported. However, with respect to a cloud-based deployment and potentially malicious administrators, we have to address the design and implementation differently.

4.2. Property-based TPM virtualization

Winandy et al. reviewed the vTPM designs published so far, identified weaknesses, and proposed a property-based vTPM design as improvement [SSW08]. The concept of a property-based TPM is already older, and a *virtualized* property-based TPM is the logical evolution of this well established concept. The original property-based TPM idea stems from Sadeghi and Stueble [SS04].

The prime weakness that Winandy et al. found, and which their design addresses, is that PCRs are a too inflexible indicator for whether the system is in a trusted state or not. For example, if the hypervisor is updated to fix a vulnerability, then after reboot neither unsealing nor attestation will succeed anymore if data is bound to the measurements of the old hypervisor code. This situation can also arise when migrating a VM from one machine to another machine and if the hypervisor versions do not match.

Their proposal is to add a layer of indirection in front of the actual PCR values, resulting in the so-called property-based vTPM displayed in Fig. 4.2. This indirection makes it possible that many different kinds of property providers can be embedded in the vTPM. For example, the traditional implementation of the ⟨*TPM_*⟩Extend command as we described it in section 2.3 is to cumulatively hash inputs in each call — this could be one of the property providers.

However, a more interesting property provider is one which, for example, does not store the plain hashes but rather stores certificates indicating whether the measured software satisfies Common Criteria[1] requirements. The idea behind this is that such a property provider does not simply store the hash that the vTPM receives from the VM during an *extend* operation, but rather that it queries a TTP about this hash. A TTP then provides a certificate to the property provider if the hash corresponds to a certain property for which it can give a certificate.

[1] The Common Criteria is a well established security engineering standard which can be used to specify security requirements [CC].

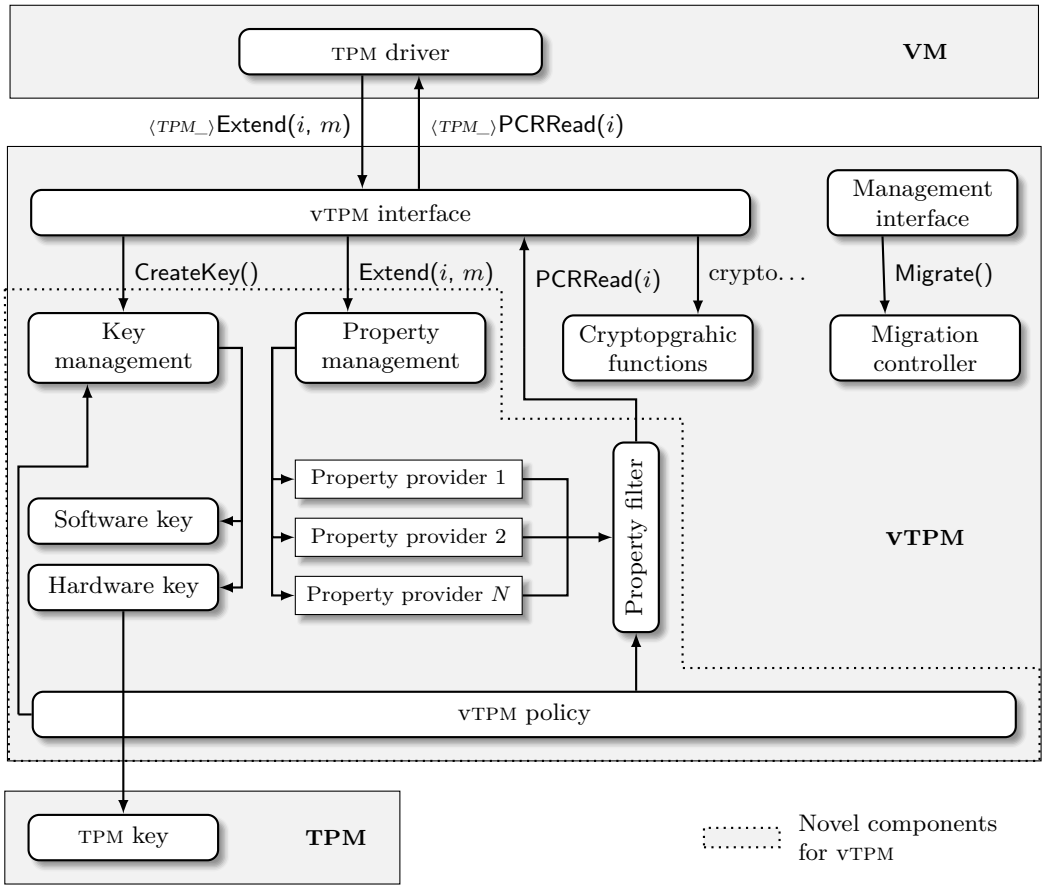


Figure 4.2: Virtualized property-based TPM architecture by Winandy et al. Observe how multiple distinct property providers can be the underlying mechanism implementing the ⟨*TPM_*⟩Extend call.

When the VM attempts to read the PCR values using e.g. the ⟨*TPM_*⟩Quote command, the vTPM internally uses a property filter with a policy which defines how the aforementioned properties are mapped to the much simpler and smaller 160-bit PCRs.[2]

The immediate advantage of this abstraction is that, even if binary hashes change, the certificates are not necessarily invalidated. Hence, the ⟨*TPM_*⟩Quote, ⟨*TPM_*⟩Unseal, and ⟨*TPM_*⟩CertifyKey operations continue to work if the properties still hold. This corresponds exactly with the observation made by Sadeghi and Stueble, namely, that users care about *properties* rather than *configurations*.

Moreover, Winandy et al. noted that their property based vTPM design has advantages over the standard vTPM that was proposed by Berger et al. earlier. Namely, the plain vTPM design map the lower PCRs of the physical TPM to the lower PCRs of the vTPM, because this allows the configuration of the underlying hypervisor to be included in attestations by the VM. However, the shortcoming of such an approach is that when the VM moves to a new hypervisor, the VM cannot use its old sealed data. On the other hand, the property-based vTPM does not suffer from this limitation, because multiple hashes can map to the same property.

Applicability. In summary, the property-based vTPM has a distinct advantage over the vTPM which was discussed earlier in this chapter. However, in spite of these advantages, practical considerations limit the role property-based vTPMs can play in this thesis.

For an ordinary vTPM, the source code is readily available; unfortunately, not so for property-based vTPMs. Moreover, the problems addressed by the property-based vTPM are orthogonal to the problem statement of this thesis. By itself, property-based vTPMs are not more secure than vTPMs with respect to the adversaries from our attacker model. This means that we will not implement property-based vTPMs for this thesis; though we include this concept in the key deployment discussion in the architecture chapter.

4.3. Disaggregating Dom0

Murray et al. reviewed the TCB of Xen, and their conclusion is that the TCB is too large primarily due to the domain building code residing in Dom0 [MMH08]. This is an important observation, and it also influences the architecture proposed in this thesis.

[2] Property-based vTPMs still comply with the TPM specifications [TCG03].

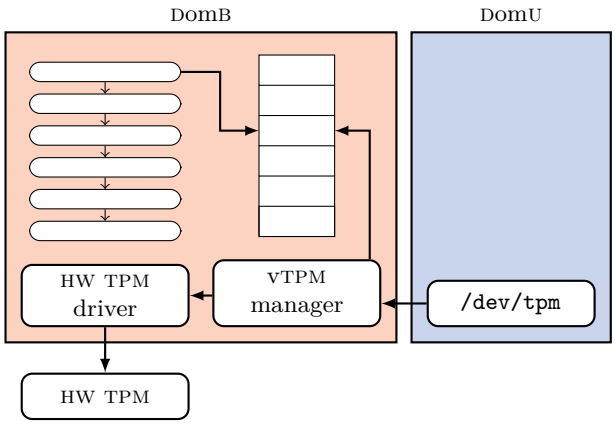


Figure 4.3: The DomB domain builder design. All vTPMs live in the DomB. Adapted from Murray et al. [MMH08].

Recall that the Xen hypervisor is a thin, type-I hypervisor, and that it explicitly defers the domain building process to the management domain. Furthermore, recall that the domain builder process involves privileged steps which include the power to introspect other VMs.

Murray et al. put forward a solution which greatly reduces the TCB by removing Dom0 from the TCB. The authors propose to put the domain building code in a very small domain, running Mini-OS. This small domain, named DomB, has the introspection powers that Dom0 used to have and is therefore part of the TCB. In contrary to Dom0, this DomB does not stand under supervision of the cloud administrator. Therefore, it can be considered a 'user space' extension to the hypervisor.

Furthermore, the authors have realized the synergy that is possible with vTPMs architectures. By placing the vTPMs inside their DomB, they claim protection of the vTPMs against Dom0, though it would be better to place a vTPM in a private, isolated domain. Figure 4.3 illustrates how Murray et al. have added vTPM functionality to DomB. Also shown is a hardware TPM driver, which they use to associate the vTPM with the hardware TPM.

Undiscussed in their paper is how DomB *itself* is booted, which is a chicken-and-egg problem.

Applicability. At first sight, it appears that their approach provides confidentiality and integrity of the cloud consumer HVKs and VM (requirements R1 and R2) because Dom0 cannot introspect anymore.

However, the authors leave some security aspects unanswered — which are important for an actual deployment. First, they do not address secure storage of the vTPMs' state during operations such as rebooting or suspension. Second, it is unclear how the cloud consumer, leveraging a vTPM hosted in DomB, can actually deploy credentials to a vTPM in a secure fashion.

With regard to the first point, they claim that the cloud consumer can take care of the protection of the secondary storage of his or her VM using full disk encryption (FDE) such as Bitlocker [**Fer06**]. While this argument itself holds, it presupposes that the vTPM is trusted, and they have not argued how they would do this — while the vTPMs are safe in DomB during runtime, this is not the case if the vTPMs themselves are flushed to secondary storage when the hypervisor (and thus also DomB) reboots.

With regard to the second point, even if we assume they would seal these vTPMs to the physical TPM, this does not explain how cloud consumers get their HVKs securely to their vTPM without the risk of interception or a man-in-the-middle attack by Dom0 over channel C5.

Hence, their proposal by itself is not sufficient for our goals. Nevertheless, the essence of their solution, namely the disaggregated domain builder, plays a role in our architecture, presented in chapter 5.

4.4. Breaking up is hard to do: security and functionality in a commodity hypervisor

Colp et al. take the concept of disaggregating Dom0 a step further in their 'Xoar' design and disaggregate Dom0 into nine classes of service VMs [**Col+11**]. These nine classes are chosen such that each class represents a single piece of functionality originally located in Dom0. For instance, a domain-builder domain, a Xenstore domain, and a disk back-end domain are service domains introduced by their Xoar design.

In addition to the immediate isolation benefits that such a far-reaching segregation provides, the authors add extra security features to control these service domains. Some of these additional features are fairly standard and are not discussed (e.g. a policy for hypercalls, off-site logging), while other features are an interesting new approach which can strengthen the disaggregation. For example, they propose automatic (policy-based or timer-based) snapshots and rollbacks of the service VMs, based on observations that it is generally easier to reason about a program's correctness at the start of execution rather than over long periods of time [**Can+04**]. A use case of such automatic rollbacks is communication with the Xenstore service domain: the policy could stipulate

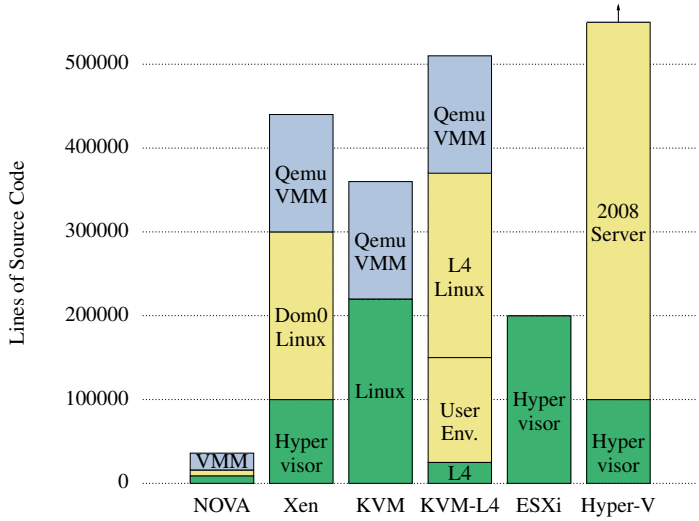


Figure 4.4: Overview of various TCBs. Figure from Steinberg and Kauer [**SK10**].

that the Xenstore is restored after each request, meaning that each request can be reasoned about as if it is the first after booting, making analysis easier.

Applicability. Xoar is the epitome of disaggregating the management domain in Xen. However, in this thesis disaggregation is a tool for achieving a higher purpose, and not a goal by itself. Disaggregation of Dom0 helps us to ensure that the cloud consumer's assets are secure against the malicious insider — but any disaggregation beyond this level does not (essentially) add security to our design. Hence, while the Xoar effort is commendable, considering that their source code is not available, we will not attempt to recreate their advanced degree of disaggregation in this thesis.

4.5. The NOVA microhypervisor

Introduction to microkernel design. Operating system kernels are generally divided into two categories: monolithic kernels and microkernels. The former refers to the architectures that contain relatively much code that runs in kernel mode, while the latter strips the code in kernel mode to the bare minimum (e.g., inter-process communication, virtual memory, and scheduling) and runs as much code as possible in user mode. Advocates of microkernels highlight the cleaner design and security benefits of the inherently reduced TCB. A typically cited

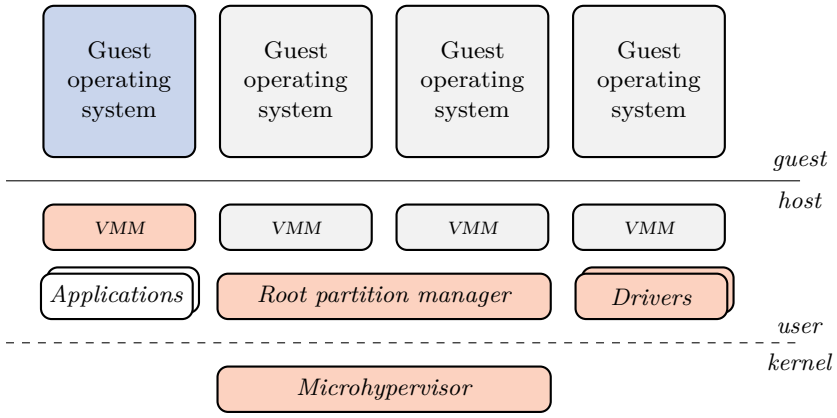


Figure 4.5: NOVA architecture. Since the microhypervisor strictly separates between processes, not all code running in host mode is in the TCB. As an example, we highlight in red the TCB components for the first VM (marked in blue) to show that the strict isolation in the microhypervisor design implies that not all hypervisor components are in the TCB. Figure based on original by Steinberg and Kauer with different coloring [**SK10**].

example is that hardware drivers do not need to run in kernel mode anymore. This is cited as a great benefit since hardware drivers tend to be the most prone to bugs in the code. On the other side, monolithic kernel proponents claim that the overhead in microkernels leads to performance degradation [**Lie96**].

While the debate fizzled out after some years, with monolithic kernels seeming to have the upper hand (Linux and Windows are both examples), there has been renewed interest. With the advent of the L4 kernels [**Lie+95**], which greatly reduced the performance issues associated with microkernels, many new variants and derivatives have been developed, although hardware vendor driver support remains problematic.

NOVA. *NOVA OS virtualization architecture* (NOVA) by Steinberg and Kauer is the logical extension of a microkernel design with virtualization capabilities [**SK10**]. It is a response to the critique that many of today's commodity hypervisors have a too large TCB. If a microkernel can be turned into a hypervisor without losing its attractive features; then they might be an excellent candidate for designing a hypervisor from scratch.

The NOVA architecture makes heavy use of the Intel and AMD virtualization CPU instructions (which were briefly introduced in section 2.3) to provide an emulated physical machine to oblivious VMs at minimal performance penalties.

These CPU instructions do not merely add an extra protection ring to the processor, but in fact they double these by cloning all these protection rings into an environment in which the VM runs (the so-called *guest* environment, as opposed to the *host* environment in which the hypervisor runs). This is visible in Fig. 4.5, where it can be seen how NOVA utilizes the rings available in the *host* environment to create different levels of privilege for code in the hypervisor itself. Examples of hypervisor code running in lower privilege are the virtual machine manager, drivers and custom administrator-provided additional programs known as *NOVA applications*.

It is claimed that by the broad application of the principle of least privilege, the NOVA design should be more secure than competitive designs. However, these claims are hard to gauge. Nevertheless, the authors of NOVA presented a comparison of source lines of code, visible in Fig. 4.4, with which they argue that their approach provides a drastically reduced TCB.

Applicability. In order to consider how NOVA could satisfy our requirements, we imagine the following scenario: A vTPM is turned into a NOVA application, which (as mentioned above) runs in the hypervisor in user mode. This vTPM then offers cryptographic services to its associated virtual machine via emulated devices. Furthermore, it also is straightforward to apply passthrough encryption in the VMM to any disk or network streams passing through.

Such a design would then allow a cloud consumer to store HVKs in this vTPM, safe from externals and from the cloud administrator — at least, as long as no malicious code is loaded into the hypervisor by the CSP.[3] Likewise, because there is no management node, the cloud administrator cannot interfere with the virtualization process. This would be satisfactory for both our two most important requirements: the security of the HVKs and VM (R1, R2).

However, the lack of a management node is very crippling, even unrealistic for CSPs. A CSP will not do the administration of all the machines manually, so some sort of management node must be introduced on which a cloud administrator has privileges.

Furthermore, all microkernels suffer from a lack of hardware support and NOVA is no exception. Hardware vendors are already struggling with providing drivers for the many different monolithic kernels out there, let alone microkernels. Neither the prospect of a hardware lock-in nor the prospect of having to write drivers themselves is much appealing for a cloud service provider. As such, we

[3] It will be necessary to use TC techniques so that the cloud consumer can verify this. A deployment scheme similar to what we will describe in the architecture chapter can be used. However, for brevity, we will not pursue this further for NOVA.

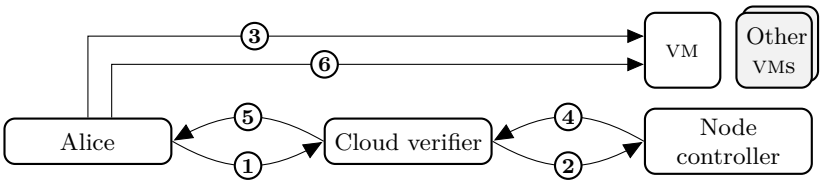


Figure 4.6: The cloud trust anchor design by Schiffman et al. [Sch+10]. ① Alice requests an attestation of the CV and its criteria. ② The CV verifies the cloud's NCs. ③ Alice starts her VM. ④ The NC sends an attestation and identity key of Alice's VM to the CV. ⑤ The CV forwards these to Alice. ⑥ Alice uses the key to authorize her VM to access her data on the cloud after validating the key signature and verifying the VM attestation.

decided not to pursue our solution using NOVA.

4.6. Seeding Clouds with trust anchors

Schiffman et al. make the observation that the trusted computing infrastructure in today's clouds is very limited [Sch+10]. The two main challenges they highlight are [Sch+10]:

1. Cloud consumers have no knowledge on which host their VMs are (or will be) deployed. However, the CSP requires them to provide their data beforehand, thus in this gap the security may already be violated.

2. Due to its nature, the TPM is a slow device and responding to many attestation requests from the many consumers located on a host could be a performance bottleneck.

Their proposed solution is a cloud verifier (CV). This is a service which vouches for the integrity of a cloud consumer's VM as well as the underlying system, i.e. it vouches for the enforcement of properties on the target hosts. As such, cloud consumers delegate to the CV to perform the job of attesting the target infrastructure. By attesting the CV, cloud consumers can verify that their trust in the CV is justified.

In Fig. 4.6, a summary of their architecture is shown. The basic principle is that Alice's VM can *only* access her data if the TCB is properly attested. The CV performs this attestation on her behalf, while she will attest the VM.

In their paper, they discuss their implementation details (such as using IPsec as well as more details about the protocols) but that does not change the design as discussed so far.

Applicability. One observation about this scheme is that Alice's VM is already booted and running before Alice has received the necessary information to make a judgment about the trustworthiness of the infrastructure. In other words, the VM that Alice supplies to the CSP, must be split across at least two stages. The first stage must not contain any data of which the confidentiality or integrity is to be protected. Only in the stage two, when the VM can access encrypted storage after having received the necessary keys from Alice, can data be stored which ought to be protected.

While their design is interesting, the dual approach to deploying VMs to the cloud is unappealing. Preferably, Alice should not need to split her data in two parts. This is a point which is addressed in the CaaS architecture proposed in this thesis.

4.7. Cloudvisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization

Cloudvisor is a tiny security monitor placed underneath a commodity hypervisor [**Zha+11a**]. The authors of Cloudvisor assume that the hypervisor is an untrusted component, therefore, they position Cloudvisor in the role of enforcing access control on the hypervisor.

Their approach is to degrade the privilege level of the hypervisor to that of an ordinary VM kernel and letting only Cloudvisor run at the highest privilege level. To achieve this, Cloudvisor interposes on updates of page tables as well as on disk input and output. The authors claim that Cloudvisor provides a complete confidentiality and integrity protection for the customer under a strong attacker model; not only other customers but also the cloud operator is assumed untrusted. Their confidentially and integrity claim — in face of this strong attacker model — relies on the established TC technologies (cf. section 2.3). Without trusted computing, a tampered Cloudvisor can void all security claims.

Cloudvisor exists only for Xen, although the authors claim it can be ported to other hypervisors.

Applicability. The functionality Cloudvisor provides can be summarized in two points: (i) protection against the cloud operator, and (ii) the reduction of the TCB of Xen. These are two orthogonal issues.

Regarding the protection against the cloud provider, the authors give only little information on how the Xen components have been changed. However,

there are some hints in which they explain what they have changed. In the latter part of their paper, they mention, "*... we will also investigate ... to support para-virtualization in Cloudvisor.*" From this we understand that Cloudvisor does not support paravirtualization but depends on hardware virtualization extensions. Paravirtualization is the default and dominant mode for running Xen on all major clouds such as Amazon and Rackspace. Furthermore, it is the only way to use Xen on platforms without hardware virtualization extensions. Certain clouds, such as EC2, do provide the opportunity to run HVM VMs, albeit with the restriction of Windows only or only available for larger instance types [**WJW12**].

The authors also write, "*Cloudvisor ensures that a physical memory page can only be assigned to one owner at a time.*" This is the primary cause for the significant Cloudvisor disk overhead of up to 55% for I/O intensive applications such as webservers [**Zha+11a**]. For all I/O, Cloudvisor has to duplicate the data from the DomU's I/O buffers into hypervisor buffers before the data is fed to the hardware device.

Not only does their insistence on a single owner per memory page cause this performance penalty, it also neglects the fact that shared memory pages play a fundamental role in paravirtualized Xen.[4] The authors stated their intent to extend CloudVisor to PV mode. If they were to proceed with that, some form of discretionary memory access control would need to be added to Cloudvisor, causing a further increase of Cloudvisor's TCB.

TCB reduction. The reduction of the Xen TCB provided by Cloudvisor is a valid goal; during the last years, many authors have shown that improvement is necessary [**SK10**; **MMH08**]. Xen's large TCB is primarily caused by the inclusion of the management domain (Dom0) in the TCB. However, the authors of Cloudvisor make the conjecture that the hypervisor component is also bloated and needs reduction. This seems a contradiction with the fact that Xen is a bare-metal hypervisor (type-I) which defers a lot of non-essential work to the management domain (e.g., domain building and communication with hardware devices). Hence, the Cloudvisor effort displays a large overlap with the Xen development which already focused on keeping the hypervisor minimal.

A consequence of this overlap could be that the claimed TCB gains by Cloudvisor might not be sustainable in a real environment, as opposed to the laboratory conditions under which Cloudvisor was developed. For instance, as soon as Cloudvisor has to support paravirtualization or all of the multiple architectures (*x*86, *x*86-64, IA-64, ARM) that the Xen hypervisor supports, the code will grow.

[4] Refer back to subsection 2.2.1 for a discussion on grant tables and how all PV drivers depend on this.

At this point, it is more logical to compare Cloudvisor not with vanilla Xen, but rather with other solutions which do not support paravirtualization. In face of these other approaches, Cloudvisor hardly comes across as an architecturally appealing solution.

Consider the following design choices they made. For instance, for operation of the full disk encryption, the Cloudvisor security monitor depends on a daemon running in the management domain, i.e., a communication channel bypassing the hypervisor and breaking the encapsulation. Furthermore, the role that remains for the hypervisor in the Cloudvisor design seems unclear. The prime existential reason for the Xen hypervisor, namely tracking page ownership and page table updates, has been overtaken by the Cloudvisor monitor. Therefore, the Xen hypervisor could be removed or stripped down in the Cloudvisor architecture. Effectively, this is a patchwork approach to a microhypervisor (cf. section 4.5) in which certain hypervisor duties run at a lower privilege level. In that respect Cloudvisor offers no real benefits over designs such as NOVA, which have a clean architecture and a TCB in the same order of magnitude. (Cloudvisor's claimed 5.5 kilo lines of code (KLOC) and NOVA's 9 KLOC are in the same magnitude when compared with a Linux of more than 200 KLOC, see Fig. 4.4.)

The orthogonal proposal the authors make, namely to use TC technologies to ensure to the cloud consumer that a trusted version of the virtualization environment is running, is indeed a viable approach and also a central theme of this thesis. However, their discussion of using TC technologies does not go beyond plugging it into their design and requiring that attestation must be used, omitting the details. They only suggest a simple protocol which relies on attestation and hybrid encryption, referring to "*the public key*" without specifying which key this is and how it is created. On the contrary, in section 5.3 of this thesis a detailed implementation of TC technologies is presented.

4.8. Self-service cloud computing

Butt et al. introduce the self-service cloud (SSC) computing model which splits administrative privileges between a system-wide domain and per-consumer administrative domains [**But+12**]. This allows cloud consumers to securely spawn their own meta-domain over which they have control, including their own user Dom0. This meta-domain is isolated from an untrusted Dom0 using a mandatory access control in the hypervisor, and everything not strictly needed to be run in Dom0 is moved to consumer control.

The self-service meta-domains can be extended using so called service domains. These allow a wide range of consumer-controlled services to be made available, for

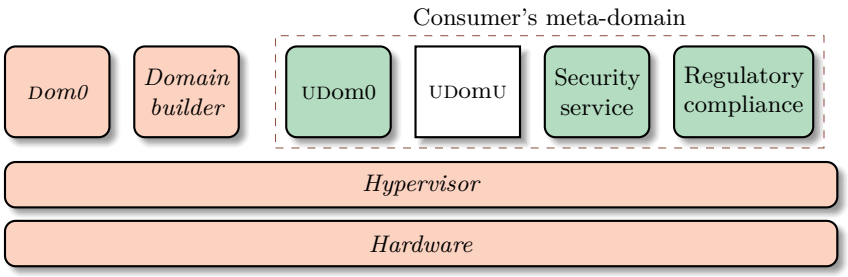


Figure 4.7: The overview of the self-service cloud approach. The system-wide TCB is shown in red, while the cloud-consumer-level TCB is shown in green. The cloud consumer's meta-domain contains a private Dom0 and a set of service domains. A subset of these service domains are the *mutually trusted* (between consumer and CSP) service domains, which includes a regulatory compliance monitor. Figure based on Butt et al. [**But+12**].

example DomU introspection, storage intrusion detection, and storage encryption. An overview of the SSC design is given in Fig. 4.7. In addition, the authors introduce mutually trusted service domains as part of their design. Such mutually trusted domains can solve a source of tension in cloud computing, namely that cloud consumers want privacy while CSPs want assurance that the resources are not used for malicious purposes. Using the mutually trusted domains, these two parties can agree that the CSP can inspect the consumer's VMs for compliance, while the consumer can verify that the mutually trusted domain does not reveal data but only discloses violations of the agreed-upon rules.

Applicability. Butt et al. developed their SSC design independently and in parallel to the work in this thesis, and there are a number of similarities between their design and the design presented in this thesis. Both designs have the concept of extending the VM of a consumer with additional service VM(s), have an encrypt on-the-fly functionality, and both designs anchor the VMs with trusted computing.

There are also differences, however. For example, we discuss the TC bootstrapping in more detail, coupled with a complete TC implementation, with a focus on the entire process of how the consumer actually gets his or her keys securely in the cloud. In addition, we base our service VM on Mini-OS, which has a minimal TCB and is well-suited for the role of service domain in Xen, while their service VMs are based on (trimmed down) Linux kernels.

5. Architecture

The contribution of this thesis, the *cryptography as a service* (CaaS) design, is introduced and discussed in this chapter. This architecture is an improvement of existing virtualization designs such that the requirements posed in chapter 3 are satisfied. This chapter is divided in two complementary sections. First, after a brief introduction, the design is introduced in a high-level overview in section 5.2. Second, in section 5.3 we explain how the CaaS design fits in an overall cloud computing environment.

Since our reference implementation is based on the Xen hypervisor, we apply the Xen terminology when describing the CaaS design. Nonetheless, the CaaS architecture and its principles can be applied to any type-I hypervisor design.

5.1. Introduction

Recall the requirements outlined in chapter 3. The protection of the HVKs necessitates a segregation between "normal" and "high-value" VM data and a reflection of this in the Xen architecture. A design which approaches the problem with a combination of *data* and *privilege* segregation could look as shown in Fig. 5.1 and is summarized in the following two points.

1. Split off the HVKs from the ordinary VM data by placing the HVKs in a separate VM. These two VMs must then communicate over a strictly defined interface.

2. Curtail the influence of the powerful malicious insider adversary on these VMs through stronger access control in the hypervisor.

However, these two measures do not address how to provision the HVKs to this isolated VM. In fact, all straightforward approaches to provision HVKs securely are subject to a chicken-and-egg problem, as exhibited in Fig. 5.2.

Observe that if a decryption key is provided directly alongside a VM then an insider has access to this key, too. On the other hand, to deploy a key at runtime requires a secure channel with a VM. Setting up such a secure channel by basing the channel on a pre-shared key only shifts the aforementioned problem one

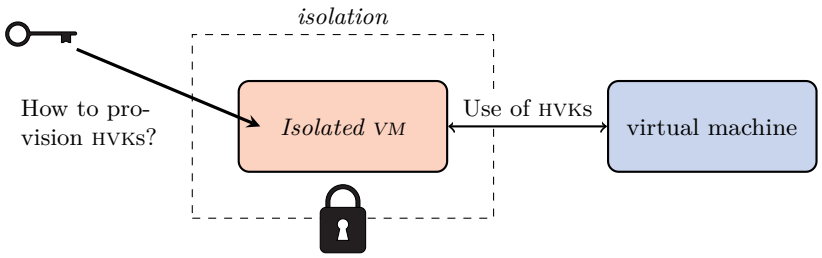


Figure 5.1: Segregation and isolation of HVKs is useful, but does not answer how to provision keys.

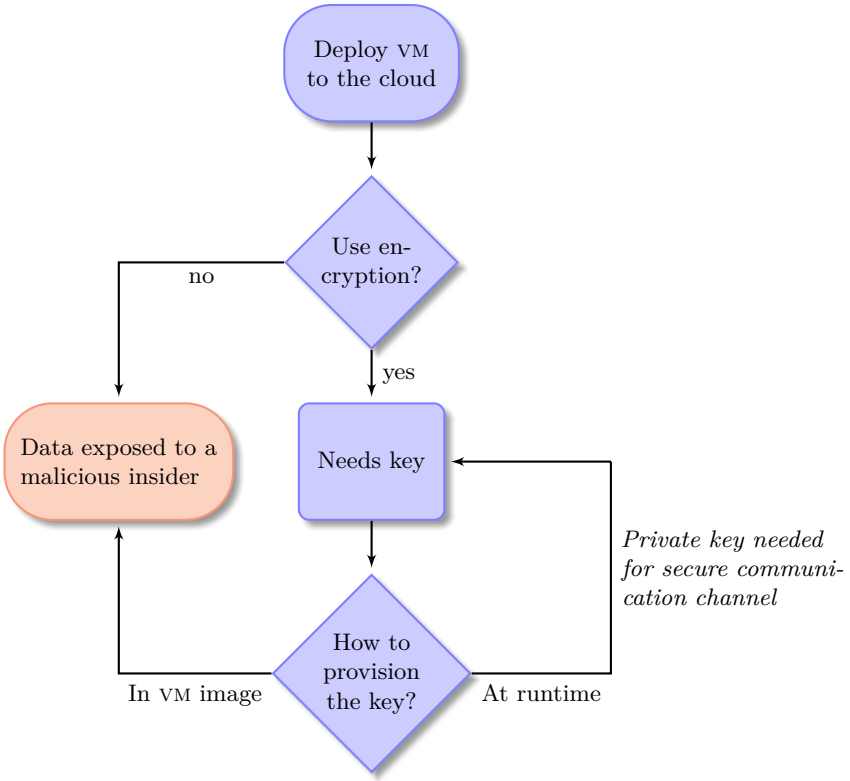


Figure 5.2: The chicken-and-egg problem with key deployment in the cloud. Even if the memory of a VM is protected, an out-of-band channel for secure deployment of keys is necessary. For instance, trusted computing.

level down (since the adversary can also read this pre-shared key). A similar argument holds if the secure channel would be based on a public key and a certificate to check it — one cannot be certain that the root certificate in the VM, which is used to verify certificates, has not been tampered with. Lastly, hoping to avoid pre-sharing a key by using a Diffie-Hellman-based approach at runtime is not viable either due to being prone to man-in-the-middle attacks [**Sta06**, p. 301]. (To see this, recall that the malicious insider controls channel C5.)

Therefore, a solution to provide HVKs out-of-band is necessary. This key role will be played by trusted computing since it allows deployment of HVKs such that they are bound to a trusted software configuration. Therefore, in addition to a segregated HVK storage (which we call the *cryptographic assistance domain*) and *hypervisor access control*, it will be necessary to introduce a third component, called the *trusted domain builder*, which is trusted and communicates with the TPM. These three components are presented in the following section.

5.2. Design

Figure 5.3, shows a high-level overview of the CaaS design. The architecture revolves around three main components, marked with numbers in the figure, which function as follows.

5.2.1. Cryptographic assistance domain

component: A minimal assistance VM which is supplied with keys by the cloud consumer and operates in an isolated environment. It exposes passthrough devices to the cloud consumer's VM. (See ① in Fig. 5.3.)

satisfies: This step provides protection against the *external adversary* for the assets which are required, that is, the HVKs. By segregating the HVKs in a shielded environment for which the keys are provisioned at deployment and not directly accessible during runtime, these keys are safeguarded even in case of a total breach of the DomU by an external attacker.

This is a step towards fulfilling requirement R1 (protection of the HVKs), for it covers the external adversary aspect of this requirement.

Furthermore, this is a step towards requirement R2 (protection of the VM) with respect to the *malicious insider* adversary; by encrypting data streams for virtual device I/O, the VM confidentiality and integrity are protected for attack channel C5.

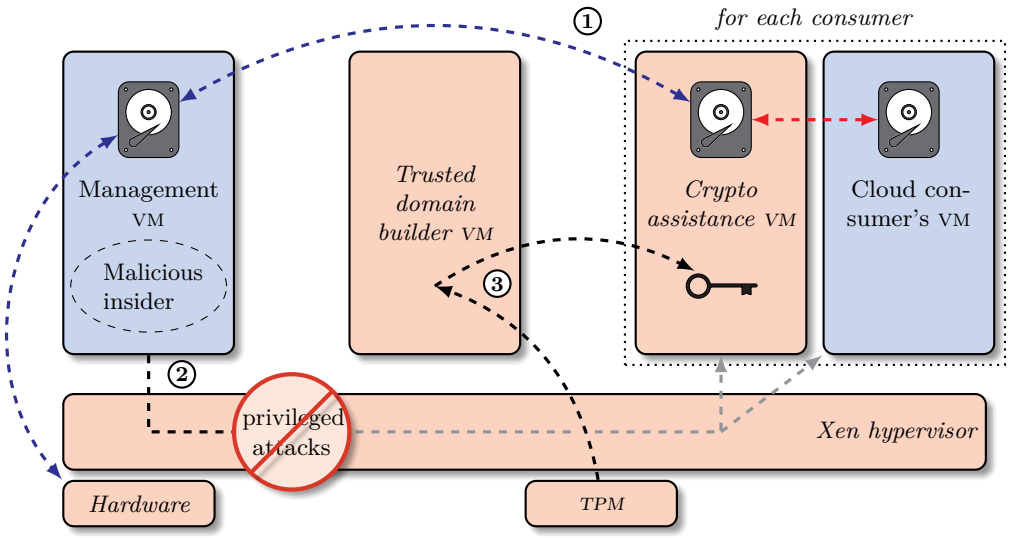


Figure 5.3: High-level overview of the CaaS architecture. The harddisk symbols indicate block drivers and the key symbol is the encryption key used for passthrough encryption. The red dashed line is encrypted traffic and the blue dashed line is unencrypted.

method: A key facet of this cryptographic assistance domain (DomC) is that it exposes a block device which the cloud consumer's VM can attach to. All data that the consumer's VM sends through this device is encrypted on the fly before forwarding it to the management domain. In this way, no matter where the actual data is stored in the cloud, the adversary cannot access it.

Not shown here (though discussed later) is that besides as a key used for the passthrough encryption, HVKs can also be exposed to the DomU through a vTPM interface, thus acting as a virtual HSM.

5.2.2. Hypervisor access control

component: The access control resides in the hypervisor, and affects primarily the management domain by curbing hypercalls. (See ② in Fig. 5.3.)

satisfies: This component is a step towards fulfilling requirements R1 and R2 (protection of the VM and HVKs) with respect to the malicious insider for attack channel C4.

method: The privileges of the management domain (Dom0), i.e. the cloud

administrator, are curtailed by the addition of *access control* in the hypervisor. The hypercalls which are restricted, are those through which the management domain can attack other VMs (in particular, privileged hypercalls with the ability to read the memory of VMs).

However, this means that some legitimate, but privileged, tasks that the management domain *used to* do, such as domain building, must now be relocated elsewhere, i.e. in the trusted domain builder — it is simply not feasible to determine at runtime when a hypercall is being used for legitimate and when for malign purposes, especially considering valid use cases such as VM suspension which occur after a VM has been launched.

5.2.3. Trusted domain builder

component: A trusted domain builder that also has exclusive access to the TPM. (See ③ in Fig. 5.3.)

satisfies: This component completes the requirements R1 and R2 (protection of the VM and HVKs), because the aforementioned components strongly rely on this component to satisfy their requirements. By implementing the necessary workflow to deploy keys out of reach of all of the adversaries on the one hand, and by taking over the domain building responsibility from Dom0 on other hand, this component has a key role in satisfying these two requirements.

This component also implements requirements R3 to R5 though these details are more of a technical nature and are discussed later.

method: The trusted domain building domain (DomT) has taken over the tasks of starting (and suspending, resuming, etc.) VMs from the management domain, since, as mentioned, Dom0 no longer possesses the necessary privileges.

Furthermore, this domain has the exclusive permission of communicating with the TPM, as guaranteed by the aforementioned access control component. The TPM is leveraged to enable the cloud consumer to encrypt HVKs such that only DomT can decrypt these, which DomT then injects into the DomC. As explained in the previous section, the use of the TPM is necessary because ordinary ways of delivering HVKs to the DomT are exposed to the adversary. Placing the TPM logic in the hypervisor would be a violation of the Xen philosophy of a slim, minimal hypervisor.

Remark: It is imperative that the malicious insider adversary cannot tamper with DomT since it is a trusted component. This implies that the

low-level steps of loading and running DomT may not be a responsibility of the cloud administrator. Rather, the hypervisor starts this domain immediately after booting, and the cloud administrator can only interact with it over a strictly defined interface.

5.3. Key provisioning

We argued that trusted computing is needed to deploy HVKs to the cloud securely. In this section, we give two approaches to using TC with the CaaS design. The first is a basic scheme, which, although secure in a limited context, does not meet the requirements of a real cloud environment. The second scheme that is described introduces a TTP which improves the usability of the deployment scheme at the cost of an external dependency. For this section, it may be necessary to refer back to the TC introduction in section 2.3.

5.3.1. The basic scheme

The basic scheme involves the most simple secure workflow. A TPM-bound public key is shared with the cloud consumer who uses this key to deliver his or her HVKs to the DomT in encrypted form. The DomT will then request the TPM to decrypt the HVKs on its behalf.

The public key used is not an ordinary asymmetric key but a *certified binding key* which guarantees that the key can only be used by trusted software. Using certified binding keys for securely delivering secrets has been successfully applied before [**Sad+07**]. (Although such previous use cases are in a non-cloud scenario, this does not affect how the TC technology works.)

Figure 5.4 exhibits the interaction with the TPM to set up and distribute a certified binding key. For clarity, we omitted the required TPM overhead such as encapsulating commands in OSAP and OIAP sessions[1] and supplying authentication data (password) parameters.[2] Likewise, we assume that the TPM has been taken ownership of, and that either the SRK authentication data (password) is set to a well-known value or that DomT possesses this information.

The first step in the scheme is that DomT creates a certified binding key which is represented as the pair (PK_{BIND}, SK_{BIND}). Second, this key and its

[1] These sessions are used to protect the freshness of communication with the TPM and also obviate the need to send passwords to the TPM in cleartext.

[2] We do not use a nonce in the key certification. The reason is that it is not important when the key has been generated and certified, but what its properties are. Because the binding key is non-migratable, these properties will never change [**SSB07**, p. 61].

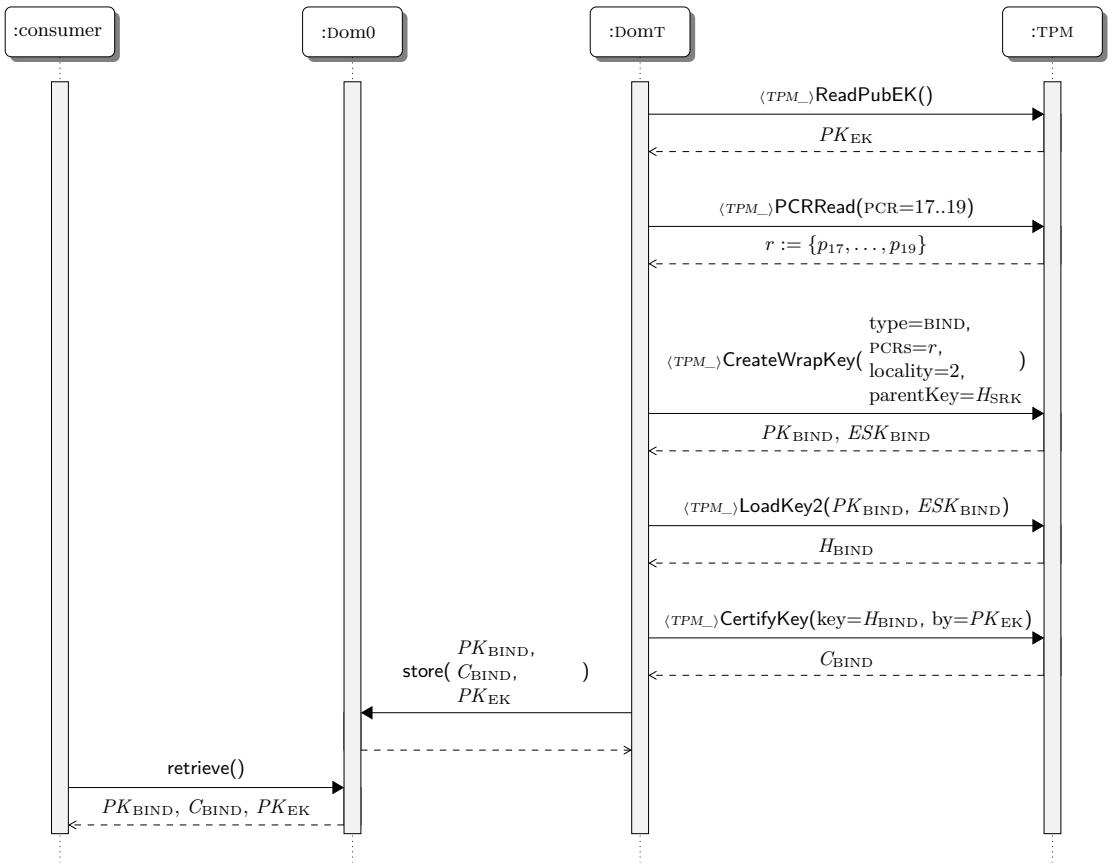


Figure 5.4: Interactions between consumer, Dom0, DomT and TPM for receiving a binding key.
PK is a public key, *ESK* is a TPM-encrypted secret key, *C* is a certificate, *H* is a TPM-specific key handle.

certificates are stored in Dom0, since DomT does not have network access. Third, Dom0 distributes the key and its certificates to the cloud consumer (though in practice this would flow via the cloud infrastructure). Before using the key, the cloud consumer will verify the certificates and check whether the software to which the key is bound corresponds to a list of known hash digests of trusted software.

Data structures. Once the cloud consumer is in possession of the certified binding key and has verified it, he or she can use this to encrypt HVKs and send these to the cloud. However, not only the HVKs are encrypted (req. R1); the VM as a whole needs to be encrypted, too (req. R2). This is captured in the following definitions. (These data structures are discussed in more detail in appendix B.2.)

- *Encrypted virtual machine* (EVM): the cloud consumer's VM, encrypted using a symmetric key.
- *Virtual machine control blob* (VMCB): a blob of important metadata regarding the VM, in particular the symmetric key used for the EVM. This blob itself is encrypted using the NPK.
- *Node public key* (NPK): a name for the set of the public binding key and its certificates (the EK public key with its endorsement credential and the certificate on the binding key which describes the PCRs used). When we use the term *node*, we refer to one of the physical VM-hosting machines in the CSP's data center.

5.3.1.1. Discussion on the basic scheme

While the thus far described scenario provides ample security against eavesdroppers, there is room for improvement.

1. The appraising party (the consumer) has no guarantee that the EK belongs to a real TPM and not a malicious, virtual TPM. A solution for this is to rely on *endorsement credentials* supplied by TPM manufacturers together with the TPM. (That is, if appraisers trust the manufacturer.) In practice, however, such certificates are not consistently issued by manufacturers, and even if we find one then we cannot rely on it for this very reason, since CSPs do not like to be locked-in to specific vendors.

2. In the protocol we encrypt the VMCB for one specific TPM. This means that the VM cannot be migrated to another machine. It also removes the freedom on the CSP's part of deciding where the VM is run (e.g., in a part of the cloud which is under the least load at that moment) but instead necessitates that the VM is deployed on the specific node for which the binding key was given to the consumer.

3. The burden of interpreting the PCRs lies at the consumer. Verifying PCR measurements is notoriously demanding: the number of valid configurations in a commodity OS is huge due to the sprawl of libraries and all the different versions of this software. The challenger has, in theory, to analyze whether every existing operating system and every patch-level has the "desired" configuration [**ss04**].

   Fortunately, the set of valid software configurations of the TCB we are considering in this thesis (the hypervisor plus a few tools) is a magnitudes smaller than the configurations in a typical Windows or Linux TCB. Moreover, we can be much more stringent in which version we allow or not. One may expect a CSP to be resolute in keeping systems up to date (even if the outdated version has not been proven vulnerable), more so than one may expect of end users in a desktop trusted computing setting.

   Nevertheless, it remains a daunting task for consumers to verify the hashes themselves if the task is not delegated. Consumers would need to compile all the software themselves in order to verify the hashes. Moreover, any proprietary changes made to Xen by the CSP will — unless the reluctant CSPs cooperate and give insight into their modifications — cause mismatches.

4. The privacy of the CSP is at risk. It is in the CSP's interest to keep the cloud appearing homogeneous to consumers, but this is hard to guarantee.

   a) Two consumers can easily see if they are co-resident by comparing their received binding key fingerprints.

   b) When recognizing a key fingerprint, it is easy for consumers to see that they have been hosted at a certain node before, and cloud consumers can use this information to determine the size of the cloud using the birthday paradox.

   Obfuscation techniques can be conceived (e.g., a fresh key for each customer for each machine) but they will require careful engineering.

5.3.2. The cloud verifier scheme

To address the issues associated with the simple scheme, we introduce a trusted third party called the cloud verifier (CV). The CV has been introduced earlier during discussion of related work in chapter 4. The CV is a concept introduced by Schiffman et al. [Sch+10] though defined in a slightly different way. In their paper, the CV attests the machines in the cloud, while we make use of binding keys — two sides of the same TC coin. Although their protocols differ from ours because of this, the principle of a TTP is similar in both cases.

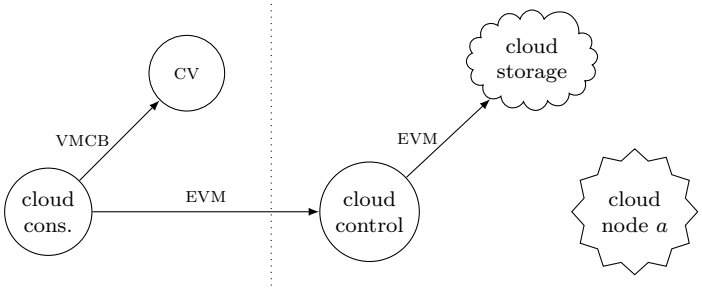
The CV we define in this subsection has two main purposes. The first purpose is to assist in the initial deployment, and the second purpose is to assist in judging whether a host is trustworthy. Both purposes are discussed as follows.

Deployment assistance. The workflow in a typical commodity cloud is that the cloud consumer starts with uploading a VM to the cloud which is stored into the cloud storage. At any later point, whenever the cloud consumer desires so, he or she can go the CSP's website and via the web interface indicate that a VM must be started.[3]

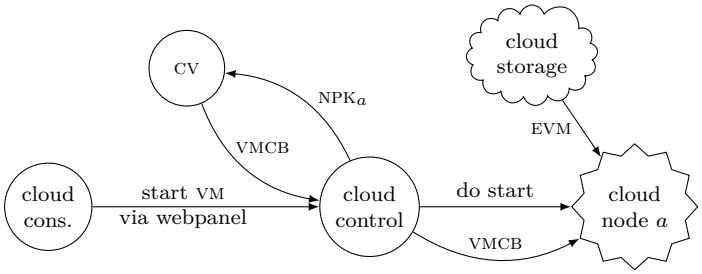
Only at the moment of launching a VM does the CSP decide on which node the VM will be run (depending on load balancing in the cloud). This implies that it is not possible for the cloud consumer to encrypt the VMCB for the cloud beforehand, since at that moment cloud consumers do not know yet where exactly their VM will be scheduled. Indeed, to not lose the flexibility of launching VMs in an automatic fashion, this implies that the CSP needs to connect to specific software at the cloud consumer and inform the cloud consumer's software to encrypt the VMCB for a certain cloud node. This is a cumbersome and involved proces and some cloud consumers may not be interested at all in running such software themselves.

Therefore, the CV takes over this role of assisting the deployment of VMs (which implies that cloud consumers encrypt their VMCB for the CV, who then re-encrypts it for the cloud during deployment). In Fig. 5.5a it is shown how the cloud consumer can delegate the deployment to the CV trusted third party and sends the EVM to the CSP. (Recall that the EVM is not usable without the VMCB.) In Fig. 5.5b it is depicted how the CSP "calls back" to the CV when the actual deployment takes place.

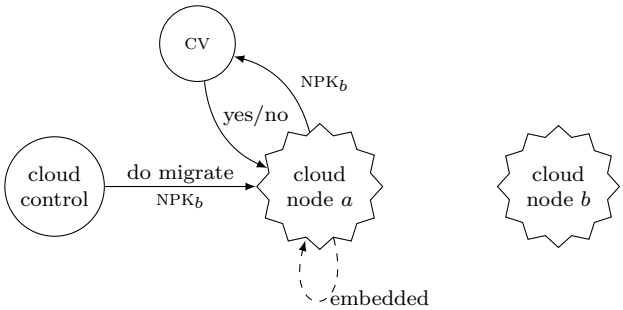
[3] In most clouds, there are a multitude of ways to start new VMs, such as via an API or automatically through elasticity. But the general workflow is the same, in that the CSP doesn't know beforehand where the new VM will be hosted.



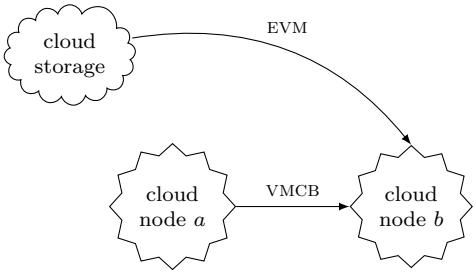
(a) Preliminary deployment of a VM to the cloud. The large EVM is sent to the cloud, while the small VMCB is kept outside of the cloud.



(b) The consumer requests the launch of a VM. The cloud controller selects a node from the pool which will host the VM and passes the node's NPK to the CV. The CV has to make a judgment on whether the NPK is trustworthy. If deemed trustworthy, the VMCB is encrypted using the NPK and the VM is started.



(c) The cloud controller decides the VM needs to be migrated or cloned to another node. The NPK from the new node needs to be judged by the old node (likely with help of the CV), and if deemed trustworthy, the operation is allowed.



(d) The VMCB is encrypted using the NPK of the new node, the EVM fetched from storage, and the VM is started.

Figure 5.5: Trusted deployment of VM to the cloud. For an explanation of the nomenclature refer to p. 53.

Judgments on trustworthiness. Making judgments about whether a host is trustworthy or not is nontrivial. The difficulty of making judgments is due to the plethora of distinct TPMs, distinct software configurations, and the fact that these change over time.[4] Such a moment of judging trustworthiness of software configurations takes place during initial deployment, or later on, when migrating; in both cases, the judgment process compares PCR values to a list of known values. In Fig. 5.5c, an example is shown of a migration request in which the CV is involved.

In the process of judging trustworthiness, the CV can play an important since the CV, as a specialized entity, can have access to more information than a cloud consumer can. For example, it might have an accurate list of PCRs by having signed a non-disclosure agreement (NDA) with the CSP, or perhaps it has an accurate list of all the EKs of the TPMs of all the nodes belonging to the CSP. In either case, the process of judgment involves (i) taking a node's NPK (in particular, of course, the listed PCRs) and (ii) evaluating it using a policy which contains lists of trustworthy PCRs, valid endorsement credentials, etc.

The policy, which is effectively a finite set of rules, could be fully embedded into the VMCB (depicted in Fig. 5.5c). This implies that the cloud consumer needs to have knowledge of all critical information such as PCRs and which nodes the cloud providers has when creating the VMCB for the first time. Unfortunately, it is infeasible for the cloud consumer to gather and embed all this information, moreover since it is constantly changing.[5] Indeed, this is a compelling argument to use a CV instead.

Alternatively, one could also consider an approach akin to property-based vTPMs, although this idea is not pursued in this thesis. In essence, instead of embedding raw PCRs into the policy, the public key of a property provider could be embedded instead. This could allow for a more flexible approach with a set of different property providers, each vouching for a different area of expertise (one for the list of valid PCRs hashes, one for the list of valid EKs, etc.) Of course, this introduces new challenges as well, such as revocation of certificates.

[4] Note that talking to a genuine TPM is not enough, even if it is seems to be coming from the IP-range of the target CSP. Our attacker model relies on the assumption that physical attacks are not possible, therefore the CSP must make known those platforms which are in the physical security area. Otherwise, malicious insiders could use machines under their physical control from the same IP-range as the data center to lure cloud consumers into sharing secrets.

[5] For instance, hypervisors do have vulnerabilities from time to time [XSA15; XSA16], and therefore it must be possible to strike out a vulnerable version from the set of valid software configurations.

5.3.2.1. Discussion on the cloud verifier scheme

A CV can, in principle, be fulfilled by anyone. However, CSPs will be highly reluctant to share the fingerprints of their nodes, if only because it shows the size of their cloud. To a lesser extent, they will be reluctant to share the source code of their proprietary changes to the hypervisor software (which is necessary, else it is not possible to decide whether a hash is trustworthy or not).

Hence, it is likely that the CV is a service provided by the CSP, or that some of the dependencies are provided by the CSP. We imagine a scenario in which only a few security officers have the power to change the rulesets used on these CVs, preventing the malicious insider from our attacker model to pose a threat.

On the other hand, the CV could also be run by another entity who has sufficient insight in the software used by the CSP. It is conceivable that a company behind a CV signs a NDA with the CSP to audit the proprietary cloud software. To guarantee the CV's independence, it would be recommendable that the CV is funded by the cloud consumer or a consortium of cloud consumers.

6. Implementation

The ideas presented in this thesis have been translated to a functional implementation based on the Xen hypervisor. In this chapter we discuss two main topics. First, we first briefly name and describe the components involved in this implementation. Second, we describe the steps involved in bootstrapping the CaaS design.

6.1. Components

The implementation involves the extension and modification of the Xen hypervisor at different levels. While one important consideration for this thesis is that readers should be able to reproduce the results, a detailed discussion of all the components is not necessary for understanding the contributions of this thesis. Therefore, in the current section the components are only identified and briefly introduced, while a detailed discussion of all the components is given in appendix B. Figure 6.1 shows an overview of the components that comprise the implementation. We briefly review each component.

① *Data structures.* The EVM (VM in encrypted form) and VMCB (essential metadata) data structures are recurring throughout the implementation. Since the decisions of what to encrypt and how plays a key role in satisfying the requirements, these data structures warrant a deeper discussion. (Appendix B.2.)

② *Access control.* The design involves a separation of duties between Dom0 and DomT using access control. Both domains have a subset of hypercalls that they are allowed to use (e.g., Dom0 can still set the system clock, but cannot introspect domains anymore). (Appendix B.3.)

③ *Virtual filesystem bridge.* A virtual filesystem bridge between Dom0 and DomT through which DomT can read files from Dom0 as if it were reading these from its own disk. This bridge is used by the domain builder as well as the TPM management. (Appendix B.4.)

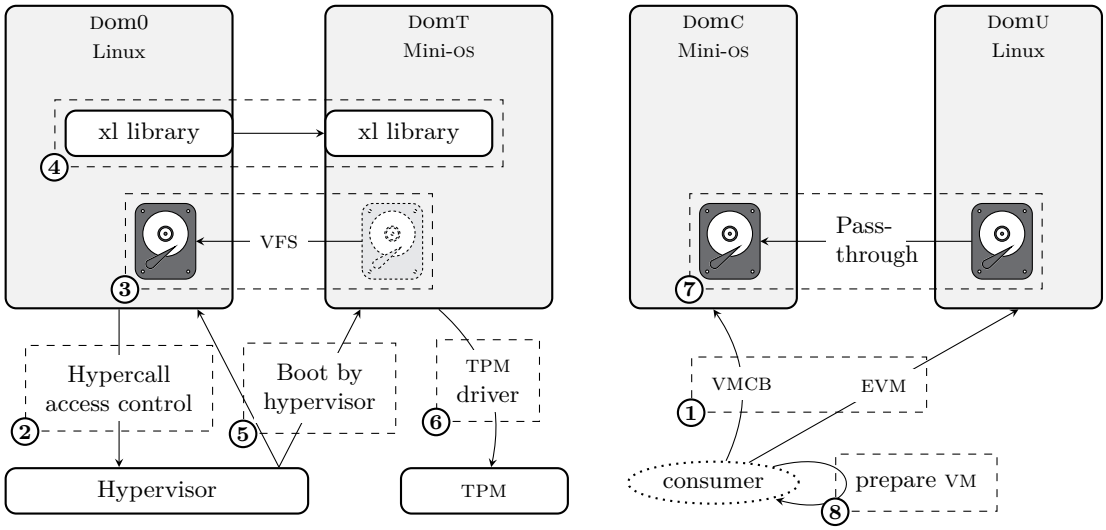


Figure 6.1: Overview of the components to be discussed in this chapter, with an emphasis on highlighting the components, rather than giving a chronological overview of the CaaS architecture.

④ *Porting domain building code.* The domain building code (the so-called Libxl) was ported to Mini-OS. (The Libxl in the Dom0 will forward any requests that it cannot perform itself anymore to the libxl in DomT.) (Appendix B.5.)

⑤ *Direct booting.* The trusted domain builder is directly booted by hypervisor after which Dom0 and DomT must discover each other. (Appendix B.6.)

⑥ *A TPM driver for Mini-OS.* The DomT includes logic for exclusively communicating with the TPM. (Appendix B.7.)

⑦ *Passthrough encryption.* A passthrough encryption is set up between DomC and DomU, which involved both (i) creating a back-end driver and (ii) creating the encryption. (Appendix B.8.)

⑧ *Service tools.* User tools help in deploying a VM to the cloud by the cloud consumer. In particular, a plain VM must be split into the aforementioned data structures. (Appendix B.10.)

6.2. CaaS bootstrapping

In this section, we describe the process of bootstrapping our design. We only discuss the paravirtualized mode of Xen, omitting the HVM mode (though porting concepts such as passthrough encryption to HVM is possible).

Introduced terminology. We briefly list the components that will be discussed in this section, notwithstanding that most of these have already been introduced in either the background chapter or the previous section.

- *DomT*: trusted domain builder which also has exclusive access to the TPM. Underlying kernel: Mini-OS.
- *DomC*: cryptographic assistance domain which provides passthrough encryption and a vTPM. Underlying kernel: Mini-OS.
- *TBoot*: a bootloader which can install a DRTM (cf. chapter 2, p. 20).
- *Pv-tgrub*: a modification of PV-GRUB (cf. appendix A.1) which cannot only boot VMs but also has the ability to extend measurements to a vTPM.
- *Templates*: a template is defined here to be a generic VM image which can be instantiated multiple times for different consumers (i.e., no consumer-specific data is part of the template itself).

  The only templates which we are considering in this work are helper VMs, instantiated only to serve a specific task. Such a helper VM template should be a minimal (i.e., no disk needs to be attached) VM image which can be run on Xen in PV mode. In practice, this means that these templates are Mini-OS based kernels since these come stripped down with all required functionality statically compiled in.

6.2.1. Initialization

This subsection exhibits the CaaS steps which are only needed once after powering on, i.e., the initialization steps.

Start of day (Fig. 6.2)

① After powering on and initializing, the bootloader is started. Because our design uses a DRTM, there are no restrictions on the type of bootloader.

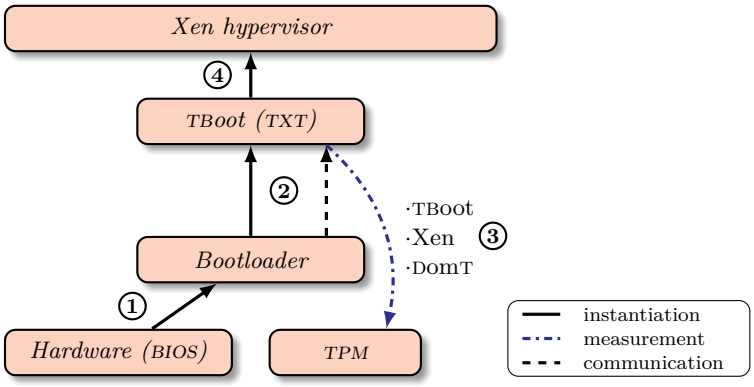


Figure 6.2: The hypervisor is booted via several intermediate steps, being measured in the process.

② The bootloader boots TBoot with (i) the Xen hypervisor, (ii) Dom0 kernel and (iii) DomT kernel as additional modules.

TBoot, by utilizing TXT to install a DRTM, marks the beginning of the use of the TPM in our design. This TXT call ensures that the TBoot binary is extended into a PCR which can only be extended by a TXT instruction.

③ Once TBoot is booted, it will measure its policy file. Its policy file will instruct it to measure the modules which are part of the TCB, namely the Xen hypervisor and DomT kernel. Since Dom0 doesn't belong to the TCB, it will be extended into an unused PCR.

④ Finally, control is passed to the Xen hypervisor.

Work domains (Fig. 6.3)

⑤ The Xen hypervisor boots the two domains that are required to start consumer VMs. The first is the familiar Dom0, stripped of certain privileged capabilities, while the second is DomT, a small Mini-OS based helper domain.

⑥ DomT will receive two templates from Dom0.

1. The first template is the DomC template, which is only the program logic without any keys; consumer keys will be injected later on.
2. The second template is the PV-TGRUB template, which is needed to boot the consumer's VM in a secure manner.

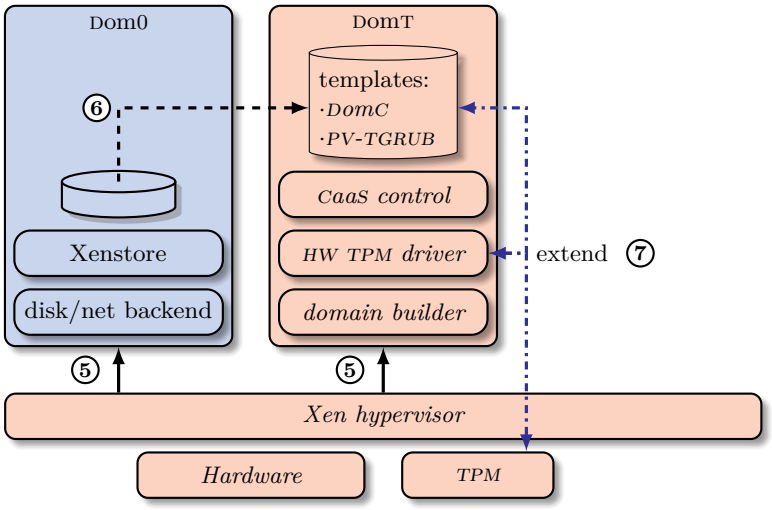


Figure 6.3: The hypervisor boots Dom0 and DomT. Dom0 retrieves the templates DomC and PV-TGRUB from local (or remote) disk and passes these to DomT who measures and caches them.

Since these templates are the same for all users, this loading needs to be done only once. Nevertheless, it is conceivable that the CSP might roll out updated versions of either template, extending the newer version to the TPM (as explained in the next step).

⑦ The templates are part of the TCB, therefore they are measured by DomT, which involves extending their hashes to the TPM. The templates remain cached in DomT's memory so that they can be directly fed to the domain builder stages later on.

At this point, the TPM contains measurements of TBoot, Xen, DomT, DomC and PV-TGRUB. This corresponds to the TCB components of the design, and are exactly the measurements which are relevant for the certified binding key.

State (Fig. 6.4)

⑧ In-between consecutive reboots of the machine, DomT's state is securely stored and reloaded. The state is encrypted and also the current value of the TPM counter is included so that rollbacks can be detected. Also considered part of the state is the private key blob of the TPM key.

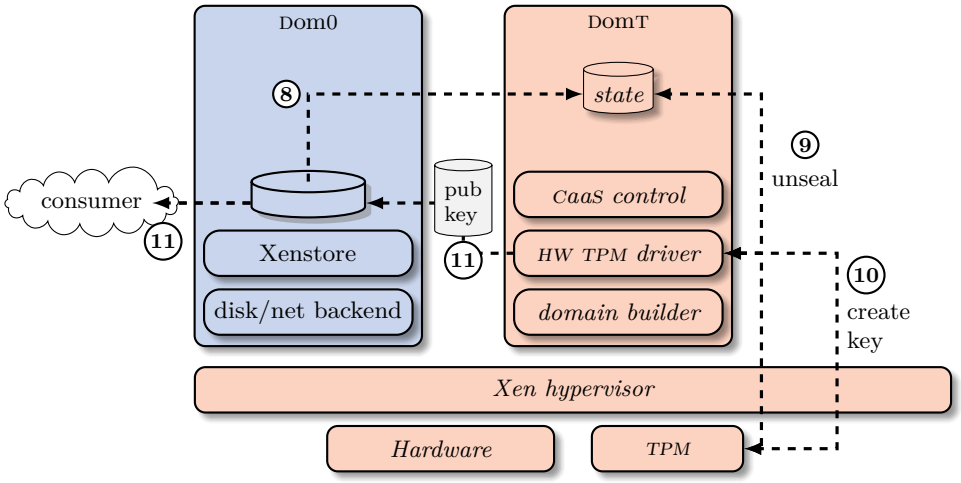


Figure 6.4: Dom0 loads the state and decrypts it using the TPM. Furthermore, a binding keypair is generated if none is found. If a key is generated, the public key is exported.

This step will be skipped if no state exists.[1]

⑨ The ⟨*TPM_*⟩Unseal operation will decrypt the blob if the PCRs are in the correct state. After unsealing, the DomT will check that its expected TPM counter value matches the value found in the TPM, otherwise it will be rejected as a replay attack.

⑩ If no key exists, a new non-migratable binding key is generated as well as a certificate over the key. This key is bound to the current software state, i.e., the values in the PCRs.

⑪ If a new key has been generated (or if Dom0 lost the previous copy of the public key) then the public key of the TPMs key is exported. This key reaches the cloud consumer through Dom0.

[1] If the malicious insider removes these then a new key will be generated by DomT. The result will be that all currently assigned VMs are invalidated, i.e. only a loss of availability.

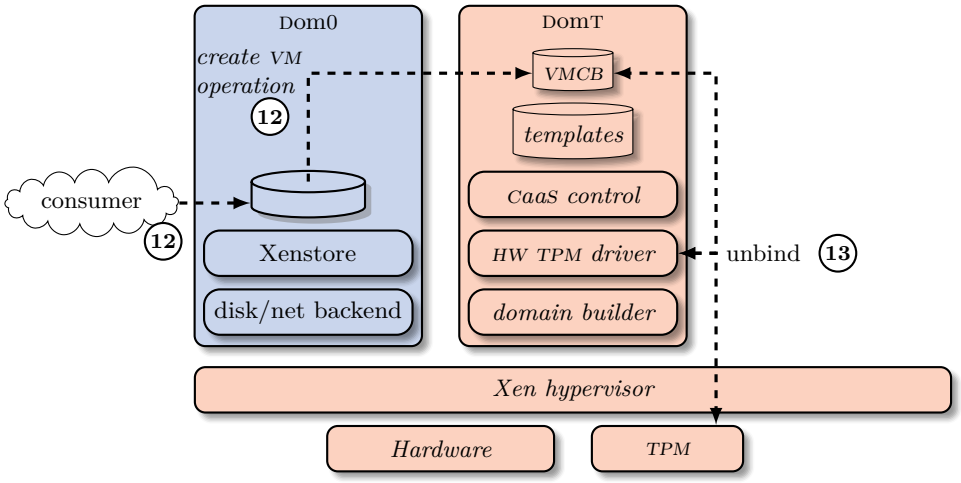


Figure 6.5: When starting a VM, Dom0 passes the settings from the domain configuration file to DomT together with an encrypted blob containing metadata: the VMCB. The blob is unbound by DomT and is dependent on a correct state in the PCRs.

6.2.2. Starting a VM

At this point, the configuration has finished initializing. This means that both the work domains are set-up and the TPM PCRs are in the state corresponding to the software running. The DomT is waiting for commands from Dom0.

When and which commands Dom0 will issue is dependent on the cloud management software. This can be a proprietary product or one of the various open source approaches such as OpenStack or CloudStack [os; cs]. For this thesis, a discussion of cloud management software is out of scope; we will instead assume an administrator issues commands via the Xen toolstack.

Command (Fig. 6.5)

(12) The Xen toolstack has received a command to create a VM from the administrator. We assume the consumer has delivered its VM to Dom0 or it is reachable via network by Dom0.

DomT receives from Dom0 the Xen details for the creation of this VM. This includes essentials such as the amount of RAM, number of virtual CPUs (vCPUs), network cards, and the location of disks. Additionally, it receives an important blob of encrypted data called the VMCB. This VMCB encapsulates various vital data such as checksums and the decryption key

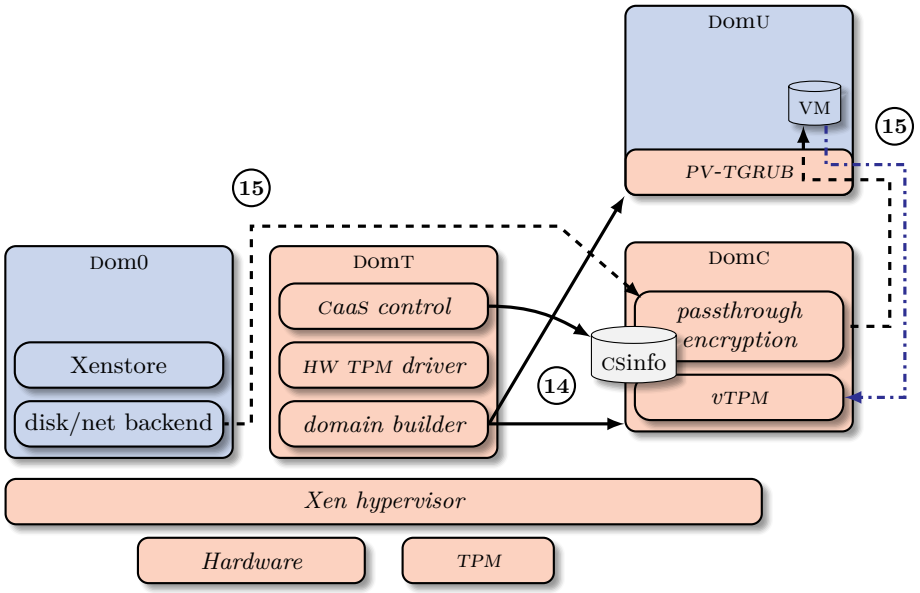


Figure 6.6: DomT builds the DomC and DomU domains using the cached templates. The core information from the decrypted VMCB is injected into the DomC as CSinfo. The DomU is started with a bootloader which chain loads the actual VM.

for the VM image, used in the passthrough encryption (cf. appendix B.2 for details). This blob can only be decrypted by the TPM.

(13) The DomT proceeds to request the physical TPM to decrypt the VMCB. This is achieved using the ⟨*TPM_*⟩Unbind operation. In this operation, the physical TPM will first verify whether the current measured configuration is in the stipulated state which the decryption key requires. If the prerequisites are satisfied, the TPM will decrypt the blob and return a decrypted VMCB to DomT.

Building (Fig. 6.6)

DomT now has all the information needed to build the cloud consumer's two domains. However, DomU will not be built directly; instead a PV-TGRUB is booted which later chain loads the real DomU.

(14) The domain builder sets up the DomU with the given settings by the administrator (i.e., the RAM size, number of vCPUs, etc.) It does not,

however, place the DomU kernel to be ran into memory; instead, the kernel from the PV-TGRUB template is placed there, which has the sole purpose of chainloading the real kernel.

The DomC domain is set-up using predefined values for the RAM and vCPUs since these requirements are homogeneous across all the DomCs. As block device(s) for DomC, the consumer's encrypted VM image is attached.[2]

The DomC receives its payload with user-specific information from DomT during domain construction. This information, which we refer to as CaaS-info (CSinfo), contains the key for the passthrough encryption as well as the non-volatile memory of the vTPM.

(15) The DomC and DomU are now both in a running state. The DomC, as part of its responsibilities, connects to the block device offered by Dom0 and offers a passthrough block device to the DomU. Inside the DomU the PV-TGRUB is in charge and mounts this passthrough block device.

At this point, a normal PV-GRUB setup would chain-load the DomU kernel and be finished. However, our bootloader takes an additional step before chain-loading. This step by the bootloader — which is the reason that our implementation is called a PV *trusted* GRUB — is to extend the hash digests of the loaded DomU kernel to the vTPM. This operation sets up the vTPM for functioning as an SRTM for the DomU.

Booting (Fig. 6.7)

(16) The only essential change that is taking place in this step is that DomU is now booting the consumer-provided DomU kernel. This kernel will connect to the passthrough device offered by DomC, just like the PV-TGRUB did before. At this point DomU can also connect to any devices offered by Dom0 which could be devices for which no passthrough encryption is needed, such as NICs or disks which contain public data and require no confidentiality or integrity protection.

An important functionality at this point is the ability for the DomU to utilize non-migratable HVKs stored in the vTPM. These secrets have been pre-loaded by the consumer and were stored in the VMCB, which was later injected into DomC as part of the CSinfo. These keys can now be used by the DomU without the risk of the keys ever leaking out of the vTPM when

[2]The VMCB contains info relating to the primary device; DomT will assure that booting happens from this disk. Other disks may be present; the CaaS design does not rule out the use of unencrypted disks next to encrypted disks.

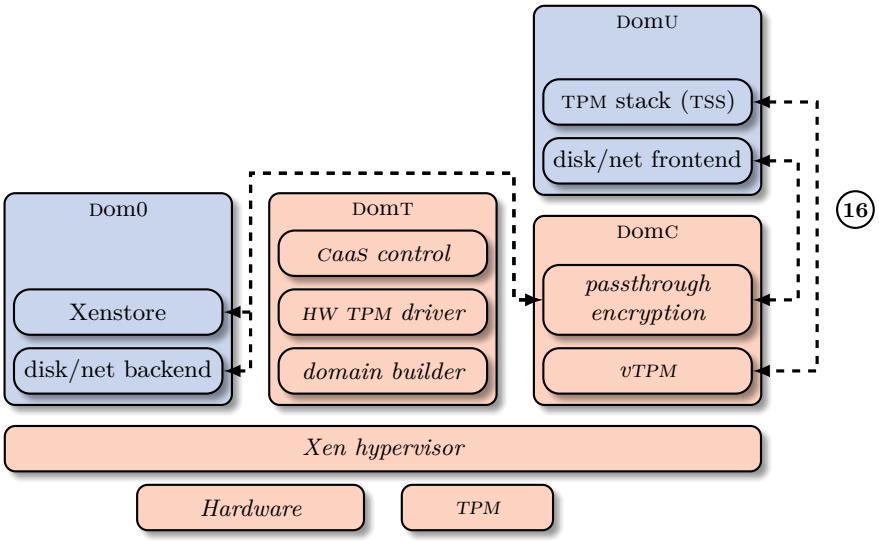


Figure 6.7: DomU utilizes the virtual devices offered by DomC. For the passthrough devices, DomC applies encryption and forwards requests to Dom0. Additionally, regular devices from Dom0 can still be used by DomU (not depicted).

the DomU is compromised by an outside attacker. An example scenario is one where the DomU will treat the vTPM as an PKCS#11 interface for creating certificates.

This is of great importance for req. R1, because it means that HVKs can be guarded even in the face of successful external attackers.

7. Evaluations

In this chapter, various aspects of our proposed architecture such as lines of code (LOC) and passthrough performance are investigated.

7.1. Magnitude of the TCB

In Fig. 7.1, a TCB breakdown of components for the CaaS architecture is shown. The bulk of the LOC is taken up by Newlib, and although we already excluded some parts of Newlib which are not used,[1] this process could be taken steps further. With respect to LOC, the CaaS approach currently does not match up to other minimal-TCB approaches, for instance:

- Colp et al. claim only 13.5 KLOC for their domain builder in the Xoar design [**Col+11**];
- Zhang et al. claim that their Cloudvisor comprises only 5.5 KLOC [**Zha+11a**];
- Steinberg et al. claim only 36 KLOC for the NOVA microhypervisor [**SK10**].

However, to achieve the smallest TCB is not the focus of this thesis. Moreover, the TCB as an absolute indicator of security is rather superficial for our design. In our case, the security would benefit the most from hardening the code in the domain builder, perhaps even by adding some lines to the TCB for extra bounds checking, rather than trimming down Newlib to a bare minimum.

In addition, we remark that our solution is the only design that actually implements a trusted computing layer. A small TCB is only of limited benefit if there is no guarantee that this minimal TCB software is actually the software being executed. In either case, when comparing our solution with the status quo, which includes Dom0 in the TCB, the improvement is still significant: the removal of the Dom0 kernel from the TCB frees 7.6 million lines of code [**Col+11**; **Zha+11a**].

[1] We ignore the math library and the Unicode conversion library (iconv) for Newlib.

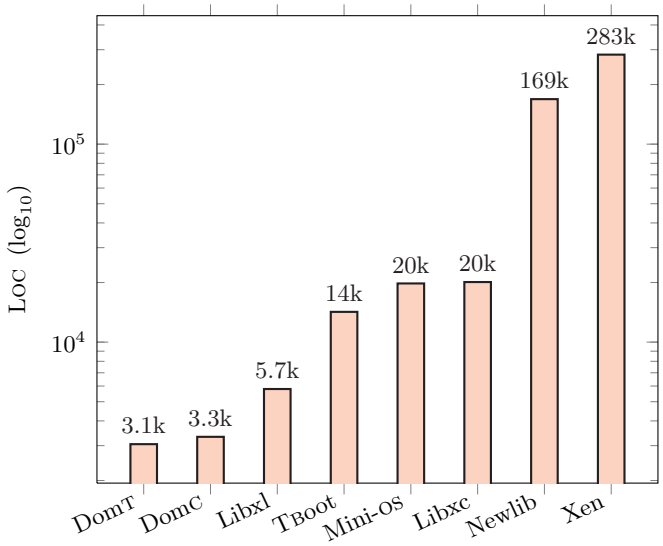
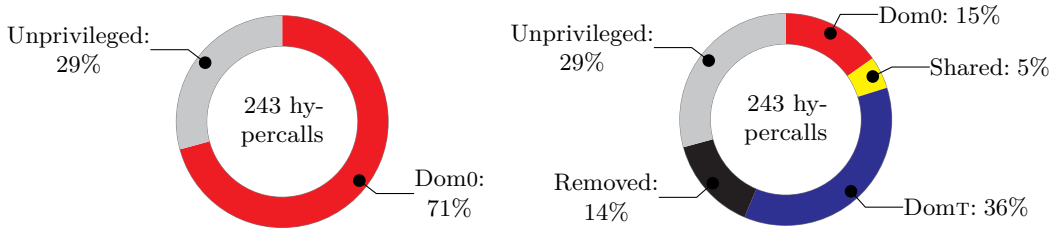


Figure 7.1: Overview of the TCB components and the respective lines of code in the CaaS architecture. The CaaS architecture adds roughly 230 KLOC onto the Xen hypervisor, resulting in a grand total of 520 KLOC. (Counted using Sloccount [SL].)



(a) Vanilla Xen. **(b)** CaaS Xen.

Figure 7.2: A comparison of the distribution of all Xen (sub-)hypercalls before and after disaggregation based on our CaaS requirements. Hypercalls with both an unprivileged and a privileged version are only counted once, and listed as a privileged call.

7.2. Access control

As discussed earlier, in order to provide better protection against the malicious insider we curtailed the power of the management domain. The approach we took, was to write a custom Xen security module (XSM) module (cf. appendix B.3). Recall that we assigned to Dom0 and DomT distinct responsibilities, and used these responsibilities in reviewing all the hypercalls (appendix F).

As shown in Fig. 7.2, we greatly reduced the number of privileged hypercalls accessible by the management domain (and therefore, by a cloud administrator). All of the remaining privileged hypercalls available to the management domain relate directly to the set of responsibilities assigned to Dom0, e.g., setting the system clock,[2] domain rebooting, etc. Moreover, we removed debugging and tracing hypercalls since these are not needed in a production environment.

In summary, this XSM module is successful in curbing the number of hypercalls available to the malicious insider adversary. Moreover, if the privileged domains are segregated even further (e.g., as proposed by Colp et al. [Col+11]), then Fig. 7.2b will show further segregation, too.

[2]We ignore the possibility of an adversary meddling with cryptographic protocols by changing the system clock.

Characteristic	Value
Disk	Western Digital WD5000AAKS - 75V0A0
Processor	Intel quadcore i7 with 3.2 GHz, 64-bits cores
RAM	2048 MiB (Dom0) / 128 MiB (DomC) / 256 MiB (DomU)

Table 7.1: Properties of the Dell Optiplex 990 benchmark machine.

7.3. Passthrough encryption

As explained earlier, we implemented a passthrough cryptographic device. The cryptographic scheme we used for this is also supported in native Linux by the dm-crypt kernel driver (see appendix B.8.1 for details). This makes it possible to compare two scenarios with each other, namely

1. the *vanilla* scenario, i.e., a DomU communicating with Dom0 without an interposing DomC domain, in both unencrypted mode as well as with a dm-crypt protected filesystem in DomU; and

2. the *passthrough* scenario, which uses an interposing DomC in both a simple forwarding mode or in forwarding mode with encryption.

We took bandwidth measurements using the *fio* tool [FI] in the DomU. For each of the aforementioned four combinations, measurements were taken over 12 distinct block sizes with each read or write run lasting 10 minutes. We disabled all buffering on the DomU side (important for reading) as well as explicitly performed only synchronous I/O (which affects writing). Clearly, these are not everyday settings, but they serve to make the experiments less biased. Moreover, we execute random I/O tests only, which is necessary to prevent the harddisk I/O buffers from playing a role.[3] In Table 7.1, we list the properties of our testing machine.

[3] A completely unbuffered experiment is difficult to guarantee. It requires further code analysis to determine whether no buffering or reordering takes place in the path through the ring buffers and the Dom0 block device subsystem.

7.3.1. Discussion of results

We briefly discuss the findings exhibited in figures 7.3 to 7.5.

- *Overhead of passthrough disk.* In Fig. 7.3, we show the relative overhead that our DomC solution incurs. With a smaller block size, more switching has to occur between front-ends and back-ends, and the overhead incurred by our DomC becomes more profound.

  The significant differences seen between the read and write is a result from the writing being slower than reading (see next graph) and the overhead thus being a relatively smaller contributing factor.

- *Inspection of spread of measurements.* in Fig. 7.4, we highlighted a section of the plots in detail. This scatter plot gives an insight in what the spreading looks like for a typical measurement sample. The writing performance had more variance and scored less than reading for the same parameters; this is a behavior we saw across all measurement samples. This seems to correspond with the specifications of our harddisk, which states a higher latency for writing than for reading.[4] In either case, the DomC dampens the variance somewhat, which is an artifact of the extra overhead.

- *Overhead of encryption.* In the previous two plots we only showed the plots corresponding to the unencrypted path. Therefore, in Fig. 7.5, we show the relative overhead that encryption incurs. The vanilla scenario uses the Linux dm-crypt kernel driver, while the passthrough device uses our custom encryption. The graph indicates that the overhead of encryption plays only a very minor role. The regression line for the writing under vanilla conditions shows a nonconforming pattern, but this is mostly the cause of a single outlier. This by itself seems at least partially a byproduct of the higher variance shown in writing, which is dampened in the DomC scenario.

For figures 7.3 and 7.5, the definition of the relative difference is, with operation $O \in (\text{read}, \text{write})$ and mean referring to data bandwidth sample mean:

$$\text{Overhead } \% = \frac{\text{mean}(O_{\text{vanilla}}) - \text{mean}(O_{\text{passthrough}})}{\text{mean}(O_{\text{vanilla}})} \times 100\%$$

[4] 8.9 ms for reading, and 10.9 ms for writing [WD].

7.3.2. Measurement conclusions

Our passthrough block device shows overhead performance values ranging from mediocre to acceptable depending on the block size. It is not easy to say which block size is the closest to reality, but most likely it is somewhere in the middle. In any event, it is a fact that OS kernels will combine I/O requests, as well as pre-fetch disk sectors and store these in RAM in anticipation of further reads. Therefore, I/O requests with block sizes as small as 512 bytes will be very rare.

Nonetheless, performance was never a design goal when developing the Mini-OS block back-end driver, on the contrary, stability and flexibility were the primary objectives. In that respect, the design goals were met since the passthrough device remained stable under all experiments. Moreover, the flexible design of parsing all requests (instead of just copying them from one ring buffer to the next) allows for new use cases in the future, for instance, a block device which is not backed by a Dom0 disk but instead by a ramdisk or by a network-based disk.

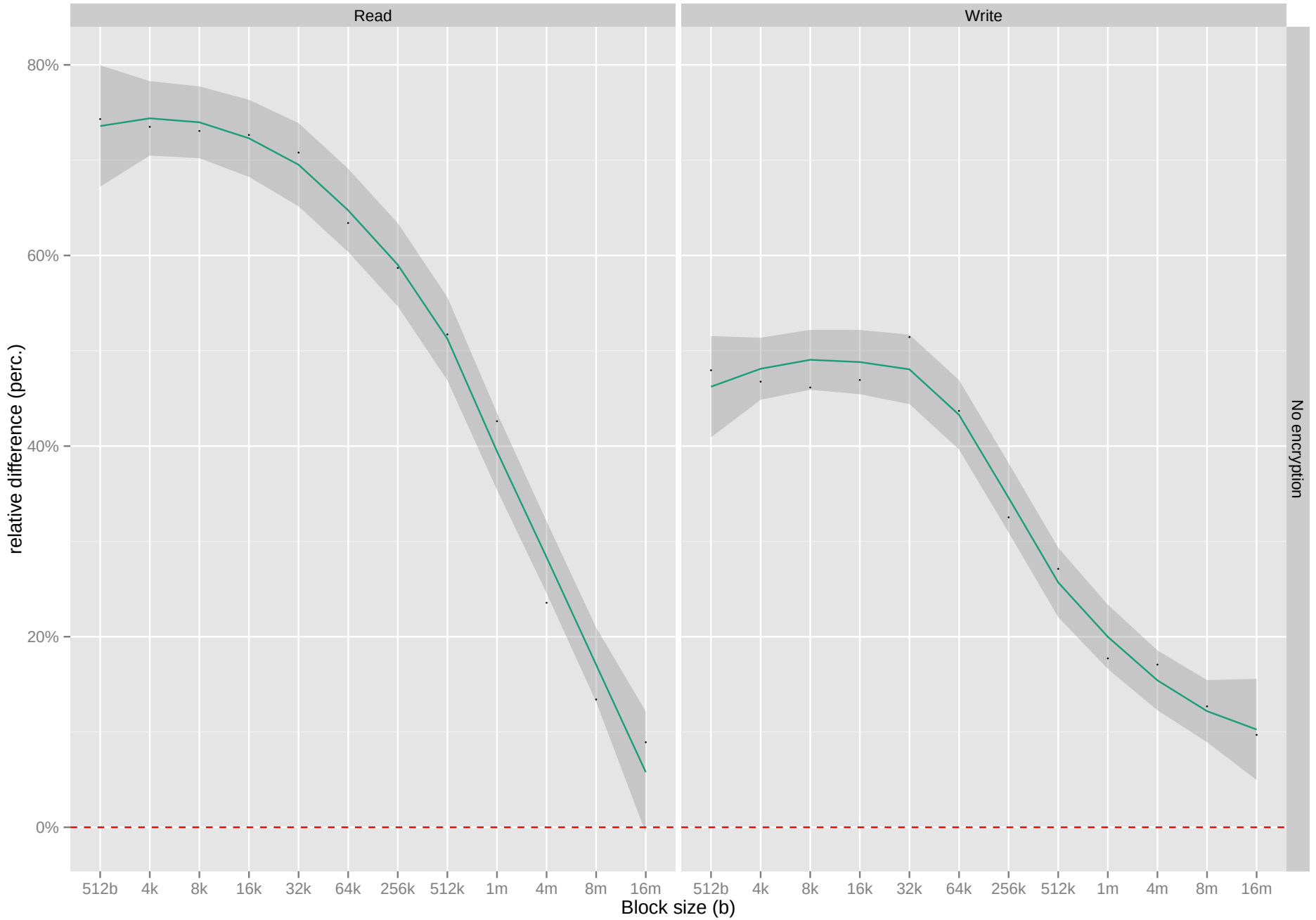


Figure 7.3: Plot of relative overhead of running in *passthrough* mode, as opposed to *vanilla*. The shaded area is the 95% confidence interval.

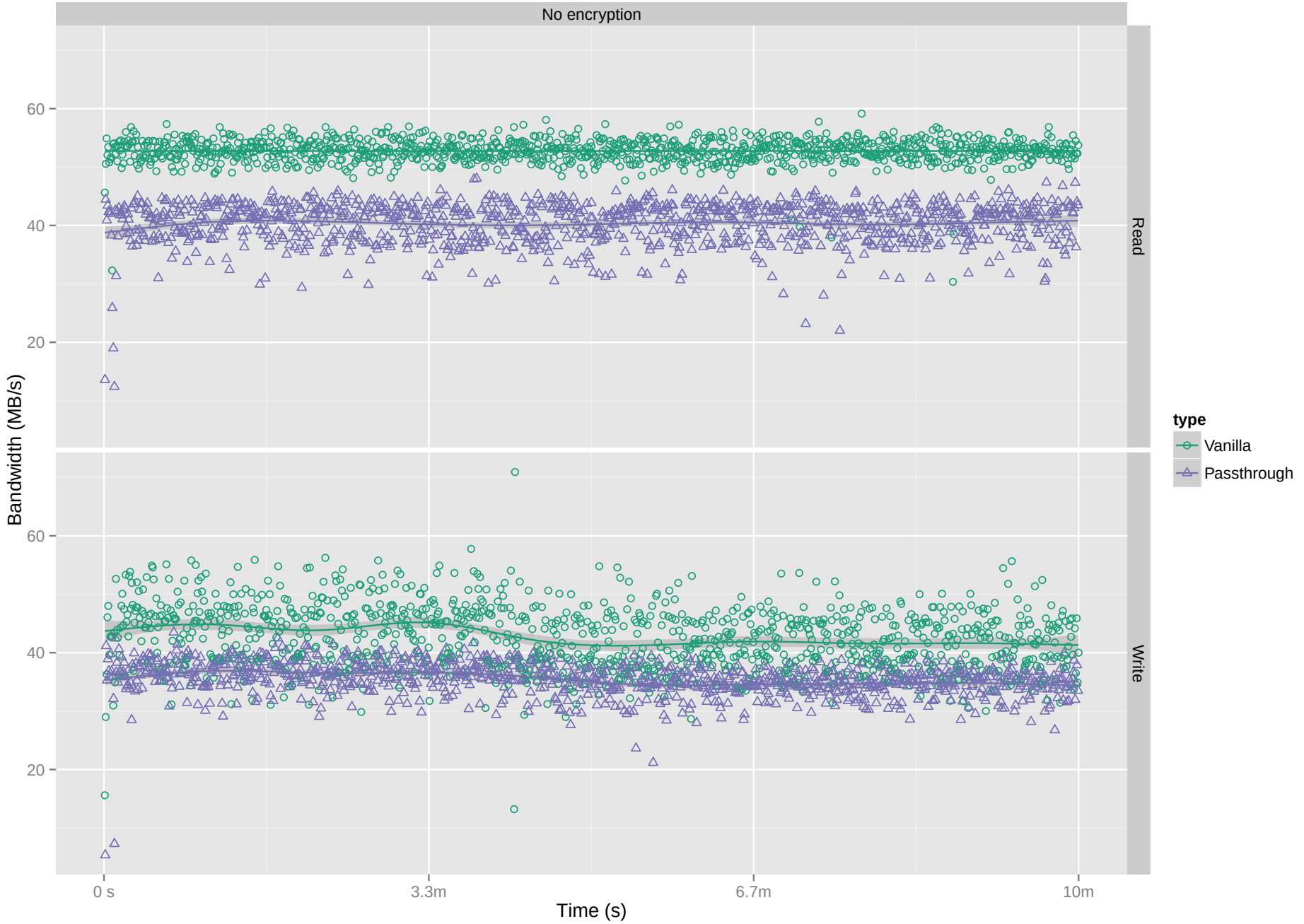


Figure 7.4: Close-up inspection scatter plot with a 4 MiB block size.

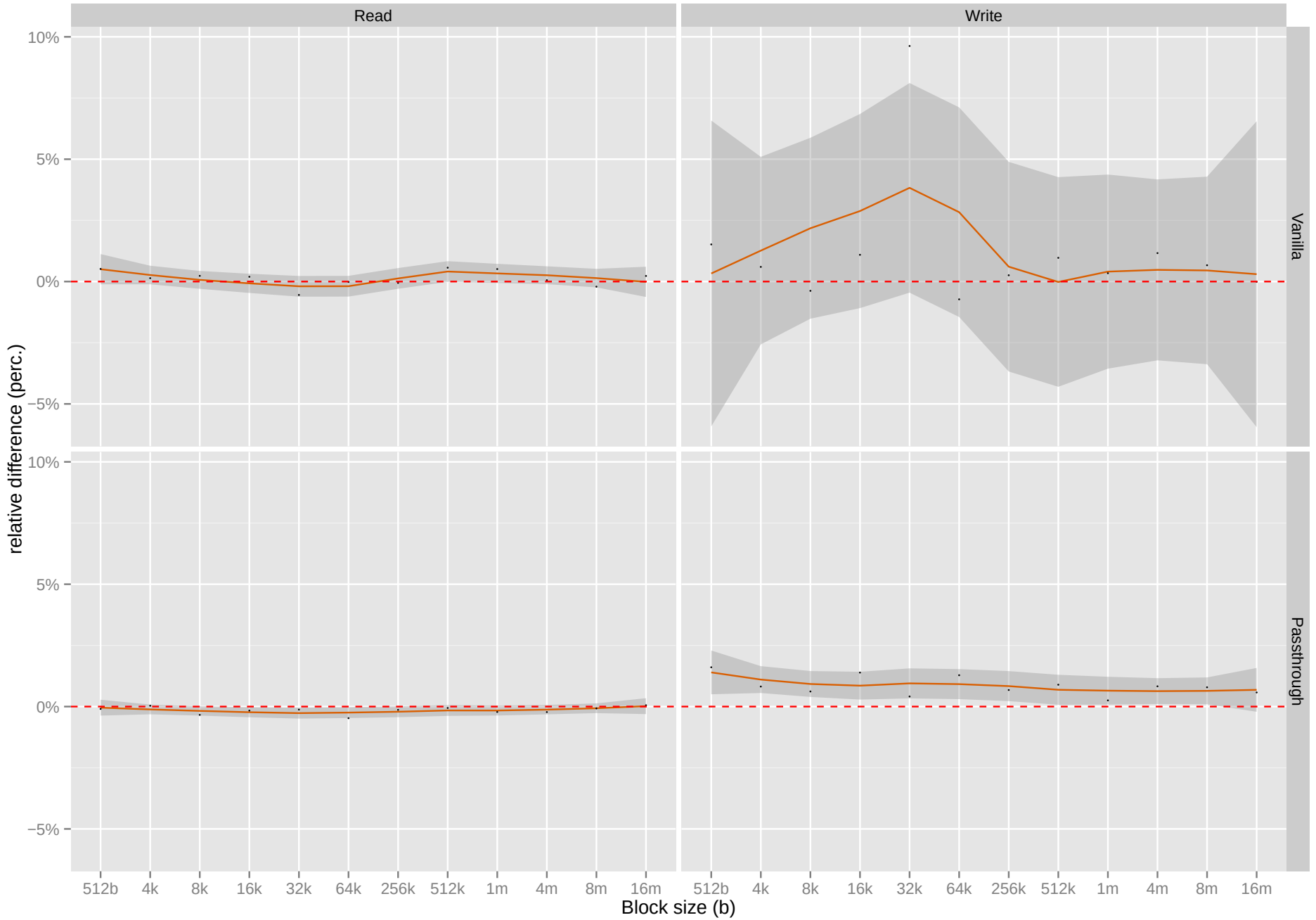


Figure 7.5: Plot of relative overhead of running with encryption, as opposed to without. The shaded area is the 95% confidence interval.

7.4. Experiments in the public cloud

The DomC relies on modifications to the hypervisor and Dom0 from a security perspective (i.e., protection against introspection as well as having guarantees using TC technology) and a functional perspective (in vanilla Xen, only Dom0 can offer back-end devices to other domains). However, the latter restriction can be circumvented if the DomU is modified.

Testing with the DomC in the public cloud is for the most part an interesting experiment. However, it is also more than that, since a DomC in the public cloud is a step toward satisfying requirement R1 (the protection of HVKs) — namely, even if the DomU is breached, an external attacker does not gain access to the DomC domain.

In the Amazon EC2 cloud we performed the following steps. Because we cannot write to the normal paths in the Xenstore (only Dom0 can do this), we need to work around this, and both domains must first share a path that they both can write to with the other party.

1. Start n VMs in a single request via the Amazon web services (AWS) API. Typically, this will give at least one pair of co-resident VMs. In our experience, $n = 20$ was a good value. Each run will cost for each VM an hour worth of computing time, though one run is typically enough.

2. For each launched VM, heuristics are used to determine co-residence. Various heuristics are possible, for example checking the first network hop [**Ris+09**], and comparing domain ID numbers (instances differing only one value in their ID are typically on the same machine). These checks are fast and efficient and useful in weeding out the many VMs which are definitely not co-resident.

   However, both these approaches generate false positives, and these need to be filtered further. The only conclusive heuristic is to test if both can access a common path in the Xenstore. Since there exist no common paths in the Xenstore to which both domains can write, one domain shares a path under its own subtree[5] world-readable, or colloquially, it "places a hook."

   The other domain tries to access this path, i.e. "search for a hook." It can do this by scanning all domain IDs until it finds a domain which has shared the respective path, or it can directly look at a suspected domain ID from the previous heuristic.

[5] More precisely, the `data` field is the only path for which this is possible.

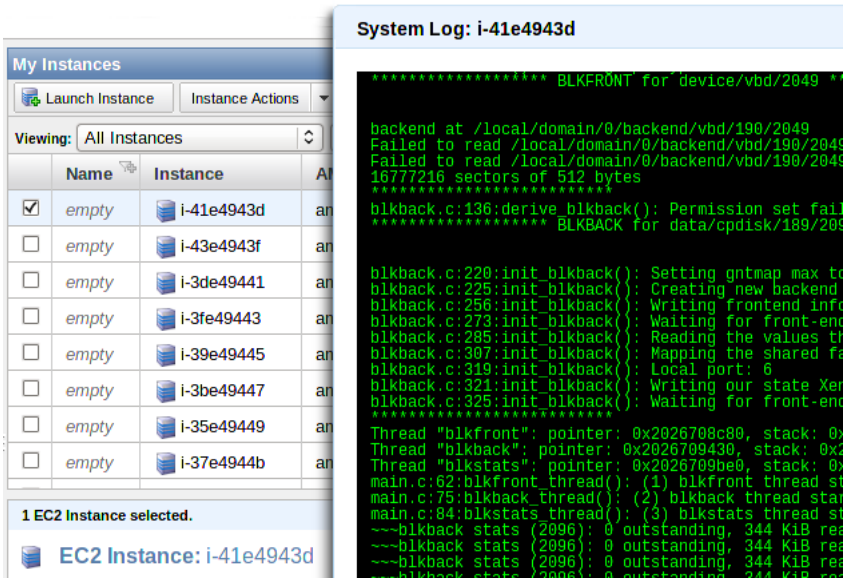


Figure 7.6: Example DomC output on EC2.

3. Given two co-resident VMs, both will need to relocate their respective front-end and back-end trees in the Xenstore to an alternative path because the default paths lack the proper permissions. The approach is the same as in the heuristic used in the previous step, by setting the world-readable (and writable) flag on a specific path which satisfies the requirements.

   On the first VM, a modified Linux kernel is uploaded and the machine is rebooted. This kernel has been modified to "place the hook". On the second VM, a modified DomC payload is uploaded and the VM is then rebooted. This modified DomC will scan for the hook, and if it finds it, the back-end will announce its device to the front-end and it appears in DomU as a new disk.

4. Now that the connection is set up, the FDE is applied transparently by DomC on any data sent by DomU. In Fig. 7.6 an example is shown how that after these steps, it is possible to view the DomC debug output via the AWS interface.

8. Future work

Extending DomC

The DomC is a flexible assistance domain which can be expanded to suit the needs of a particular use case. For instance, a system which stores personal health records (PHRs) is an example of a system which can benefit from the CaaS design [Den+12]. If the CaaS is used in such a healthcare environment, a secure auditing component could be added to DomC. Moreover, such a tool could benefit from the synergy with the HVKs by not only recording data, but also returning audit logs signed with an HVK for authenticity.

Another use case is when we take trusted virtual domains (TVDs) into account. TVDs are an established design in which a node can only join the trusted domain if the node is in a trusted state [Cat+10].

In our design, the DomC could very easily be extended such that access to the passthrough devices is prohibited if the state in the vTPM does not correspond with the defined TVD policy. In fact, the passthrough devices would not include only encrypted disk, but also a passthrough network device which automatically connects to a virtual private network.

Hardening Mini-OS

We have made extensive use of Mini-OS in order to reduce the TCB to a minimum. By doing so, we have placed critical components in a Mini-OS environment. While these components check and sanitize the input received from the untrusted Dom0 source, in the light of the established security doctrine of multiple lines of defense, additional security measures in Mini-OS would be commendable.

Currently, Mini-OS has no additional security features as for instance found in commodity kernels such as Windows or Linux. The memory layout of Mini-OS is simple. We would suggest the following improvements.

- The Mini-OS stacks for each thread reside directly after each other, making it dangerously simple to overflow the stack and corrupt data. We propose an intermediate page (which is not mapped to a physical page) placed

between the thread stacks so that a page fault is generated when a stack is overflowing.

- The principle of address space layout randomization (ASLR) and stack canaries have been extensively studied for commodity operating systems such as Linux and Windows, and could find suitable application on Mini-OS as well.

In fact, by not having to take legacy software in account, Mini-OS has the potential for drastic security improvements which might be considered too stringent for a commodity OS. Mini-OS could be able to sport having both a hardened environment and a TCB far smaller than those of commodity hardened operating systems.

Final components

Not all components of the CaaS design were implemented in the limited time available for this thesis. The PV-TGRUB and vTPM still require implementation, though for both, existing work is already available, and aligning this work with the design from the architecture chapter should not prove too difficult. Lastly, the migration of VMs (req. R5) between nodes is supported by the CaaS design (by communication between the DomTs), but not yet implemented.

9. Conclusions

We started this thesis with setting out a goal that recurs throughout all chapters. We defined the high-level goal as enabling cloud consumers to securely deploy and run their virtual machines in the cloud, while protecting their high-value cryptographic credentials against external as well as internal attackers. During the problem description, we defined this goal more clearly and translated it to a set of requirements. In particular, we further specified the abstract goal of securely deploying and running VMs in the cloud, as security in face of a powerful malicious insider. To achieve this, the HVKs, which are deployed securely and protected during runtime, play an invaluable role in enabling a passthrough encryption which closes down the last attack channel available to the malicious insider.

The design proposed in this thesis comprises many components in order to fulfill the requirements. These components can be categorized into three pillars, namely an improved access control in the hypervisor, a cryptographic assistance domain, and a trusted domain builder. The confidentiality and integrity of the cloud consumer's assets is only guaranteed if all three pillars are setup and working correctly.

While we see room for further improvement of our implementation, the essential components which lend the design its security guarantees have been implemented. Moreover, the implementation is one without significant reservations — while many authors incorporate trusted computing in their designs only by stating that their design *should* use TC to provide secure deployment, our implementation goes one step further and implements the TC steps from the encryption tool at the cloud consumer, up to unbinding at the trusted domain builder.

In summary, the CaaS design presents a multifaceted approach towards solving trust issues with cloud computing at the IaaS level. While many of the components in CaaS are not strictly unique to this thesis, the integration and composition of these solutions does result in a synergy not yet presented before. Indeed, the deployment of HVKs in a secure environment in the cloud, as well as a passthrough encryption which leverages these keys for a complete VM protection, are not only immediately usable in the status quo, but also function as a building block from which higher level solutions can directly benefit.

Appendix

A. Further introduction to Xen

In section 2.2, during the background information chapter, we introduced the overall Xen architecture and briefly mentioned Mini-OS. In this appendix we take a further look at Xen. In the first section we view Xen from an implementation perspective, e.g., the libraries and tools involved. Then, in the second section we review Mini-OS which is an important building block for the helper domains.

A.1. Xen components

In Fig. A.1 we outline the components[1] which are used in the management domain, Dom0. This figure highlights the central role assigned to user space in Dom0 to perform the management tasks — when it comes to management tasks, the Dom0 kernel is merely forwarding commands to the hypervisor on behalf of the user space tools. We briefly discuss the most important components shown.

Libxl. For easy management of the machine, the cloud administrator needs a tool via which he or she can interface with the Xen components. As mentioned previously, in Xen, this role is fulfilled by the *Libxenlight* (Libxl) library and the accompanying `xl` command line tool.[2]

Domain building is a core task of Libxl. However, the range of commands the administrator wishes to execute comprises much more than domain building alone. Other commands supported by Libxl are domain (un)pausing, saving and restoring, setting quotas, viewing currently running domains, and more. These commands are not only callable with `xl`, but through Libxl they are exposed to various other tools and libraries (e.g., cloud management software to manage a large group servers).

[1] We do not claim that this is a complete overview of Xen components. For instance, we do not show ballooning, transcendent memory, or user space device emulation. However, this overview is sufficient for discussing the implementation.

[2] The etymology of Libxenlight is that it is a successor to the previous interface to Xen: Xend and `xm` (the Xen daemon and "Xen master" tool). Some readers might be familiar with these tools, though these are now deprecated.

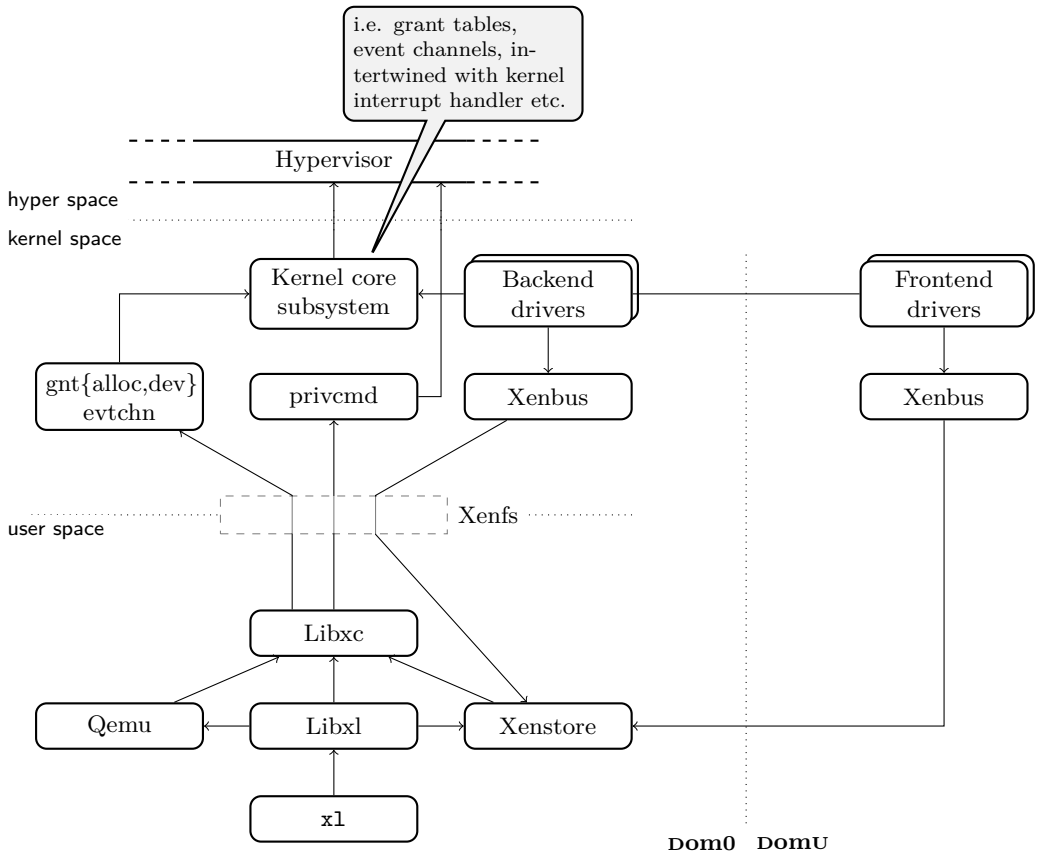


Figure A.1: An overview focused on the user space Xen libraries living in Dom0, with kernel space and hyperspace drawn above them. This diagram conveys that the management tools live in user space. The Xenfs is mounted at `/proc/xen` through which kernel and user space communicate.

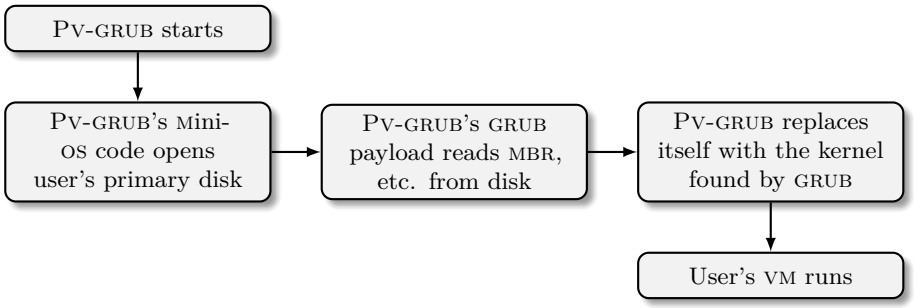


Figure A.2: The flowchart of booting a DomU using PV-GRUB.

Libxc. The Libxl library itself partly defers responsibilities to another library: *libxencontrol* (Libxc).[3] The Libxc library is the user space library which wraps around the hypercalls and does most of the low-level primitive operations required by Libxl.

Therefore, it plays a key role in for instance the domain building process, discussed in the next appendix. The CaaS design also makes use of the Libxc in the helper domains which run Mini-OS. Fortunately, libxc has already been ported to Mini-OS by Xen developers for use in the PV-GRUB project, and we can build on this effort.

Xenfs. As can be seen in Fig. A.1, most of the management logic lives in user space in Dom0. This is an application of the principle of least privilege; i.e., only the most time critical components (e.g. disk back-ends) need to live in the kernel space.[4]

The link between user space and kernel space is provided by the Xenfs component. Xenfs is a virtual filesystem which is to be mounted at `/proc/xen` and provides sockets and information for the Xenstore and Libxl to connect to.

Bootloaders. Normally, Xen expects to boot a kernel binary image directly, not a virtual disk.[5] However, from a management perspective, it is a hindrance for cloud consumers to have to ask the CSP to set up a new kernel image each time they wish to update their kernel.

[3] Do not confuse this library with Libxl; they differ one letter.

[4] Not all back-ends are necessarily handled in the Dom0 kernel. Slightly slower but more flexible back-ends can be created in user space. For instance, the emulator Qemu is typically used to back file-based virtual disks.

[5] In other words, if you take a snapshot of a partition, the hypervisor does not know how to use it. What is needed, is the kernel image used for booting, typically only a few megabytes in size.

Therefore, in order to empower users to have control over their own booted kernel, Xen makes use of a modified version of the well known GRUB bootloader [GR]. The Xen modifications, branded as PV-GRUB[6], work by first loading the modified GRUB in the DomU's address space, which then replaces itself with the real kernel the user is interested in.

In Fig. A.2 we exhibit the workflow of this enhanced GRUB. We remark that the PV-GRUB bootloader tool is an example of a tool which leverages a Mini-OS to be small and compact.

A.2. Mini-OS

Mini-OS is a minimal kernel which is designed exclusively for Xen. The origins of Mini-OS is as a proof-of-concept kernel to showcase the features of paravirtualization on Xen [TD08]. However, besides being a showcase, various interesting use cases for Mini-OS have been developed. Because Mini-OS contains no legacy code like a commodity kernel, it has great potential for executing small and isolated Xen specific tasks. However, that does not mean that Mini-OS is only suitable for unprivileged tasks. On the contrary, by privileging Mini-OS, it can in fact take over various specialized tasks from the large, generic Dom0. The advantages of Mini-OS are:

1. Smaller TCB than Dom0 for equal tasks due the lack of native hardware drivers, filesystem or system libraries. (However, the corollary is that Mini-OS is not suited for every kind of task.)

2. Isolation from manual interference — Mini-OS has no concepts of users or terminals, so there exists less room for a malicious insider to directly interact with Mini-OS and cause mischief.

3. Because all code is statically compiled, the entire OS is a single binary and therefore easily measurable.[7]

These three advantages compelled us to use Mini-OS for our helper domains DomC and DomT. While Mini-OS has distinct advantages, porting C programs from a POSIX environment to a bare-metal kernel such as Mini-OS is a nontrivial

[6] Some readers might be familiar with the so-called PyGrub system instead. This is the preceding system, which used a deprecated design, and will not be discussed here.

[7] That is not to say that a Linux setup cannot be made static and that a Mini-OS cannot be made dynamic. Linux can read a pre-packed filesystem from a ram disk, and Mini-OS can pull in scripts via the block driver to feed to a statically compiled interpreter. These cases will not be dealt with in this thesis.

task. Most C programs depend in the very the least on LibC,[8] but generally have much more dependencies. Without a LibC, Mini-OS will only run the most simple programs. The naive solution, porting the Linux GNU LibC to Mini-OS, would be complicated due the large size and dependencies on Linux specifics.

Therefore, the *Newlib* library [**Joh**+] has been ported to Mini-OS [**TD08**]. This library is a minimal C library geared towards embedded systems. Hence, porting to new environments such as Mini-OS is relatively painless. By leveraging newlib, normal C programs can be ported to Mini-OS, granted they can be sufficiently disentangled from any C libraries used besides of LibC.

[8] The LibC library provides all the basic C functions which programmers take for granted, such as the string functions (strcpy and friends), memory functions (malloc and others), and many more.

B. Implementation details

In this appendix we give a detailed discussion of how the components in our implementation work. It is recommended to consult appendix A for a deeper introduction to Xen in order to understand the technical topics in this chapter. The structure of this chapter will be component oriented, in the order presented in Fig. 6.1.

The implementation in this thesis is based on Xen version 4.1.2.

B.1. Overview

In Fig. B.1 we give an overview of how our modifications to the Xen management libraries look in the overall picture. This only shows Dom0 and DomT, since the DomC plays no role in domain management. This figure shows that most of our modifications and hooks live in Dom0 user space, where they forward functionality to DomT.

In addition, our modifications and extensions to Mini-OS are exhibited in Fig. B.2, showing the core Mini-OS components and our additions contained in DomC and DomT. These components will be explained in the sections that follow.

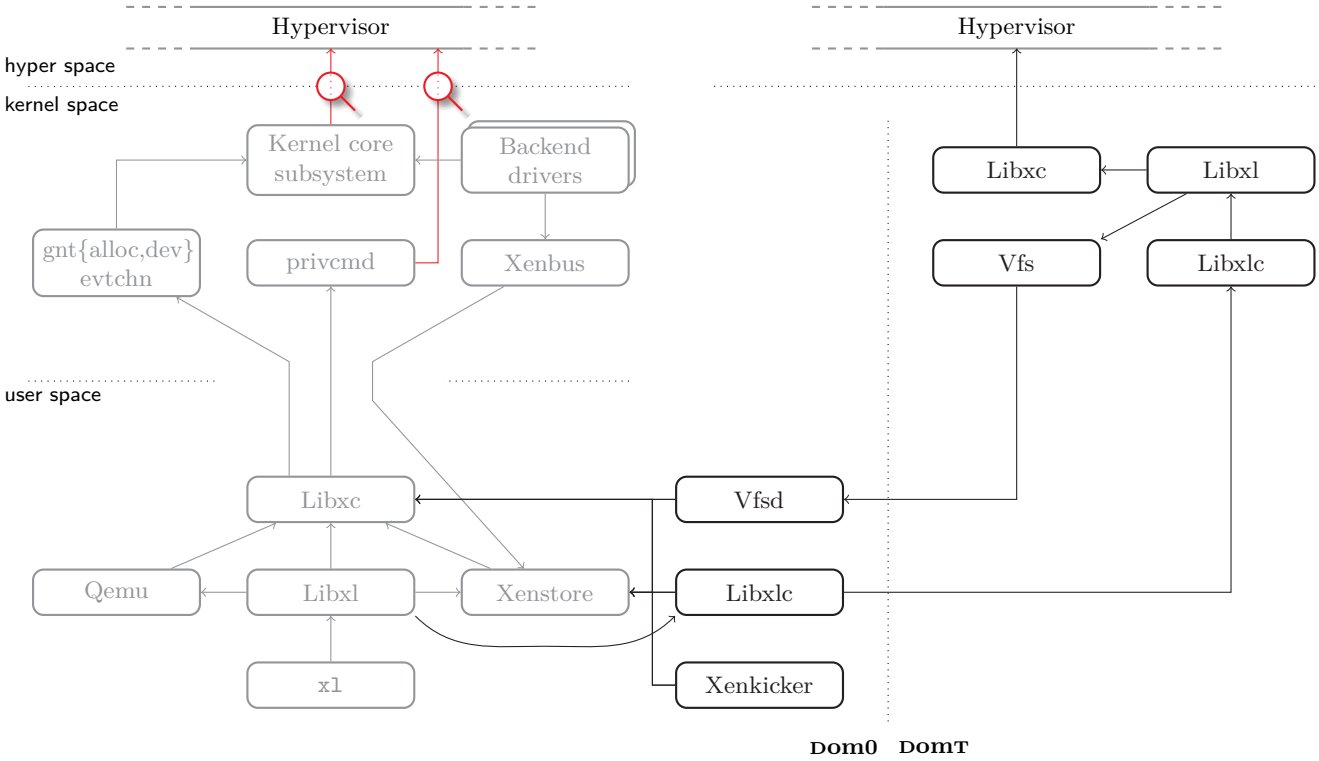


Figure B.1: An overview of the original management libraries from Fig. A.1 in gray together with the CaaS additions in black. In red, we highlighted the Dom0 privileged hypercalls which are now subjected (but weren't in vanilla Xen) to checks of access control by our CaaS security module.

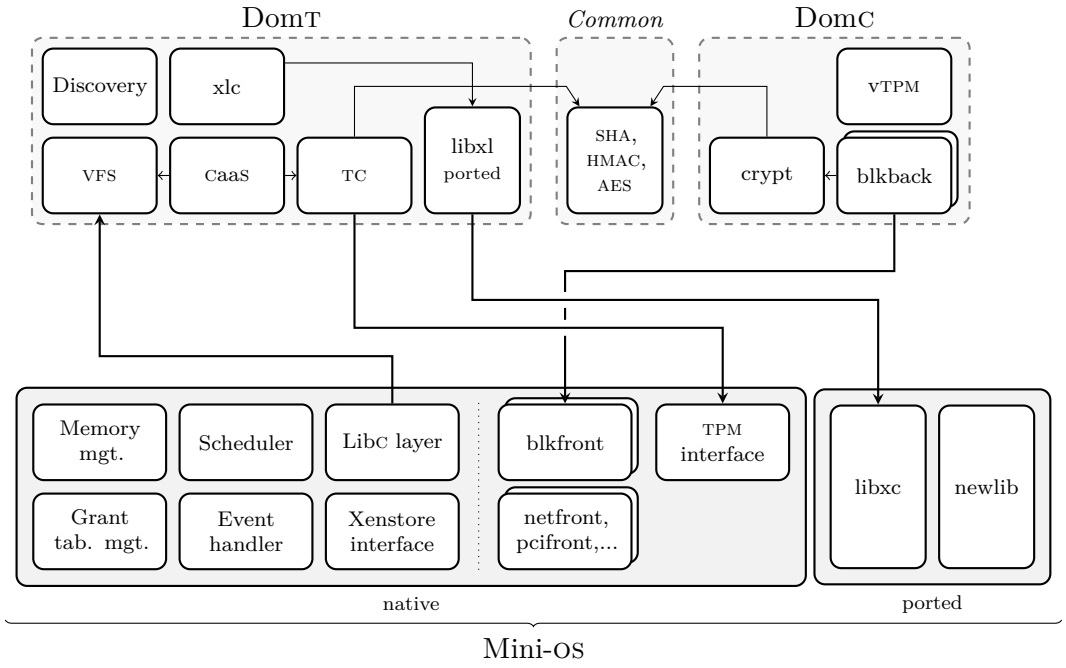


Figure B.2: Mini-OS components, with the original components shown at the bottom and our added components shown above it. The layering is only conceptual, all code runs in the same address space and protection ring. For brevity, we only highlighted the most important interactions using edges.

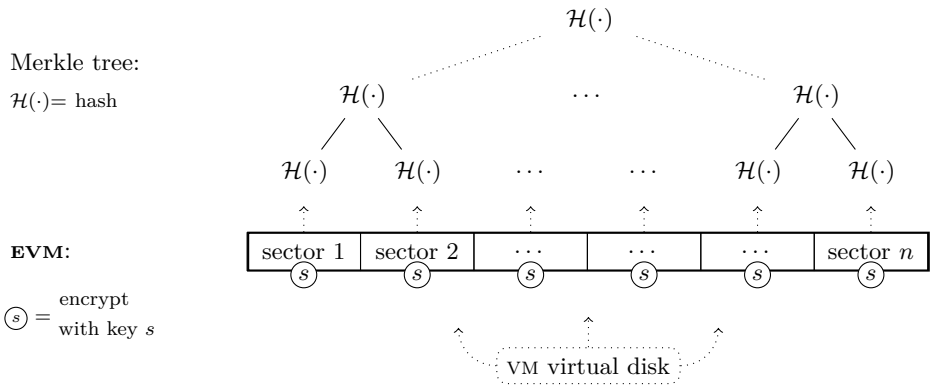


Figure B.3: The $E(\text{VM})$ data structure (from now on, EVM) is the virtual disk encrypted with key s. A Merkle tree is derived from the EVM by hashing sectors and ordering these in a tree.

B.2. Data structures

In this appendix we discuss the two most important data structures which play a role in our implementation.

Encrypted VM. In Fig. B.3 we exhibit how we apply a full disk encryption (FDE) scheme to secure the VM (the precise FDE scheme follows later in appendix B.8.1). The notion of a VM that the cloud consumer might have, is in fact simply a filesystem on a virtual disk with n sectors. By applying encryption on each sector of this virtual disk using key s, we get the data structure we refer to as the *encrypted virtual machine* (EVM).

We remark that the symmetric key s is not only used for encrypting the filesystem but also for important pieces of metadata which also need to be protected (discussed below). Hence, key s plays a central role in our data structures. For maximum entropy, and considering that the key never needs to be entered by the user, this key should be generated randomly by the cloud consumer's software the first time.

Merkle tree. For efficiency reasons, FDE schemes are typically designed such that decrypting a single sector does not depend on the decryption of other sectors (although the position with respect to other sectors matter, i.e., swapping will be detected).

This has no consequences for *confidentiality* as long as the block cipher is sufficient and the FDE scheme does not leak information. However, for integrity,

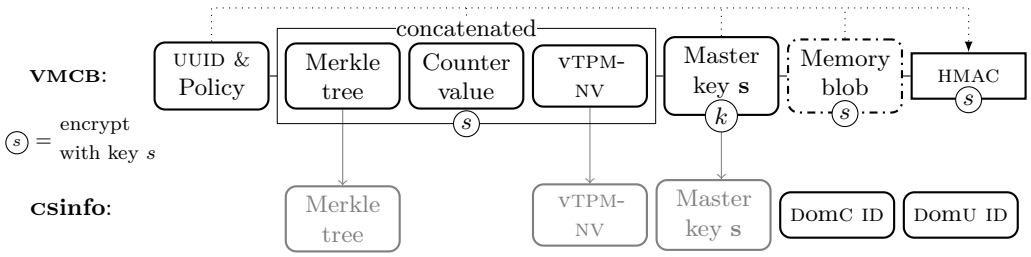


Figure B.4: The VMCB data structure (which is provided to the DomT by the consumer), and the CSinfo data structure (which is injected into the DomC later). Variable s is a randomly-generated symmetric key also used in EVM (cf. Fig. B.3) and k is the public binding key. As visible, s itself is encrypted by k. For all encryption, padding and CBC are applied if necessary.

the implications are that any tampering with the ciphertext at a single position will only scramble parts of the plaintext and not the complete filesystem. Such tampering might go undetected with the user (violation of integrity of VM, R2). Furthermore, by default such a scheme offers no protection against an adversary rolling back certain blocks with older versions (freshness, R4).

Integrity protection using message authentication codes (MACs) is a well known approach in information security. Moreover, as identified by Yun et al., it is inevitable that each block will have to be hashed in order to achieve complete integrity protection of a disk — this cannot be reduced further [**YSK09**]. Therefore, we calculate MACs of all the disk sectors and construct a hash tree of these, i.e., a Merkle tree [**Mer87**]. In the upper part of Fig. B.3 an example of such a Merkle tree is shown. After the construction of the Merkle tree is complete, the encrypted virtual disk `EVM` is the first component of the tuple that will be sent to the cloud later on. For the sake of integrity, the Merkle tree much be protected as secure as the key s, though it suffices to store only the topmost hash, since the tree can be rebuilt with only the topmost hash.

Metadata. Figure B.4 shows the *virtual machine control blob* (VMCB) structure. The VMCB contains metadata regarding the VM which the consumer wants to virtualize. The UUID is uniquely generated for this VM and is unencrypted to ease administration for the CSP.

The virtual TPM non-volatile memory (vTPM-NV) contains the non-volatile memory of the consumer's vTPM. By determining the contents of the vTPM-NV beforehand, the consumer is able to secure high value keys under all circum-

stances (req. R1); even if the VM is compromised, the HVKs remain secure.

The virtual counter value (cf. TC introduction, section 2.3) is the value the virtual counter had at the time of VMCB update. The value will be initialized to zero by the cloud consumer when creating the VMCB. The DomT will maintain a set of virtual counters indexed on the UUID of the VMCBs. This ensures that a malicious insider cannot rollback the VMCB — nor the EVM of which the Merkle tree is a digest (req. R4).

The memory blob is an optional parameter and will not be used directly by the cloud consumer. When Xen saves the execution state of a VM to disk, the RAM of the VM is an additional asset that needs to be protected with the same requirements as the EVM.

The hash-based message authentication code (HMAC) glues all the properties of the VMCB together and prevents individual properties from being rolled back by a malicious insider (req. R4). The master key s will be encrypted using an asymmetric key k, which refers to the certified binding key of the target platform.

The CSinfo shown in Fig. B.4 refers to the data which is injected into DomC during domain creation (cf. Fig. 6.6). The CSinfo contain certain data from the VMCB as well as two domain identifiers; by injecting these values in DomC by a trusted source (DomT), DomC cannot be tricked into offering the passthrough encryption to the wrong domain by a malicious insider abusing the Xenstore (req. R3, coupling of VM and HVKs). We discuss this attack and its implications in more detail in appendix E.

Figure B.5 exhibits the interaction between the entities when deploying a VM. In the first phase, the consumer takes preparatory steps which are merely the implementation steps of constructing the entities shown in Fig. B.4, namely the EVM and VMCB structures. Because DomU is indirectly built (cf. Fig. 6.6), the EVM (variable d) stays in Dom0 and only the VMCB (variable b) is transfered to DomT.

B.3. Deprivileged management domain

In the CaaS design, the system has stopped relying on Dom0 for domain building, and thus Dom0 is evicted from the TCB. However, unless the hypervisor is modified to reflect this, Dom0 still has the privileges to execute attacks. Therefore, it is critical that the hypervisor's access control reflects the new division of work between the Dom0 and DomT domains.

In vanilla Xen there exists by default a very basic access control, which is nothing more than a boolean privileged versus unprivileged status flag. This

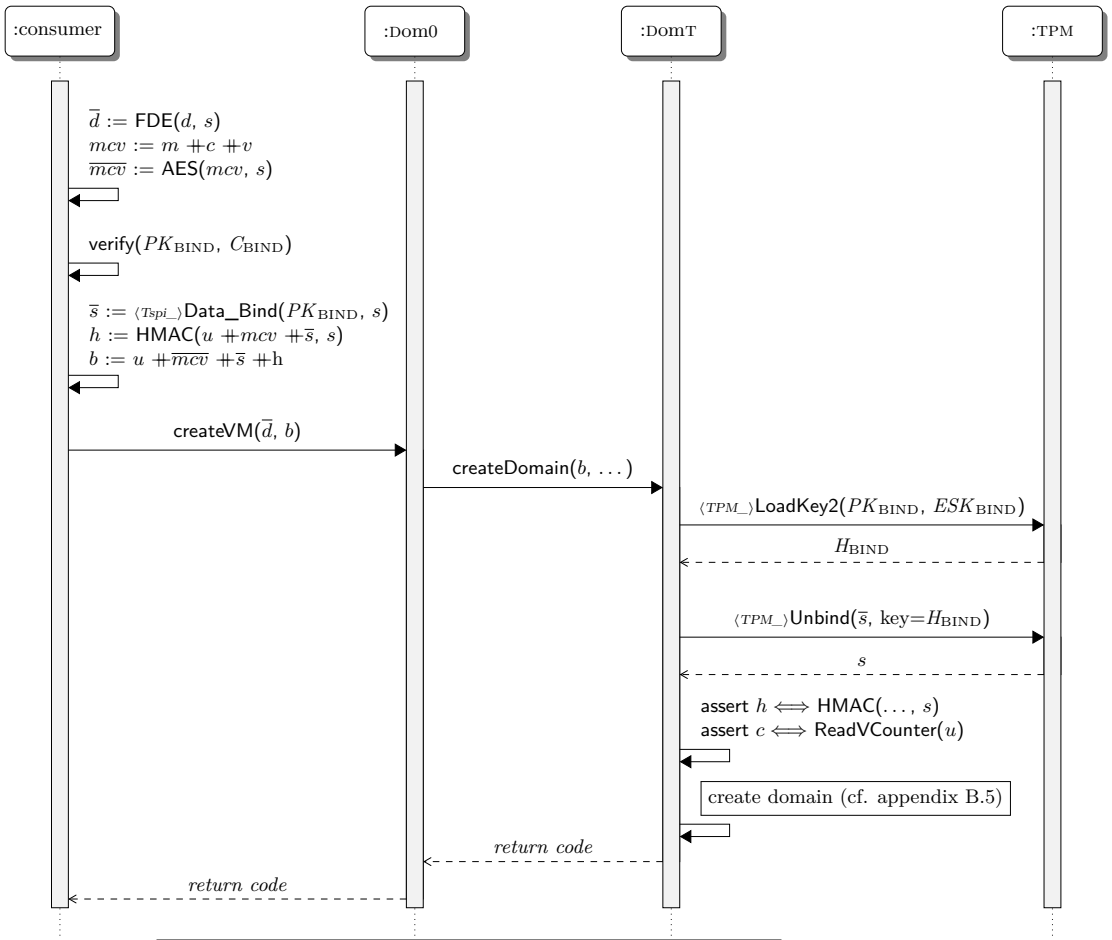


Figure B.5: The interactions between the entities for deploying a VM. The + symbol indicates concatenation and an overline indicates a ciphertext.

suffices only for the most simple scenario in which there is one domain which fulfills all of the privileged tasks. Fortunately, there already exist a more expressive development in this area on which we can build.

B.3.1. The XSM framework

The XSM framework, also known as sHype, is a framework for writing security modules for Xen [Sai+05]. The XSM framework design is an application of mandatory access control at hypervisor level. The discussion of mandatory access control is out of scope as it is a large topic; however, its application here is simple, it allows to express strict regulations which curb the free deployment of VMs more than what is possible in vanilla Xen. For instance, one might desire a strict Chinese wall policy such that customers Alice and Bob are never co-resident on a single machine. Another example is the desire to prevent the VMs from Alice and Charley to communicate with each other under any circumstance.

The XSM framework is implemented as modifications to the hypervisor source code. To allow for the expressiveness of aforementioned example policies, the XSM framework adds many hooks throughout the hypervisor source code via which a XSM security module can regulate the control to hypercalls.

Xen comes with two available implementations which hook into the XSM framework: the access control module (ACM) and Flask.

1. The ACM framework, which appears unmaintained, is oriented at implementing Chinese Wall or simple type enforcement policies, both limited in expressiveness.

2. The Flask system borrows much from SELinux. One compiles a policy using a SELinux policy compiler in which one describes how to act at all the hook points. This policy is then compiled to a binary format which is passed to the Xen hypervisor at boot time.

   For each VM that is started, the VM configuration file will hold information on which category the VM belongs to. For instance:

   ```
   access_control = ["policy=,label=system_u:system_r:domU_t"]
   ```

   Expresses that the booted domain is a PV guest.

B.3.2. The CaaS security module

Neither of the two distributed XSM plugins satisfied our needs. While ACM is limited in expressiveness, the Flask approach did not compile for us and did

not cover all the hooks we wanted.[1] (Though if these problems did not exist, Flask might have been a good candidate.) On the other hand, we were required to make numerous changes to the XSM hooks as they were laid out by default.

- For example, many hooks would record the *target* of a certain hypercall, but only a few hooks would record the *invoking* domain. This is a reflection of the original design of XSM where it serves as an *addition* to the existing privileged vs. unprivileged access control — i.e., it bluntly assumes the invoker is Dom0 for certain hypercalls. Hence, we expanded many XSM hooks to allow for our disaggregation of Dom0 privileges.
- Furthermore, not all hypercalls were fitted with XSM hooks.

As a result of these considerations, we decided to create our own XSM module which we simply named the CaaS XSM (or security) module.

From one to two privileged domains. For our XSM module, we first asked ourselves the question what the division of responsibilities between Dom0 and DomT is. In vanilla Xen, Dom0 is many different roles at the same time. However, in our design we discern the following division of privileges.

- Dom0: disk and network emulation, VM management, platform configuration (e.g., system clock, power management).
- DomT: domain building, controlling the TPM.

Our conclusion is that the situation is not black-and-white, rather, Dom0 and DomT together share the privileged hypercalls (with the critical notion that, for our requirements, DomT exclusively controls the memory). With the assignment of roles laid out, the next step was to investigate all the hypercalls found in the Xen source code. The result of this large analysis is found in appendix F — for each hypercall we analyzed under which responsibility the hypercall falls. In summary, there are four categories:

1. hypercalls belonging to Dom0;
2. hypercalls belonging to DomT;
3. hypercalls assigned to both;
4. hypercalls assigned to neither.

[1] However, improvements seem to be in the pipeline for the upcoming Xen 4.2 release.

Finally, in the implementation step of our CaaS security module, we implemented our security module using the XSM hooks. A detailed overview of which XSM hooks we used is presented in appendix G. For each XSM hook, we applied the category restriction defined in the assignment of hypercalls found in Table F.1.

In our design, the CaaS security module is compiled into Xen and enabled at all times. We remark that removing the module from the compilation will lead, obviously, to a different hash of the Xen hypervisor, and thus be detected as tampering by the appraising party.

Remarks regarding CaaS security module.

- Since DomT is part of the TCB, we could have decided to make DomT omnipotent and give DomT access to all available hypercalls. However, this would not correspond with the principle of least privilege; moreover, we clearly defined the responsibilities for DomT, so it is sensible to restrict DomT to only those hypercalls that it needs to fulfill its duties.

- We removed a set of privileged hypercalls by assigning them to neither Dom0 nor DomT. The removal of these hypercalls follows from the division of responsibilities — for instance, we did not assign the responsibility of "debugging VMs" to any of the domains, hence, these hypercalls do not need to be enabled in a production environment. The small set of shared hypercalls are composed of overlapping hypercalls (e.g., unpausing a VM is needed for Dom0's management taks as well as for the domain builder in DomT) or due to the innate nature of both work domains (e.g., writing to the Xen debug console is desired for all helper domains, both Dom0 and DomT).

- Implementing all the XSM hooks in correspondence with how we assigned the responsibilities is a considerable amount of work; in particular for those hooks which are not yet in the form desired for expressing our policy or for those hypercalls for which XSM hooks are vacant. Therefore, we chose to restrict our implementation to only the current available hooks.

  Notwithstanding that our CaaS security module can be improved, all the dangerous hypercalls — such as those used for introspection — have been curtailed properly in this implementation. For all intents and purposes, the CaaS security module provides full security against the adversary from the attacker model (section 3.1). Trial runs with our provisional CaaS security module did not lead to any instabilities, however, we cannot rule

out the possibility that our hypercall and XSM analysis in appendices F and G can be improved.

B.4. Virtual filesystem driver

For various components in DomT, such as the domain builder and loading of TC keys and blobs, it is necessary to read data from Dom0 files. However, as DomT contains no filesystem drivers, adding these would bloat Mini-OS' small codebase. The compromise is to rely on the virtual filesystem (VFS) kernel driver in the Dom0, rather than a block device.

In Linux, the virtual filesystem (formerly called virtual filesystem switch) ties together all the various filesystems the kernel can operate on. For instance, the main partition is mounted on `/` and the proc filesystem on `/proc`. Regardless of whether opening a file on `/example` or `/proc/example`, the kernel makes sure that the request ends up at the correct filesystem driver.

As we know from the attacker model (section 3.1) availability is not considered. This means that, as convenience, we can rely on reading files from Dom0 as long as we make the assumption that this data may be untrustworthy.

This observation has been made earlier by Murray et al. who made a VFS back-end driver between their DomB and Dom0 [MMH08]. However, inquiry revealed that their code was never upstreamed and that their patches do not apply on a current version of Xen. Moreover, Thibault and Deegan also speak of a project with a VFS split-driver [TD08], but also of this work we could find no further details. Therefore, we implemented a complete VFS solution for Mini-OS which provides transparant hooking of the I/O calls in Mini-OS.

B.4.1. The CaaS VFS bridge

As we exhibited in the Xen overview figure during the introduction (Fig. 2.4), virtual devices in Xen typically have the form of a front-end and a back-end communicating over shared memory. This paradigm is well suited for devices with a strict defined interface. On the other hand, it is quite involved to write such drivers, and creating a high performance VFS interface is not a core goal of this thesis. Hence, we take a simpler route by using a hybrid approach which uses the Xenstore for commands and grant tables for data transfer.

Our VFS bridge. The system calls that our VFS bridge implements (and reroutes to Dom0) are listed in Table B.1. This coverage is sufficient for (i)

System call	*Description*
open	Open a file descriptor
close	Close a file descriptor
stat	Get details on a given file
lseek	Move the offset around in file descriptor
read	Read from file descriptor
write	Write to file descriptor
mmap	Map parts of a given file directly into memory
majorminor	Return major and minor numbers for a device[a]

[a]Each Linux hardware device has been assigned a unique major minor and minor number. The ported domain builder relies on this.

Table B.1: An overview of the small set of system calls that need to be available in the VFS bridge.

loading VM images used in the domain builder, and (ii) for reading and writing DomT state and TPM keys.

In Fig. B.6 we present an example of how our VFS daemon approach operates. An important point which can be seen here, is that DomT treats Dom0 as untrusted. Moreover, for each I/O buffer of data in DomT's memory, a zeroed bounce buffer is created of equal length. This is necessary because we can only share complete pages with Dom0 and we have no desire to expose unrelated data to a potentially malicious Dom0. In the next step, all the pages in the bounce buffer are shared (since the grant table works at page-level), and each grant reference (an integer) is then accumulated into a large list — which by itself is in a page which is shared with Dom0 as well.

Summary. For our implementation, we built a rudimentary but stable VFS bridge between Dom0 and DomT which meets our functional requirements. Although we did not perform measurements on this part, we can conclude from the design that our approach is not the fastest attainable.

However, this was never the goal; from the outset, it was clear that DomT will only seldom need to perform disk I/O, and then a few milliseconds overhead are negligible. On the other hand, by keeping the VFS interface simple, we prevented creating exploitable coding errors.

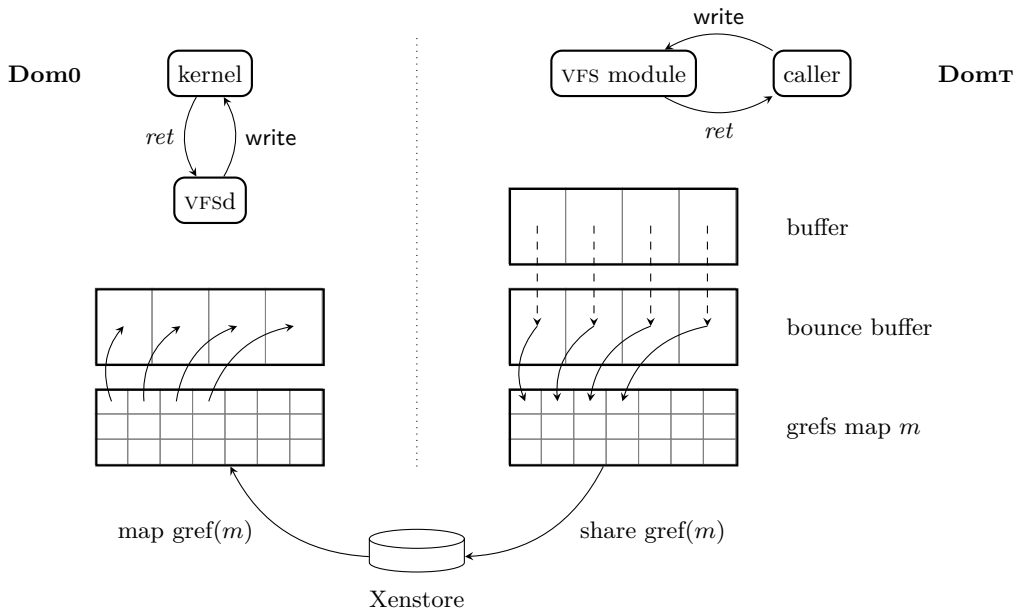


Figure B.6: VFS daemon operations for a write call. On the DomT side, a component (e.g., domain builder) writes to a file. This call is handled by the VFS handler, who creates a bounce buffer to which it duplicates the buffer. The buffer is composed of several pages, and each of these pages is shared using the grant table. All the grant references accumulate into a page, which itself is again shared using the grant table. This grant reference is shared through the Xenstore and expanded at Dom0. The VFS daemon then executes the system call on behalf of DomT, returning the return value through the Xenstore.

B.5. Domain builder port

As noted before, the domain building process in Xen comprises an intricate number of steps. Among things, it is necessary to parse the kernel binary blob, setup initial page tables, place any ramdisks, setup domain bookkeeping metadata, and connect the Xenstore and Xenconsole to the new domain. Due to the minimalistic nature of the Xen hypervisor, these tasks are for the most part performed in Dom0.

Moreover, these are not atomic operations. In other words, in one hypercall the hypervisor is asked to allocate a memory range for the new domain, while

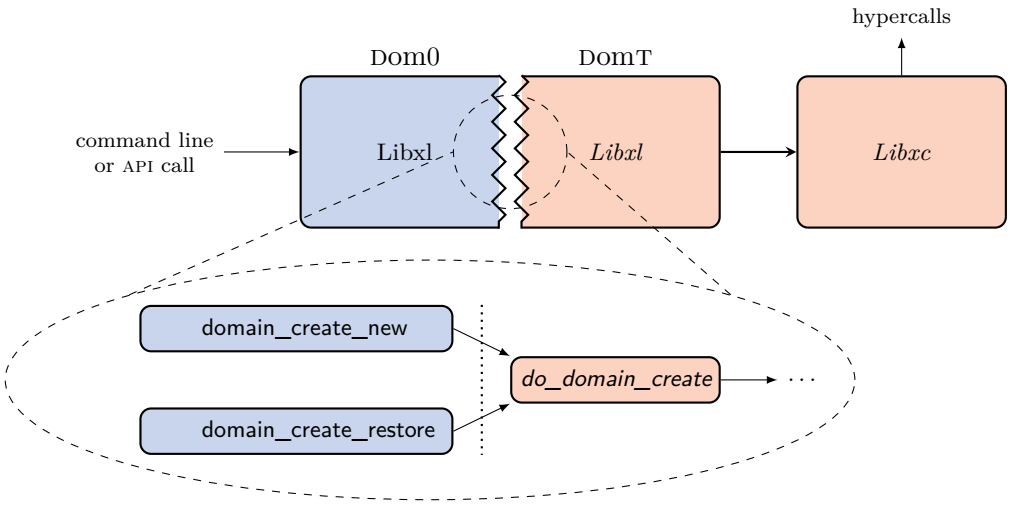


Figure B.7: The break-up of libxl over an unprivileged Dom0 and privileged DomT part during the domain builder process.

in another hypercall this memory range is mapped into the page tables of Dom0, in order that Dom0 can write to this new domain's memory. There is no single hypercall that merges all the operations into one hypercall, and neither would this be feasible due to the diverse operations that Dom0 needs to perform (e.g., PV differs from HVM, during suspension Dom0 needs to read the memory again, etc.)

B.5.1. The CaaS domain builder

Our approach generally follows the solution as laid down by Murray et al. [MMH08], though we had to create our solution from scratch.[2] In this architecture, the domain building process (which includes various sensitive privileged instructions) is moved to a separate domain builder domain who has read access to the Dom0 disk.

In our design, we consider a real-world scenario in which privileges are divided between Dom0 and DomT and not fully assigned to either of these (cf. appendix B.3). This implies that some privileged commands such as assigning VM quotas and (un)pausing VMs remain with Dom0, and this needs to be reflected in the design of the Libxl splitting.

[2] An inquiry with the authors learned us that their code had never been upstreamed and based on a Xen version of a considerable time ago.

In Fig. B.7, it is exhibited how the Libxl is broken apart over a Dom0 part and a DomT part. The DomT part of Libxl relies on a Libxc which exclusively lies in the DomT. The functions shown in Fig. B.7 at which we break the build process have been carefully chosen; the determination of where to break is under influence of two opposing interests.

- On the one hand, one wishes to place as much code in the DomT as possible. By doing so, we can reduce the number of calls that need to be made between Dom0 and DomT. However, more importantly, this would ensures that we have sufficient context information available to determine whether a call is to be allowed or not. This plays a key role because calls to Libxc during the building process are scarcely distinguishable from calls outside the building process, when they are used for malign introspection instead.

- On the other hand, however, it is unnecessary to route functions for which Dom0 retains privileges (i.e., management related hypercalls) through DomT. Moreover, there are parts of the domain building code in Libxl which are strictly Dom0 related; for example, attaching to the output console of the newly started domain makes no sense outside of Dom0 and would be complex to create in DomT.

  Furthermore, an important factor in our considerations is to make as little changes to the build process as possible. The 'break' in the build process should not necessitate additional rewriting of Libxl or Libxc code. Such excessive refactoring risks breaking dependent behavior (i.e., migration code) and makes it more difficult to rebase our code on newer versions of Xen in the future.

In our approach, we use C preprocessor directives which check if we are compiling Libxl for DomT or for Dom0 and then include only the relevant code. By doing so, we are able to develop our CaaS solution on the code base of a pristine Xen source tree with minimal rewriting.

Comparison of the break point. With the two opposing influences in mind, we determined that the optimal breaking point is the location shown in Fig. B.7.

On the other hand, Murray et al. break the build process at a different point, namely not in Libxl but in a function from Libxc [MMH08]. However, the function they used, ⟨xc_⟩linux_build, is now deprecated in Xen due to design changes [XML1]. Rolling back these Xen changes would require significant rewriting of Libxl and Libxc and would have no prospect of ever being accepted

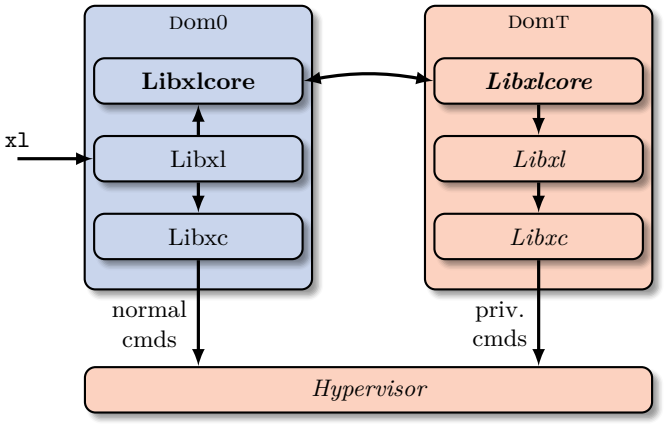


Figure B.8: Schematic overview of the *xlcore* communication channel. Domain building calls initiated in Dom0 are forwarded to DomT.

upstream. Hence, the approach of Murray et al. was sensible at that time, but is no longer applicable to a modern Xen version.

B.5.2. Inter-VM pipe

In networked environments, a 'breaking line' in processes is a well understood problem, for which the remote procedure call (RPC) methodology is the standard solution. In analogy to inter-process communication, the term inter-VM communication (IVMC) has been coined in literature for the communication taking place between VMs [**MMH08**].

In our implementation for IVMC between the management domain and the trusted domain builder, we created a new library in Dom0 to wrap those functions which are now handled in DomT. This library, which we called *Libxlc* (Libxl-core) because it handles the core Libxl operations, is shown in Fig. B.8. Observe that there are two Libxls and two Libxcs, either for normal or for privileged commands.

The use of the Libxlcore communication channel operates in a manner similar to how we handled the VFS daemon (appendix B.4). In fact, the Xenstore-based communication channel (cf. VFS example in Fig. B.6) is again used, but with sender and receiver swapped.

How the VFS underlies domain building. The VFS interface plays a fundamental role for the domain building process in DomT. By allowing our Libxlc library to forward ordinary build instructions (which inform the domain builder

where on Dom0 to find the VM config file) to DomT we do not need to modify the domain building code.

The VFS code in DomT is developed in such a way that it reroutes ordinary POSIX file I/O to Dom0. That is, if the code in in our Mini-OS port of Libxl tries to access the file `/etc/xen/example.cfg`, in fact the underlying VFS code is used to fetch the file from Dom0 in a fashion completely transparent to the calling code.[3] Again, by not having to consider file I/O from a Mini-OS as something different than in native Linux, we minimize the changes we need to make to Libxl.

B.6. Direct booting of DomT in conjunction with Dom0

As we explained in chapter 5, TBoot loads the hypervisor, after which the hypervisor loads *both* Dom0 and DomT.

B.6.1. TBoot

In section 2.3 we introduced the various trusted computing concepts. We also mentioned that nowadays, it is possible to create a *dynamic root of trust for measurements* (DRTM) using the new tools provided by Intel and AMD, respectively TXT and AMD-V. In the architecture chapter, we introduced TBoot which helps us with bootstrapping the components in a secure matter. Here we briefly review the essentials regarding TBoot which show why it is useful for CaaS. In appendix C we give a full and complete overview of how TBoot works and how we precisely configure it (including which PCRs).

Localities. A brief point that is noteworthy regarding the new DRTM instructions, is that they are part of the TPM specification version 1.2 which also foresees in the addition of eight new, dynamic PCRs. These new PCRs are *locality* bound; this means that only software running at the required locality can access these PCRs. The list of locaties, with a brief description, is (from restricted to less restricted):

- *Locality 4* — Intel TXT hardware use only
- *Locality 3* — *authenticated code* module (see appendix C)
- *Locality 2* — trusted OS
- *Locality 1* — an environment for use by the trusted OS

[3] As shown in the Mini-OS overview picture, Fig. B.2, the Libc layer in Mini-OS reroutes file I/O calls to our VFS module.

- *Locality 0* — non-trusted and legacy TPM operation

The important contribution of localities to the CaaS design is the fact that we can allow the use of the lowest-level locality to Dom0, without compromising the security of our design. To achieve this, our access control module (appendix B.3) ensures that the higher localities can only be assigned to DomT and not to Dom0. Hence, only DomT can talk to the TPM at the locality of a trusted OS — since the certified binding key that we create is bound to PCRs *and* localities, it is not possible for Dom0 to use (and abuse) this binding key.

Protection against DMA. As we mentioned in the problem description in chapter 3, we recognize DMA as a risk to the confidentiality and integrity of the memory state of a VM. Fortunately, Xen already supports VT-D in both the hypervisor as well as domain building.[4] Therefore we leverage this functionality to protect against DMA attacks.

B.6.2. Direct booting of DomT

In the vanilla Xen design, the hypervisor boots *only* the Dom0 domain, which then takes care of booting other domains. In our design, however, it is vital that the hypervisor, which is part of the TCB, directly boots the other TCB component, DomT. Booting DomT in the same manner as Dom0 is in principle not too involved, since both domains are paravirtualized. But there are differences.

- A normal DomU expects to find in its start info page the grant references for the Xenstore and the console.
- Likewise, it expects to find event channel numbers in the start info page to which it can immediately connect to signal Dom0 for waiting data on these channels.

Our approach involved changes in three areas.

Hypervisor. We modified the Xen hypervisor to consider the first multiboot module passed to it (by TBoot) as the DomT module. The first module following DomT will now be considered the Dom0 kernel instead, and the first subsequent module will be considered as Dom0 ramdisk.

We also extend the VT-D protection to the DomT memory range, protecting DomT against DMA attacks.

[4] The VT-D technology is Intel's implementation of an input/ouput memory management unit (IOMMU), the device which adds a layer of indirection to DMA, preventing access to undesired memory ranges.

Property	Value
Key type:	Binding key
PCRs:	TBoot, Xen, DomT, and the templates
Locality:	Locality 2 (only a trusted OS can use the key)
Migratability:	Non-migratable

Table B.2: Key properties for the certified binding key created in the TPM.

Mini-OS modifications. Mini-OS is not designed to be started without the console or Xenstore being available. Hence, without any modifications it will crash immediately when writing to the console during booting.

Our solution involves establishing the Xenstore and console pipes at runtime. We modified the boot order for Mini-OS components to bring up as very first the grant table mechanism and interrupt handler. Any Mini-OS debugging output up to this point will be outputted to the Xen console instead of the ordinary console from Dom0. Mini-OS will sleep until Dom0 is ready to make a connection, after which Mini-OS proceeds with the normal initialization of its remaining components, and after that, the launching our payload. This work is embodied in the 'discovery' module we created, as depicted in the Mini-OS overview (Fig. B.2).

Dom0 helper tools. The communication channel between Dom0 and DomT suffers from a chicken-and-egg problem: we would like to share the grant reference and event channel port number of the Xenstore ring buffer with the other party — but normally one shares this information through the Xenstore! To solve this, we could either (i) add an hypercall to pass these numbers around, or (ii) use conventions that reserve grant references and event channel port numbers for this use. We opted for the latter approach.

While Dom0 and DomT boot simultaneously, in the typical scenario the Mini-OS based DomT will be quicker to initialize than the Linux based Dom0. As described in the previous paragraph, DomT will wait until Dom0 is ready. In the final phase of booting Dom0, it will start up our Xenkicker program, which is a simple C program that, using Libxc, maps the grant references and connects to the event channels of which the numbers are fixed by convention. After this step, the Mini-OS unpauses and continues to boot the DomT.

B.7. Trusted platform module driver

The DomT creates a certified binding key with the properties shown in Table B.2 and performs various operations using the TPM. Therefore, as shown in the Mini-OS overview (Fig. B.2), we created a TC module for DomT which implements the TC operations from section 5.3. Our TC module communicates with the TPM through the TPM interface, although this is merely a thin layer. The TPM is an example of a device accessible through memory mapped input/output (MMIO). Hence, basic communication is not very complex since the TPM is addressable as if it were RAM memory.

While on Linux and Windows there are libraries available for interacting with the TPM (e.g., TrouSerS [**TR**]), no such effort exists for Mini-OS. Our TC module fills this gap; it adds support for many of the TPM commands based on the TPM specifications [**TCG03**]. This includes adding support the typical TPM overhead such as OSAP and OIAP sessions.

CaaS control module. While the TC module implements the TPM commands, the CaaS module (cf. Fig. B.2) handles the steering of DomT and uses the TPM commands exported by the TC module.

After booting DomT, the CaaS module reads in the persistent TPM state from the Dom0 filesystem using simple POSIX I/O calls, which are backed transparently by the VFS module (cf. Fig. 6.4). This state is untrusted, but that is not a vulnerability since it only affects availability. The state comprises

- the TPM public key part of the certified binding key;
- the private key part of this key (encrypted by the TPM);
- miscellaneous items such as the current value of the TPM counter.

We did not complete the counters support; hence, only the first two items are implemented.

B.8. Passthrough encryption

We utilize vTPMs so that the cloud consumer's high value keys can be safeguarded against the external adversary (req. R1). Instead of letting these vTPMs live in in Dom0, we choose to give each vTPM its own domain, a DomC domain.

We considered two possible approaches for protecting the confidentiality and integrity of the consumer's I/O data streams (as opposed to RAM) from the adversary.

1. *Exert control over the entire workflow.* This would mean to disaggregate the Dom0 such that the administrator cannot interact or interfere with the components that perform disk I/O. Since Dom0 is so heavily involved in providing the back-end of block devices, it is challenging to disaggregate such functionality away from the Dom0, although Colp et al. achieve some success here [Col+11]. The situation becomes increasingly difficult when considering network mounted storage as is common in clouds nowadays; these all would need to be attested with TC techniques to assure that there also the disk data is not exposed to the cloud administrators.

2. *Encrypting when leaving the TCB.* One way would be to have the consumer run a FDE scheme in his or her VM. This key needs to stem from somewhere; possibly from a vTPM. But *only* if such a vTPM is rooted in the hardware TPM, with the vTPM-NV being safeguarded, is such an approach viable.

We remark that the most commonly used disk encryption schemes such as Truecrypt [TC] or Bitlocker [Fer06] lack the support for integrity protection that is necessary to protect against tampering and rollback. While this is fine for the typical scenario of protecting against laptop theft, these schemes are inadequate for protecting integrity in the cloud. While there exist integrity preserving filesystems (e.g., ZFS), that could perhaps be used ontop on one of the aforementioned FDE schemes, the DomC design relieves consumers from the burden of having to worry about setting up and maintaining such a filesystem; in the DomC design they can use whatever they are used to.

The passthrough encryption is achieved by DomC offering a block device to the DomU which looks indistinguishable from a block device offered by Dom0. An immediate advantage of this approach is that no modifications in the cloud consumer's VM are needed; it will work with a vanilla PVOps Linux kernel.[5]

Our kernel of choice for DomC is (just like for DomT) a Mini-OS kernel because of its tiny TCB. Our passthrough block device for Mini-OS relies on three components working together:

1. first, we need to connect a block front-end device to the (encrypted) disk offered by Dom0;

2. second, we need to apply encryption inside Mini-OS for any data that we wish to forward from one side to the other;

3. third, we need a block back-end driver for Mini-OS to talk to DomU.

[5] PVOps refers to paravirtual operations being natively embedded in the kernel source as opposed to distributed as a separate patchset. Linux versions of 3.2 and higher are PVOps enabled.

There exist block back-end implementations for Linux, various BSDs and OpenSolaris. However, none exists for Mini-OS, probably due to the fact that there are no drivers for Mini-OS which talk directly to hardware. Nevertheless, a front-end block device driver is available for Mini-OS, which meant we only needed to write one of the two split-drivers, namely the block back-end.

B.8.1. Cryptography

In its primitive form, a VM at rest can be represented by a virtual disk, i.e. as a large byte array that represents a filesystem. As the user wishes to protect the confidentiality and integrity of this VM (req. R2), this virtual disk is encrypted. Clearly, the naive approach of encrypting the complete disk using cipher-block chaining (CBC) is unsuitable; for decrypting a file at the end of the disk, the entire disk would need to be traversed and decrypted. Therefore, we encrypt the disk using an established FDE scheme.

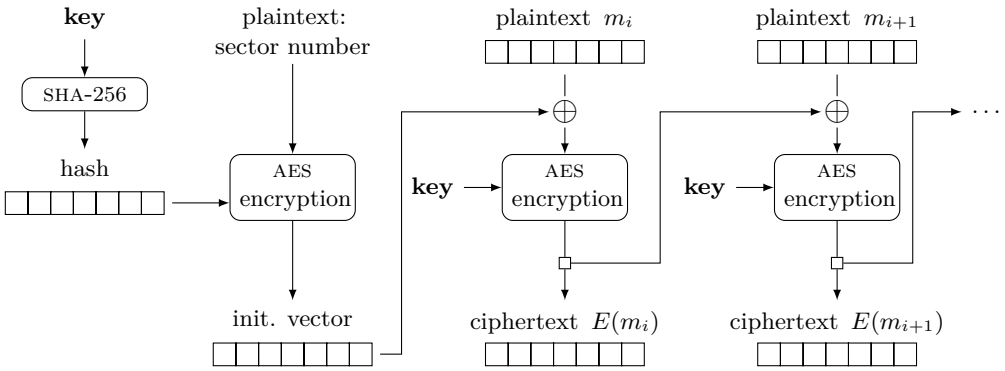
ESSIV. The scheme we chose for our passthrough encryption is *encrypted salt-sector initialization vector* (ESSIV) [**Fru05**], although any FDE scheme can be used in the DomC. This scheme is relatively simple (i.e., there exist more sophisticated schemes with better performance), but due to its simplicity it is a good candidate for our implementation. Nonetheless, we were not aware of vulnerabilities with respect to its confidentiality guarantees at the moment of writing this thesis.[6] But even if there were vulnerabilities, the FDE scheme used in CaaS can be upgraded without loss of generality. Moreover, this scheme is simple to implement and is also the default scheme used by the Linux dm-crypt kernel driver [**Sau**].

In Fig. B.9 the encrypt and decryption operations under ESSIV with CBC are exhibited. This scheme is the application of CBC at sector level combined with a nonstandard way of calculating the initialization vector (IV). The IV for sector i is calculated as [**Fru05**]:

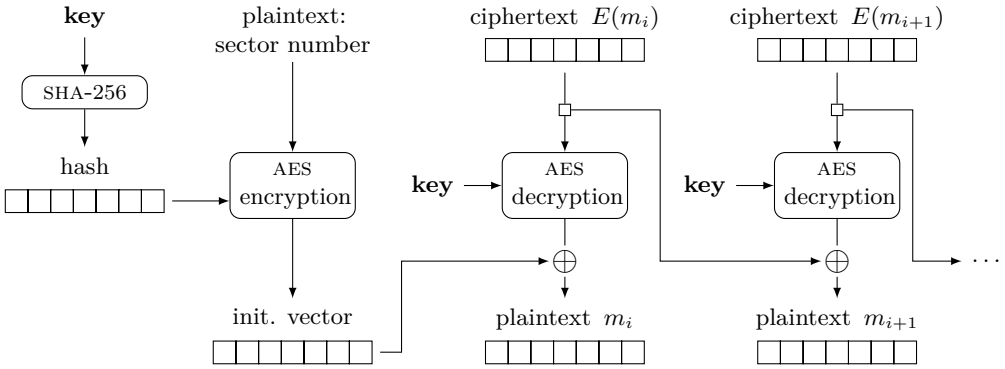
$$IV(i) = \mathcal{E}_s(i), \qquad \text{where } \; s = \text{hash}(k)$$

The hash function used in ESSIV must output a digest length which is suitable as key length for the block cipher. In our implementation, we use SHA-256 together with AES.

[6] This is true for confidentiality. However, with respect to integrity, the situation is quite different. In fact, as seen in in Fig. B.9b, due to the CBC a single bit flip in the ciphertext block $E(m_i)$ scrambles only m_i and an attacker-chosen bit in m_{i+1}. This is a undesired property, and therefore integrity protection measures are needed.



(a) Encryption operation.



(b) Decryption operation.

Figure B.9: Disk encryption with AES and CBC using ESSIV for IVs.

B.8.2. DomC back-end driver

In the Mini-OS overview figure (Fig. B.2), it is visible that the DomC incorporates a disk back-end driver. This back-end driver that we wrote for Mini-OS follows the same paradigm as its Linux counterpart. How this block back-end driver is set-up in relation to the other components is shown in Fig. B.10. This figure presents a simple summary of how the DomC services are configured after the creation of the cloud consumer's VM.

In Fig. B.11, the structure of a request on the block interface between the front-end and back-end is exhibited. All of the Xen virtual devices work using a *ring buffer* with a producer and consumer that can operate simultaneously without locking. Requests are placed on the ring buffer by the requester (i.e., DomU) and removed by the responder (i.e., DomC, or Dom0 in case of normal devices). A block interface request has several attributes describing the kind of request, as shown in Fig. B.11. In particular, each request contains (aside from the sector number) a list of *segments*. Each segment points to a page frame and indicates the first and last sector in the page frame to transfer. This does not need to be contiguous (e.g., as shown on the right hand side in Fig. B.11) but can be spread out over many page frames. The reason for this fragmentation is that the front-end kernel might split the I/O buffer over several noncontiguous pages due to its paging mechanism, in what it perceives as its physical memory. Our block back-end has to take this fragmentation into account when mapping and reading data requests. With a typical page size of 4096 bytes, up to 8 sectors can be read or written in a segment. Our block-backend can handle multiple disk back-ends in a single session, within the limits set by Mini-OS (i.e., no pre-emptive threads, no SMP).

B.8.3. Applying passthrough encryption

The basic application of our cryptographic module in DomC is simple — we apply the operations in-between the copying between the front and back ring buffers. The key used in these operations is supplied by the cloud consumer. As we have listed in Fig. B.4 (p. 93), there are a few data items which need to be provided to DomC. These data items are coupled in what we call CSinfo. Sharing this info through the Xenstore would be rather unwise, since it is ill-suited as a secure transport medium. Instead, during the domain building, DomT injects this CSinfo into DomC's memory range. During the Mini-OS booting process in in DomC, the CSinfo is read and processed.

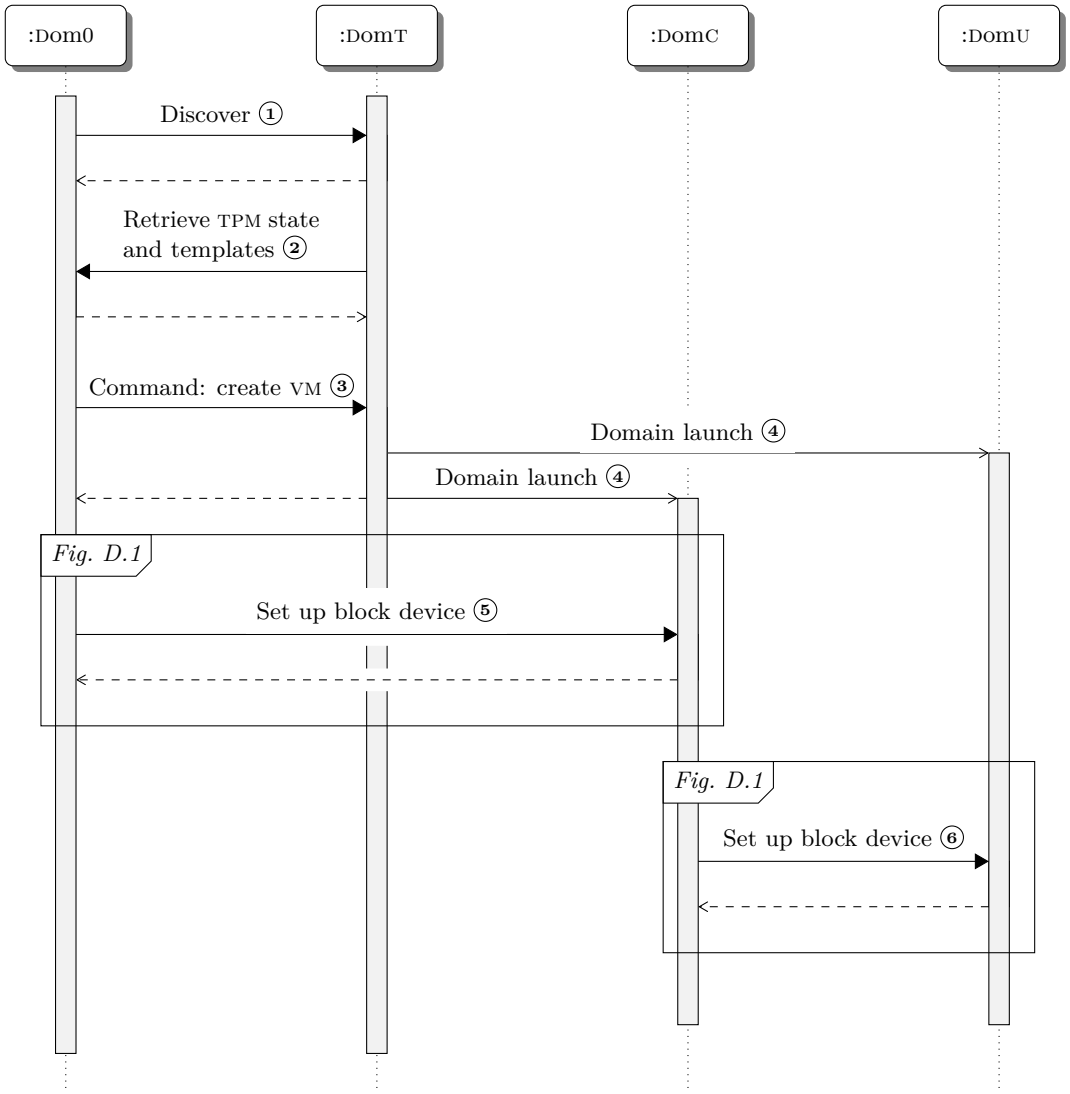


Figure B.10: Overview of the initialization of the DomC devices. ① After booting by the hypervisor, the Dom0 and DomT must find each other after their respective initializations (appendix B.6). ② The DomT fetches the TPM state and templates. ③ Whenever a request is placed in Dom0 for the creation of a VM, this request is then forwarded to DomT. ④ Both domains are launched together, and the key is injected into the DomC. ⑤ First the DomC attaches to the block device offered by Dom0. ⑥ Then DomC offers a block device to the DomU, who does not need to be modified for this.
The last two steps are shown in detail in appendix D.

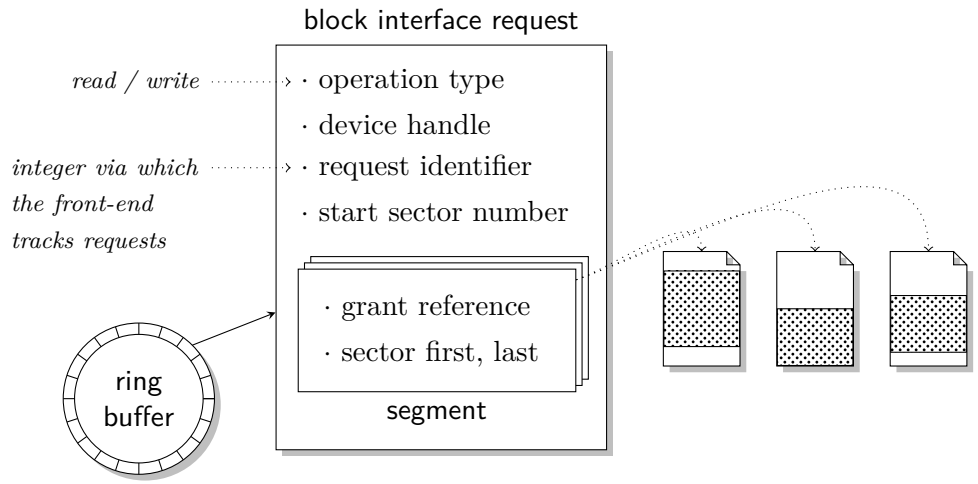


Figure B.11: Structure of block interface requests which are placed in the ring buffer.

B.9. vTPMs and PV-TGRUB

In the Xen introduction we briefly mentioned a Xen-specific bootloader called PV-GRUB (cf. subsection 2.2.2, p. 14). In the architecture, we mentioned that our design expands on this tool by adding the concept of extending measurements of the loaded kernel to the vTPM. We looked at what solutions already exist for trusted booting. In a non-virtualized environment, two bootloader designs are known which make use of the TPM.

- TGRUB [**TG**], which installs a SRTM during boot; and
- OSLO [**Kau07**] for AMD and TBoot [**TB**] for Intel, which install a DRTM during boot.

For a proper (that is, using a root of trust) vTPM operation, it is necessary that the first loaded piece of software (the kernel) is always measured. The normal PV-GRUB has no such functionality, therefore, we propose to extend the PV-GRUB with the ability to *extend* the first chain-loaded piece of software to the vTPM, in an approach similar to the trusted boot loaders listed above.

This approach using PV-TGRUB is architecturally cleaner than the alternative, which is to have DomT perform the initial measurement and store the value in the vTPM. Such an approach would be a complex task, because normally the domain builder does not actually look into the virtual devices which are attached to a new domain.

Implementation status. A lack of time meant PV-TGRUB as well as the vTPM did not see it to the final development phase.

- Regarding the vTPM, there are improvements in the pipeline for the upcoming Xen release which will compile a vTPM and a vTPM manager on Mini-OS. Using these improvements it will become possible to include a vTPM in the Mini-OS DomC, and the vTPM manager in the DomT. Our CSinfo implementation already reserves space for this extension.
- While we did do successful trial runs with booting a PV-GRUB, we were still in the phase of making PV-GRUB work, and the PV-TGRUB extensions were an option yet.

B.10. Services

In section 5.3 we discussed in the light of key provisioning various parties involved in the deployment of VMs, such as the cloud consumer and a CV. We implemented only the cloud consumer tools.

B.10.1. User VM deployment tool

We created a Python based tool which assists cloud consumers in deployment of their VMs to a CaaS system. This tool, `deployer.py`, currently has one mode of operation, which takes the following parameters.

In parameters:
- path to public key file of the target TPM
- path to the VM disk image

Out parameters:
- path where to write the encrypted VM (i.e., EVM structure)
- path where to write the VMCB

The tool operates in two stages. In the first stage, it creates a random symmetric key and applies the FDE scheme onto the VM disk image (resulting in the EVM). In the second stage, it encrypts this symmetric key with the public key of the TPM. This ciphertext comprises the VMCB.[7]

The `deployer.py` tool is written in Python so that it is easily extensible. In the underlying engine code, we wrap C calls to the TSS service provider interface (TSPI) library so that we can easily work with the TPM public key and certificate.

[7] The other fields of the VMCB structure are not implemented in the proof of concept.

This is fine for the proof of concept, although in the future the TSPI dependency might be augmented with a native Python version since the operations are essentially simply RSA operations. We note that the TSPI library does not need a TPM; the cloud consumer's TPM is not used for creating an EVM or VMCB.

C. Details on the use of TBoot

Due to our lab equipment comprising only an Intel machine, we restrict our discussion to TXT, although all concepts apply to AMD-V as well. These details originate from the Intel *measured launch environment* (MLE) development guide [**TXT**] and the TBoot documentation [**TB**].

Intel TXT works by starting a measured launch to guarantee a secure boot and revolves around a new CPU instruction: GETSEC[SENTER]. This command works in two stages; in the first stage it takes an *authenticated code* module, also referred to as SINIT blob, while in the second stage it takes a MLE blob. An implementation of an MLE is for example Trusted Boot (TBoot) or Flicker.[1] We briefly review these two components.

SINIT. As soon as SENTER is called, the processor enters a secure environment (i.e., halt CPUs, disable interrupts and protect memory), resets the dynamic PCRs (PCR 17 to 23) and executes the SINIT. This SINIT blob is signed by Intel and this signature is verified by the SENTER instruction. The job of the SINIT module is to verify a proper chipset configuration. Trusting the SINIT module (based on Intel's endorsement) is essential and forms the core root of trust that replaces the trust normally placed into the BIOS.

Furthermore, the SINIT reads from the TPM non-volatile memory (TPM-NV) the launch control policy (LCP).[2] The LCP specifies, using an approved hash, which MLE is allowed to be ran. (Optionally, it can contain more conditions such as a BIOS hash.)

Before launching the MLE, PCR-17 will be extended with several chipset values, and most importantly, a hash of the LCP policy.

Measured launch environment. The MLE that we will discuss here is TBoot. TBoot is an open source, pre-kernel module that uses Intel TXT to perform a

[1] Flicker allows the execution of small pieces of code in complete isolation during runtime. See McCune et al. [Mc+08].

[2] Writing to the TPM-NV requires to be authenticated as owner of the TPM, hence, this data can be considered as coming from a trusted source.

```
kernel /boot/tboot.gz ⟨Tboot arguments⟩
# Xen hypervisor and DomT TCB modules:
module /boot/xen.gz ⟨Xen arguments⟩
module /boot/domt
# Dom0 kernel plus initial ramdisk:
module /boot/vmlinuz-linux ⟨Linux arguments⟩
module /boot/initramfs-linux.img
# Modules used exclusively by Tboot:
module /boot/pol/list.data
module /boot/sinit.bin
```

Listing C.1: GRUB with TBoot example.

measured and verified launch of an OS kernel [TB]. Because the Xen hypervisor behaves as a multiboot[3] compliant kernel, it works well with TBoot.

TBoot examines the given multiboot modules and verifies each one against a policy stored in the TPM-NV by the owner (i.e., in similar fashion to the LCP). The flexibility of these verified launch (VL) policies is quite high. One can give a decision type (halt or continue on PCR mismatch) and support a variable list of approved modules. One can also specify to which PCR the specific measured module should be extended (for modules after the first one).

TBoot will extend the hash of TBoot to PCR-18, as well as the hash of the first module which will be booted (e.g. Xen or Linux kernel). Finally, it will chainload the first module that was passed to it.

Application in CaaS

We will clarify the previous text with a description of how TBoot is used in our situation. In our GRUB configuration file we boot the modules in the fashion shown in listing C.1. Note that TBoot is the first module to be launched; it will take care of starting the SENTER with the SINIT module, after which the SINIT will pass control to TBoot in the manner described above.

In listing C.1 we load seven modules in total. The last two modules will be used by TBoot itself, which leaves four modules to be handled with a verified launch policy: Xen, DomT, Dom0 Linux and the initial ramdisk used by the Dom0 Linux kernel.

In order for TBoot to verify these modules, we write these policies to the

[3]The multiboot specification by the Free Software Foundation is a specification which is used, amongst others, by Linux, Xen and TBoot. An important part of the specification is the ability to pass a variable list of modules to a kernel.

```
# Creation of a policy which continues on mismatch:
tb_polgen --create --type continue cont.pol
tb_polgen --add --num 0 --pcr none --hash image \
        --cmdline ⟨Xen arguments⟩ --image xen.gz cont.pol
tb_polgen --add --num 1 --pcr 19 --hash image \
        --image domt.gz cont.pol
      ⋮                  ⋮
# Write policy to TPM:
lcp_writepol -i 0x20000001 -f cont.pol -p ⟨TPM password⟩
```

Listing C.2: Writing verified launch policies to TPM-NV.

TPM-NV beforehand. In listing C.2 we show how we generate a policy file which we write to the TPM-NV.

PCR. One of the benefits of our CaaS design is that Dom0 is no longer part of the TCB. However, TBoot expects to find a policy for *all* given modules. Therefore, we need to find a way to exclude Dom0 from tainting our measurements.

The first component of our solution is to set a 'continue' policy (as shown in listing C.2), which will continue execution on a verification error (i.e., if the Dom0 kernel is upgraded which results in an updated hash). This does not affect the security of our solution since we rely on a PCR-bound binding key, and not on a verified launch policy.

The second component involves using the VL policy to write the Dom0 Linux and ramdisk to a separate PCR which we will not include in the PCRs used for creating a certified binding key. Our PCRs will look like this:

- PCR-17: hash chipset firmware and of LCP
- PCR-18: hash of MLE (TBoot)
- PCR-19: hash of DomT
- PCR-20: hash of Dom0 kernel and ramdisk *(ignored for the binding key)*

Using this approach, we guarantee that a trusted TCB (hypervisor plus DomT) has been booted. We remark, as highlighted earlier, that this authenticated boot is not intended to form a protection against runtime attacks (e.g., buffer overflows). The use of TBoot creates a DRTM, but the execution of secure components does not live in a protected environment as is the case with for instance Flicker [**Mc+08**].

D. Xenstore device handshaking

All of the Xen virtual devices work using a ring buffer with a producer and consumer which is a straightforward building block. The establishment of such devices happens via the Xenstore. In Fig. D.1 we show the sequence diagram of handshaking a device between the front-end and back-end.

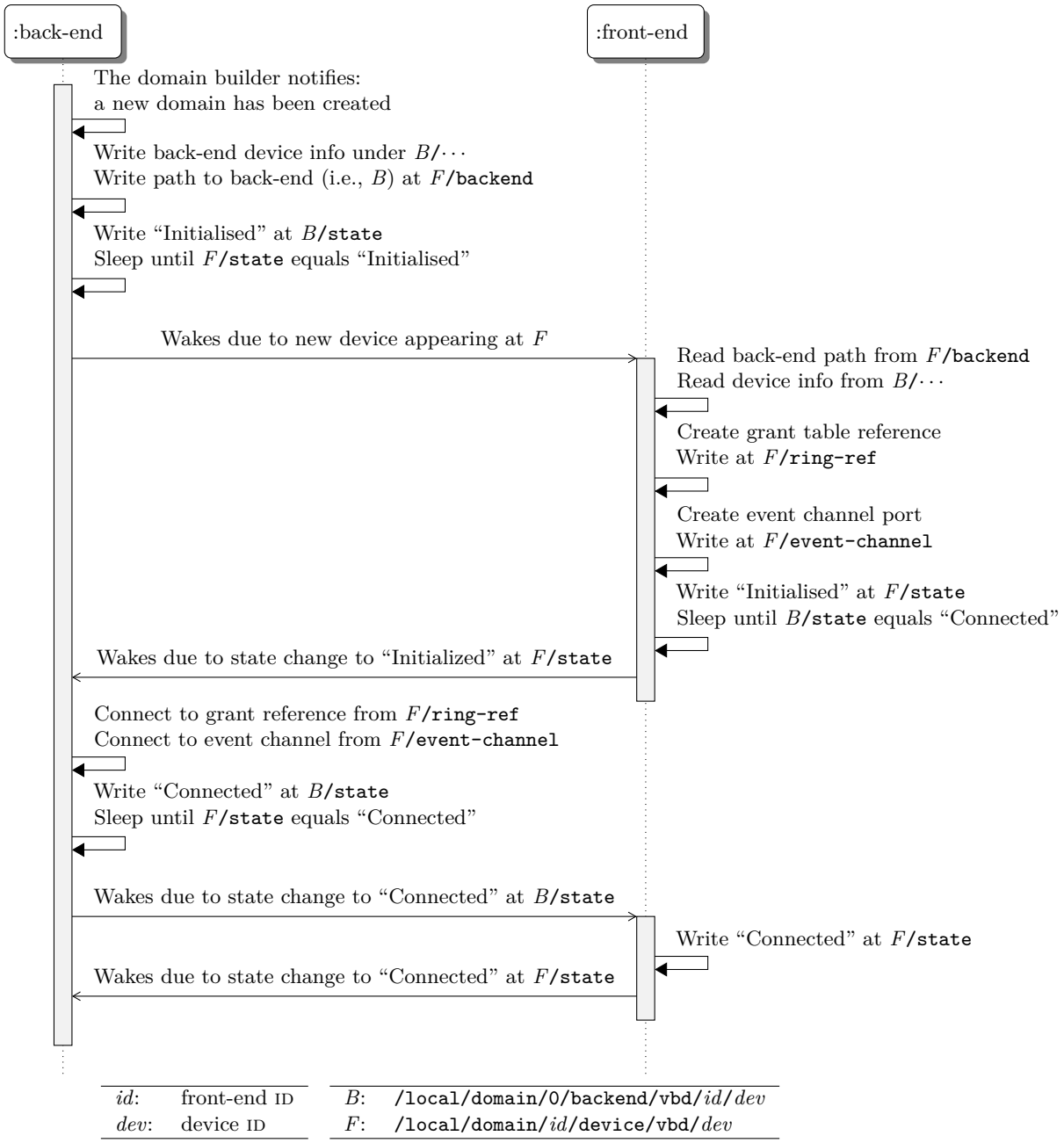


Figure D.1: The Xenstore handshaking between the front-end and back-end for establishing a common ring buffer. For simplicity, the back-end is shown as being domain 0.

E. Xenstore security

Because the CSinfo (see appendix B.2) includes the domain IDs of the DomC and the DomU, the DomC will not offer any of its services (e.g., the decrypted virtual disk) to any other domain than its designated DomU. Therefore, it is not possible for a malicious DomU — such as one under control by the malicious insider — to attach to this DomC and fake to be the corresponding DomU.

However, this is only one side of story. The DomU has no guaranteed way of knowing which domain is the DomC, hence, a rogue domain could pretend to be a DomC and offer a block device to DomU.[1] While this does not compromise the confidentiality of the secrets, it could be considered as a loss of integrity (requirements R1 and R2). For instance, if the VM involved is a service that makes access control decisions based on secrets offered by the (rogue) DomC, the validity of these decisions could be undermined — with dire consequences. There are two straightforward solutions. An easy solution to this problem could be to inject this information at build time in DomU, too. However, this contradicts our design goal that no modifications to DomU are needed. It is more appealing to lock down the Xenstore such that Dom0 cannot tamper with it any longer.

Trusted Xenstore. A locked-down and trusted Xenstore would mean that only domains designated by DomT can offer back-end devices to other domains.

In Xoar, Colp et al. created a new domain which holds the Xenstore exclusively, out of reach of Dom0 [Col+11].[2] Our CaaS architecture can easily be adapted to work with such a system: our CaaS security module would shield this Xenstore domain, while the DomT would be designated as the only domain which has administrative privileges over the contents of the Xenstore. This results in the inclusion of the Xenstore helper domain into the TCB. To prevent tampering, DomT has to extend the hash of this Xenstore helper domain to the TPM (in the step depicted in Fig. 6.3).

[1] By default, the Xenstore will not allow this, hence, the only adversary who can do this would be the cloud administrator.

[2] There a changes in the pipeline for inclusion of a Mini-OS based Xenstore domain into the upcoming Xen release.

F. Xen hypercalls

In Table F.1 we list the hypercalls for the Xen version we examined (version 4.1.2). For each hypercall we have three columns with properties. In the first column, **S**elf, we mark if the hypercall can be invoked with the calling domain as affected entity. In the second column, **O**ther, we mark if the hypercall can be called with another domain as subject. For both these columns, the symbol $\times$ indicates that this property holds, while the symbol $\otimes$ indicates that, additionally, a privileged status is mandatory.

In the third column we indicate how we make the preconditions for the hypercall stronger. A $\mathcal{Z}$ means that that only the Domain **Z**ero (Dom0) can execute the hypercall, while a $\mathcal{T}$ indicates that only DomT can execute the hypercall. These restrictions affect only the privileged hypercall versions and come *ontop* of existing requirements. Sometimes, we do not wish to restrict the privileged command because we need it in both domains. This is denoted by the $\cdot$ symbol. Furthermore, we abolish certain privileged hypercalls, indicated with the symbol $\lightning$. These are generally debugging hypercalls which belong neither in the category of Dom0 nor DomT in a production environment. For each hypercall the superscript denotes the relevant XSM hook and in which source file the hypercall is implemented.[1]

In general, the unprivileged hypercalls map to functionality which the guest kernel needs to do its everyday tasks but for which it in a PV environment lacks access. As an archetypal example, the hypercall #2 in Table F.1 requests the hypervisor to update page tables on behalf of the calling domain.

The privileged hypercalls deal with all the other cases, such as providing emulation for other domains and managing shared resources.

The table was compiled based on analyzing the Xen source code. We are not aware of any such hypercall table for Xen being available; the nearest approximation (that is, the list of hypercalls from the Xen documentation) only covers a very small set.

[1] These can be looked up in Table F.2 (p. 132) and tables G.1 and G.2 (p. 133) from appendix G.

Table F.1: Overview of Xen hypercalls. Hypercalls are defined at various granularities in Xen; we list the subcalls of each hypercall if these are sufficiently distinct. Most of the nomenclature used in the descriptions is detailed in the list of acronyms.

	Hypercall	S	O	R	Description
			Primary hypercalls		
1	set_trap_table[L]	×			Installs the virtual interrupt descriptor table for the domain. Xen will inform the domain via callbacks.
2	mmu_update[G59,58]	×[†]	⊗	$\mathcal{T}$	Update the specified domain's page tables or the machine to physical (M2P) table. This function is efficient on large batches of requests packed together in one hypercall invocation.
3	update_va_mapping[G80]	×			This hypercall is designed for a single page table update when the overhead from hypercall #2 for operating on batches of requests is not needed.
4	update_va_mapping_otherdomain[G80]		⊗	$\mathcal{T}$	Identical to hypercall #3, but applies only on other domains.
5	set_gdt[G]	×			Installs a (new) global descriptor table for a VCPU of the domain.
6	stack_switch[C]	×			Switches the kernel stack for the domain.
7	set_callbacks[D]	×			Register the entry functions via which the hypervisor informs the domain of virtual interrupts.
8	fpu_taskswitch[L]	×			An optimization feature in which the hypervisor sets the task switch (TS) bit in control register 0 for a VCPU for the domain.[‡]
9	set_debugreg[L]	×			Sets the debug registers for a VCPU of the domain.
10	get_debugreg[L]	×			Returns the debug registers for a VCPU of the domain.
11	update_descriptor[G]	×			Set the domain's thread local storage descriptors.
12	multicall[U]	×			Execute multiple hypercalls in one go.
13	set_timer_op[V]	×			Sets a one-shot timer for the domain's current VCPU.
14	xen_version[R]	×			Returns various Xen version information (major, minor, compiler info, etc.) on demand.
15	console_io[R1]	⊗		·	Read and write to the Xen emergency console.
16	vm_assist[N]	×			Enable various VM assist functionality to lessen the burden for PV porting efforts.

[†]This is an example of a hypercall with both a privileged and unprivileged version. A domain requires no privileges to update its page tables with machine pages it owns. However, privileges are required when mapping pages which are not owned or when changing another domain's page tables.

[‡]Normally, after a context switch the OS needs to restore the floating point registers for the active thread. This is a waste of cycles if the floating point operations are never used. The solution is the TS flag. Once the TS flag is set, the CPU will trap as soon as a floating point operation is invoked, allowing the kernel to interpose and set the floating point registers.

	Hypercall	S	O	R	Description
17	iret[D]	×			Asks Xen to perform an *interrupt return* (IRET) instruction on behalf of the domain. The IRET instruction is used to return from the kernel to usermode.
18	set_segment_base[M]	×			In the x86-64 architecture, segment addressing has been has been largely removed; however the FS and GS registers have been retained because operating systems have come to rely on them as extra base pointers.
19	mmuext_op[G16]	×			Allows a domain to request various MMU operations not covered by hypercall #2, such as pinning pages, TLB flush, local descriptor table update, and more.
20	xsm_op[Y20]	×			Allows a domain to communicate with the loaded XSM module (cf. appendix B.3), if any. No privileges are required, XSM handles the access control itself.
21	nmi_op[R]	×			Register a non-maskable interrupt handler. NMIs generally indicates nonrecoverable hardware errors.
22	callback_op[D]	×			Register a callback with a wider range of types than either #7 or #21. It can also unregister callbacks.
23	vcpu_op[N]	×			Allows the domain to perform several actions on a VCPU, for instance bringing up or down a VCPU or setting a timer. Analogous to the operations possible on a CPU core without virtualization. (Spread over 14 omitted subcalls.)
24	hvm_op[F11,13,12,14]	×	⊗	$\mathcal{T}$	Various HVM related commands related to injecting traps, flushing the TLB, etc. (Spread over 16 omitted subcalls.)
25	tmem_op	×			Xen transcendent memory interface, allowing domain kernels to share memory caches. (Spread over 13 omitted subcalls.)
26	kexec_op[S51]	⊗		$\lightning$	Replace the hypervisor kernel with any other kexec supported kernel such as Linux or Xen *without rebooting*. It can also start an emergency kernel in case of hypervisor or Dom0 crash. (Spread over 4 omitted subcalls.)
27	xenoprof_op[X65]	⊗	⊗	$\lightning$	Profile domains with a module in the Xen hypervisor. (Spread over 18 omitted subcalls.)
	Memory reservation and information (HYPERVISOR_memory_op)				
28.1	increase_reservation[T54]	×	⊗	$\mathcal{Z}$	Increases specified domain's current memory reservation within the maximum allowance.
28.2	decrease_reservation[T54]	×	⊗	$\mathcal{Z}$	Decreases specified current domain's memory reservation.
28.3	populate_physmap[T54]	×	⊗	$\mathcal{T}$	Populates the memory of specified domain with pages.
28.4	exchange[T]	×	⊗	$\mathcal{T}$	Exchange memory reservations in atomic fashion. For instance, this is used to accumulate fragments into one contiguous block. All the memory pages must be owned by the same domain.
28.5	maximum_ram_page[T]	×			Returns the highest machine frame number of mapped RAM in this system.
28.6	current_reservation[T55]	×	⊗	$\mathcal{Z}$	Returns the *current* memory reservation for specified domain.
28.7	maximum_reservation[T55]	×	⊗	$\mathcal{Z}$	Returns the *maximum* memory reservation for specified domain.
28.8	machphys_mfn_list[B]	×			Returns the list of MFNs comprising the M2P table.

	Hypercall	S	O	R	Description
28.9	maximum_gpfn[A55]	×	⊗	$\mathcal{T}$	Returns the highest guest page frame number (GPFN) in use by specified domain.
28.10	machphys_mapping[A]	×			Returns the virtual address of the M2P table.
28.11	add_to_physmap[A23]	×	⊗	$\mathcal{T}$	Sets the GPFN at which a certain page appears to the specified domain.
28.12	memory_map[A]	×			Returns the domain's *physical* memory map, also known as the E820 map, as it was when the domain was started.
28.13	machine_memory_map[A15]	⊗		·	Returns the *machine* memory map. Dom0 needs this information to configure its hardware devices. The table returned is the E820 map offered by the BIOS and gives no information about the current status of the RAM.
28.14	set_memory_map[A33]	×	⊗	$\mathcal{T}$	Set the physical memory map of the specified domain. This map is consulted at domain start. It is much more simple than the one offered by the BIOS because domains see only abstracted memory.
28.15	set_pod_target[A]		⊗	$\mathcal{T}$	Sets the populate-on-demand (PoD) target (only makes sense for HVM domains).
28.16	get_pod_target[A]		⊗	$\mathcal{T}$	Returns the PoD target.
28.17	get_sharing_freed_pages[A]	×			Part of experimental code for page sharing between VMs; this hypercall returns the number of MFNs saved. Should be considered part of memory sharing hypercalls listed below.
	Grant table operations (HYPERVISOR_grant_table_op)				
29.1	map_grant_ref[Q8]	×			Map a list of grant references into the domain's page tables.
29.2	unmap_grant_ref[Q9]	×			Release a list of mapped grant references.
29.3	setup_table[Q48]	×	⊗	$\mathcal{T}$	Initialize the grant table for a domain.
29.4	transfer[Q10]	×			Instead of merely sharing a page, this hypercall irrevocably transfers control.
29.5	copy[Q7]	×			Asks the hypervisor to do memory copying between (shared) pages, obviating the need for a guest to map grant references and perform the memory copies himself. The hypervisor can do this more efficient because it already has all the memory pages of the machine in its page tables.
29.6	query_size[Q47]	×	⊗	$\mathcal{T}$	Returns the current and maximum sizes of a domain's grant table.
29.7	unmap_and_replace[Q]	×			Unmaps a grant reference, but atomically replaces the page table entry to point to a new machine page.
29.8	set_version[Q]	×			Before any grants are activated, it can be chosen by the domain whether it wants version 2 or version 1 (backwards compatibility) of grant tables.
29.9	get_status_frames[Q]	×	⊗	$\mathcal{T}$	Gets the list of frames used to store grant status for a specified domain; in version 2 of the grant tables this can speed up synchronization.
29.10	get_version[Q]	×	⊗	$\mathcal{T}$	Retrieves the version (either 1 or 2) used for the grant table of a specified domain.
	Scheduler operations (HYPERVISOR_sched_op)				

	Hypercall	S	O	R	Description
30.1	yield[V]	×			Yield the vCPU for others to use.
30.2	block[V]	×			Block the vCPU until an event is received for processing.
30.3	shutdown[V]	×			Halt the entire domain.
30.4	poll[V]	×			Poll a set of event channel ports.
30.5	remote_shutdown[V70]		⊗	$\mathcal{Z}$	Shutdown another domain.
30.6	shutdown_code[V]	×			Set the shutdown code which will reported to the tools, e.g., for telling the tools this domain crashed or whether it exited normally.
30.7	watchdog[V]	×			Start, poke or destroy a domain watchdog timer.
	Event channel operations (HYPERVISOR_evtchn_op)				
31.1	alloc_unbound[P25,37]	×	⊗	$\mathcal{T}$	Allocates and returns an unused port number for communication with a specified domain.
31.2	bind_interdomain[P4]	×			Bind a local port to a specified remote port and domain number.
31.3	bind_virq[P]	×			Bind a local event channel to a virtual interrupt request on a specified vCPU.
31.4	bind_pirq[P]	×§			Bind a local event channel to a *machine* IRQ.
31.5	bind_ipi[P]	×			Bind a local event channel to receive inter-processor interrupt events.
31.6	close[P3]	×			Close an event channel.
31.7	send[P5]	×			Send an event over an event channel.
31.8	status[P36]	×	⊗	$\mathcal{T}$	Get the status of an event channel connected at a specified domain and port.
31.9	bind_vcpu[P]	×			Change the vCPU at which event for a specified port are delivered.
31.10	unmask[P]	×			Unmask the local event channel (and deliver any pending events).
31.11	reset[P35]	×	⊗	$\mathcal{T}$	Close all event channels associated with a specified domain.
	Platform hypercalls¶ (HYPERVISOR_platform_op)				
32.1	settime[I84]	⊗		$\mathcal{Z}$	Set the machine wall clock time.
32.2	add_memtype[I56]	⊗		$\mathcal{Z}$	Configures ranges of RAM to be enabled for processor caching using the CPU *memory type range registers* (MTRRs).
32.3	del_memtype[I56]	⊗		$\mathcal{Z}$	Reverses the MTRR configurations.
32.4	read_memtype[I56]	⊗		$\mathcal{Z}$	Reads the current type of an MTRR.
32.5	microcode_update[I57]	⊗		$\lightning$	Allows the CPU microcode to be updated.
32.6	platform_quirk[I63]	⊗		$\mathcal{Z}$	On certain hardware it might be necessary to disable IRQ balancing or a specific *I/O advanced programmable interrupt controller* (IOAPIC) register.

§The operation does not require privileged status; instead, it mandates that the domain possesses the *irq* capability. This is normally only in possession of Dom0.

¶There are various machine settings for which the hypervisor cannot or does not want to have the software logic. For instance, time synchronization would add complexity to the hypervisor which does not suit well with the notion of a bare-metal hypervisor.

	Hypercall	S	O	R	Description
32.7	firmware_info[I39]	⊗		$\mathcal{Z}$	Returns the BIOS enhanced disk driver info, which is a uniform way for the BIOS to tell the OS what disk to boot from, among other things.
32.8	enter_acpi_sleep[I21]	⊗		$\mathcal{Z}$	Puts the machine into advanced Configuration and power interface sleep state.
32.9	change_freq[I30]	⊗		$\mathcal{Z}$	Adjusts a CPU core frequency.
32.10	getidletime[I43]	⊗		$\mathcal{Z}$	Returns a CPU core idle time.
32.11	set_processor_pminfo[I]	⊗		$\mathcal{Z}$	Adjust a CPU core states, e.g., power saving and throttling.
32.12	get_cpuinfo[I]	⊗		$\mathcal{Z}$	Returns info about a CPU core, such as its online status.
32.13	cpu_online[I]	⊗		$\mathcal{Z}$	Brings a CPU core up.
32.14	cpu_offline[I]	⊗		$\mathcal{Z}$	Brings a CPU core down.
32.15	cpu_hotadd[I]	⊗		$\mathcal{Z}$	Add a CPU at runtime.
32.16	mem_hotadd[I]	⊗		$\mathcal{Z}$	Add RAM at runtime.
	System control (HYPERVISOR_sysctl_op)				
33.1	readconsole[W66]	⊗		$\mathcal{Z}$	Reads content from hypervisor console buffer.
33.2	tbuf_op[W78]	⊗		$\lightning$	Trace buffer operations allows to debug the performance of domains.
33.3	sched_id[W69]	⊗		$\mathcal{Z}$	Get the ID of the current scheduler.
33.4	getdomaininfolist[W42]	⊗		·	Gets a *list* of domaininfo structures (cf. 35.12).
33.5	perfc_op[W60]	⊗		$\lightning$	Provides access to hardware performance counters.
33.6	lockprof_op[W]	⊗		$\lightning$	Allows for profiling of locks in the hypervisor.
33.7	debug_keys[W31]	⊗		$\lightning$	Inject debug keys into Xen (as if entered over serial line).
33.8	getcpuinfo[W41]	⊗		$\mathcal{Z}$	Get physical CPU information.
33.9	availheap[W28]	⊗		$\mathcal{T}$	Returns number of free domain heap pages.
33.10	get_pmstat[W40]	⊗		$\mathcal{Z}$	Retrieve power management statistics.
33.11	pm_op[W64]	⊗		$\mathcal{Z}$	Configure power management.
33.12	page_offline_op[W]	⊗		$\mathcal{Z}$	Provides ability to hotplug memory at logical level (take page offline).
33.13	cpupool_op[W]	⊗		$\mathcal{Z}$	CPU pool operations.
33.14	scheduler_op[W]	⊗		$\mathcal{Z}$	Adjust scheduler for all CPU pools.
33.15	physinfo[K61]	⊗		$\mathcal{Z}$	Get physical information about the host machine.

	Hypercall	S	O	R	Description
33.16	topologyinfo[K]	⊗		$\mathcal{Z}$	Retrieve CPU core/socket/node identifiers.
33.17	numainfo[K]	⊗		$\mathcal{Z}$	Returns NUMA info.
	Passthrough devices (HYPERVISOR_physdev_op)				
34.1	manage_pci_add[H]	⊗		$\mathcal{T}$	Adds a PCI device to specified domain.
34.2	manage_pci_add_ext[H]	⊗		$\mathcal{T}$	Add external PCI.
34.3	manage_pci_remove[H]	⊗		$\mathcal{T}$	Removes a PCI device.
34.4	eoi[H]	⊗		$\mathcal{T}$	Notify EOI for the specified IRQ.
34.5	pirq_eoi_gmfn[H]	⊗		$\mathcal{T}$	Mange EOIs through a shared page.
34.6	irq_status_query[H]	⊗		$\mathcal{T}$	Query the status of an IRQ line.
34.7	set_iopl[H]	⊗		$\mathcal{T}$	Set the current VCPU I/O privilege level.
34.8	set_iobitmap[H]	⊗		$\mathcal{T}$	Set the current VCPU I/O port permission bitmap.
34.9	apic_write[H83]	⊗		$\mathcal{T}$	Write IOAPIC register.
34.10	apic_read[H83]	⊗		$\mathcal{T}$	Read IOAPIC register.
34.11	alloc_irq_vector[H27]	⊗		$\mathcal{T}$	Allocate physical upcall vector for an IRQ.
34.12	free_irq_vector[H]	⊗		$\mathcal{T}$	Free physical upcall vector.
34.13	map_pirq[H]	⊗		$\mathcal{T}$	Map physical IRQ.
34.14	unmap_pirq[H]	⊗		$\mathcal{T}$	Unmap physical IRQ.
34.15	restore_msi[H]	⊗		$\mathcal{T}$	Restore message signaled interrupt for PCI device.
34.16	setup_gsi[H]	⊗		$\mathcal{T}$	Setup global system interrupt.
	Domain management operations (HYPERVISOR_domctl_op)				
35.1	setvcpucontext[O18]		⊗	·	Sets the VCPU context of the specified domain and VCPU.
35.2	getvcpucontext[O6]		⊗	·	Gets the VCPU context.
35.3	pausedomain[O17]		⊗	·	Pauses execution of the specified domain.
35.4	unpausedomain[O19]		⊗	·	Unpauses execution of the specified domain.
35.5	resumedomain[O68]		⊗	$\mathcal{Z}$	Used by the Xen tools in the final part of domain resumption, brings up the VCPUs.
35.6	createdomain[O32]		⊗	$\mathcal{T}$	Create an empty shell for a new domain by setting up data structures in the hypervisor.
35.7	max_vcpus[O53]		⊗	$\mathcal{Z}$	Set the maximum number of VCPUs for the specified domain.
35.8	destroydomain[O2]		⊗	·	Destroy the specified domain and its data structures.
35.9	setvcpuaffinity[O81]		⊗	·	Set the VCPU affinity, i.e., specifying on which preferred *machine* CPU cores the VCPU runs.

	Hypercall	S	O	R	Description
35.10	getvcpuaffinity[O81]		⊗	·	Get the vCPU affinity.
35.11	scheduler_op[O71]		⊗	$\mathcal{Z}$	Adjust scheduling properties of the specified domain.
35.12	getdomaininfo[O42]		⊗	·	Get the domain info of a specified list of domains. The domain info structure contains vCPU info, CPU time, and more.
35.13	getvcpuinfo[O46]		⊗	$\mathcal{Z}$	Returns various details such as online and CPU time for the specified domain and vCPU.
35.14	max_mem[O76]		⊗	·	Set the maximum allowed memory for the specified domain.
35.15	setdomainhandle[O75]		⊗	$\mathcal{T}$	Set the domain handle, which by convention is filled by the domain builder with the UUID of the VM.
35.16	setdebugging[O74]		⊗	$\lightning$	Attach a Xen debugger for the specified domain.
35.17	irq_permission[O22,67]		⊗	$\mathcal{T}$	Grant the *irq* capability to the specified domain which is obligatory for hypercall #31.4.
35.18	iomem_permission[O22,67]		⊗	$\mathcal{T}$	Grant the *iomem* capability to the specified domain which enables it to use MMIO devices, for instance, a TPM.
35.19	ioport_permission[E22,67]		⊗	$\mathcal{T}$	Grant the *ioport* capability to the specified domain for communicating with hardware ports.
35.20	settimeoffset[O34]		⊗	$\mathcal{Z}$	Set the wall clock time in seconds since 1-1-1970 for the specified domain.
35.21	set_target[O73]		⊗	$\mathcal{T}$	Sets a target for the specified domain, over whom it then gains full privileges.
35.22	subscribe[O]		⊗	$\mathcal{T}$	Set the specified domain's suspend event channel.
35.23	disable_migrate[O]		⊗	$\mathcal{T}$	Disable (or re-enable) the migratability of specified domain.
35.24	shadow_op[E77]		⊗	$\mathcal{T}$	Control shadow page tables operation (cf. subsection 2.1.3).
35.25	getpageframeinfo[E45]		⊗	$\mathcal{T}$	Check if a page is pinned to a type.
35.26	getpageframeinfo2[E45]		⊗	$\mathcal{T}$	Check for an array of pages the pin status.
35.27	getpageframeinfo3[E45]		⊗	$\mathcal{T}$	Check for an array of pages the pin status (in x86-64).
35.28	getmemlist[E44]		⊗	$\mathcal{T}$	Returns a list of MFNs belonging to a specified domain.
35.29	hypercall_init[E50]		⊗	$\mathcal{T}$	Initialize for given domain a hypercall page: A small layer of indirection to give the hypervisor some flexibility in how hypercalls are performed.
35.30	sethvmcontext[E49]		⊗	$\mathcal{T}$	Sets the HVM context (i.e., the VMCS) for a domain, necessary for migrating HVM machines.

	Hypercall	S	O	R	Description
35.31	gethvmcontext[E]		⊗	$\mathcal{T}$	Gets the HVM context for a domain.
35.32	gethvmcontext_partial[E]		⊗	$\mathcal{T}$	Gets the HVM context for a domain (partially).
35.33	set_address_size[E24]		⊗	$\mathcal{T}$	Set the address size for a domain (i.e., to switch to compatibility 32-bits).
35.34	get_address_size[E24]		⊗	$\mathcal{T}$	Gets the address size.
35.35	set_machine_address_size[E52]		⊗	$\mathcal{T}$	Define the maximum machine address size which should be allocated for a specified domain.
35.36	get_machine_address_size[E52]		⊗	$\mathcal{T}$	Gets the maximum machine address size.
35.37	sendtrigger[E72]		⊗	$\mathcal{Z}$	Send triggers such as 'reset button pressed' to specified HVM domain.
35.38	assign_device[E26]		⊗	$\mathcal{T}$	Assigns a PCI device to specified HVM domain, relying on IOMMU.
35.39	test_assign_device[E79]		⊗	$\mathcal{T}$	Check if a device is already assigned.
35.40	deassign_device[E]		⊗	$\mathcal{T}$	Remove the assignment of a device.
35.41	get_device_group[E]		⊗	$\mathcal{T}$	Retrieve sibling information for a device.
35.42	bind_pt_irq[E29]		⊗	$\mathcal{T}$	Sets up a passthrough IRQ from HW directly to specified HVM domain.
35.43	unbind_pt_irq[E]		⊗	$\mathcal{T}$	Removes passthrough IRQ.
35.44	memory_mapping[E]		⊗	$\mathcal{T}$	Bind MMIO range to specified HVM domain.
35.45	ioport_mapping[E]		⊗	$\mathcal{T}$	Binds I/O ports to specified HVM domain.
35.46	pin_mem_cacheattr[E62]		⊗	$\mathcal{T}$	Pin caching type of RAM for specified HVM domain.
35.47	set_ext_vcpucontext[E38]		⊗	$\mathcal{T}$	Set extended vCPU context.
35.48	get_ext_vcpucontext[E38]		⊗	$\mathcal{T}$	Get extended vCPU context.
35.49	setvcpuextstate[E82]		⊗	$\mathcal{T}$	Set extended vCPU state.
35.50	getvcpuextstate[E82]		⊗	$\mathcal{T}$	Get extended vCPU state.
35.51	set_cpuid[E]		⊗	$\mathcal{T}$	Set the processor information visible for the specified domain.

	Hypercall	S	O	R	Description
35.52	settscinfo[E]		⊗	$\mathcal{T}$	Set time stamp counter (TSC) information for specified domain.
35.53	gettscinfo[E]		⊗	$\mathcal{T}$	Retrieve TSC information for specified domain.
35.54	suppress_spurious_page_faults[E]		⊗	$\mathcal{T}$	Disable the injection of spurious page faults into specified domain.
35.55	mem_sharing_op[E]		⊗	$\mathcal{T}$	Memory sharing operations (e.g., copy on write pages between domains).
35.56	debug_op[E]		⊗	$\lightning$	Debug specified HVM domain stepwise.
35.57	mem_event_op[E]		⊗	$\lightning$	Allows memory access listener to perform single stepping for HVM domain.
35.58	set_access_required[E]		⊗	$\lightning$	Stipulates that the memory access listener must be present, otherwise pauses the HVM domain.
35.59	gdbsx_guestmemio[E]		⊗	$\lightning$	Debug a domain using GDBSX, inspecting the memory.
35.60	gdbsx_pausevcpu[E]		⊗	$\lightning$	For domain debugging, pause VCPU.
35.61	gdbsx_unpausevcpu[E]		⊗	$\lightning$	For domain debugging, unpause VCPU.
35.62	gdbsx_domstatus[E]		⊗	$\lightning$	For domain debugging, retrieve status of domain.

A. `arch/⟨arch⟩/mm.c`
B. `arch/⟨arch⟩⟨subarch⟩/mm.c`
C. `arch/x86/⟨subarch⟩/mm.c`
D. `arch/x86/⟨subarch⟩/traps.c`
E. `arch/x86/domct.c`
F. `arch/x86/hvm/hvm.c`
G. `arch/x86/mm.c`
H. `arch/x86/physdev.c`
I. `arch/x86/platform_hypercall.c`
J. `arch/x86/sysctl.c`
K. `arch/x86/sysctl.c`
L. `arch/x86/traps.c`
M. `arch/x86/x86_64/mm.c`
N. `common/domain.c`
O. `common/domctl.c`
P. `common/event_channel.c`
Q. `common/grant_table.c`
R. `common/kernel.c`
S. `common/kexec.c`
T. `common/memory.c`
U. `common/multicall.c`
V. `common/schedule.c`
W. `common/sysctl.c`
X. `common/xenoprof.c`
Y. `xsm/xsm_core.c`

Table F.2: Locations where hypercalls are processed. The placeholders are for hypercalls which are implemented differently across several architectures.

G. XSM hooks

To enforce the division of responsibilities between Dom0 and DomT outlined in appendix F, we made ample use of the XSM framework. The XSM framework largely covered our needs, in particular for the hypercalls which most threaten confidentiality and integrity. In fact, it offers hooks even for hypercalls which are not privileged, of which we made no use. On the other hand, not all hypercalls that we wished to curtail have an appropriate hook available. We experimented with adding our own hooks, however, the magnitude of that operation meant we could not create all the hooks for this thesis.

In Table G.2 we list all the hooks that were used in our access control module. In Table G.1 we list the hooks that we did not use in our access control module, either because it was not necessary (not a privileged operation), or because we chose to share the hypercall between Dom0 and DomT and thus no curtailing was necessary. Both tables list the relevant XSM hook and also which hypercalls from Table F.1 depend on it.

1. ⟨*xsm_*⟩console_io 28.15
2. ⟨*xsm_*⟩destroydomain 35.8
3. ⟨*xsm_*⟩evtchn_close_post 31.6
4. ⟨*xsm_*⟩evtchn_interdomain 31.2
5. ⟨*xsm_*⟩evtchn_send 31.7
6. ⟨*xsm_*⟩getvcpucontext 35.2
7. ⟨*xsm_*⟩grant_copy 29.5
8. ⟨*xsm_*⟩grant_map_ref 29.1
9. ⟨*xsm_*⟩grant_map_unref 29.2
10. ⟨*xsm_*⟩grant_transfer 29.4
11. ⟨*xsm_*⟩hvm_param 24
12. ⟨*xsm_*⟩hvm_set_isa_irq_level 24
13. ⟨*xsm_*⟩hvm_set_pci_intx_level 24
14. ⟨*xsm_*⟩hvm_set_pci_link_route 24
15. ⟨*xsm_*⟩machine_memory_map 28.13
16. ⟨*xsm_*⟩memory_pin_page 19
17. ⟨*xsm_*⟩pausedomain 35.3
18. ⟨*xsm_*⟩setvcpucontext 35.1
19. ⟨*xsm_*⟩unpausedomain 35.4
20. ⟨*xsm_*⟩xsm_op 20

Table G.1: XSM hooks that were *not* used for our access control.

21. ⟨xsm_⟩acpi_sleep 32.8
22. ⟨xsm_⟩add_range 35.18, 35.19, 35.17
23. ⟨xsm_⟩add_to_physmap 28.11
24. ⟨xsm_⟩address_size 35.33, 35.34
25. ⟨xsm_⟩alloc_security_evtchn 31.1
26. ⟨xsm_⟩assign_device 35.38
27. ⟨xsm_⟩assign_vector 34.11
28. ⟨xsm_⟩availheap 33.9
29. ⟨xsm_⟩bind_pt_irq 35.42
30. ⟨xsm_⟩change_freq 32.9
31. ⟨xsm_⟩debug_keys 33.7
32. ⟨xsm_⟩domain_create 35.6
33. ⟨xsm_⟩domain_memory_map 28.14
34. ⟨xsm_⟩domain_settime 35.20
35. ⟨xsm_⟩evtchn_reset 31.11
36. ⟨xsm_⟩evtchn_status 31.8
37. ⟨xsm_⟩evtchn_unbound 31.1
38. ⟨xsm_⟩ext_vcpucontext 35.47, 35.48
39. ⟨xsm_⟩firmware_info 32.7
40. ⟨xsm_⟩get_pmstat 33.10
41. ⟨xsm_⟩getcpuinfo 33.8
42. ⟨xsm_⟩getdomaininfo 35.12 ,33.4
43. ⟨xsm_⟩getidletime 32.10
44. ⟨xsm_⟩getmemlist 35.28, 35.28, 35.28
45. ⟨xsm_⟩getpageframeinfo 35.25, 35.26, 35.27
46. ⟨xsm_⟩getvcpuinfo 35.13
47. ⟨xsm_⟩grant_query_size 29.6
48. ⟨xsm_⟩grant_setup 29.3
49. ⟨xsm_⟩hvmcontext 35.30, 35.31, 35.32
50. ⟨xsm_⟩hypercall_init 35.29
51. ⟨xsm_⟩kexec 28.26
52. ⟨xsm_⟩machine_address_size 35.35, 35.36
53. ⟨xsm_⟩max_vcpus 35.7
54. ⟨xsm_⟩memory_adjust_reservation 28.1, 28.2, 28.3
55. ⟨xsm_⟩memory_stat_reservation 28.6, 28.7, 28.9
56. ⟨xsm_⟩memtype 32.2, 32.3, 32.4
57. ⟨xsm_⟩microcode 32.5
58. ⟨xsm_⟩mm_machphys_update 28.2
59. ⟨xsm_⟩mm_normal_update 28.2
60. ⟨xsm_⟩perfcontrol 33.5
61. ⟨xsm_⟩physinfo 33.15
62. ⟨xsm_⟩pin_mem_cacheattr 35.46
63. ⟨xsm_⟩platform_quirk 32.6
64. ⟨xsm_⟩pm_op 33.11
65. ⟨xsm_⟩profile 27
66. ⟨xsm_⟩readconsole 33.1
67. ⟨xsm_⟩remove_range 35.18, 35.19, 35.17
68. ⟨xsm_⟩resumedomain 35.5
69. ⟨xsm_⟩sched_id 33.3
70. ⟨xsm_⟩schedop_shutdown 30.5
71. ⟨xsm_⟩scheduler 35.11
72. ⟨xsm_⟩sendtrigger 35.37
73. ⟨xsm_⟩set_target 35.21
74. ⟨xsm_⟩setdebugging 35.16
75. ⟨xsm_⟩setdomainhandle 35.15
76. ⟨xsm_⟩setdomainmaxmem 35.14
77. ⟨xsm_⟩shadow_control 35.24
78. ⟨xsm_⟩tbufcontrol 33.2
79. ⟨xsm_⟩test_assign_device 35.39
80. ⟨xsm_⟩update_va_mapping 28.3, 28.4
81. ⟨xsm_⟩vcpuaffinity 35.9, 35.10
82. ⟨xsm_⟩vcpuextstate 35.49, 35.50
83. ⟨xsm_⟩xen_apic 34.10, 34.9
84. ⟨xsm_⟩xen_settime 32.1

Table G.2: XSM hooks that were used for our access control.

Bibliography

[**AL12**] Alert Logic. *An emperical analysis of real world threats*. 2012. URL: http://www.alertlogic.com/resources/state-of-cloud-security-report (visited on 11/2012).

[**AMDV**] AMD corporation. *AMD Programmer's manual, volume 2*. URL: http://support.amd.com/us/Processor_TechDocs/24593_APM_v2.pdf (visited on 11/2012).

[**And01**] Ross J. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. 1st. New York, NY, USA: John Wiley & Sons, Inc., 2001. ISBN: 0471389226.

[**Arm+10**] M. Armbrust et al. "A view of cloud computing". In: *Communications of the ACM* 53.4 (2010), pp. 50–58.

[**Bar+03**] P. Barham et al. "Xen and the art of virtualization". In: *ACM SIGOPS Operating Systems Review*. Vol. 37. 5. ACM. 2003, pp. 164–177.

[**Bel05**] F. Bellard. "QEMU, a fast and portable dynamic translator". In: USENIX. 2005.

[**Ber+06**] Stefan Berger et al. "vTPM: Virtualizing the trusted platform module". In: *In USENIX Security*. 2006, pp. 305–320.

[**Bis04**] M. Bishop. *Introduction to computer security*. Addison-Wesley Professional, 2004.

[**But+12**] Shakeel Butt et al. "Self-service cloud computing". In: *Proceedings of the 2012 ACM conference on Computer and communications security*. CCS '12. Raleigh, North Carolina, USA: ACM, 2012, pp. 253–264. ISBN: 978-1-4503-1651-4. DOI: 10.1145/2382196.2382226. URL: http://doi.acm.org/10.1145/2382196.2382226.

[**Can+04**] G. Candea et al. "Microreboot–a technique for cheap recovery". In: *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*. Vol. 6. 2004.

[**Cat+10**] L. Catuogno et al. "Trusted Virtual Domains–design, implementation and lessons learned". In: *Trusted Systems* (2010), pp. 156–179.

[**CC**] Common Criteria. *Portal*. URL: `http://www.commoncriteriaportal.org/cc` (visited on 12/2012).

[**Chi07**] David Chisnall. *The Definitive Guide to the Xen Hypervisor (Prentice Hall Open Source Software Development Series)*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2007. ISBN: 013234971X.

[**Col+11**] Patrick Colp et al. "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. Cascais, Portugal: ACM, 2011, pp. 189–202. ISBN: 978-1-4503-0977-6. DOI: `10.1145/2043556.2043575`. URL: `http://doi.acm.org/10.1145/2043556.2043575`.

[**CS**] Apache foundation. *Cloudstack*. URL: `http://incubator.apache.org/cloudstack` (visited on 11/2012).

[**CSA10**] Cloud Security Alliance (CSA). *Top threats to cloud computing, version 1.0*. Mar. 2010. URL: `http://www.cloudsecurityalliance.org/topthreats/csathreats.v1.0.pdf` (visited on 11/2012).

[**Den+12**] Mina Deng et al. "Towards Trustworthy Health Platform Cloud". In: *Secure Data Management*. Ed. by Willem Jonker and Milan Petković. Vol. 7482. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 162–175. ISBN: 978-3-642-32872-5. DOI: `10.1007/978-3-642-32873-2_12`. URL: `http://dx.doi.org/10.1007/978-3-642-32873-2_12`.

[**Des+08**] T. Deshane et al. "Quantitative comparison of Xen and KVM". In: *Xen Summit, Boston, MA, USA* (2008), pp. 1–2.

[**DY83**] D. Dolev and A. Yao. "On the security of public key protocols". In: *Information Theory, IEEE Transactions on* 29.2 (1983), pp. 198–208.

[**Fer06**] N. Ferguson. "AES-CBC+ Elephant diffuser: A disk encryption algorithm for Windows Vista". In: *Microsoft Corp* (2006).

[**FI**] Various authors. *Fio tool*. URL: `http://freecode.com/projects/fio` (visited on 11/2012).

[**Fru05**] C. Fruhwirth. "New methods in hard disk encryption". In: *Institute for Computer Languages, Theory and Logic Group, Vienna University of Technology* (2005).

[**Gen09**] Craig Gentry. "Fully homomorphic encryption using ideal lattices". In: *Proceedings of the 41st annual ACM symposium on Theory of computing*. STOC '09. Bethesda, MD, USA: ACM, 2009, pp. 169–178. ISBN: 978-1-60558-506-2. DOI: 10.1145/1536414.1536440. URL: http://dx.doi.org/10.1145/1536414.1536440.

[**Gol73**] R.P. Goldberg. *Architectural principles for virtual computer systems*. Tech. rep. DTIC Document, 1973.

[**GR**] GNU. *GRand Unified Bootloader*. URL: http://www.gnu.org/software/grub (visited on 11/2012).

[**GR03**] T. Garfinkel and M. Rosenblum. "A virtual machine introspection based architecture for intrusion detection". In: *Proc. Network and Distributed Systems Security Symposium*. 2003.

[**Joh+**] Jeff Johnston et al. *Newlib*. URL: http://sourceware.org/newlib (visited on 10/2012).

[**Kau07**] B. Kauer. "OSLO: Improving the security of Trusted Computing". In: *Proceedings of 16th USENIX security symposium on usenix security symposium*. 2007, pp. 1–9.

[**Kel+10**] Eric Keller et al. "NoHype: virtualized cloud infrastructure without the virtualization". In: *Proceedings of the 37th annual international symposium on Computer architecture*. ISCA '10. Saint-Malo, France: ACM, 2010, pp. 350–361. ISBN: 978-1-4503-0053-7. DOI: 10.1145/1815961.1816010. URL: http://doi.acm.org/10.1145/1815961.1816010.

[**Kiv+07**] A. Kivity et al. "kvm: the Linux virtual machine monitor". In: *Proceedings of the Linux Symposium*. Vol. 1. 2007, pp. 225–230.

[**Lie+95**] J. Liedtke et al. "On-kernel construction". In: *Proceedings of the 15th ACM Symposium on OS Principles*. 1995, pp. 237–250.

[**Lie96**] J. Liedtke. "Toward real microkernels". In: *Communications of the ACM* 39.9 (1996), pp. 70–77.

[**Mc+08**] J.M. McCune et al. "Flicker: An execution infrastructure for TCB minimization". In: *SIGOPS Operating Systems Review* 42.4 (2008), pp. 315–328.

[**Mer87**] Ralph Merkle. "A Digital Signature Based on a Conventional Encryption Function". In: *Advances in Cryptology – CRYPTO '87*. Ed. by Carl Pomerance. Vol. 293. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2006, pp. 369–378. ISBN: 978-3-540-18796-7.

[**MG11**] P. Mell and T. Grance. "The NIST definition of cloud computing (draft)". In: *NIST special publication* 800 (2011), p. 145.

[**MMH08**] Derek Gordon Murray, Grzegorz Milos, and Steven Hand. "Improving Xen security through disaggregation". In: *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*. VEE '08. Seattle, WA, USA: ACM, 2008, pp. 151–160. ISBN: 978-1-59593-796-4. DOI: 10.1145/1346256.1346278. URL: http://doi.acm.org/10.1145/1346256.1346278.

[**NBH08**] K. Nance, M. Bishop, and B. Hay. "Virtual Machine Introspection: Observation or Interference?" In: *Security Privacy, IEEE* 6.5 (2008), pp. 32 –37. ISSN: 1540-7993. DOI: 10.1109/MSP.2008.134.

[**OS**] Openstack. *Openstack*. URL: http://www.openstack.org (visited on 10/2012).

[**PG74**] G.J. Popek and R.P. Goldberg. "Formal requirements for virtualizable third generation architectures". In: *Communications of the ACM* 17.7 (1974), pp. 412–421.

[**RC11**] F. Rocha and M. Correia. "Lucy in the sky without diamonds: Stealing confidential data in the cloud". In: *Dependable Systems and Networks Workshops (DSN-W), 2011 IEEE/IFIP 41st International Conference on*. IEEE. 2011, pp. 129–134.

[**Ris+09**] Thomas Ristenpart et al. "Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds". In: *Proceedings of the 16th ACM conference on Computer and communications security*. CCS '09. Chicago, Illinois, USA: ACM, 2009, pp. 199–212. ISBN: 978-1-60558-894-0. DOI: 10.1145/1653662.1653687. URL: http://doi.acm.org/10.1145/1653662.1653687.

[**Sad+07**] A.R. Sadeghi et al. "Enabling fairer digital rights management with trusted computing". In: *Information Security* (2007), pp. 53–70.

[**Sad11**] Ahmad-Reza Sadeghi. *Trusted Computing lecture slides*. 2011. URL: http://www.trust.informatik.tu-darmstadt.de/fileadmin/user_upload/Group_TRUST/LectureSlides/ESS-SS2011/Chap3_-_TCG_Concepts.pdf (visited on 12/2012).

[**Sai+05**] R. Sailer et al. "Building a MAC-based security architecture for the Xen open-source hypervisor". In: *Computer Security Applications Conference, 21st Annual*. IEEE. 2005, 10–pp.

[**Sar+06**] L.F.G. Sarmenta et al. "Virtual monotonic counters and count-limited objects using a TPM without a trusted OS". In: *Proceedings of the first ACM workshop on Scalable trusted computing*. ACM. 2006, pp. 27–42.

[**Sau**] C. Sauot. *dm-crypt: A device-mapper crypto target*. URL: `http://www.saout.de/misc/dm-crypt` (visited on 10/2012).

[**Sch+10**] Joshua Schiffman et al. "Seeding clouds with trust anchors". In: *Proceedings of the 2010 ACM workshop on Cloud computing security workshop*. CCSW '10. Chicago, Illinois, USA: ACM, 2010, pp. 43–46. ISBN: 978-1-4503-0089-6. DOI: `10.1145/1866835.1866843`. URL: `http://doi.acm.org/10.1145/1866835.1866843`.

[**SGR09**] Nuno Santos, Krishna P. Gummadi, and Rodrigo Rodrigues. "Towards trusted cloud computing". In: *Proceedings of the 2009 conference on Hot topics in cloud computing*. HotCloud'09. San Diego, California: USENIX Association, 2009. URL: `http://dl.acm.org/citation.cfm?id=1855533.1855536`.

[**SK10**] U. Steinberg and B. Kauer. "NOVA: A microhypervisor-based secure virtualization architecture". In: *Proceedings of the 5th European conference on Computer systems*. ACM. 2010, pp. 209–222.

[**SL**] D.A. Wheeler. *SLOCCount*. URL: `http://www.dwheeler.com/sloccount` (visited on 12/2012).

[**SS04**] A.R. Sadeghi and C. Stüble. "Property-based attestation for computing platforms: caring about properties, not mechanisms". In: *Proceedings of the 2004 workshop on New security paradigms*. ACM. 2004, pp. 67–77.

[**SSB07**] P. Sevinç, M. Strasser, and D. Basin. "Securing the distribution and storage of secrets with trusted platform modules". In: *Information Security Theory and Practices. Smart Cards, Mobile and Ubiquitous Computing Systems* (2007), pp. 53–66.

[**SSW08**] Ahmad-Reza Sadeghi, Christian Stüble, and Marcel Winandy. "Property-Based TPM Virtualization". In: *Information Security*. Ed. by Tzong-Chen Wu et al. Vol. 5222. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2008, pp. 1–16. ISBN: 978-3-540-85884-3. URL: `http://dx.doi.org/10.1007/978-3-540-85886-7_1`.

[**Sta06**] William Stallings. *Cryptography and Network Security*. 4th. Pearson Education India, 2006.

[**TB**] Intel corporation. *TBoot on SourceForge*. URL: `http://sourceforge.net/projects/tboot` (visited on 11/2012).

[**TC**] TrueCrypt Developers Association et al. *TrueCrypt-Free open-source disk encryption software for Windows 7/Vista/XP, Mac OS X, and Linux*. URL: `http://www.truecrypt.org` (visited on 10/2012).

[**TCG**] Trusted Computing Group. *About and members*. URL: `http://www.trustedcomputinggroup.org/about_tcg/tcg_members` (visited on 06/2012).

[**TCG03**] Trusted Computing Group. *TPM main specification*. 2003. URL: `http://www.trustedcomputinggroup.org/developers/trusted_platform_module` (visited on 06/2012).

[**TCGb**] Trusted Computing Group. *How to make your system and data truly secure*. URL: `http://www.trustedcomputinggroup.org/resources/trusted_computing_how_to_make_your_systems_and_data_truly_secure` (visited on 11/2012).

[**TD08**] S. Thibault and T. Deegan. "Improving performance by embedding HPC applications in lightweight Xen domains". In: *Proceedings of the 2nd workshop on System-level virtualization for high performance computing*. ACM. 2008, pp. 9–15.

[**TG**] Sirrix AG and Ruhr-University Bochum. *Trusted GRUB*. URL: `http://projects.sirrix.com/trac/trustedgrub` (visited on 12/2012).

[**TR**] TrouSerS developers. *TrouSerS – The open-source TCG Software Stack*. URL: `http://trousers.sourceforge.net` (visited on 11/2012).

[**TXT**] Intel corporation. *Intel Trusted Execution Technology*. URL: `http://download.intel.com/technology/security/downloads/315168.pdf` (visited on 11/2012).

[**Vaq+08**] L.M. Vaquero et al. "A break in the clouds: towards a cloud definition". In: *ACM SIGCOMM Computer Communication Review* 39.1 (2008), pp. 50–55.

[**VB**] Oracle corporation. *Oracle VM Virtualbox*. URL: `http://www.virtualbox.org` (visited on 11/2012).

[**VDJ10**] M. Van Dijk and A. Juels. "On the impossibility of cryptography alone for privacy-preserving cloud computing". In: *Proceedings of the 5th USENIX conference on Hot topics in security*. USENIX Association. 2010, pp. 1–8.

[**VM**] VMware corporation. *VMware ESX Server, VMWare Player*. URL: `http://www.vmware.com/` (visited on 11/2012).

[**VV09**] A. Velte and T. Velte. *Microsoft virtualization with Hyper-V*. McGraw-Hill, Inc., 2009.

[**VZ09**] J. Voas and J. Zhang. "Cloud Computing: new wine or just a new bottle?" In: *IT professional* 11.2 (2009), pp. 15–17.

[**WD**] Western Digital corporation. *WD Caviar SE16 family*. URL: `http://www.direktronik.se/pdfer/i28-0314.pdf` (visited on 12/2012).

[**WJW12**] D. Williams, H. Jamjoom, and H. Weatherspoon. "The Xen-Blanket: virtualize once, run everywhere". In: *ACM EuroSys* (2012).

[**XML1**] Xen mailing list. *New domain builder in xen-unstable*. URL: `http://lists.xen.org/archives/html/xen-ia64-devel/2007-01/msg00249.html` (visited on 11/2012).

[**XSA15**] Xen developers et al. *Xen security advisory 15*. URL: `http://lists.xen.org/archives/html/xen-announce/2012-09/msg00006.html` (visited on 10/2012).

[**XSA16**] Xen developers et al. *Xen security advisory 16*. URL: `http://lists.xen.org/archives/html/xen-announce/2012-09/msg00005.html` (visited on 10/2012).

[**YSK09**] A. Yun, C. Shi, and Y. Kim. "On protecting integrity and confidentiality of cryptographic file system for outsourced storage". In: *Proceedings of the 2009 ACM workshop on Cloud computing security*. ACM. 2009, pp. 67–76.

[**Zha+11a**] Fengzhe Zhang et al. "CloudVisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization". In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. SOSP '11. Cascais, Portugal: ACM, 2011, pp. 203–216. ISBN: 978-1-4503-0977-6. DOI: `10.1145/2043556.2043576`. URL: `http://doi.acm.org/10.1145/2043556.2043576`.

[**Zha+11b**] Y. Zhang et al. "Homealone: Co-residency detection in the cloud via side-channel analysis". In: *Security and Privacy (SP), 2011 IEEE Symposium on*. IEEE. 2011, pp. 313–328.

[**Zha+12**] Yinqian Zhang et al. "Cross-VM side channels and their use to extract private keys". In: *Proceedings of the 2012 ACM conference on Computer and communications security*. CCS '12. Raleigh, North Carolina, USA: ACM, 2012, pp. 305–316. ISBN: 978-1-4503-1651-4. DOI: 10.1145/2382196.2382230. URL: http://doi.acm.org/10.1145/2382196.2382230.

Acronyms

A

ACM access control module.
AES Advanced Encryption Standard.
AIK attestation identity key.
AMD-V AMD virtualization.
API application programming interface.
AWS Amazon web services.

B

BIOS basic input output interface.

C

CA certificate authority.
CaaS cryptography as a service.
CBC cipher-block chaining.
CIA confidentiality, integrity and availability.
CPU central processing unit.
CRTM core root of trust for measurements.
CSINFO CaaS-info.
CSP cloud service provider.
CV cloud verifier.

D

DAA direct anonymous attestation.
DMA direct memory access.
Dom0 domain zero.
DomB the builder domain.
DomC crypto domain.
DomT trusted domainbuilder domain.
DomU domain unprivileged.
DRTM dynamic root of trust for measurements.

E

EC2 Amazon Elastic Compute Cloud.
EK endorsement key.
EOI end of interrupt.
ESSIV encrypted salt-sector initialization vector.
EVM encrypted virtual machine.

F

FDE full disk encryption.
FHE fully homomorphic encryption.

G

GPFN guest page frame number.
GRUB GNU grand unified bootloader.

H

HMAC hash-based message authentication code.
HSM hardware security module.
HVK high value key.
HVM hardware virtual machine.
HW hardware.

I

IaaS infrastructure as a service.

IOAPIC I/O advanced programmable interrupt controller.
IOMMU input/ouput memory management unit.
IPSec internet protocol security.
IRET interrupt return.
IRQ interrupt request.
IV initialization vector.
IVMC inter-VM communication.

K

KLoC kilo lines of code.
KVM Kernel-based Virtual Machine.

L

LCP launch control policy.
LOC lines of code.

M

M2P machine to physical.
MAC message authentication code.
MBR master boot record.
MFN machine frame number.
MLE measured launch environment.
MMIO memory mapped input/output.
MMU memory management unit.
MTRR memory type range register.

N

NC node controller.
NDA non-disclosure agreement.
NIC network interface card.
NIST National Institute of Standards and Technology.
NOVA NOVA OS virtualization architecture.
NPK node public key.

O

OIAP object-independent authorization protocol.
OS operating system.
OSAP object-specific authorization protocol.

P

PaaS platform as a service.
PC personal computer.
PCI peripheral controller Interface.
PCR platform configuration register.
PHR personal health record.
PoD populate-on-demand.
POSIX portable operating system interface.
PV paravirtualized.
PVOps paravirtual operations.

R

RAM random access memory.
RNG random number generator.
RPC remote procedure call.
RSA Rivest Shamir Adleman.

S

SaaS software as a service.
SHA-1 secure hash algorithm #1.
SMP symmetric multiprocessor architectures.
SRK storage root key.
SRTM static root of trust for measurements.
SSC self-service cloud.

T

TBoot Trusted Boot.
TC trusted computing.
TCB trusted computing base.

TCG Trusted Computing Group.
TLB translation lookaside buffer.
TOCTOU time of check, time of use.
TPM trusted platform module.
TPM-NV TPM non-volatile memory.
TS task switch.
TSC time stamp counter.
TSPI TSS service provider interface.
TSS TCG software stack.
TTP trusted third party.
TVD trusted virtual domain.
TXT trusted execution technology.

U

UUID universally unique identifier.

V

VCPU virtual CPU.
VFS virtual filesystem.
VL verified launch.
VM virtual machine.
VMCB virtual machine control blob.
VMM virtual machine monitor.
vTPM virtual TPM.
vTPM-NV virtual TPM non-volatile memory.

X

XSM Xen security module.

Index of symbols and identifiers

This index is sorted on the name of the identifier without the leading prefix.

Index of concepts