SELF-SERVICE CLOUD COMPUTING

BY

SHAKEEL BUTT

A dissertation submitted to the

Graduate School—New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

Graduate Program in Computer Science

Written under the direction of

Vinod Ganapathy

and approved by

____

____

____

____

New Brunswick, New Jersey

January, 2015

ABSTRACT OF THE DISSERTATION

SELF-SERVICE CLOUD COMPUTING

by SHAKEEL BUTT

Dissertation Director: Vinod Ganapathy

Cloud computing has transformed the IT industry. Clients can acquire computing resources on demand from the cloud, and can drastically reduce their maintenance, management and startup cost. Many new companies rely exclusively on the cloud and according to Gartner's study [23], by 2015, 90% of government agencies and large companies will be using the cloud. However, many challenges remain in ensuring wide adoption of the cloud. In this work, we have focused on two such challenges.

The first challenge is that of security and privacy. When clients choose to use public cloud infrastructure, the confidentiality and integrity of their code and data can be compromised by insider attacks (e.g., malicious system administrators). The second challenge is that of inadequate flexibility provided to the clients. Clients must typically rely on the cloud provider to deploy useful services, such as security services (NIDS or Rootkit and Malware detectors) or deduplication services *e.g.,*memory or storage deduplication.

In virtualized cloud infrastructures, a Virtual Machine Monitor (VMM) governs the execution of client virtual machines (VMs). Both the challenges discussed above arise from the way VMMs assign privilege to client VMs. In this work, we have designed and implemented Self-service Cloud Computing (SSC), a new cloud computing model that introduces novel abstractions to improve the security and privacy of client code and data, and gives clients more flexible control over their VMs.

In SSC, the privilege model of a commodity VMM is modified and a new cloud management platform is designed and implemented, which uses the modified VMM to solve the security and flexibility problem without affecting the benefits of cloud computing like low maintenance and management cost. SSC incorporates protocols based on Trusted Platform Module (TPM) to establish client's trust on the SSC enabled infrastructure. To demonstrate the utility of SSC, we have implemented and evaluated multiple security, storage and networking services.

Acknowledgements

First and foremost I would like to thank my graduate adviser Professor Vinod Ganapathy. Vinod's constant support and push has made this dissertation a reality. His vision, passion and drive have been a constant source of motivation and confidence for me. I am really thankful to him for believing in me and supporting me wherever I stumbled during my graduate journey. I also would like to express my gratitude to the rest of my dissertation committee members Professor Liviu Iftode, Professor Ricardo Bianchini and Dr. Cristian Ungureanu for providing feedback.

I have also had the good fortune of being able to work on the initial part of this dissertation with Dr. H. Andrés Lagar-Cavilla, my mentor at AT&T Labs Research. His pragmatism and in-depth knowledge of virtualization has greatly influenced the core ideas underlying this dissertation. I would also like to thank Dr. Abhinav Srivastava, my mentor during my second tenure at AT&T Labs Research, for his support and invaluable advise.

I have had the privilege of sharing my time at Rutgers with several excellent colleagues. I especially thank Mohan Dhawan, Mudassir Shabbir, Liu Yang, Rezwana Karim, Amruta Gokhale, Liu Yang, Nader Boushehrinejadmoradi, Steve Smaldone, Pravin Shankar, Lu Han and Chetan Tonde.

I would also like to thank Nabeel Butt and Sharjeel Ahmed Qureshi for providing me feedback on the early draft of this dissertation. Last but not the least, I am thankful to my wife, Saher, for her patience while I was working on this dissertation.

Dedication

To my spirtual leader and murshid, Hazrat Khwaja Moeen-ud-din Chisti Ajmari (R.A).

Table of Contents

List of Figures

Chapter 1

Introduction

This dissertation proposes a new cloud computing model, called Self-Service Cloud computing (SSC). SSC introduces new abstractions that improve the security and privacy of client code and data and offer clients flexible control over their virtual machines. SSC is evaluated using a number of case studies that showcase its utility and demonstrate its potential to improve the state-of-the-art in cloud platform security.

1.1 Motivation

In recent years, cloud computing has gained popularity among many enterprises. A large number of enterprises have migrated their applications to the cloud. Several startups, such as like edX and Instagram, exclusively rely on the cloud to provide services to their customers. A study by Gartner [23] forecasting the future trends of the computer software industry claims that 90% of government agencies and large enterprises will be using the cloud by 2015.

One reason behind this popularity is the attractive economic incentives cloud computing has to offer to end-users. Cloud computing frees clients from having to procure computing infrastructure, thereby reducing the barrier to entry which benefits startups. It reduces the cost of managing this infrastructure by providing automated management mechanisms and offloads the maintenance cost to the cloud provider. In addition, cloud computing provides elasticity in the use of computing resources and allows cloud customers to improve the reliability of their services via replication and redundant placement of computation across geographically distributed different data centers.

In this dissertation, we are only interested in *public* cloud platforms, such as Amazon EC2, Google Compute Engine and Microsoft Azure. Despite the apparent popularity and benefits of cloud computing, many enterprises hesitate to migrate their applications to the cloud, opting instead to use private or in-house cloud offerings. Doing so often negates many of the benefits that public cloud computing has to offer, *e.g.,* in an in-house cloud infrastructure, the enterprise must have to procure and manage the computing infrastructure.

There are two major reasons why enterprises are often reluctant to use public cloud infrastructure. The first reason is security and privacy of client data and computation. Enterprises that host sensitive or proprietary data (*e.g.,* banks and pharmaceutical industries) want to protect this data from outsiders. On public cloud, the provider controls and manages the infrastructure running the customers applications and has privileges to inspect the memory contents and network traffic of the application. Thus, sensitive client data is vulnerable to attacks by malicious cloud operators. Cloud Security Alliance has characterized this threat as "Malicious insider working for cloud provider" and marked it as one of the top security threats to cloud computing [9, 10].

The second reason is the inflexible dependence of customers on the provider for new and customized infrastructure level services for their applications. Delegating management to the cloud provider is often a two-edged sword. While doing so can reduce operating costs, it also has the disadvantage of making the cloud provider exclusively responsible for services that the enterprise client may wish to avail. These problems arise due to the core technology used to power the cloud infrastructure *i.e.,*virtualization. To be more precise, these problems are due to the way the existing software systems implementing virtualization are designed.

Cloud providers use virtualization to multiplex physical resources among their customers. Customers are given the abstraction of a physical machine, known as virtual machine (VM). They can run their applications in these VMs similar to running applications on real physical machines. Virtualization allows running multiple VMs on a single physical machine and provides isolation between them. Thus, using virtualization, cloud providers can reduce the infrastructure cost without compromising the isolation among the customers. The virtualization software stack is also known as Virtual Machine Monitors (VMMs).

Virtual machine monitors implement a trusted computing base (TCB) that virtualizes the underlying hardware (CPU, memory and I/O devices) and manages VMs. In commodity

VMMs, such as Xen and Hyper-V, the TCB has two parts—*the hypervisor* and an *administrative domain*. The hypervisor directly controls physical hardware and runs at the highest processor privilege level. The administrative domain, henceforth called *dom0*, is a privileged VM that is used to control and monitor client VMs. Dom0 has privileges to start/stop client VMs, change client VM configuration, monitor their physical resource utilization, and perform I/O for virtualized devices. Dom0 also runs device drivers and virtualizes I/O for the VMs.

Endowing dom0 with such privileges leads to the underlying two problems mentioned before:

- ***Security and privacy of client VMs.***

  Dom0 has the privilege to inspect the state of client VMs, *e.g.,* the contents of their vCPU registers and memory. Dom0 is used by the cloud provider to control and manage the virtual machines (VM) running on the system. This privilege can be misused by attacks against the dom0 software stack (*e.g.,* because of vulnerabilities or misconfigurations) and malicious system administrators or cloud operators. This is a realistic threat [13–17, 22, 28], since dom0 typically executes a full-fledged operating system with supporting user-level utilities that can be configured in complex ways.

- ***Inflexible control over client VMs.***

  Virtualization has the potential to enable novel services, such as security via VM introspection [6, 21], migration [8] and checkpointing. However, the adoption of such services in modern cloud infrastructures relies heavily on the willingness of cloud service providers to deploy them. Clients have little say in the deployment or configuration of these services. It is also not clear that a "one size fits all" configuration of these services will be acceptable to client VMs. For example, a simple cloud-based security service that checks network packets for malicious content using signatures will not be useful to a client VM that receives encrypted packets. The client VM may require deeper introspection techniques (*e.g.,* to detect rootkits), which it cannot deploy on its own. Even if the cloud provider offers such an introspection service, the client may be reluctant to use it because dom0's ability to inspect its VMs may compromise its privacy.

1.2 Self-service Cloud Computing

The main contribution of this dissertation is a new ***Self-service cloud (SSC) computing*** model that simultaneously addresses the problems of malicious insiders and inflexible control. SSC consists of two components, a commodity hypervisor with modified privilege model, named ***SSC hypervisor***, and software stack to manage the systems equipped with such hypervisor, named ***SSC control plane***.

1.2.1 SSC Hypervisor

SSC introduces a novel hypervisor privilege model that partitions the responsibilities traditionally entrusted with dom0 to two new kinds of administrative domains. *Sdom0*, short for System-dom0, manages the resources of the physical platform, schedules VMs for execution, and manages I/O quotas. Sdom0 also executes the device drivers that control the physical hardware of the machine. Each physical platform is equipped with one Sdom0 instance.

Each client gets its own *Udom0* instance, short for User-dom0, which the client can use to monitor and control its VMs executing on that physical platform. While Sdom0 holds the privileges to pause/unpause client VMs, access their read-only state (*e.g.,* number of vCPUs assigned or their RAM allocation), and manage their virtual I/O operations, the hypervisor disallows Sdom0 from mapping the memory and vCPU registers of any of the client's VMs (*i.e.,* Udom0, UdomUs and SDs, introduced below).

Aside from these administrative domains, each platform also hosts the client's work VMs, called *UdomUs*. Each client's Udom0 has the privileges to administer that client's UdomUs. Although Udom0 has the privileges to perform the aforementioned tasks, SSC's design aims to keep Udom0 stateless for the most part. Thus, SSC also supports *service domains (SDs)*, which can perform these administrative tasks on client VMs. We use the term ***meta-domain*** to refer to the collection of a client's domains (Udom0, UdomUs and SDs).

The final component of SSC is a platform-wide *domain builder (domB)*. The sole responsibility of domB is to create VMs in response to client requests. Building a VM on a platform requires accessing the VM's memory and registers onto the physical platform. Because this task is privacy-sensitive, it cannot be performed by Sdom0 although it involves mapping the client

VM's state onto the physical platform. The TCB therefore includes domB, which is entrusted with this responsibility. DomB also interacts with the TPM and hosts the code to virtualize the TPM for individual clients [4].

1.2.2 SSC Control Plane

SSC control plane is a distributed system for managing physical platforms equipped with a SSC hypervisor. It facilitates the interaction between hosts in the cloud infrastructure as well as between the client and the cloud. It presents a unified administrative interface to clients, and transparently manages all of a client's VMs running across different hosts on the cloud. Simultaneously, it also protects clients from the cloud by preserving SSC's twin objectives of security/privacy and flexible VM control.

The control plane has three components, a *cloud controller*, a *node controller* and a *dashboard*. *Cloud controller* has a global view of the provider's infrastructure, and is tasked with allocating resources to clients elastically, provisioning and scheduling virtual machines. In SSC, the Sdom0 on each physical platform runs an instance of *node controller*, which is tasked to monitor the resource utilization on the host and control VM scheduling. *Dashboard* acts as a communication layer between the client and the cloud.

SSC enables new services for the clients VMs, which introduces dependencies among the VMs. The control plane provides the ability to specify such relationships among VMs. Clients can use the *dashboard* to provide VM specifications along with inter-VM relationship specifications and the *cloud controller* produces a VM placement satisfying these specifications. The *cloud controller* is also responsible for cloud wide maintenance operations such as VM migrations. SSC's control plane also incorporates new VM migration protocols for migrating VMs having dependencies.

1.3 Benefits of SSC

This section presents additional potential benefits of SSC to the cloud providers and their customers.

1.3.1 Services

Virtualization, the underlying technology powering cloud computing, enables many new services, like VM introspection, checkpointing, I/O manipulation and deduplication, for the customers of the cloud providers. While virtualization enables these services, their availability on modern cloud infrastructures depends on the willingness of providers to deploy them. For example, Amazon's EC2 does not expose a checkpointing API to its clients. Likewise, cloud services do not currently allow clients to independently deploy "security domains" to monitor their VMs. The extent of protection of client VMs is limited to that provided by user-space anti-virus tools that can execute within VMs (in addition to firewalls or intrusion detection systems employed by the provider). Such tools are incapable of deeper introspection, as would be needed to detect kernel-level malware such as rootkits.

Even if cloud providers were to offer services such as the above, their implementation will likely conflict with client security and privacy. Services such as VM introspection, checkpointing and memory deduplication are privileged because they must access client state, such as its VM memory pages, virtual registers and configuration parameters. These services are therefore implemented in dom0. A malicious cloud administrator would be able to misuse these services and gain access to a trove of sensitive client data.

SSC gives clients the freedom to administer their own VMs, and deploy services that today require provider support. SSC provides clients with unprecedented flexibility to deploy customized cloud-based services and allows clients to administer their own VMs. However, this does not necessarily mean that clients need to have increased technical know-how or manpower to leverage the benefits of SSC. Clients can use services developed by third party vendors [55].

1.3.2 Mutual Trust

Cloud computing allows even small enterprises access to vast amounts of computing power. Procuring and maintaining tens of thousands of servers would be difficult for most organizations but renting servers on-demand from a cloud computing provides much more affordable alternative. However, this conveniently available computing power can be abused by malicious attackers to stage distributed denial of service attacks, serve malwares or distribute illegal contents. Cloud Security Alliance has identified abuse of cloud services as one of top threats to cloud computing [9]. Indeed, there were real cases of distributed denial of service attacks and spamming attacks using Amazon EC2 infrastructure and recently Nvidia's try grid service was misused by miners to mine virtual currencies like Bitcoin [48].

Cloud providers usually mitigate such abuse by inspecting the client VMs but such accesses may conflict with the client's privacy goals. There is often such a tension between the client's privacy policies and the cloud provider's need to retain control over client VMs executing on its platform. SSC introduces notion of mutual trust to resolve the tension between client's privacy policies and provider's need to retain control over the software executing on its platform.

SSC introduces ***mutually-trusted service domains (MTSDs)***, mutually trusted by the clients and the provider. The cloud provider and the client agree upon policies and mechanisms that the provider will use to control the client's VMs. The cloud provider implements its code in a MTSD, which runs similar to a SD, and can therefore inspect a client's VMs. Clients can leverage trusted computing technology [4,25,29] to verify that a MTSD only runs code that was mutually agreed-upon with the cloud provider. Clients that have verified the trustworthiness of the platform and the MTSD can rest assured that their privacy will not be compromised. Likewise, the cloud provider can ensure liveness of MTSDs for regulatory compliance.

1.4 Threat Model

SSC's threat model is similar to those used in recent work on protecting client VMs in the cloud [31, 51, 68], and differentiates between *cloud service providers* and *cloud system administrators*. Cloud providers, such as Amazon EC2 and Microsoft Azure, have a vested interest in protecting their reputations. On the other hand, cloud system administrators are individuals entrusted with performing system-level tasks and maintaining the cloud infrastructure. To do so, they need access to dom0 and the privileges that it entails.

We assume that *cloud system administrators are adversarial* (or could make mistakes), and by extension, that the *administrative domain is untrusted*. Administrators have both the technical means and the monetary motivation to misuse dom0's privileges to snoop client data at will. Even if system administrators are benign, attacks on client data can be launched via exploits directed against dom0. Such attacks are increasing in number [13–17, 28] because on commodity VMMs, dom0 often runs a full-fledged operating system with a complex software stack. Likewise, misconfigured services in dom0 can also pose a threat to the security and privacy of client data.

SSC protects clients from threats posed by exploits against Sdom0 and cloud administrators who misuse Sdom0's privileges. SSC prevents Sdom0 from accessing the memory contents of client VMs and the state of their virtual processors (vCPUs). This protects all of the client's in-memory data, including any encryption keys stored therein. SSC's core mechanisms by themselves do not prevent administrators from snooping on network traffic or persistent storage. Security-conscious clients can employ end-to-end encryption to protect data on the network and storage.

SSC assumes that *the cloud service provider is trusted*. The provider must supply a TCB running an SSC-compliant VMM. We assume that the physical hardware is equipped with an IOMMU and a Trusted Platform Module (TPM) chip. Using TPM clients can obtain cryptographic guarantees about the software stack executing on the machine. The cloud provider must also implement procedural controls (security guards, cameras, auditing procedures) to ensure the physical security of the cloud infrastructure in the data center. This is essential to prevent hardware-based attacks, such as cold-boot attacks, against which SSC cannot defend. SSC does not attempt to defend against denial-of-service attacks. Such attacks are trivial to launch in a cloud environment, *e.g.,* a malicious administrator can simply configure Sdom0 so that a client's VMs is never scheduled for execution, or power off the server running the VMs. Clients can ameliorate the impact of such attacks via off-site replication. Finally, SSC does not aim to defend against subpoenas and other judicial instruments served to the cloud provider to monitor specific clients.

1.5 Summary of Contributions

The thesis this dissertation supports is:

It is possible to improve the level of security, privacy and control that cloud clients have over their code and data by modifying the hypervisor's privilege model.

This dissertation supports the above thesis statement by presenting one such model *i.e.,*Self-Service Cloud computing. We demonstrate its utility via a number of case studies and develop the supporting infrastructure to make SSC a viable alternative to today's cloud platforms.

The following list iterates the contributions of this dissertation in more detail:

- ***The SSC hypervisor.*** We have designed and implemented a new hypervisor privilege model that allows clients to administer their own VMs, while disallowing the cloud's administrative domain from inspecting client VM state. It enables clients to deploy new and customized services for their VMs and protects clients VMs from malicious cloud operators.

- ***The SSC control plane.*** We have designed and implemented software stack to manage systems equipped with SSC hypervisors. The novel features of the control plane include giving clients the ability to specify inter-VM dependencies and new VM migration protocols.

- ***Mutually-trusted service domains.*** SSC introduces the notion of mutual trust between clients and providers. We have implemented mutually trusted service domains that check regulatory compliance in a manner that is mutually agreed upon between the cloud provider and the client.

- ***Service domains.*** We have demonstrated the benefits of SSC by implementing and evaluating several VM services as SDs, which were traditionally implemented in dom0.

1.6 Dissertation Organization

This dissertation is organized as follows. Chapter 2 presents the design and implementation of the SSC privilege model while Chapter 3 presents SSC control plane in detail. The VM services we implemented to show the benefits of SSC are described in detail in Chapter 4 along with their evaluations. In the same chapter, we also included the VM migration evaluations as well. Chapter 5 overviews the related work and finally, Chapter 6 concludes the dissertation and discusses possible future directions.

1.7 Statement of Contributions

The SSC project is the result of joint collaboration between Rutgers University and AT&T Labs Research. From Rutgers, my dissertation adviser Professor Vinod Ganapathy and from AT&T Labs, my mentors, Dr. H. Andrés Lagar-Cavilla and Dr. Abhinav Srivastava contributed in the design, implementation and evaluation of the SSC project. Specifically Dr. Lagar-Cavilla provided the source code for Patagonix [35] and implemented the memory deduplication service while Dr. Srivastava provided the source code for VMWall [56].

Chapter 2

The SSC Hypervisor

This chapter describes the design and implementation of the SSC hypervisor, focusing on the new abstractions, their operations, and the privilege model.

2.1 Components

As Figure 2.1 shows, an SSC platform has a single system-wide administrative domain (Sdom0) and a domain-building domain (domB). Each client has its own administrative domain (Udom0), which is the focal point of privilege and authority for a client's VMs. Udom0 orchestrates the creation of UdomUs to perform client computations, and SDs, to which it delegates specific privileges over UdomUs. SSC hypervisor prevents Sdom0 from inspecting the contents of client meta-domains. The implementation of SSC hypervisor is based on Xen. More specifically SSC hypervisor is Xen with modified privilege model.

One of the main contributions of the SSC model is that it splits the TCB of the cloud infrastructure in two parts, a ***system-level TCB***, which consists of the hypervisor, domB, BIOS and the bootloader, and is controlled by the cloud provider, and a ***client-level TCB***, which consists of the client's Udom0, SDs, and MTSDs. Clients can verify the integrity of the system-level TCB using trusted hardware. They are responsible for the integrity of their client-level TCBs. Any compromise of a client-level TCB only affects that client.

Sdom0 runs all device drivers that perform actual I/O and wields authority over scheduling and allocation decisions. Although these privileges allow Sdom0 to perform denial-of-service attacks, such attacks are not in our threat model (Section 1.4); consequently, Sdom0 is not part of the TCB.

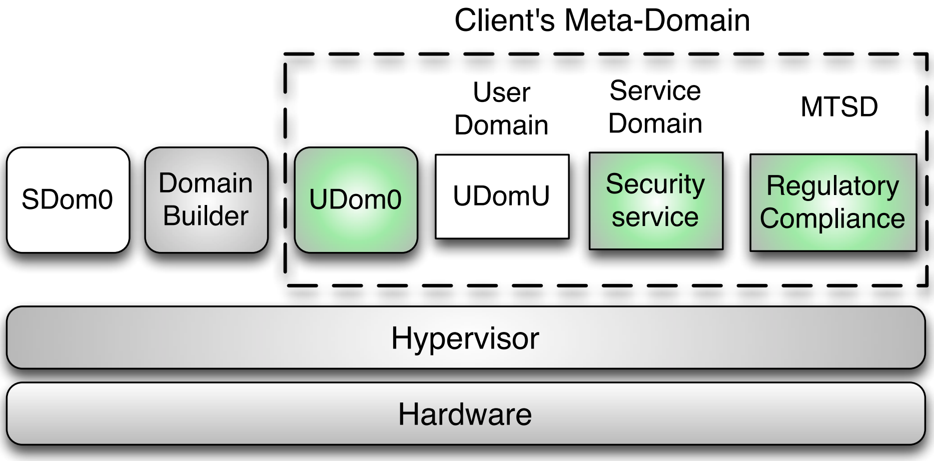


Figure 2.1: **The design of a self-service cloud (SSC) computing platform.** SSC splits the TCB of the system (indicated using the shaded components) into a ***system-level TCB***, with the hardware, the SSC hypervisor, and the domain builder, and a ***client-level TCB***, with the Udom0 and service domains.

The components of SSC must be able to communicate with each other for tasks such as domain creation and delegating privileges. In our prototype, VMs communicate using traditional TCP/IP sockets. However, domB receives directives for domain creation through hypervisor-forwarded hypercalls (see Figure 2.2 and Figure 2.3). Images of domains to be created are passed by attaching storage volumes containing this information.

2.2 Bootstrapping

Hosts in the cloud infrastructure are assumed to be equipped with TPM and IOMMU hardware, which is available on most modern chipsets. DomB, a part of TCB, interacts with the hardware TPM and also contains the components to virtualize TPM. The protocols described in this section assume client interaction with a vTPM instance. We use the vTPM protocols as described in the original paper [4], although it may also be possible to use recently-proposed variants [18].

During system boot, the BIOS passes control to a bootloader, and initializes the hardware TPM's measurement. In turn, the bootloader loads our modified version of the Xen hypervisor, Sdom0's kernel and ramdisk, and domB's kernel and ramdisk. It also adds entries for the

- Create_Udom0 (backend_id, nonce, Enc_Params, SigClient)
  Description: This hypercall is issued by Sdom0 to initiate a client meta-domain by creating a Udom0. The backend_id argument is a handle to a block device provided by Sdom0 to the client to pass Udom0 kernel image, ramdisk and configuration to domB. The nonce supplied by the client is combined with the vTPM's measurement list, which is returned to the client for verification following domain creation. Enc_Params denotes a set of parameters that are encrypted under the vTPM's AIK public key. SigClient is the client's digital signature of key parameters to the Create_Udom0 call. These parameters are used by the protocol in Figure 2.3(a) to bootstrap a secure communication channel with the client after Udom0 creation.
- Create_Userdomain (backend_id, nonce)
  Description: Issued by Udom0 to provide VM images of SDs or UdomUs to domB. The parameters backend_id and nonce are as described above.
- Create_MTSD (client_id, backend_id, nonce_provider, nonce_client, privilege_list)
  Description: Sdom0 uses this hypercall to start an MTSD within a client's meta-domain. The configuration parameters, which are included in the block device specified by backend_id, contain the command-line arguments used to initiate the service provided by the MTSD. MTSDs are also assigned specific privileges over UdomUs in the client meta-domain. This hypercall returns an identifier for the newly-created MTSD. It also returns two signed vTPM measurements, each appended with the nonces of the provider and the client.
- Grant_Privilege (SD_id, UdomU_id, privilege_list)
  Description: This hypercall is used by Udom0s to delegate specific privileges to a SD over an UdomU. Udom0s can issue this hypercall only on SDs and UdomUs within their own meta-domain.

Figure 2.2: **Summary of new hypercalls introduced to enable SSC.** Figure 2.3 shows their usage.

hypervisor and domB to the measurement stored in the TPM's PCR registers. The hypervisor then builds Sdom0 and domB. Finally, it programs the IOMMU to restrict Sdom0's access to only the pages that Sdom0 owns. Following bootstrap and initialization, the hypervisor unpauses Sdom0 and schedules it for execution. Sdom0 then unpauses domB, which awaits client requests to initialize meta-domains. SSC forbids Sdom0 from directly interacting with the TPM; all TPM operations (both with the hardware TPM and vTPM instances) happen via domB.

Sdom0 starts the XenStore service, which is a database used traditionally by Xen to maintain information about virtual device configuration. Each user VM on the system is assigned its own subtree in XenStore with its virtual device configurations.

(a) Protocol for Udom0 creation (initializing a new meta-domain) and bootstrapping an SSL communication channel

1. client → Sdom0	:	n_{TPM}, Udom0_image, $\mathsf{Enc}_{\mathsf{AIK}}(\mathsf{freshSym} \| n_{\mathrm{SSL}} \| \mathsf{hash}(\text{Udom0\_image}))$, Sig_{client}
2. Sdom0 → domB	:	CREATE_UDOM0(Udom0_image, n_{TPM}, $\mathsf{Enc}_{\mathsf{AIK}}(\mathsf{freshSym} \| n_{\mathrm{SSL}} \| \mathsf{hash}(\text{Udom0\_image}))$, Sig_{client}) → ID$_{client}$
3. domB → client	:	ID$_{client}$, $\mathsf{TPMSign}(n_{\mathrm{TPM}} \| \mathsf{PCR})$, ML
4. domB → Sdom0	:	Unpause Udom0 (denoted by ID$_{client}$) and schedule it for execution
5. Udom0 → client	:	n_{SSL}
6. client → Udom0	:	$\mathsf{Enc}_{\mathsf{freshSym}}(\mathsf{SSLpriv})$

Notes: In step 1, Udom0_image is passed via a block device provided by Sdom0 to the client. The key AIK denotes the public part of the vTPM's AIK (attestation identity key), freshSym is a fresh symmetric key chosen by the client, and Sig$_{client}$ is the digital signature, under the client's private key, of $\mathsf{freshSym} \| n_{\mathrm{SSL}} \| \mathsf{hash}(\text{Udom0\_image}) \| n_{\mathrm{TPM}}$. In step 2, when domB executes CREATE_UDOM0, it requests the vTPM to decrypt $\mathsf{Enc}_{\mathsf{AIK}}(\ldots)$, checks the hash of Udom0_image, verifies the client's digital signature Sig$_{client}$, and places freshSym and n_{SSL} into Udom0's memory. In step 3, ML denotes the measurement list, while PCR denotes the content of the vTPM's platform control register (storing the measurements); $\mathsf{TPMSign}(\ldots)$ denotes that the corresponding content is signed with the private part of the vTPM's AIK key. ID$_{client}$ is a unique identifier assigned to the newly created Udom0 (and meta-domain). In steps 5 and 6, Udom0 interacts with the client, who sends it the SSL private key (denoted by SSLpriv) encrypted under freshSym. Udom0 decrypts this to obtain SSLpriv, which is then used for all future SSL-based communication with the client.

(b) Protocol for UdomU and SD creation

1. client → Udom0	:	n_{client}, VM_image (*this message is sent via SSL*)
2. Udom0 → domB	:	CREATE_USERDOMAIN(VM_image, n_{client}) → ID$_{VM}$
3. domB → Udom0	:	ID$_{VM}$, $\mathsf{TPMSign}(n_{client} \| \mathsf{PCR})$, ML
4. Udom0	:	GRANT_PRIVILEGE(ID$_{VM}$, ID$_{UdomU}$, SD_privileges) (*this step is necessary only for VMs that are SDs*)
5. domB → Sdom0	:	Unpause ID$_{VM}$ and schedule it for execution

(c) Protocol for MTSD creation

1. Udom0 → Sdom0	:	n_{client}, identifier of the MTSD to be installed (VM image resides with provider)
2. Sdom0 → domB	:	CREATE_MTSD(ID$_{client}$, MTSD_image, $n_{provider}$, n_{client}, MTSD_privileges) → ID$_{MTSD}$
3. domB → Sdom0	:	ID$_{MTSD}$, $\mathsf{TPMSign}(n_{provider} \| \mathsf{PCR})$, ML
4. domB → Udom0	:	ID$_{MTSD}$, $\mathsf{TPMSign}(n_{client} \| \mathsf{PCR})$, ML
5. domB → Sdom0	:	Unpause ID$_{MTSD}$ and schedule it for execution

Notes: In step 2, ID$_{client}$ is the meta-domain identifier obtained during Udom0 creation.

Figure 2.3: **Protocols used in SSC for the creation of Udom0, UdomUs, SDs and MTSDs.**

2.3 Building Client Meta-Domains

In SSC, domB receives and processes all requests to create new domains, including Udom0s, UdomUs, SDs, and MTSDs. Client requests to start new meta-domains are forwarded to domB from Sdom0. In response, domB creates a Udom0, which handles creation of the rest of the meta-domain by itself sending more requests to domB (*e.g.,* to create SDs and UdomUs). To allow clients to verify that their domains were built properly, domB integrates domain building with standard vTPM-based attestation protocols developed in prior work [4, 50].

Udom0

Upon receiving a client request to create a new meta-domain, Sdom0 issues the CREATE_UDOM0 hypercall containing a handle to the new domain's bootstrap modules (kernel image, ramdisk, *etc*). DomB builds the domain and returns to the client an identifier of the newly-created meta-domain. In more detail, the construction of a new meta-domain follows the protocol shown in Figure 2.3(a). This protocol achieves two security goals:

1. *Verified boot of Udom0.* At the end of the protocol, the client can verify that the Udom0 booted by the SSC platform corresponds to the image supplied in step 1 of Figure 2.3(a). To achieve this goal, in step 1, the client supplies a challenge (n_{TPM}) and also provides hash(Udom0_image), encrypted under the vTPM's public key (AIK). These arguments are passed to domB, as part of the CREATE_UDOM0 hypercall in step 2. In turn, DomB requests the vTPM to decrypt the content enciphered under its public key, thereby obtaining hash(Udom0_image). DomB then creates the domain after verifying the integrity of the VM image (using hash(Udom0_image) and Sig_{client}), thereby ensuring that Sdom0 has not maliciously altered the VM image supplied by the client. It then returns to the client an identifier of the newly-created meta-domain, a digitally-signed measurement from the vTPM (containing the contents of the vTPM's PCR registers and the client's challenge) and the measurement list. The client can use this to verify that the domain booted with the expected configuration parameters.

2. *Bootstrapping SSL channel with client.* In SSC, the network driver is controlled by

Sdom0, which is untrusted, and can eavesdrop on any cleartext messages transmitted over the network. Therefore, the protocol in Figure 2.3(a) also interacts with the client to install an SSL private key within the newly-created Udom0. This SSL private key is used to authenticate Udom0 during the SSL handshake with the client, and helps bootstrap an encrypted channel that will then be used for all further communication with the client.

Installation of the SSL private key proceeds as follows. In step 1, the client supplies a fresh symmetric key (freshSym), and a nonce (n_{SSL}), both encrypted under the vTPM's public key. In step 2, domB creates Udom0 after checking the integrity of the Udom0 image (using Sig_{client}). When domB creates Udom0, it requests the vTPM to decrypt this content, and places freshSym and n_{SSL} in Udom0's memory, where SSC's privilege model prevents them from being accessed by Sdom0. Recall from Section 2.2 that Sdom0 cannot directly access the TPM or vTPM (only domB can do so), and therefore cannot obtain the value of freshSym. In step 5, Udom0 sends n_{SSL} to the client, which responds in step 6 with the SSL private key encrypted under freshSym. Udom0 can now decrypt this message to obtain the SSL private key. Assuming that both freshSym and n_{SSL} are random and generated afresh, the protocol allows the client to detect replay attempts.

This protocol significantly restricts the power of *evil twin attacks* launched by a malicious Sdom0. In such an attack, Sdom0 would coerce domB to create a malicious Udom0 domain, and trick the client into installing its SSL private key within this domain. This malicious domain would then transfer the SSL private key to Sdom0, thereby compromising client confidentiality. In our protocol, domB checks the integrity of Udom0_image before booting the domain, thereby ensuring that the only "evil" twin that Sdom0 can create will have the same VM image as supplied by the client. Sdom0 therefore cannot include arbitrary malicious functionality in the evil twin (*e.g.,* code to transmit secret keys to it) without being detected by the client. Further, SSC's privilege model prevents Sdom0 from directly inspecting the memory of the twin VM, thereby protecting the the value of freshSym that is installed in it during creation. Finally, steps 5 and 6 of the protocol detect replay attempts, thereby ensuring that even if a twin VM is created, exactly one of the twins can interact with the client to obtain its SSL private key. This twin VM then becomes the Udom0 of the client's meta-domain, while the other twin can no longer interact with the client.

UdomUs and SDs

Udom0 accepts and processes client requests to start UdomUs and SDs. Clients establish an SSL connection with Udom0, and transmit the kernel and ramdisk images of the new domain to Udom0. Udom0 forwards this request to domB, which then builds the domain. See Figure 2.3(b).

We aim for Udom0s and SDs to be stateless. They perform specialized tasks, and do not need persistent state for these tasks. The lack of persistent state eases the clients' task of verifying the integrity of these domains (*e.g.,* via inspection of their code), thereby minimizing risk even if they are compromised via attacks directed against them. The lack of state also allows easy recovery upon compromise; they can simply be restarted [11]. In our design, we do not assign persistent storage to SDs. They are neither extensible nor are they allowed to load kernel modules or extensions outside of the initial configuration. All relevant configuration values are passed via command line parameters. This design does require greater management effort on the part of clients, but is to be expected in SSC, because it shifts control from the provider to clients.

We have implemented SDs and Udom0s in our prototype using a carefully-configured paravirtualized Linux kernel; they only use ramdisks. The file system contains binaries, static configuration and temporary storage. SSC elides any unnecessary functionality in SDs and Udom0s to minimize their attack surface. Udom0s in our prototype integrates a replica of the `libxl` to provide an administrative interface. It may be possible to reduce the size of the client-level TCB using a simpler software stack (*e.g.,* based on Mini-OS, which is part of the Xen distribution). However, we have not done so in our current prototype.

MTSDs

Like SDs, each MTSD belongs to a client meta-domain. MTSDs can be given specific privileges (via the CREATE_MTSD hypercall) to map the state of client VMs, checkpoint, fingerprint, or introspect them. This allows the cloud provider to inspect client domains for regulatory compliance. Section 2.6 discusses regulatory compliance with MTSDs in further detail.

Both the cloud provider and client cooperate to start the MTSD, as shown in the protocol in Figure 2.3(c). The client initiates the protocol after it has agreed to start the MTSD in its meta-domain. DomB creates the MTSD, and both the provider and the client can each ensure that the MTSD was initialized properly using signed measurements from the vTPM. The provider or the client can terminate the protocol at this point if they find that the MTSD has been tampered with.

2.4 SSC Privilege Model

At the heart of SSC, there is a new privilege model enforced by the hypervisor. This model enables clients to administer their own VMs securely, without allowing cloud administrators to eavesdrop on their data. For purposes of exposition, we broadly categorize the privileged operations performed by a VMM into six groups.

1. ***VM control operations*** include pausing/unpausing, scheduling, and destroying VMs.

2. ***Privacy-sensitive operations*** allow the mapping of memory and virtual CPU registers of a VM.

3. ***Read-only operations*** expose non-private information of a VM to a requester, including the number of vCPUs and RAM allocation of a VM, and the physical parameters of the host.

4. ***Build-only operations*** include privacy-sensitive operations and certain operations that are only used during VM initialization.

5. ***Virtual I/O operations*** set up event channels and grant tables to share memory and notifications in a controlled way for I/O.

6. ***Platform configurations*** manage the physical host. Examples of these operations include programming the interrupt controller or clock sources.

	Sdom0	**domB**	**Udom0**	**SD/MTSD**
VM control (**C**)	✓		✓	✓
Privacy-sensitive (**P**)			✓	✓
Read-only (**R**)	✓		✓	✓
Build-only (**B**)		✓		
Virtual I/O (**I**)	✓		✓	✓
Platform config. (**L**)	✓			

Figure 2.4: **Actors and operations in the privilege model.** Each ✓ in the table denotes that the actor can perform the corresponding operation.

In addition to these operations, VMMs also perform hardware device administration that assigns PCI devices and interrupts to different VMs. We expect that hardware device administration may rarely be used in a dynamic cloud environment, where VM checkpointing and migration are commonplace, and leave for future work the inclusion of such operations in the SSC privilege model.

In SSC, Sdom0 has the privileges to perform VM control, read-only, virtual I/O and platform operations. VM control operations allow VMs to be provisioned for execution on physical hardware, and it is unreasonable to prevent Sdom0 from performing these tasks. A malicious system administrator can misuse VM control operations to launch denial-of-service attacks, but we exclude such attacks from our threat model. Sdom0 retains the privileges to access read-only data of client VMs for elementary management operations, *e.g.,* listing the set of VMs executing in a client meta-domain. Sdom0 executes backend drivers for virtual devices and must therefore retain the privileges to perform virtual I/O operations for all domains on the system. As discussed earlier, SSC also admits the notion of driver domains, where device drivers execute within separate VMs [34]. In such cases, only the driver domains need to retain privileges to perform virtual I/O. Finally, Sdom0 must be able to control and configure physical hardware, and therefore retains privileges to perform platform operations.

The domain builder (domB) performs build-only operations. Building domains necessarily involves some operations that are categorized as privacy-sensitive, and therefore includes them. However, when domB issues a hypercall on a target domain, the hypervisor first checks that the domain has not yet accrued a single cycle (*i.e.,* it is still being built), and allows the hypercall to succeed only if that is the case. This prevents domB from performing privacy-sensitive

	Sdom0	**domB**	**Udom0**	**SD**	**MTSD**
Hardware	**L**				
Sdom0					
domB	**C,R,I**		**I**		
Udom0	**C,R,I**	**B**			
SD	**C,R,I**	**B**	**C,P,R,I**	**C,P,R,I**	**C,P,R,I**
MTSD	**C,R,I**	**B**	**R,I**	**R,I**	**R,I**
UdomU	**C,R,I**	**B**	**C,P,R,I**	**C,P,R,I**	**C,P,R,I**

Figure 2.5: **Actors, objects, and operations in the privilege model.** Each column denotes an actor that performs an operation, while each row denotes the object upon which the operation is performed. Operations are abbreviated as shown in Figure 2.4.

operations on client VMs after they have been built.

Udom0 can perform privacy-sensitive and read-only operations on VMs in its meta-domain. It can also perform limited VM control and virtual I/O operations. Udom0 can pause/unpause and destroy VMs in its meta-domain, but *cannot* control scheduling (this privilege rests with Sdom0). Udom0 can perform virtual I/O operations for UdomUs in its meta-domain. Udom0 can delegate specific privileges to SDs and MTSDs as per their requirements. A key aspect of our privilege model is that it groups VMs by meta-domain. Operations performed by Udom0, SDs and MTSDs are restricted to their meta-domain. While Udom0 has privileges to perform the above operations on VMs in its meta-domain, it cannot perform VM control, privacy-sensitive, and virtual I/O operations on MTSDs executing in its meta-domain. This is because such operations will allow Udom0 to breach its contract with the cloud provider (*e.g.,* by pausing, modifying or terminating an MTSD that the Udom0 has agreed to execute). Figure 2.4 and Figure 2.5 summarize the privilege model of SSC.

We implemented this privilege model in our prototype using the Xen Security Modules (XSM) framework [49]. XSM places a set of hooks in the Xen hypervisor, and is a generic framework that can be used to implement a wide variety of security policies. Security policies can be specified as modules that are invoked when a hook is encountered at runtime. For example, XSM served as basis for IBM's sHype project, which extended Xen to enforce mandatory access control policies [49]. We implemented the privilege described in this section as an XSM policy module.

Although the privilege model described above suffices to implement a variety of services, it can possibly be refined to make it more fine-grained. For example, our privilege model can currently be used to allow or disallow an SD from inspecting UdomU memory. Once given the privilege to do so, the SD can inspect arbitrary memory pages. However, it may also be useful to restrict the SD to view/modify specific memory pages, *e.g.,* on a per-process granularity, or view kernel memory pages alone. We plan to explore such extensions to the privilege model in future work.

2.5 Virtual I/O

In our SSC prototype, device drivers execute within Sdom0, thereby requiring clients to depend on Sdom0 to perform I/O on their behalf. Naïvely entrusting Sdom0 with I/O compromises client privacy. Our prototype protects clients via modifications to XenStore.

In Xen, domUs discover virtual devices during bootstrap using a service called XenStore, which runs as a daemon in dom0. Each domU on the system has a subtree in XenStore containing its virtual device configurations. Dom0 owns XenStore and has full access to it, while domUs only have access to their own subtrees.

In SSC, we modified XenStore allowing domB to create subtrees for newly-created VMs, and give each Udom0 access to the subtrees of all VMs in its meta-domain. Udom0 uses this privilege to customize the virtual devices for its UdomUs. For instance, it can configure a UdomU to use Sdom0 as the backend for virtual I/O. Alternatively, it can configure the UdomU to use an SD as a backend; the SD could modify the I/O stream (*e.g.,* a storage SD; see Figure 4.8). An SD can have Sdom0 as the backend, thereby ultimately directing I/O to physical hardware, or can itself have an SD as a backend, thereby allowing multiple SDs to be chained on the path from a UdomU to the I/O device. We also modified XenStore to allow Sdom0 and Udom0 to insert block devices into domB. This is used to transfer kernel and ramdisk images during domain building.

Xen traditionally uses a mechanism called *grant tables* for fine-grained control on virtual I/O. Grant tables are used when domUs communicate with the backend drivers in dom0. DomU uses grant tables to share a single page of its memory with dom0, which redeems the grant to access the page. The hypervisor enforces any access restrictions specified by domU, and does not even disclose the actual page number to dom0. SSC benefits from the grant tables mechanism in allowing meta-domains to ultimately connect to and communicate I/O payloads to their backend drivers in Sdom0. As long as these payloads are encrypted (*e.g.,* using an SD within the meta-domain), client privacy is protected.

Ultimately, Sdom0 is responsible for I/O operations by communicating with physical hardware. Malicious Sdom0s can misuse this privilege to enable a number of attacks. For example, a client's Udom0 attaches a virtual device via a handshake with Sdom0. Sdom0 can launch attacks by corrupting this handshake or firing spurious virtual interrupts. As long as client payloads are encrypted, none of these attacks will breach client privacy; they merely result in denial-of-service attacks.

A final possibility for attack is XenStore itself. In our prototype, XenStore resides within Sdom0, which can possibly leverage this fact implement a variety of denial of service attacks. (Note that even if XenStore is abused to connect client VMs to the wrong backend, grant tables prevent client payloads from being leaked to Sdom0). Techniques for XenStore protection have recently been developed in the Xoar project [11], and work by factoring XenStore into a separate domain (akin to domB). SSC can employ similar techniques, although we have not done so in our prototype.

2.6 Regulatory Compliance using MTSDs

As previously discussed, an MTSD executes within a client meta-domain. The MTSD can request specific privileges over client's VMs in this meta domain (via a manifest) to perform regulatory compliance checks. These privileges include access to a VM's memory pages, vCPU registers and I/O stream. For example, an MTSD to ensure that a client VM is not executing malicious code may request read access to the VM's memory and registers. The client can inspect the manifest to decide whether the requested privileges are acceptable to it, and then start the MTSD. The privileges requested in the manifest are directly translated into parameters for the CREATE_MTSD hypercall. Both the client and the provider can verify that the MTSD was started with the privileges specified in the manifest.

Clients may wish to ensure that the MTSD's functionality does not compromise their privacy. For example, the client may want to check that an MTSD that reads its VM memory pages does not inadvertently leak the contents of these pages. One way to achieve this goal is to inspect the code of the MTSD to ensure the absence of such undesirable functionality. However, we cannot reasonably expect most cloud clients to have the economic resources to conduct thorough and high-quality security evaluations of MTSDs.

In this dissertation, we took a more practical and pragmatic approach. We propose to outsource the verification of MTSDs to mutually trusted third party. The cloud provider can provide the source code of MTSDs to the third party to verify the absence of data leakage and maliciousness and will get the signed certificate. The clients of the provider will just need to check the validity of the MTSDs certificates. This process is similar to SSL certifications except that the data leakage detection is more complex and error prone. As part of future work we are planning to develop tools for easing the verification process. Another approach the cloud providers can take is to use the standard tools developed by third parties instead of developing MTSDs themselves. For example there are different malware detection products developed by Symantec and McAfee. The provider can deploy such products as MTSDs.

Chapter 3

The SSC Control Plane

This chapter presents the design and implementation of SSC's control plane. We present the motivation behind SSC's control plane, its components and their operations.

3.1 Motivation

The control plane of traditional cloud platforms is responsible for monitoring resource usage of individual hosts, and suitably placing or migrating client VMs using the cloud provider's load balancing policies. It is also responsible for deciding where any client-chosen services offered by the cloud provider (*e.g.,* an intrusion detection service) will be placed on the network. Examples of such software include VMWare's vCloud, Amazon Web Services, OpenStack, OpenNebula, CloudStack, and Eucalyptus.

The control plane of an SSC-based cloud platform must have two key features:

1. ***Allowing clients to specify inter-VM dependencies.*** By introducing per-client administrative domains (Udom0s), SSC gives clients the ability to create SDs that can have specific administrative privileges over UdomUs, and the ability to interpose on their I/O paths. Thus, clients must be able to specify how the VMs that they propose to create are inter-related. For example, a client may wish to specify that an SD that he creates has the privileges to map the memory of one of its UdomUs. A consequence of this dependency is that these two VMs, *i.e.,* the SD and the UdomU, must reside on the same physical host. Likewise, the client can specify that an SD is to serve as the network middlebox for a collection of UdomUs. The control plane must respect this dependency in placing VMs, ensuring that all outgoing and incoming network connections to any UdomU in that collection will first traverse the SD. This invariant must be maintained even if any of

the UdomUs or the SD itself are migrated.

While traditional control planes do support middleboxes, the key difference is that these middleboxes are offered as services by the cloud provider. Both client-defined middleboxes and privileged SDs are unique to the SSC platform. The control plane of an SSC platform must be aware of and enforce these dependencies during execution.

2. ***Presenting a unified administrative interface to the client.*** On SSC, each platform that hosts a client VM must also have a Udom0 instance executing on it. This is because the Udom0 hosts the client's SSL private keys, which are required for the SSL handshake with the client, as described in Section 2.3.

   Naïvely using traditional control plane software with such a setup can lead to security holes. For example, a client may strategically create VMs in a manner that forces the cloud provider to place them on different physical hosts *e.g.,* by creating a large number of VMs that place large resource demands on individual hosts. The client could leverage the fact that there is a Udom0 instance on each physical platform that executes at least one of the client's VMs to estimate the number of physical hosts in the cloud, or to map the cloud provider's network topology.

   SSC's control plane must provide the illusion of a single administrative interface to the client, while hiding the presence of individual Udom0 instances. The control plane must suitably relay administrative operations from this interface to the corresponding Udom0 instances. It is also responsible for transparently handling VM migrations across hosts.

3.2 Traditional Control Plane

The control plane of most traditional cloud platforms has three key components. First is a platform-wide *cloud controller*. This component has a global view of the provider's infrastructure, and is tasked with allocating resources to clients elastically, provisioning and scheduling virtual machines. Cloud providers may choose to partition their infrastructure into multiple zones for fault tolerance and scalability, and execute multiple instances of cloud controllers in each zone. Second is a per-host *node controller*. This component typically executes as a daemon within dom0, and interacts with the hypervisor on the platform to monitor local resource consumption, and reports these statistics to the cloud controller. The third is a *dashboard*, which is the interface via which clients interact with the cloud. Each client's dashboard instance reports the state of the client's slice of the cloud, *e.g.,* the number of VMs executing, the resources consumed, and the current bill.

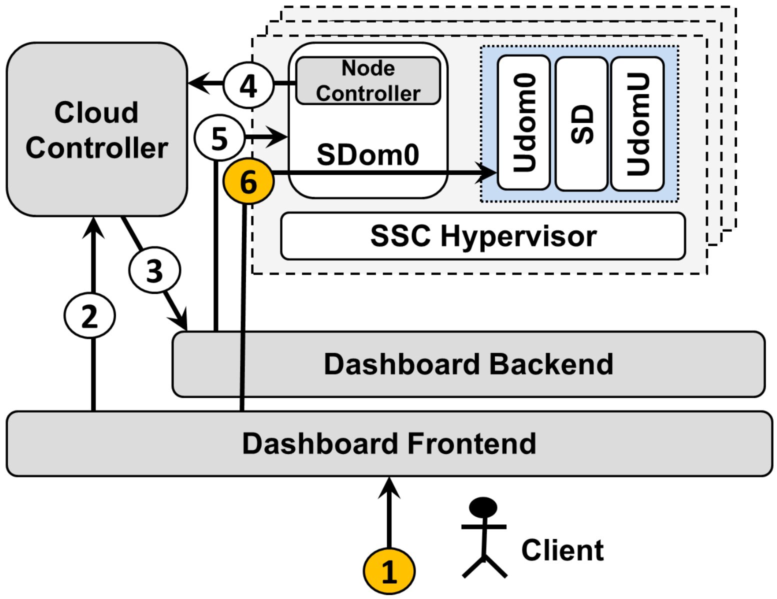
3.3 SSC's Control Plane

SSC's control plane functionally enhances each of the components discussed in previous section and introduces new protocols for inter-component communication. It introduces new client-centric features, such as the ability to specify relationships among VMs and manage client-deployed middleboxes. From the cloud provider's perspective, the new features include VM migration protocols, and the ability to provision resources while respecting client VM dependencies. Figure 3.1 summarizes the components of the control plane, and their interactions.

1. ***Cloud Controller.*** SSC allows clients to specify inter-VM dependencies (discussed in more detail in Section 3.5). These dependencies may imply that certain VMs must be colocated on the same physical host. They may also specify how the I/O path of a client's work VM must be routed through the cloud. For example, an SD that serves as the backend for a UdomU must lie on the I/O path of that UdomU, irrespective of the machines on which the SD and UdomU are scheduled for execution.

   The cloud controller must accept these specifications from the client (via the dashboard) and produce a VM placement satisfying these specifications. In doing so, it has to account for the current load on various hosts on the network and the resource requirements of the client's VMs. The cloud controller's scheduler therefore solves a constraint satisfaction problem, and produces a placement decision that is then communicated back to the dashboard. The dashboard interacts with individual hosts to schedule VMs for execution.

   The cloud controller initiates VM migrations. Based upon the resource usage information received from node controllers, the cloud controller may decide that a client's VMs need



(1) Clients interact with the dashboard frontend to specify VM dependencies and to provide VM images to the cloud. All client communication with the dashboard happens over SSL.
(2) The dashboard frontend communicates with the Cloud Controller and provides it with inter-VM dependencies and the VM configurations requested by the client.
(3) The Cloud Controller communicates with the dashboard backend to provide VM placements. This may be in response to client requests to create VMs or to initiate VM migration for load balancing.
(4) The Node Controller periodically communicates with the Cloud Controller to provide diagnostic information about individual cloud nodes.
(5) The dashboard backend communicates with the Sdom0s of individual hosts to create client meta-domains on the host (*i.e.,* the Udom0), as well as to initiate VM migration.
(6) Once a Udom0 is created on a host, the client communicates with it via the dashboard frontend. This happens via SSL, and is used to create new UdomUs and SDs as well as to configure and operate them. The client remains oblivious to the number of physical hosts used to execute its VMs, and is presented with the illusion of communicating with a single administrative interface. The dashboard routes requests to Udom0s of relevant hosts.

Figure 3.1: **Components of the control plane and their interactions.** Communications labeled with shaded circles are secured by SSL.

to be migrated for load balancing. It produces a new VM placement and communicates this to the dashboard. The dashboard then communicates with the source and target hosts to initiate the actual migration of the VMs.

The client never directly interacts with the cloud controller, nor does the cloud controller interact with the client's VMs. It simply produces VM placement decisions and communicates them with the dashboard. The client only trusts the cloud controller to produce a fair VM placement decision. Violation of this trust can potentially lead to denial of service attacks, which are outside SSC's threat model.

2. ***Node Controller.*** A node controller executes as a daemon within an individual platform's Sdom0. It can therefore monitor the resource utilization on the host and control VM scheduling, but *cannot* create user VMs (done by domB) or read/modify individual client VMs.

   The client never interacts directly with the node controller. In fact, as an entity that executes within the Sdom0, it is untrusted. The node controller cannot compromise the security of client VMs in any manner besides launching denial of service attacks by failing to schedule the VM for execution, or by reporting false resource utilization to the cloud controller, thereby triggering frequent VM migrations.

3. ***Dashboard.*** In SSC, the dashboard serves as the layer between the client and the cloud platform and has two responsibilities: to interact with the client, and to interface with various components of the cloud platform. Accordingly, we logically split the dashboard into a *frontend* and a *backend*, as shown in Figure 3.1.

   The dashboard frontend is an interface presented to the client via which it can enter VM images to be booted, and specify inter-VM dependencies. The frontend communicates any inter-VM dependencies specified by the client to the cloud controller, which uses this information to create a placement decision. However, the contents of the VM images itself are never passed to the cloud controller. The dashboard directly passes these images to the end hosts via an SSL channel (whose setup we discuss in Section 3.4).

   The dashboard backend orchestrates all communication between components of the cloud platform on behalf of the client. It obtains placement decisions from the cloud controller and transmits the client's VM images to the hosts on which they must be created. All communication between the dashboard and the end hosts is secured by the aforementioned SSL channel to protect it from any network snooping attacks launched

1. client $\rightarrow$ Sdom0 : n_{TPM}, $\mathsf{Enc}_{\mathsf{AIK}}(\mathsf{freshSym}||n_{\mathrm{SSL}})$, Sig_{client}
2. Sdom0 $\rightarrow$ domB : CREATE_DASHBOARD(n_{TPM},
$\mathsf{Enc}_{\mathsf{AIK}}(\mathsf{freshSym}||n_{\mathrm{SSL}})$, Sig_{client})
3. domB $\rightarrow$ client : $\mathsf{TPMSign}(n_{\mathrm{TPM}}||\mathsf{PCR})$, ML
4. domB $\rightarrow$ Sdom0 : Schedule the dashboard for execution
5. DashVM $\rightarrow$ client : n_{SSL}
6. client $\rightarrow$ DashVM : $\mathsf{Enc}_{\mathsf{freshSym}}(\mathsf{SSLpriv})$
7. client $\leftrightarrow$ DashVM : SSL handshake

Figure 3.2: **Protocol to create a dashboard VM instance.**

by malicious cloud operators.

Because the dashboard handles sensitive information (client VM images), it is part of the TCB. While it is possible to implement the dashboard in any manner that allows the client to trust it, we chose to implement the dashboard itself as a VM that executes atop an SSC hypervisor. We assume that the cloud provider will dedicate a set of physical hosts simply to execute dashboard VMs for its clients. We call this machine the *dashboard host*. Each new client gets a dashboard VM instance that executes as a Udom0 on the SSC hypervisor. This ensures that even the cloud operator on these physical hosts cannot tamper with the execution of the client's dashboard VM instance.

3.4 Operations of the SSC's Control Plane

A client begins its interaction with an SSC platform by first requesting a dashboard VM instance. During the creation of this VM instance, the client configures it so that the client can communicate with with the dashboard VM via SSL. It then provides VM images to this dashboard instance over the SSL channel, which then starts the VMs on physical cloud hosts on behalf of the client. In this section, we present and analyze the protocols that are used for these steps.

Dashboard Creation and Operation.

A client creates a dashboard VM instance using the protocol shown in Figure 3.2. It communicates with the "cloud provider," in this case the Sdom0 VM of the dashboard host. The first message of the protocol consists of a nonce (n_{TPM}), together with a piece of ciphertext

($\mathsf{freshSym}||n_{\mathrm{SSL}}$) encrypted using the AIK public key of the TPM of the dashboard host.[1] Here $\mathsf{freshSym}$ is a fresh symmetric key produced by the client, while n_{SSL} is a nonce; we explain their roles below. The client also digitally signs the entire message (denoted in Figure 3.2 as $\mathsf{Sig}_{\mathrm{client}}$).

The Sdom0 of the dashboard host communicates these parameters to its domB via the CREATE_DASHBOARD command, which is a new hypercall to the underlying hypervisor. This command instructs domB to use the VM image used by the cloud provider for dashboard VMs, and create an instance of the VM for the client. DomB does so after verifying the client's digital signature. In this step, domB also communicates with the TPM to decrypt the message encrypted under its AIK public key, and places $\mathsf{freshSym}$ and n_{SSL} in the newly-created dashboard VM instance. Because the dashboard host executes an SSC hypervisor, the contents of this dashboard VM instance are not accessible to Sdom0, which therefore cannot read $\mathsf{freshSym}$ and n_{SSL}.

At this point, the client can verify that the dashboard VM has been created correctly. DomB sends a digitally-signed measurement from the TPM, containing the contents of its PCR registers and n_{TPM}, together with the measurement list (as is standard in TPM-based attestation protocols [4, 50]). We assume that cloud provider will make the measurements of dashboard VM instances publicly-available. This is a reasonable assumption because the dashboard VM does not contain any proprietary code or data. All information proprietary to the cloud provider is in other components of the platform, such as the cloud controller. The dashboard VM simply encodes the protocols to interact with the client as well as with components of the cloud's control plane. Thus, the cloud provider can even make the code of the dashboard VM available for public audit.

In the last two steps of the protocol, the dashboard VM instance interacts with the client to obtain the private key portion of the client's SSL key pair ($\mathsf{SSLPriv}$). This key is used in the SSL handshake between the client and the dashboard VM, thereby allowing the establishment of an SSL channel between the two. The dashboard VM sends the nonce n_{SSL} to the client,

[1] We assume that the TPM is virtualized [4], and that the corresponding vTPM drivers execute in the dashboard host's domB. Thus, the AIK public key used in this message is that of the virtual TPM instance allocated to this client. We assume that the client interacts with the cloud out of band to obtain this AIK public key.

who sends in turn SSLPriv encrypted under freshSym, which is known only to the dashboard VM. This allows the dashboard VM to retrieve SSLPriv.

The protocol is designed to prevent attacks whereby a malicious cloud operator attempts to impersonate the client to start a dashboard VM instance as the client. The use of freshSym in the protocol ensures that the dashboard VM that is created has a secret value known only to the client. Two key features of SSC prevent the cloud operator from learning the value of freshSym: (a) the fact that the ciphertext sent in the first message can only be decrypted by the TPM (via domB), and (b) the fact that Sdom0 cannot obtain values stored in the memory of a dashboard VM instance. These features together allow the client to bootstrap the SSL handshake in a secure fashion. Finally, the nonces n_{TPM} and n_{SSL} prevent attempts by a malicious cloud provider to replay the protocol.

Creation and Operation of Client VMs.

Once the dashboard VM is set up using the protocol discussed above, the client can create its VMs. It provides these VM images via the SSL channel to the dashboard. However, to boot these VMs on a physical host, the dashboard VM must still communicate with the Sdom0 on that host, which is untrusted.

To protect client VMs, which may contain sensitive code and data, from untrusted Sdom0s, we require that the first client VM that is booted on a physical platform be its Udom0. This is also the case during cloud controller-initiated VM migration, where client VM instances are moved from a source host to a target host that does not have any of the client's VMs. As discussed in chapter 2, Udom0 is a stateless administrative interface for the client. It could therefore run a standard OS distribution, and not contain any sensitive client code or data. As a result, its image can be provided to the Sdom0 on the physical host. The dashboard does so on behalf of the client. The Udom0 can then accept VM images from the dashboard to create UdomUs and SDs on the physical host.

Note that the client never interacts directly with the Udom0. Rather, the dashboard VM serves as a trusted intermediary. The dashboard presents the client with the illusion of a unified administrative interface, regardless of the number of Udom0 instances executing on various physical hosts. The dashboard interfaces with each of the physical hosts to start VMs and verifies TPM measurements after the VMs boot. The only virtual TPM key that is exposed to the client is the AIK public key of the dashboard host.

3.5 Specifying Inter-VM Dependencies

Clients on an SSC platform can use SDs to implement middleboxes offering novel security and system services. These middleboxes can either hold specific privileges over the client's VMs, or serve as their I/O backends. SDs can also serve as I/O backends for other SDs, allowing services to be composed flexibly (Figure 3.5 presents an example).

SSC's control plane provides a language (Figure 3.3) for clients to specify such inter-VM dependencies. The client specifies these dependencies via the dashboard, which forwards it to the cloud controller. SSC's control plane allows two kinds of inter-VM dependencies:

1. **GRANT_PRIVILEGE dependencies.** This rule allows the client to specify that an SD must have specific privileges over a UdomU. These privileges may include mapping the user- or kernel-level memory of the UdomU, or reading vCPU registers. In the example program shown in Figure 3.4, *memscan_vm* is given the privileges to map the kernel memory of *webserver_vm*, *e.g.,* to detect rootkit infections.

   This dependency implicitly places a co-location constraint. The SD and the UdomU must be started on the same physical node. This is required because the operations to assign privilege (*e.g.,* mapping memory) are local to a node's hypervisor and can only be performed when both the SD and the UdomU run atop the same hypervisor.

2. **SET_BACKEND dependencies.** This rule specifies that one VM must serve as the I/O backend for another VM for a specific device type (we currently support storage and network devices). It also allows the client to specify whether the two VMs must be co-located (MUST_COLOCATE), or whether the cloud controller can possibly place them on different physical hosts (MAY_COLOCATE).

   For example, a network back-end SD that runs an encryption/decryption service for a client's UdomU must ideally be colocated on the same physical host as the UdomU. This

```
  Program := (Decl;)∗ (Init;)∗ (VMDep;)∗
     Decl := VM vm // vm is a variable, VM is a keyword
     Init := vm.name = String; vm.image = VM image
    VMDep := GrantRule | BackRule
GrantRule := GRANT_PRIVILEGE(sd, udomu, PrivType)
 BackRule := SET_BACKEND(backvm, frontvm, Device, Location)
 PrivType := USER_MEM | KERN_MEM | vCPU | FULL
   Device := STORAGE | NETWORK | . . .
 Location := MUST_COLOCATE | MAY_COLOCATE
```

Figure 3.3: **Language used to specify VM dependencies.**

```
VM webserver_vm; // Client's Web server
VM empDB_vm; // Client's employee database
VM memscan_vm; // Memory introspection SD
VM enc_vm; // SSL proxy for the employee DB
VM Snort_vm; // SD running the Snort NIDS

webserver_vm.name = “MyWebServer”;
webserver_vm.image = ApacheVM.img;
empDB_vm.name = . . .; empDB_vm.image = . . .;
memscan_vm.name = . . .; memscan_vm.image = . . .;
enc_vm.name = . . .; enc_vm.image = . . .;
Snort_vm.name = . . .; Snort_vm.image = . . .;

GRANT_PRIVILEGE(memscan_vm, webserver_vm, KERN_MEM);
SET_BACKEND(Snort_vm, webserver_vm, NETWORK, MAY_COLOCATE);
SET_BACKEND(enc_vm, empDB_vm, NETWORK, MUST_COLOCATE);
```

Figure 3.4: **Example inter-VM dependency specification.**

is necessary to protect the secrecy of the client's data before it reaches the device drivers hosted in Sdom0. If on the other hand, the client is not concerned about the secrecy of his network traffic, but wishes only to check for intrusions, the SD that performs network intrusion detection can potentially be placed on another machine, provided that all inbound traffic traverses the SD first before being routed to the UdomU.

Figure 3.4 presents an example of such a scenario. The client specifies that the network traffic to *webserver_vm* must be checked using Snort. The client does not consider the traffic to and from the Web server to be sensitive, so it may potentially be exposed to the cloud provider. However, any interactions with its employee records database *empDB_vm* must happen over SSL. The *enc_vm* SD serves as the network I/O backend for the *empDB_vm* database, encrypting all outgoing traffic and decrypting all incoming traffic, while residing on the same host as *empDB_vm*.

Note that on traditional cloud platforms, customers usually do not control how their VMs are placed. Rather, the cloud controller determines VM placements based upon the global state of the cloud platform, the configurations requested by the client, and the cloud provider's load balancing policies. However, SSC's control plane gives clients *some* control over how their VMs are placed. For example, a client can specify that a rootkit detection SD that inspects the memory of its UdomUs must be placed on the same physical host as the UdomUs.

The cloud controller's scheduling algorithm uses these requests as additional constraints. It processes the entire program specified by the client to determine VM placements. For example, consider the dependencies shown in Figure 3.5. The client has a *webserver_vm* that receives data over an encrypted channel. The client dedicates the *enc_vm* VM to handle encryption and decryption, while *memscan_vm* scans the kernel memory of *webserver_vm*. Additionally, the client has specified that all packet headers destined for the *webserver_vm* must be inspected by *firewall_vm*. These rules imply that *memscan_vm*, *webserver_vm* and *enc_vm* must be colocated on the same machine. However, *firewall_vm*, which only inspects packet headers and is set as the backend for *enc_vm*, can be located on a different host.

In general, it is possible to depict these dependencies as a *VM dependency graph*, as shown in Figure 3.5. In this graph, an edge *vm1*→*vm2* depicts that *vm1* either serves as the backend for *vm2* or that *vm1* has privileges over *vm2*. Edges are also annotated with MUST_COLOCATE or MAY_COLOCATE to denote co-location constraints.

In some cases, it may not be possible to resolve the client's VM dependencies, given hardware constraints or the current load on the cloud infrastructure. In such cases, the client is suitably notified, so that it can modify the dependencies. The dashboard also performs certain sanity checks on the program input by the client. For example, it checks to determine that the VM dependency graph implied by the program is acyclic (the acyclic property is required to make migration decisions, as discussed in Section 3.6). In such cases, the dashboard raises a warning, akin to a compile-time error, so that the client can correct the program.

Once the specifications are accepted, and the cloud controller produces a placement, the

```
GRANT_PRIVILEGE(memscan_vm, webserver_vm, KERN_MEM);
SET_BACKEND(enc_vm, webserver_vm, NETWORK, MUST_COLOCATE);
SET_BACKEND(firewall_vm, enc_vm, NETWORK, MAY_COLOCATE);
```

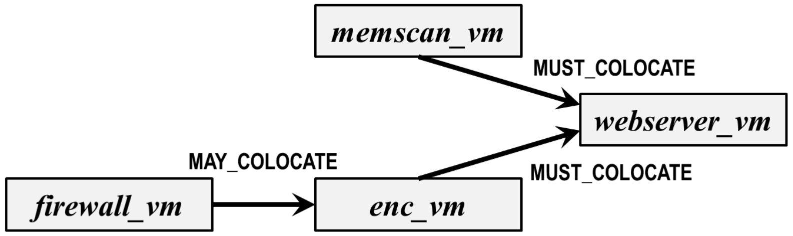


Figure 3.5: **Example showing inter-VM dependencies that require certain VMs to be co-located.**We have elided VM declarations and initializations for brevity. Also shown is the VM dependency graph for this example.

dashboard orchestrates the actual setup of the VMs. For a pair of dependent VMs *vm1*→*vm2* that are located on the same physical host, enforcing the dependencies is relatively straightforward. In case they are dependent via a GRANT_PRIVILEGE, the dashboard simply instructs the Udom0 to assign suitable privileges over *vm2* to *vm1*. Likewise, the dashboard can instruct Udom0 to set *vm1* as the device backend for *vm2* in case of a SET_BACKEND dependency.

However, *vm1* and *vm2* could be located on different physical hosts (S and T, respectively) if they related via a SET_BACKEND(*vm1*, *vm2*, . . ., MAY_COLOCATE). In this case, the dashboard configures switches on the network to route traffic destined for *vm2* via *vm1*. More concretely, SSC uses the Open vSwitch [44] software switch for this purpose. In this case, the dashboard instructs the Udom0s on S and T to create SDs on these hosts running the Open vSwitch software. On S, traffic from *vm1* is sent to the Open vSwitch SD running on that host, which routes it to the Open vSwitch SD on T via a Generic Routing Encapsulation (GRE) tunnel. On T, this Open vSwitch SD is configured to be the backend of *vm2*, thereby routing traffic to *vm2*. Outbound traffic from *vm2* is routed via *vm1* in a similar fashion. Figure 3.6 illustrates the setup using the VMs for Figure 3.5 as an example.

The co-location constraints implied by SSC's inter-VM dependencies can possibly be miused by malicious clients to infer proprietary information about the cloud platform using *probe-response* attacks. In such attacks, the client provides a program with a sequence of

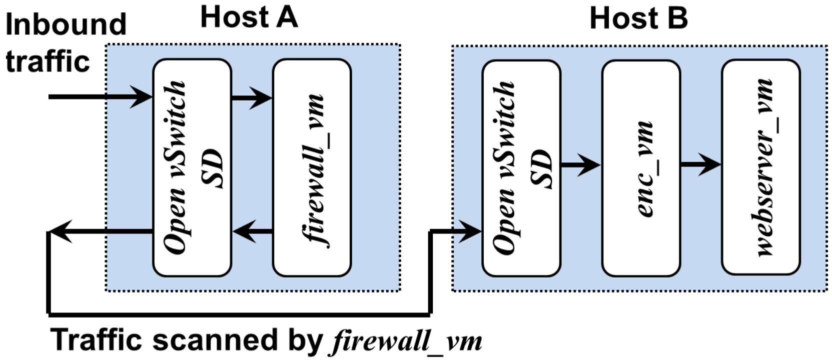


Figure 3.6: **Open vSwitch setup** showing the path followed by inbound network traffic to the web server example from Figure 3.5. Outbound network traffic follows the reverse path. The Udom0 instances and the *memscan_vm* instance on Host B are not shown.

GRANT_PRIVILEGE dependencies, requiring a certain number of VMs (say, n VMs) to be co-located on the same physical host. The client could repeatedly probe the cloud with programs that successively increase the value of n. When the cloud controller is no longer able to accommodate the client's requests, the client can use this failed request to gain insight into the limitations of the hardware configurations of the cloud platform's hosts or into the current load on the cloud.

Such threats are definitely a possibility, and we view them as the cost of giving honest clients the flexibility to control VM placements to enable useful services. Nevertheless, there are defenses that the cloud provider could use to offset the impact of such threats. For example, the provider could pre-designate a cluster of machines to be used for clients with larger-than-usual VM co-location constraints, and try to satisfy the client's requests on this cluster. This would ensure that the effects of any probe-response attacks that give the client insight into the provider's proprietary details are constrained to that cluster alone.

Finally, recent work in cloud security has focused on the possibility of attacks enabled by VM co-location, enabling a variety of malicious goals (*e.g.,* [47, 61]). These results are not directly applicable in our setting, because SSC allows a client to specify VM co-location constraints for *its own VMs*. In contrast, the works cited above require co-location of a victim VM with the attacker's VM. Does allowing malicious clients to co-locate their own VMs on a host ease the possibility of launching attacks on *other* VMs co-located on that host? We do not know the answer to this question, but it would be prudent to acknowledge that such attacks may be possible. Exploring the full extent of such attacks is beyond the scope of this dissertation.

3.6 VM Migration

Most cloud platforms employ *live migration* [12] to minimize VM downtimes during migration. Live migration typically involves an *iterative push phase* and a *stop-and-copy phase*. When the cloud controller initiates the migration of a VM, the source host commences the iterative push phase, copying pages of the VM over to the target host, even as the VM continues to deliver service from the source. This phase has several iterations, each of which pushes VM pages that have been dirtied since the previous iteration. When the number of dirty pages falls below a threshold, a *stop-and-copy phase* pauses the VM on the source, copies over any remaining dirty pages of the VM, and resumes the VM on the target. Because the number of dirty pages is significantly smaller than the size of the VM itself, live migration usually has small downtimes (sub-second in several cases).

In SSC, the decision to migrate is made by the cloud controller, which produces a new VM placement and communicates it to the dashboard. The dashboard coordinates with the source and target hosts to actually perform the migration. The decision to migrate a client VM can also result in the migration of other VMs that are related to it. When a UdomU is migrated, all the SDs that service the UdomU and must be co-located with it must also be migrated. The cloud controller uses the dependencies supplied by the client when producing a new placement decision after migration.

Dependency specifications also implicitly dictate the order in which VMs must be migrated. Consider the example in Figure 3.5. Both *memscan_vm* and *enc_vm* must be colocated when *webserver_vm* is migrated. However, because *memscan_vm* and *enc_vm* service *webserver_vm*, they must both continue to work as long as *webserver_vm* does. During the stop-and-copy phase of live migration, *webserver_vm* must be paused before *memscan_vm* and *enc_vm* are paused. Likewise, on the target host, *memscan_vm* and *enc_vm* must be resumed before *webserver_vm*. In general, the order in which VMs must be paused and resumed can be inferred using the VM dependency graph, which must be acyclic. All of a VM's children in the graph must be paused before it is paused (and the opposite order for resumption).

(1) Cloud controller decides to migrate a group of MUST_COLOCATE VMs (vm_1, vm_2, $\ldots$, vm_n) from source S to target T.
(2) Dashboard uses VM dependency graph to determine the order in which VMs must be paused on S and resumed at T.
(3) Dashboard checks whether client has a Udom0 instance running on T. If not, starts it as described in Section 3.4, and specifies order in which VMs received must be started.
(4) Dashboard requests Udom0 on S to initiate migration to T, and specifies the order in which to pause the VMs.
(5) Udom0 on S establishes an encrypted channel to communicate with Udom0 on T
(6) Udom0 on S iteratively pushes VM pages to T.
(7) Udom0 on S pauses VMs (stop-and-copy phase) and sends VM pages to Udom0 on T.
(8) Dashboard obtains TPM measurements from domB on S, containing hashes of paused VMs.
(9) Dashboard identifies any May_Colocate VM backends with dependencies on (vm_1, vm_2, $\ldots$, vm_n), and instructs switches to update network routes from these backends to T instead of S.
(10) Udom0 on T resumes the VMs, and forwards TPM measurements obtained from domB on T to dashboard.
(11) Dashboard checks the TPM measurements obtained from S and T for equality. If not equal, raises security alert.
(12) Dashboard determines whether there are any remaining client VM instances on S. If not, it initiates a shutdown of the Udom0 on S.

Figure 3.7: **Migrating a group of co-located VMs.**

Figure 3.7 summarizes the steps followed during migration. The dashboard orchestrates migration and checks TPM attestations when VMs resume after being migrated. It also parses the VM dependency graph and identifies the order in which VMs must be migrated and paused. The actual task of copying VM pages is carried out by the Udom0 of the source host. When a set of co-located VMs with MUST_COLOCATE dependencies is migrated from a source to a target, there may be VMs with MAY_COLOCATE dependencies that must be suitably updated. For example, switches on the cloud must be updated so that traffic to and from *firewall_vm* are routed to the target machine to which the VMs *webserver_vm*, *enc_vm* and *memscan_vm* are migrated. In SSC, this is accomplished by suitably modifying the configurations of the Open vSwitch SDs running on the machines that host these four VMs after migration.

Chapter 4

Evaluation

The main goals in evaluating SSC were the following:

1. To demonstrate the flexibility of the SSC model in enabling various virtualization-based services as SDs.

2. To compare the performance of these SD-based services against their traditional, dom0-based counterparts.

3. To understand the performance overhead introduced by the components of SSC's control plane.

To achieve these goals, we implemented ten different services and conducted experiments to measure overhead of running these services as SDs rather than within dom0. To measure the overhead of SSC's control plane, we conducted VM migration experiments under SSC control plane and report VM downtimes for various configurations of the migration algorithm.

Our experiments were performed on a Dell Poweredge R610 system equipped with 24GB RAM, eight 2.3GHz Xeon cores with dual threads (16 concurrent executions), Fusion-MPT SAS drives, and a Broadcom NetXtreme II gigabit NIC. All virtual machines started in our experiments (dom0, domU, Sdom0, Udom0, UdomU, SDs and domB) were configured to have 2GB RAM and 2 virtual CPUs. The experimental numbers reported in this section are averaged over five executions; we also report standard deviations.

In SSC, all VM creation requests are communicated to domB. DomB neither has any persistent state nor does it require a file system. During startup, domB prepares XenStore devices that are necessary for block interface communication between domB and other control VMs; it does not require any other I/O devices. Kernel images and the initial ramdisk along with the

Platform	Time (seconds)
Traditional Xen	2.131±0.011
SSC	2.144±0.012 (0%)

Figure 4.1: **Cost of building domains.**

configuration of the VM to be created are presented to domB as a virtual disk via the block device interface. Figure 4.1 compares the cost of building VMs on a traditional Xen VMM and on an SSC platform. As these numbers demonstrate, the costs of building domains on these platforms is near-identical. We now illustrate the utility of SSC by using it to build several SDs that implement common utilities.

4.1 Networked Services

In SSC, the network SDs can be co-located with UdomUs. We conducted experiments for both deployment scenarios *i.e.,*SDs co-located with the corresponding UdomUs and SDs running on hosts different from the hosts running UdomUs. We compare the overheads of network services running as SDs in SSC against the same network services implemented in traditional setup (*i.e.,* within dom0 executing on an unmodified Xen hypervisor).

4.1.1 Baseline Overhead

Our first experiment aims to evaluate the baseline overhead of running networked services atop the SSC infrastructure. For this experiment, we create a network SD *netsd* VM and set it as the backend of a work VM *udomu* using SET_BACKEND(*netsd*, *udomu*, NETWORK, MAY_COLOCATE). The SD *netsd* does no additional processing on the packets that it receives, and simply forwards them to *udomu*. (The remainder of this section talks about network SDs to achieve various goals, and their overhead must be compared against this baseline, which reports the best performance achievable for a networked service implemented as an SD). Under this setup, *netsd* may either be co-located on the same host as *udomu*, or be located on a different host, depending on the placement decision of the cloud controller. Each setup results in a different network topology, as illustrated in Figure 4.2. We evaluate both setups, and use the

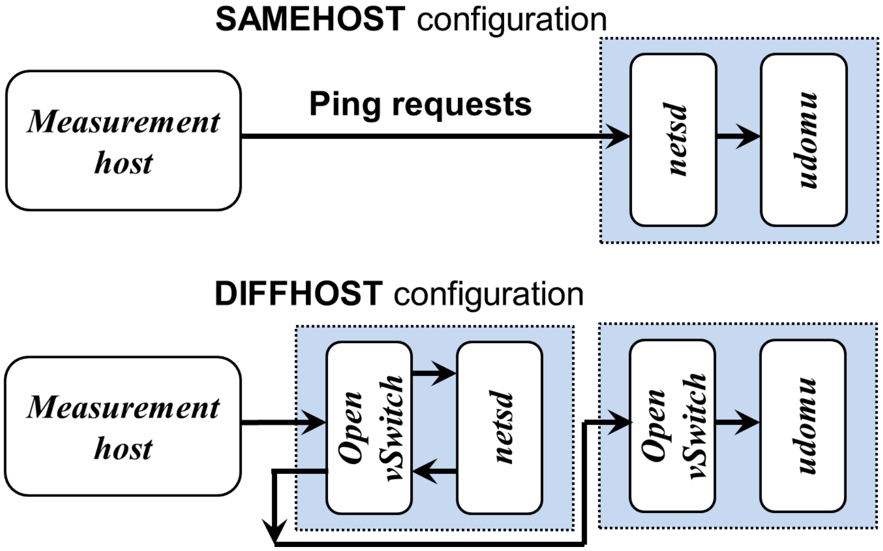


Figure 4.2: **The network topologies used to evaluate the baseline overhead of networked services executing atop SSC.**We only show the inbound network path. The outbound path is symmetric.

Setup	Throughput (Mbps)	RTT (ms)
SAMEHOST configuration		
Traditional	925.4±0.5	0.38±0
SSC	924.0±1.2 (0%)	0.62±0 (1.6×)
DIFFHOST configuration		
Traditional	848.4±11.2	0.69±0
SSC	425.8±5.5 (49.8%)	1.6±0 (2.3×)

Figure 4.3: **Baseline overhead of networked services.**

keywords SAMEHOST and DIFFHOST to differentiate the cases when *netsd* and *udomu* are co-located or not. We compare these against a traditional setup, where the networked service executes within dom0, either on the same host or on a different host.

We dedicated a separate *measurement host* in the same local network as our experimental infrastructure to transmit and receive network packets to the *udomu*. We measured the network throughput to *udomu* using the iperf3 [27] tool, and used ping to measure the round-trip time (RTT) of network traffic to and from the *udomu*. Our results report average over five executions along with the standard deviations.

Figure 4.3 presents the results of our experiment. If *netsd* is co-located with *udomu*, the throughput remains unaffected. However, the RTT drops as a result of having to traverse another element (*netsd*) on the path to the *udomu*. When *netsd* and *udomu* are on different hosts, the RTT overhead increases to (2.7×) as a result of new network elements encountered in the path to *udomu*. However, we observed that the throughput also reduces to nearly 50% compared to the traditional non-SSC-based setup. Upon further investigation, we found that the reason for this is that the way Xen currently implements support for backend network drivers prevents concurrent bidirectional network transmission.

On Xen, dom0 executes the network driver (called netback) that serves as the backend for all domU network communication. Xen uses only one kthread in netback to process domU's transmit and receive queues [67]. The SSC hypervisor inherits this limitation, and uses only one kthread in the netback drivers of SDs that serve as backends for UdomUs. Thus, if we consider the DIFFHOST configuration on a traditional Xen-based platform, where the functionality of *netsd* executes within dom0, the network driver simply receives inbound traffic from the measurement host, and forwards it to the dom0 of the machine that hosts *udomu*. In contrast, on an SSC-based platform, the netback driver within the Open vSwitch SD receives inbound traffic from the measurement host, forwards it to *netsd*, receives traffic from *netsd*, and forwards this to the Open vSwitch SD of the machine that hosts *udomu* (as shown using the arrows in Figure 4.2). As a result, even though the network hardware used in our experiments supports concurrent bidirectional network bandwidth of 1Gbps, the inability of the netback drivers to support concurrent bidirectional transmission cuts the throughput by approximately half.[1]

4.1.2 Network Access Control SD

Network access control services (*e.g.,* firewalls) are often the first layer of defense in any operational network. Traditional network access control services in the cloud, such as security groups, allow clients to specify rules on network packets. However, security groups are quite restrictive and only filter incoming packets. Our network access control SD is implemented as a middlebox that can be customized by clients. In our implementation, we used a set of rules

[1] The throughput gap between the SAMEHOST case and the DIFFHOST case in SSC allows a malicious client to infer whether *netsd* and *udomu* are co-located. However, an enhanced implementation of the netback driver in Xen, with separate kthreads to process transmit and receive queues will address this attack (and improve network throughput in the DIFFHOST case!)

Setup	Throughput (Mbps)
SAMEHOST configuration	
Traditional	925.1±0.7
SSC	923.2±1.6 (0%)
DIFFHOST configuration	
Traditional	846.7±17.2
SSC	425.2±7.2 (49.7%)

Figure 4.4: **Network access control service.**

that included a list of IP addresses and open ports for which packets should be accepted. The SD has a MAY_COLOCATE dependency on the VM(s) it protects.

Figure 4.4 presents the performance of this SD, implemented both in the traditional setting within dom0, and atop SSC. The numbers here report overheads very similar to the baseline, thereby showing that the extra CPU processing overhead imposed by the SD is minimal. (RTT numbers were also similar to the baseline numbers).

4.1.3 Trustworthy Network Metering

On traditional cloud platforms, clients trust the cloud provider to correctly charge them based upon the resources that they consume. If a client has reason to believe that a cloud provider is charging it for more than its share of resources consumed, it cannot prove that the cloud provider is cheating. Therefore what is needed is a trustworthy service that performs resource accounting. Both the cloud provider and the client must be able to access the service and verify that the charges correspond to the resource utilization of the client. Unfortunately, it is impossible to design such a service on today's cloud platforms. Recent work [5] has investigated the possibility of using nested virtualization to implement such a service.

SSC allows the creation of such trustworthy resource accounting services. The key mechanism to do so is *mutually-trusted service domains (MTSDs)*, supported by SSC. MTSDs resemble SDs in all aspects but two. First, unlike SDs, which are started and stopped by the client, the cloud provider and the client collaborate to start or stop an MTSD. Second, although an MTSD executes within a client's meta-domain with privileges to access other client VMs, a client cannot tamper with or modify an MTSD once it has started.

MTSDs can be used to implement trustworthy network metering as follows. The client and

Setup	Throughput (Mbps)
SAMEHOST **configuration**	
Traditional	924.8±1.1
SSC	924.1±0.4 (0%)
DIFFHOST **configuration**	
Traditional	845.4±11.1
SSC	424.3±3.1 (49.8%)

Figure 4.5: **Trustworthy network metering service.**

cloud provider agree upon the software that will be used to account for the client's network bandwidth utilization. This metering software executes as an MTSD and serves as the network backend for all of the client's network-facing VMs. The client and cloud provider can both verify that the MTSD was started correctly (using TPM attestations), and the SSC hypervisor ensures that neither the cloud provider nor the client can tamper with the MTSD once it has started execution. Thus, both the cloud provider and the client can trust the network bandwidth utilization reported by the MTSD.

Our network metering MTSD captures packets using the libpcap [58] library, simply counts the number of packets captured, and reports this number when queried. Because it measures network bandwidth utilization for *all* the client's VMs, it must have a MAY_COLOCATE dependency with all of them. Figure 4.5 shows the impact of network metering service on the network throughput of a single work VM (setup similar to Figure 4.2). As before, the additional overhead imposed over the baseline is minimal.

4.1.4 Network Intrusion Detection

SSC allows clients to deploy and configure customized network intrusion detection systems as middleboxes. On traditional cloud platforms, this is not possible. Rather, they are forced to accept the offerings that the cloud provider has. Moreover, they cannot configure the placement of these middleboxes and must rely on the cloud provider to do so.

As an example, we used Snort to set up an intrusion detection system as a middlebox before our work VMs. Snort uses libpcap [58] to capture network traffic. Our setup uses the Stream5 preprocessor that performs TCP reassembly and handles both TCP and UDP sessions. We used the latest snapshot of signatures available from the Snort website in our setup. The Snort SD

Setup	IDS
SAMEHOST configuration	
Traditional	922.8±1.1
SSC	920.9±1.9 (0%)
DIFFHOST configuration	
Traditional	841.2±14.2
SSC	422.6±7.1(49.7%)

Figure 4.6: **Network intrusion detection (Snort) service.**

Setup	Time (μsec)
Traditional	1014±6
SSC	1688±31 (66%)

Figure 4.7: **Time to establish a TCP connection in VMWall.**

has a *May_Colocate* dependency on the UdomU(s) it monitors. Figure 4.6 presents the results of our experiments and shows that the overhead imposed by an SD implementation of Snort is minimal.

4.1.5 VMWall Service

VMWall [53] is a virtualized application-level firewall. In the traditional setting, VMWall operates as a daemon within dom0, and intercepts network packets originating from the VM that it monitors. It then performs memory introspection of the VM to identify the process that is bound to the network connection. VMWall permits the flow only if the process belongs to a whitelist. Implemented as an SD, VMWall serves as the network backend for the UdomU that it monitors. It must also have the privileges to inspect the memory of the UdomU, so that it can identify the process from which the flow originates.

It is created with the following dependency rules, which imply that it must be co-located with the UdomU.

GRANT_PRIVILEGE(*vmwall_vm*, *udomu*, KERN_MEM);

SET_BACKEND(*vmwall_vm*, *udomu*, NETWORK, MUST_COLOCATE);

Our re-implementation of VMWall uses libvmi [42, 62] for memory introspection. Figure 4.7 presents the results of our experimental evaluation of VMWall. We measured the TCP connection setup time using a simple client/server setup. Compared to the traditional setup, establishing a TCP connection with the VMWall SD incurs an overhead of 66%. The main reason for this overhead is the latency introduced by routing all the traffic through VMWall-SD along with the three-way handshake of TCP connection creation. Also this is a one-time cost at the setup phase and will be amortized across the duration of connection.

4.2 Storage Services

Cloud providers supply clients with persistent storage. Because the actual storage hardware is no longer under the physical control of clients, they must treat it as untrusted. They must therefore have mechanisms to protect the confidentiality and integrity of data that resides on cloud storage. Such mechanisms can possibly be implemented within the client's VMs itself (*e.g.,* within a custom file system). However, virtual machine technology allows such services to be conveniently located outside the VM, where they can also be combined flexibly. It also isolates these services from potential attacks against client VMs. Because all I/O from client VMs is virtualized, storage encryption and integrity checking can easily be implemented as cloud-based services offered by the provider.

Cloud providers would normally implement such services as daemons within dom0. However, this approach entails clients to trust dom0, and hence cloud administrators. SSC provides clients the ability to implement a variety of storage services as SDs (storageSD) without trusting cloud administrators. We describe two such SDs below, one for integrity checking and another for encryption. Our implementation of both SDs is set up as illustrated in Figure 4.8. Each SD executes as a VM. The SD is set as the backend of the work VM *udomu* using SET_BACKEND(*storageSD*, *udomu*, STORAGE, MUST_COLOCATE).

4.2.1 Encryption Storage Service

Storage encryption protects the confidentiality of client data by enciphering it before storing it on disk. Using SSC, clients can deploy their own storage encryption SD that enciphers their data before it is transmitted to Sdom0, which stores it on disk (or further processes the encrypted data, *e.g.,* to implement replication). Conversely, Sdom0 reads encrypted data from disk, and passes it to the SD, which decrypts it and passes it to the client. SSC ensures that

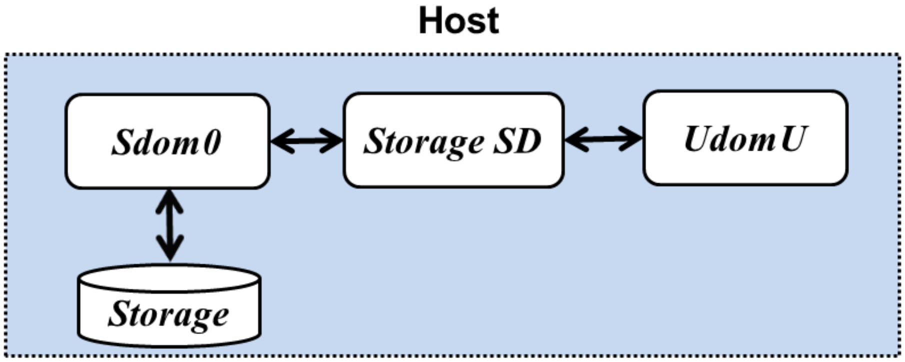


Figure 4.8: **Storage service VM architecture.**

Platform	**Unencrypted (MB/s)**	**Encrypted (MB/s)**
Traditional	81.72±0.15	71.90±0.19
SSC	75.88±0.15 (7.1%)	70.64±0.32 (1.5%)

Figure 4.9: **Cost incurred by the storage encryption service VM.**For the first experiment, the SD runs a loopback device that performs no encryption. For the second, the SD runs a crypto loopback device with 128-bit AES encryption.

Sdom0 cannot access the encryption keys, which are stored in SD memory, thereby protecting client data.

Udom0 initiates the storage encryption SD using a key passed as a kernel parameter. The SD encrypts client data before it reaches Sdom0, and decrypts enciphered disk blocks fetched by Sdom0. Data is never presented in the clear to the cloud provider, and the encryption key is never exposed to Sdom0. In our implementation, we uses crypto loopback device for the encryption and configured the loopback device to use AES 128-bit encryption.

We evaluated the cost of our SD using two experiments. In the first experiment, we simply used a loopback device (rather than a crypto loopback device) as the backend within our SD, and compared the achieved disk throughput against traditional I/O on Xen where domU communicates with a backend driver in dom0 (*i.e.,* data is stored in the clear). This experiment allows us to measure the baseline overhead of introducing a level of indirection in the I/O path (*i.e.,* the SD itself). In the second experiment, we used the crypto loopback device as the backend and measured the overhead of encryption. In our experiments, we emptied buffer caches

Platform	**Throughput (MB/s)**
Xen (dom0)	71.7±0.1
SSC (SD)	66.6±0.3 (7.1%)

Figure 4.10: **Cost incurred by the storage integrity checking service VM.**

so that each disk operation results in a disk access, thereby traversing the entire I/O path and emulating the worst-case scenario for storage encryption.

We used the Linux `dd` utility to perform a large read operation of size 2GB. Figure 4.9 presents the results of our experiments. These experiments show that the reduction in disk throughput introduced by the extra level of indirection is about 7%. With encryption enabled, the raw disk throughput reduces in both cases, thereby reducing the overhead of SSC-based encryption to about 1%.

4.2.2 Integrity Checking Service

Our integrity checking SD offers a service similar to the one proposed by Payne *et al.* [42]. The SD implements a loopback device, which runs as a kernel module. This device receives disk access requests from UdomUs at the block level, enforces the specified integrity policy, and forwards the requests to/from disk.

In our prototype SD, users specify important system files and directories to protect. The SD intercepts all disk operations to these targets, and checks that the SHA256 hashes of these disk blocks appear in a database of whitelisted hashes. Since all operations are intercepted at the block level, the SD needs to understand the high-level semantics of the file system. We use an offline process to extract known-good hashes at the block level from the client VM's file system, and populate the hash database, which the SD consults at runtime to check integrity.

We evaluated the cost of the integrity checking SD using the same workload as for the encryption SD. We checked the integrity of disk blocks against a whitelist database of 3000 hashes. Figure 4.10 compares the throughput achieved when this service is implemented as an SD versus as a daemon in dom0. The SD service incurs an overhead of about 7%, mainly because of the extra level of indirection.

4.3 Memory Introspection Service

Memory introspection tools, such as rootkit detectors (*e.g.,* [1, 35, 43, 53]), rely on the ability to fetch and inspect raw memory pages from target VMs. In commodity cloud infrastructures, memory introspection must be offered by the provider, and cannot be deployed independently by clients, who face the unsavory option of using the service but placing their privacy at risk.

Using SSC, clients can deploy memory introspection tools as SDs. We illustrate such an SD by implementing an approach developed in the Patagonix project [35]. Patagonix aims to detect the presence of covertly-executing malicious binaries in a target VM by monitoring that VM's page tables. As originally described, the Patagonix daemon runs in dom0, maps all the memory pages of the target VM, and marks all pages as non-executable when the VM starts. When the target VM attempts to execute a page for the first time, Patagonix receives a fault. Patagonix handles this fault by hashing the contents of the page (*i.e.,* an md5sum) requested for execution, and comparing it against a database of hashes of code authorized to execute on the system (*e.g.,* the database may store hashes of code pages of an entire Linux distribution). If the hash does not exist in the database, Patagonix raises an alarm and suspends the VM.

We implemented Patagonix as an SD. Each Patagonix SD monitors a target UdomU, a reference to which is passed to the SD when the UdomU boots up. Udom0 delegates to Patagonix SD the privileges to map the UdomU's pages using GRANT_PRIVILEGE (*PatagonixSD*, *udomu*, FULL). Patagonix SD marks UdomU's pages as non-executable. The SD receives and handles faults as the UdomU executes new code pages. Our Patagonix SD can detect maliciously-executing binaries with the same effectiveness as described in the original paper [35]. To measure this SD's performance, we measured the boot time of a monitored UdomU. The SD validates all code pages that execute during boot time by checking each of them against the hash database. We compared the time taken by this SD to a traditional setup where the Patagonix daemon executed within dom0. Figure 4.11 presents the results of our experiment, again showing that using an SD imposes minimal overhead.

Platform	Time (seconds)
Traditional	6.471±0.067
SSC	6.487±0.064 (0%)

Figure 4.11: **Cost of the memory introspection service VM**, measured as the time to boot a Linux-based domain.

4.4 System Call Monitor

There is a large body of work on system call-based anomaly detection tools. While we will not attempt to summarize that work here (see Giffin's thesis [24] for a good overview), these techniques typically work by intercepting process system calls and their arguments, and ensuring that the sequence of calls conforms to a security policy. The anomaly detector executes in a separate VM (dom0), and capture system call traps and arguments from a user VM for analysis. Using SSC, clients can implement their own system call anomaly detectors as SDs. The SD simply intercepts all system calls and arguments from a target UdomU and checks them against a target policy.

On a paravirtualized platform, capturing system calls and their arguments is straightforward. Each trap from a UdomU transfers control to the hypervisor, which forwards the trap to the SD if it is from a user-space process within the UdomU. The SD captures the trap address and its arguments (passed via registers). However, the situation is more complex on an HVM platform. On such a platform, traps are directly forwarded to the kernel of the HVM by the hardware without the involvement of the hypervisor. Fortunately, it is still possible to capture traps, albeit differently on AMD and Intel hardware. AMD supports control flags that can be set to trigger VMExits on system calls. On the Intel platform, traps can be intercepted by placing dummy values in the MSR (model-specific register) corresponding to the `syscall` instruction to raise a page fault on a system call. On a page fault, the hypervisor determines the source of the fault; if due to a system call, it can forward the trap address and registers to the SD.

We evaluated the cost of this approach by simply building an syscall monitor SD to capture system calls and their arguments (*i.e.,* our SD only includes the system call capture tool; we do not check the captured calls against any policies). We used the syscall microbenchmark of the UnixBench benchmark suite [59] as the workload within the target UdomU to evaluate the

Platform	System calls/second
Traditional	275K ±0.95
SSC	272K ±0.78 (1%)

Figure 4.12: **Cost incurred by the system call monitoring service VM**, measured using the UnixBench syscall microbenchmark.

overhead of this SD. The syscall microbenchmark runs mix of *close, getpid, getuid and umask* system calls and outputs the number of system calls executed in a fixed amount of time. In our experiments we compared the number of system calls executed by the syscall microbenchmark when the system call capture tool runs as SD to the traditional scenario where the system call capture tool runs in dom0. Figure 4.12 presents the result of the experiment and shows that running system call monitor as an SD incurs negligible overhead.

4.5 Other Services

So far, we have illustrated several security services implemented as SDs. However, the utility of SDs is not limited to security alone, and a number of other services can be implemented as SDs. We illustrate two such examples in this section.

4.5.1 VM Checkpointing Service

It is commonplace for cloud service providers to checkpoint client VMs for various purposes, such as live migration, load balancing and debugging. On commodity cloud architectures, checkpointing is implemented as a user daemon within dom0, which copies client VM memory pages and stores them unencrypted within dom0. If dom0 is untrusted, as is usually the case, it is challenging to create trustworthy checkpoints [54]. SSC simplifies checkpointing by allowing it to be implemented as an SD. The SD maps the client's memory pages, and checkpoints them akin to the dom0 checkpointing daemon (in fact, we reused the same code-base to implement the SD). Clients can chain the storage encryption SD with the checkpointing SD to ensure that the checkpoint stores encrypted data.

We implemented a checkpointing SD and grant it privileges using GRANT_PRIVILEGE(*CheckpointSD*, *udomu*, FULL). We evaluated checkpointing SD by checkpointing VMs

Platform	VM size (MB)	No encryption (seconds)	With encryption (seconds)
Traditional	512	0.764±0.001	5.571±0.004
SSC	512	0.803±0.006 (5.1%)	5.559±0.005 (-0.2%)
Traditional	1024	1.840±0.005	11.419 ±0.008
SSC	1024	1.936±0.001 (5.2%)	11.329 ±0.073 (-0.8%)

Figure 4.13: **Cost incurred by the checkpointing service VM.**

with two memory footprints: 512MB and 1024MB. We also conducted an experiment where we chained this SD with storage encryption SD using SET_BACKEND(*ESSD*, *CheckpointSD*, STORAGE, MUST_COLOCATE); the checkpoint file is therefore encrypted in this case. To mask the effects of disk writes, we saved the checkpoint files on a memory-backed filesystem. Figure 4.13 presents the results of our experiments, comparing the costs of our checkpointing SD against a checkpointing service implemented in dom0. Our results show that the costs of implementing checkpointing within an SD are within 5% of implementing it within dom0. In fact, we even observed minor speedups in the case where we chained checkpointing with encryption. SSC therefore offers both security and flexibility to customers while imposing mimimal overhead.

4.5.2 Memory Deduplication Service

When multiple VMs have memory pages with identical content, one way to conserve physical memory using a mechanism where VMs share memory pages [63]. Such a mechanism benefits cloud providers, who are always on the lookout for new techniques to improve the elasticity of their services. It can also benefit cloud clients who may have multiple VMs on the cloud and may be billed for the memory consumed by these VMs. Identifying and exploiting memory sharing opportunities among VMs allows clients to judiciously purchase resources, thereby reducing their overall cost of using the cloud. In commodity cloud computing environments, providers implement memory deduplication to consolidate physical resources, but such services are not exposed to clients, thereby limiting their applicability.

SSC allows clients to deploy memory deduplication on their own VMs without involving the cloud provider. To illustrate this, we implemented a memory deduplication SD. This SD accepts as input a list of domains (UdomUs) in the same meta-domain, and identifies pages with

Platform	VM size (MB)	Time (seconds)
Traditional	512	6.948±0.187
SSC	512	6.941±0.045 (0%)
Traditional	1024	15.607±0.841
SSC	1024	15.788±0.659 (1.1%)

Figure 4.14: **Cost incurred by the memory deduplication service VM.**

identical content (using their md5 hashes). For each such page, the SD instructs the hypervisor to keep just one copy of the page, and free the remaining copies by modifying the page tables of the domains. The hypervisor marks the shared pages as belonging to special “shared memory” domain. When a domain attempts to write to the shared page, the hypervisor uses copy-on-write to create a copy of that page local to the domain that attempted the write, and makes it unshared in that domain.

We evaluated the performance of the memory deduplication SD by measuring the time taken to identify candidate pages for sharing, and marking them as shared. We conducted this experiment with a pair of VMs with memory footprints of 512MB and 1024MB each. As before, we compared the performance of the SD with that of a service running in dom0 on stock Xen. Table 4.14 presents the results, and shows that the performance of the SD is comparable to the traditional approach.

4.6 Evaluating VM Migration

We measure the performance of VM migration using two metrics: *VM down time* and *overall migration time*. Recall from Section 3.6 that migration happens in two phases, an iterative push phase, and a stop-and-copy phase. The VM down time metric measures the time taken by the stop-and-copy phase, while the overall migration time measures the time from the initialization of VM migration to its completion.

We perform three sets of experiments to evaluate VM migration. In the first experiment, we migrate a single VM in SSC and compare it against migration on a traditional Xen platform. In our second experiment, we consider the case in SSC where a group of co-located VMs must be migrated together. In this experiment, we evaluate the performance implications of two migration policies. Third, we evaluate how the length of a dependency chain in the VM

Setup	Time (seconds)
Traditional	23.27±0.11
SSC	23.81±0.03 (2%)

Figure 4.15: **Total migration time for one virtual machine.**

dependency graph affects the performance of migration. The first two experiments report the overall migration time only, while the third experiment explores VM down time in detail. For all the experiments reported in this section, we assume that the VMs to be migrated are 1GB in size, and are configured with 1 virtual CPU. The setup is otherwise identical to the one used in all other SD experiments except when otherwise mentioned.

4.6.1 Migrating a Single VM

Figure 4.15 reports the time to migrate a single VM from one host to another in a traditional setting and on an SSC platform. The small overhead (2%) in the SSC setting can be attributed to the extra steps involved in migrating a VM in SSC, in particular, setting up a Udom0 at the target host. Note that because migration is live, the VM is still operational on the source as it is being migrated to the target. The down time in this case is approximately 100ms (as discussed in more detail in Section 4.6.3).

4.6.2 Migrating a Group of VMs

When a group of dependent co-located VMs (vm_1, vm_2, …, vm_n) is live migrated from one host to another, there are two options to implement iterative push. The first is to iteratively push vm_1, vm_2, …, vm_n sequentially. The second option is to iteratively push all n VMs using the available parallelism in the underlying physical platform. The tradeoff is that while parallel migration approach can lead to lower migration times, it can saturate the network link and increase CPU utilization.

Figure 4.16(a) presents the overall time required to migrate a group of 2 VMs and 4 VMs, respectively, using the sequential and parallel migration policies. Naturally, the overall time to migrate using the parallel policy is smaller than for the sequential policy. We also used the iftop utility to measure the peak network utilization during VM migration. We found that with

# of VMs	Sequential (seconds)	Parallel (seconds)
2	47.29±0.18	27.91±0.16
4	128.89±0.76	57.78±0.49

(a) Udom0 configured to have 2 virtual CPUs.

# of VMs	Sequential (seconds)	Parallel (seconds)
2	47.41±0.29	28.01±0.26
4	103.96±0.20	39.21±0.50

(b) Udom0 configured to have 4 virtual CPUs.

Figure 4.16: **Migrating multiple virtual machines using sequential and parallel migration policies.**

the sequential policy network utilization never exceeded 40%, while for the parallel migration policy, peak network utilization never exceeded 70%, even when four VMs are migrated in parallel. For this experiment, the Udom0 (which performs migration) is configured to have 2 virtual CPUs and 2GB RAM, as discussed before.

To determine whether increasing the number of virtual CPUs assigned to the Udom0 can increase network utilization (and thereby reduce overall VM migration time), we repeated the experiments with the Udom0 configured to have 4 virtual CPUs and 2 GB RAM. Figure 4.16(b) shows that in this case, the time to migrate 2 VMs remains relatively unchanged and so does the network utilization (at 40%). When 4 VMs are migrated, the Udom0 is able to exploit the underlying parallelism in the host to complete migration faster. However, this comes at a cost, and the network utilization of the host shoots to 100%.

4.6.3 VM Downtime

Recall from Section 3.6 that dependent co-located VMs are migrated in the order specified by the VM dependency graph, *i.e.,* all children of a VM must be paused before it is paused, and vice versa for resumption. Typically, this means that a client's UdomUs that are serviced by several SDs (which may themselves be serviced by other SDs) must be paused before the SDs, and must be resumed on the target host only after all the SDs have been resumed. Thus, the length of a *dependency chain* in this graph affects the performance of the stop-and-copy phase.

To evaluate the down time of a UdomU serviced by several SDs, we created dependency chains of varying length using the SET_BACKEND rule, *i.e.,* we created a chain of SDs $sd_1 \rightarrow sd_2 \rightarrow \ldots \rightarrow sd_n \rightarrow udomu$ each of which was the backend for another. We migrated these

Chain length	**Down time (ms)**
1	97±4
2	308±3
3	528±8
4	778±7

Figure 4.17: **Down time for migrating VMs.**

VMs from one host to another, and measured the down time of the *udomu*. We only used the parallel migration policy for this experiment. Figure 4.17 presents the result of this experiment, showing the number of VMs in the dependency chain. As expected, the result shows that the down time increases with the length of this chain, adding ∼200ms for each VM in the chain.

Chapter 5

Related Work

This chapter compares SSC with prior work in related areas: security and privacy of client VMs in the cloud, extending the functionality of the cloud, accountability in the cloud and related techniques in software-defined networking.

5.1 Security and Privacy of Client VMs

Popular cloud services, such as Amazon's EC2 and Microsoft's Azure rely on hypervisor-based VMMs (Xen [2] and Hyper-V [37], respectively). In such VMMs, the TCB consists of the hypervisor and an administrative domain. Prior attempts to secure the TCB have focused on both these entities, as discussed below.

Historically, hypervisors have been considered to be a small layer of software. Prior work has argued that the architecture of hypervisors resembles that of microkernels [26]. The relatively small code size of research hypervisors [36, 52, 57], combined with the recent breakthrough in formally verifying the L4 microkernel [32], raises hope for similar verification of hypervisors. However, commodity hypervisors often contain several thousand lines of code (*e.g.,* 150K LoC in Xen 4.1) and are not yet within the realm of formal verification. Consequently, researchers have proposed architectures that completely eliminate the hypervisor [31].

The main problem with these techniques (*i.e.,* small hypervisors and hypervisor-free architectures) is that they often do not support the rich functionality that is needed in cloud computing. Production hypervisors today need to support different virtualization modes, guest quirks, hardware features, and software features like memory deduplication and migration. In SSC, we work with a commodity hypervisor-based VMM (Xen), but assume that the hypervisor is part of the TCB. While this exposes an SSC-based VMM to attacks directed against hypervisor vulnerabilities, it also allows the SSC model to largely resemble commodity cloud computing. Recent advances to strengthen hypervisors against certain classes of attacks [64] can also be applied to SSC, thereby improving the overall security of the platform.

In comparison to hypervisors, the administrative domain is large and complex. It typically executes a complete OS kernel with device drivers and a user-space control toolstack. The hypervisor gives the administrative domain privileges to control and manipulate client VMs. The complexity of the administrative domain has made it the target of a number of attacks [13–17,28],

To address threats against the administrative domain, the research community has focused on adopting the principle of separation of privilege, an approach that we also adopted in SSC. Murray *et al.* [38] disaggregated the administrative domain by isolating in a separate VM the functionality that builds new VMs. This domain builder has highly-specific functionality and a correspondingly small code-base. This feature, augmented with the use of a library OS enhances the robustness of that code. Murray *et al.*'s design directly inspired the use of domB in SSC. Disaggregation is also advocated by Nova [57]. The Xoar project [11] extends this approach by "sharding" different parts of the administrative toolstack into a set of domains. Previous work has also considered separate domains to isolate device drivers [34], which are more defect-prone than the rest of the kernel.

SSC is similar to these lines of research because it also aims to reduce the privilege of Sdom0, which can no longer inspect the code, data and computation of client VMs. However, SSC is unique in delegating administrative privileges to clients (via Udom0). It is this very feature that enables clients to deploy custom services to monitor and control their own VMs.

The CloudVisor project [68] leverages recent advances in nested virtualization technology to protect the security and privacy of client VMs from the administrative domain. In CloudVisor, a commodity hypervisor such as Xen executes atop a small, trusted, bare-metal hypervisor. This trusted hypervisor intercepts privileged operations from Xen, and cryptographically protects the state of client VMs executing within Xen from its dom0 VM, *e.g.,* dom0 only has an encrypted view of a client VM's memory.

The main advantage of CloudVisor over SSC is that its TCB only includes the small, bare-metal hypervisor, comprising about 5.5KLOC, whereas SSC's system-wide TCB includes the entire commodity hypervisor and domB. Moreover, the use of cryptography allows CloudVisor to provide strong guarantees on client VM security and privacy. However, SSC offers three concrete advantages over CloudVisor. First, SSC offers clients more flexible control over their own VMs than CloudVisor. For example, because CloudVisor only presents an encrypted view of a client's VM to dom0, many security introspection tools (*e.g.,* memory introspection, as in Section 4.3) cannot be implemented within dom0. Second, unlike CloudVisor, SSC does not rely on nested virtualization. Nesting fundamentally imposes overheads on client VMs because privileged operations must be handled by both the bare-metal and nested hypervisors, which can slow down I/O intensive client applications, as reported in the CloudVisor paper. Third, SSC's MTSDs allow the cloud provider and clients to execute mutually-trusted services for regulatory compliance. It is unclear whether the CloudVisor model can achieve mutual trust of shared services.

Finally, the Excalibur system [51] operates under the same threat model as SSC, and aims to prevent malicious cloud system administrators from accessing client data. It introduces a new abstraction, called policy-sealed data, which allows encrypted client data to only be decrypted on nodes that satisfy a client-specified policy, *e.g.,* only those running the CloudVisor hypervisor, or those located in a particular geographic region. However, Excalibur's threat model excludes certain classes of attacks via the dom0 management interface, *e.g.,* attacks via direct memory inspection, that SSC explicitly addresses.

5.2 Extending the Functionality of VMMs

There has been nearly a decade of research on novel services enabled by virtualization, starting with Chen and Noble's seminal paper [6]. These include new techniques to detect security infections in client VMs (*e.g.,* [1,7,20]), arbitrary rollback [19], and VM migration [8]. However, most of these techniques are implemented within the hypervisor or the administrative domain. On current cloud infrastructures, deploying these techniques requires the cooperation of the cloud provider, which greatly limits their impact. SSC enables clients to deploy their own privileged services without requiring the cloud provider to do so. The primary advantage of such an approach is that clients need no longer expose their code and data to the cloud provider.

The xCloud project [65,66] also considers the problem of providing clients flexible control over their VMs. The original position paper [65] advocated several approaches to this problem, including by extending hypervisors, which may weaken hypervisor security. The full paper [66] describes XenBlanket, which realizes the vision of the xCloud project using nested virtualization. XenBlanket implements a "blanket" layer that allows clients to execute paravirtualized VMMs atop commodity cloud infrastructures. The key benefit of XenBlanket over SSC is that it provides clients the same level of control over their VMs as does SSC but without modifying the hypervisor of the cloud infrastructure. However, unlike SSC, XenBlanket does not address the problem of protecting the security and privacy of client VMs from cloud administrators.

It may be possible to achieve the goals of both CloudVisor and XenBlanket using two levels of nesting. However, research has shown that the overheads of nesting grow exponentially with the number of nested levels [30].

5.3 Cloud Accountability

There is a line of work that targets cloud provider accountability, which concerns the correctness of the services that the cloud provides and accounting for the resources consumed. Most prior work in this area concerns accounting the usage of particular kinds of resources (such as memory or CPU) or correctness of a specific service provided by the provider.

ALIBI [5] aims to provide verifiable resource accounting for cloud clients by using nested virtualization. It places a trusted hypervisor below providers commodity hypervisor to monitor the CPU and memory usage of the clients virtual machines. SSC allows clients to trust resource usage billing by enabling trusted resource metering through MTSDs.

CloudProof [45] provides cloud storage to the customer in which violation of integrity, write-serializability and freshness can detected and also provide mechanism to prove the violation if occurred. Hourglass Schemes [60] aim to detect if the provider has advertised encrypted storage service but does not really encrypt it. SSC solves these problems by giving control to the clients over their data and computation through SDs and allowing them to implement such services themselves.

5.4 Techniques based on Software-defined Networking

SDN technologies allow programmatic control over the network's control elements. Clients implement policies using a high-level language, and SDN configures individual network elements to enforce these policies. The SDN-based effort most closely related to SSC is CloudNaaS [3], which develops techniques allow clients to flexibly introduce middleboxes. Recent work on SIMPLE [46] enhances this basic model to allow composition of middleboxes.

We view this line of work as being complementary to SSC. SSC enables a number of new features that cannot be implemented using SDN alone—protecting client VMs from cloud operators, endowing SDs with specific privileges over client VMs via GRANT_PRIVILEGE, specifying rich inter-VM dependencies, and offering VM dependency-aware migration policies. SSC currently uses Open vSwitch-based VMs to suitably route traffic to client VMs that have a network middlebox hosted on a different physical machine. It may be possible to leverage SDN technology to enable such routing.

Chapter 6

Conclusion

The objective of this dissertation is to solve two key issues faced by the organizations that wish to adopt cloud computing. First, the security and privacy of their data in the public cloud and second, relinquishing their control over their computation. This dissertation presented a new cloud computing model, SSC, that improves client security and privacy, and gives clients the flexibility to deploy privileged services for their own VMs. SSC introduces new abstractions, a supporting privilege model and an infrastructure to achieve these goals. We integrated SSC with a commodity hypervisor (Xen), and presented case studies showing SSC's benefits.

SSC provides clients with unprecedented flexibility to deploy customized cloud-based services and holds clients responsible for administering their own VMs. However, this does not necessarily mean that clients need to have increased technical knowhow or manpower to leverage the benefits of SSC, *e.g.,* to implement their own services as SDs. Cloud providers can ease the deployment path for SSC by following an *SD app store* [55] model akin to mobile application markets. Both cloud providers as well as third-party developers can contribute SDs to such an app store, from where they can be downloaded and used by clients.

One of the main advertised benefits of cloud computing is that it frees clients from having to administer their own VMs. By allowing clients to administer their own VMs, SSC apparently diminishes this benefit. We feel that this is a fundamental tradeoff, and the price that clients need to pay for increased security, privacy, and control over their VMs. One of the consequences of this tradeoff is that clients without the appropriate technical knowhow may commit administrative errors, *e.g.,* giving a UdomU or an SD more privileges than it needs. Nevertheless, SSC ensures that the effects of such mistakes are confined to the client's meta-domain, and do not affect the operation of other clients on the same platform.

Finally, SSC restricts some virtualization features for the cloud provider like memory sharing between VMs of different clients. Cloud providers can use VM memory sharing to reduce cost but SSC by design restricts this functionality among VMs of different clients. Thus potentially increasing the cost for the cloud provider. This is a fundamental tradeoff for providing more security and isolation to the cloud clients.

Below we present a few future directions for extending and improving SSC:

- **VM placement and load balancing:** The current SSC prototype implements a very basic VM placement policy. Upon receiving the specifications from the client, SSC's cloud controller generates a VM placement plan taking into account for the current load on various hosts. This VM placement is static in nature and SSC prototype lacks dynamic load balancing.

  One of the main advantage SSC's cloud controller has over traditional controllers is the availability of inter-VM dependencies. This information can be used for better VM placement but current prototype does not avail this opportunity. Further investigation is needed regarding how inter-VM dependencies can help in the VM placement decisions and also how load balancing should be implemented in SSC.

- **Mutually trusted service domain (MTSD) verification:** Currently a third party, mutually trusted by provider and the client, is assigned for the verification of MTSDs. This third party has to verify and attest that the privacy and integrity of client's data and computation is not being compromised by MTSDs.

  In this dissertation we have not explored the verification process of the MTSDs. For better usability and adoption of SSC, tools are needed for easing the process of MTSD verification. Further techniques should be explored to establish trust on MTSDs like proof-carrying code [40,41] or using formally verified micro-kernels for MTSDs [33,39].

- **Real world deployment:** The other area we have not investigated in this dissertation is the real world deployment of SSC, used by real users. All the evaluations in this dissertation are done on a couple of physical machines. The production scale evaluation of SSC is needed to enable it for such environment. Also to improve SSC's usability, user study is needed to measure SSC’s effects on the user experience in the production environment.

Bibliography

[1] A. Baliga, V. Ganapathy, and L. Iftode. Detecting kernel-level rootkits using data structure invariants. *IEEE TDSC*, 8(5), 2011.

[2] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *ACM SOSP*, 2003.

[3] Theophilus Benson, Aditya Akella, Anees Shaikh, and Sambit Sahu. Cloudnaas: a cloud networking platform for enterprise applications. In *SoCC*, page 8, 2011.

[4] S. Berger, R. Caceres, K. Goldman, R. Perez, R. Sailer, and L. van Door. vTPM: Virtualizing the Trusted Platform Module. In *USENIX Security*, 2006.

[5] Chen Chen, Petros Maniatis, Adrian Perrig, Amit Vasudevan, and Vyas Sekar. Towards verifiable resource accounting for outsourced computation. In *VEE*, pages 167–178, 2013.

[6] P. M. Chen and B. Noble. When virtual is better than real. In *HotOS*, 2001.

[7] M. Christodorescu, R. Sailer, D. Schales, D. Sgandurra, and D. Zamboni. Cloud Security Is Not (Just) Virtualization Security. In *ACM Cloud Computing Security Workshop*, 2009.

[8] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live Migration of Virtual Machines. In *USENIX NSDI*, 2005.

[9] Cloud Security Alliance. The Notorious Nine: Cloud Computing Top Threats in 2013. `https://cloudsecurityalliance.org/the-notorious-nine-cloud-computing-top-threats-in-2013/`.

[10] Cloud Security Alliance. Top Threats to Cloud Computing, Version 1.0. `https://cloudsecurityalliance.org/topthreats/csathreats.v1.0.pdf`.

[11] P. Colp, M. Nanavati, J. Zhu, W. Aiello, G. Coker, T. Deegan, P. Loscocco, and A. Warfield. Breaking Up is Hard to Do: Security and Functionality in a Commodity Hypervisor . In *ACM SOSP*, 2011.

[12] Brendan Cully, Geoffrey Lefebvre, Dutch T. Meyer, Mike Feeley, Norman C. Hutchinson, and Andrew Warfield. Remus: High availability via asynchronous virtual machine replication. In *NSDI*, 2008.

[13] CVE-2007-4993. Xen guest root escapes to dom0 via pygrub.

[14] CVE-2007-5497. Integer overflows in libext2fs in e2fsprogs.

[15] CVE-2008-0923. Directory traversal vulnerability in the shared folders feature for VMWare.

[16] CVE-2008-1943. Buffer overflow in the backend of XenSource Xen paravirtualized frame buffer.

[17] CVE-2008-2100. VMWare buffer overflows in VIX API let local users execute arbitrary code in host OS.

[18] B. Danev, R. Masti, G. Karame, and S. Capkun. Enabling secure VM-vTPM migration in private clouds. In *ACSAC*, 2011.

[19] G. Dunlap, S. T. King, S. Cinar, M. Basrai, and P. Chen. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. In *USENIX/ACM OSDI*, 2002.

[20] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *ACM SOSP*, 2003.

[21] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *NDSS*, 2003.

[22] Gartner. Assesing the Security Risks of Cloud Computing. `http://www.gartner.com/DisplayDocument?id=685308`.

[23] Gartner. Market Trends: Application Development Software, Worldwide, 2012-2016. `http://www.gartner.com/DisplayDocument?id=2098416`.

[24] J. T. Giffin. *Model Based Intrusion Detection System Design and Evaluation*. PhD thesis, University of Wisconsin-Madison, 2006.

[25] Trusted Computing Group. TPM main spec., l2 v1.2 r116. `http://www.trustedcomputinggroup.org/resources/tpm_main_specification`.

[26] S. Hand, A. Warfield, K. Fraser, E. Kotsovinos, and D. Magenheimer. Are VMMs Microkernels Done Right? In *HotOS*, 2005.

[27] iperf3. `http://code.google.com/p/iperf/`.

[28] K. Kortchinsky. Hacking 3D (and breaking out of VMWare). In *BlackHat USA*, 2009.

[29] B. Kauer. OSLO: Improving the Security of Trusted Computing. In *USENIX Security*, 2007.

[30] B. Kauer, P. Verissimo, and A. Bessani. Recursive virtual machines for advanced security mechanisms. In *1st International Workshop on Dependability of Clouds, Data Centers and Virtual Computing Environments*, 2011.

[31] E. Keller, J. Szefer, J. Rexford, and R. Lee. Eliminating the hypervisor attack surface for a more secure cloud. In *ACM CCS*, 2011.

[32] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal Verification of an OS Kernel. In *ACM SOSP*, 2009.

[33] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: formal verification of an os kernel. In *SOSP*, pages 207–220, 2009.

[34] J. LeVasseur, V. Uhlig, J. Stoess, and S. Gotz. Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines. In *ACM/USENIX OSDI*, 2004.

[35] L. Litty, H. A. Lagar-Cavilla, and D. Lie. Hypervisor Support for Identifying Covertly Executing Binaries. In *USENIX Security*, 2008.

[36] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. TrustVisor: Efficient TCB reduction and attestation. In *IEEE Symposium on Security & Privacy*, 2010.

[37] Microsoft. Hyper-V Architecture. `http://msdn.microsoft.com/en-us/library/cc768520(BTS.10).aspx`.

[38] D. Murray, G. Milos, and S. Hand. Improving Xen Security Through Disaggregation. In *ACM VEE*, 2008.

[39] Toby C. Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. sel4: From general purpose to a proof of information flow enforcement. In *IEEE Symposium on Security and Privacy*, pages 415–429, 2013.

[40] George C. Necula. Proof-carrying code. In *POPL*, pages 106–119, 1997.

[41] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In *OSDI*, pages 229–243, 1996.

[42] B. Payne, M. Carbone, and W. Lee. Secure and Flexible Monitoring of Virtual Machines. In *ACSAC*, 2007.

[43] B. Payne, M. Carbone, M. Sharif, and W. Lee. Lares: An architecture for secure active monitoring using virtualization. In *IEEE Symposium on Security & Privacy*, 2008.

[44] B. Pfaff, J. Pettit, K. Amidon, M. Casado, T. Koponen, and S. Shenker. Extending networking into the virtualization layer. In *HotNets*, 2009.

[45] Raluca Ada Popa, Jacob R. Lorch, David Molnar, Helen J. Wang, and Li Zhuang. Enabling security in cloud storage slas with cloudproof. In *Proceedings of the 2011 USENIX conference on USENIX annual technical conference*, pages 31–31, 2011.

[46] Z. Qazi, C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu. SIMPLE-fying middlebox policy enforcement using SDN. In *SIGCOMM*, 2013.

[47] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *ACM CCS*, 2009.

[48] Sage Cutely. Mine BCN, Monero, BTC and Others with Nvidia Grid Test Drive! `http://thecleangame.net/2014/05/mine-bcn-monero-btc-others-nvidia-grid-test-drive/`. Accessed: 2014-07-01.

[49] R. Sailer, T. Jaeger, E. Valdez, R. Caceres, R. Perez, S. Berger, J. Griffin, and L. van Doorn. Building a MAC-based Security Architecture for the Xen Hypervisor. In *ACSAC*, 2005.

[50] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *USENIX Security*, 2004.

[51] N. Santos, R. Rodrigues, K. Gummadi, and S. Saroiu. Policy-sealed data: A new abstraction for building trusted cloud services. In *USENIX Security*, 2012.

[52] A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. In *ACM SOSP*, 2007.

[53] A. Srivastava and J. Giffin. Tamper-resistant, application-aware blocking of malicious network connections. In *RAID*, 2008.

[54] A. Srivastava, H. Raj, J. Giffin, and P. England. Trusted VM snapshots in untrusted cloud infrastructures. In *RAID*, 2012.

[55] Abhinav Srivastava and Vinod Ganapathy. Towards a richer model for cloud app markets, 2012.

[56] Abhinav Srivastava and Jonathon T. Giffin. Tamper-resistant, application-aware blocking of malicious network connections. In *RAID*, pages 39–58, 2008.

[57] U. Steinberg and B. Kauer. NOVA: A Microhypervisor-Based Secure Virtualization Architecture. In *ACM Eurosys*, 2010.

[58] TCPDump and libpcap. `http://www.tcpdump.org`.

[59] byte-unixbench: A Unix benchmark suite. `http://code.google.com/p/byte-unixbench`.

[60] Marten van Dijk, Ari Juels, Alina Oprea, Ronald L. Rivest, Emil Stefanov, and Nikos Triandopoulos. Hourglass schemes: how to prove that cloud files are encrypted. In *ACM Conference on Computer and Communications Security*, pages 265–280, 2012.

[61] V. Varadarajan, T. Kooburat, B. Farley, T. Ristenpart, and M. Swift. Resource-freeing attacks: Improve your cloud performance (at your neighbor's expense). In *ACM CCS*, 2012.

[62] vmitools. `http://code.google.com/p/vmitools/`.

[63] C. A. Waldspurger. Memory Resource Management in VMWare ESX Server. In *USENIX/ACM OSDI*, 2002.

[64] Z. Wang and X. Jang. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *IEEE Symposium on Security & Privacy*, 2010.

[65] D. Williams, E. Elnikety, M. Eldehiry, H. Jamjoom, H. Huang, and H. Weatherspoon. Unshackle the Cloud! In *HotCloud*, 2011.

[66] D. Williams, H. Jamjoom, and H. Weatherspoon. The Xen-Blanket: Virtualize Once, Run Everywhere. In *ACM EuroSys*, 2012.

[67] [xen-devel] bidirectional network throughput for netback, July 2013. `http://lists.xen.org/archives/html/xen-devel/2013-07/msg02709.html`.

[68] F. Zhang, J. Chen, H. Chen, and B. Zang. CloudVisor: Retrofitting Protection of Virtual Machines in Multi-tenant Cloud with Nested Virtualization. In *ACM SOSP*, 2011.