On the Control Plane of a Self-service Cloud Platform

Rutgers Department of Computer Science Technical Report 707, filed June 2014

Shakeel Butt
Rutgers and NVidia Inc.

Vinod Ganapathy
Rutgers

Abhinav Srivastava
AT&T Labs–Research

Abstract

Self-service Cloud Computing (SSC) [5] is a recently-proposed model that empowers clients of public cloud platforms in two ways. First, it improves the security and privacy of client data by preventing cloud operators from snooping on or modifying client VMs. Second, it provides clients the flexibility to deploy services, such as VM introspection-based tools, on their own VMs. SSC achieves these goals by modifying the hypervisor privilege model.

This paper focuses on the unique challenges involved in building a control plane for an SSC-based cloud platform. The control plane is the layer that facilitates interaction between hosts in the cloud infrastructure as well as between the client and the cloud. We describe a number of novel features in SSC's control plane, such as its ability to allow specification of VM dependencies, flexible deployment of network middleboxes, and new VM migration protocols. We report on our design and implementation of SSC's control plane, and present experimental evaluation of services implemented atop the control plane.

1. Introduction

In recent years, an increasing number of enterprises have migrated their applications to the cloud. Migrating applications to the cloud offers enterprises a number of attractive benefits, such as freeing them from procuring computing infrastructure, allowing elasticity in the use of computing resources, and offloading a number of management and maintenance tasks to cloud providers.

In this paper, we are interested in *public* cloud infrastructures, such as Amazon EC2 and Microsoft Azure. Despite their apparent popularity, many enterprises hesitate to migrate their applications to such infrastructures, 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 platform, the enterprise must still procure and manage the infrastructure.

We see two main reasons why enterprises are often reluctant to use public cloud infrastructures. The first is security and privacy of client data. Enterprises that host sensitive or proprietary data (*e.g.,* banks and pharmaceutical industries) may wish to protect this data from outsiders. On public cloud infrastructures, the provider controls the virtual machine (VM) management interface. This management interface typically has the privileges to inspect the state of individual VMs hosted on the cloud. For example, it can inspect the memory contents and network traffic of all the work VMs hosted on the platform. Thus, sensitive client data is vulnerable to attacks by malicious cloud operators or exploits directed against vulnerabilities in the management interface [9–14, 16].

The second reason is the inability of clients to flexibly control their VMs. Delegating management of VMs to the cloud provider can reduce operating costs in some cases. But it also has the disadvantage of making the cloud provider responsible for services that are available to the enterprise client. For example, a client may wish to use a custom network intrusion detection system (NIDS) or a memory introspection service on its VMs. It is often unusual for public cloud providers to deploy custom services for individual clients. In the unlikely case that this does happen, each client request to configure or modify these services must go through the cloud provider.

These two problems have garnered significant interest and a number of recently-proposed solutions address one or both of the problems [7, 18, 29, 36, 37]. Our focus is on a recent solution called *Self-service Cloud (SSC)* Computing [5]. SSC aims to address both the problems discussed above by modifying the privilege model of the hypervisor. Traditional hypervisors

endow the management VM (henceforth called *dom0*) with unrestricted privileges over client VMs. In contrast, an SSC hypervisor employs privilege separation to offer two kinds of management VMs: (1) a deprivileged system-wide management domain, used by cloud operators, and (2) per-client management domains, used by clients to administer and manage their own VMs. This new privilege model, discussed in further detail in Section 2, simultaneously prevents cloud operators from snooping on or modifying client VM state, and allows clients to flexibly control their own VMs.

The original SSC paper [5] studiedthe aforementioned privilege model within the hypervisor, and developed protocols to operate VMs on such a hypervisor. The main technical contribution of the present paper is the design and implementation of the control plane of the SSC platform. The control plane 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.

SSC's control plane includes several novel features not usually seen in contemporary cloud infrastructures:

(**1**) ***Specifying VM dependencies.*** SSC's privilege model allows clients to create VMs that have privileges over other VMs. For example, a client can create a memory introspection VM that has the privileges to inspect the memory contents of a work VM for malware infection (*e.g.,* [1, 20, 23, 24]). SSC's control plane presents an enhanced *dashboard* interface via which clients can specify inter-VM dependencies (*e.g.,* that the work VM depends on the memory introspection VM). SSC's control plane uses these dependencies to automatically determine how VMs must be placed on individual hosts. Thus, the memory introspection VM, for instance, will be co-located on the same host as the work VM(s) that it monitors.

(**2**) ***Flexible middlebox deployment.*** Existing cloud infrastructures do not offer a flexible way to deploy network middleboxes. SSC's control plane offers clients the flexibility of plumbing the I/O path of the work VMs, thereby allowing easy deployment of networked services, such as NIDS, trace anonymization services, and network metering. Clients simply specify VM dependencies using the interface described above, and the control plane automatically configures the I/O path to enable middlebox placement.

(**3**) ***New VM migration protocols.*** VM migration is an essential component of elastic cloud infrastructures. Contemporary VM migration tools focus on migrating individual VMs from one host to another. In SSC, VM migration must respect inter-VM dependencies. SSC's control plane includes algorithms to safely migrate VMs in the presence of inter-VM dependencies.

2. The Self-service Cloud Platform

In this section, we present background material on the SSC platform and motivate the need for a control plane.

2.1 Threat Model

SSC uses a practical threat model that we believe applies well in the real world. Similar threat models have also been adopted in other related projects [18, 29, 37].

In SSC, the *cloud provider* is assumed to be an entity such as Amazon, Microsoft, or Rackspace, and is *trusted*. The client believes that the cloud provider has a vested interest to protect its reputation, and that it will not deliberately compromise the security and privacy of the client's code and data. The threat model also assumes that physical machines on the cloud platform are equipped with trusted hardware (*i.e.,* TPM) and IOMMU units to enable I/O virtualization.

Thus, SSC assumes that the cloud provider will supply a trusted computing base (TCB) equipped with a hypervisor that implements the privilege model described in Section 2.2. The original SSC paper included TPM-based protocols that would allow the client to verify the contents of the TCB before starting its VMs on the platform. SSC's TCB also consists of various components in its control plane. We defer a detailed discussion of these components to Section 4.

On the other hand, *cloud operators*, who may include human administrators employed by the cloud provider, are *untrusted*. By extension, we assume that all actions originating from the cloud platform's dom0, including relaying I/O on behalf of client VMs, are untrusted. Note that even if cloud operators are not overtly malicious in intent, dom0 may contain remotely-exploitable vulnerabilities. SSC's threat model protects the security of client VMs against the effects of such exploits as well.

One of the key implications of this threat model is that SSC cannot defend against attacks launched by the

VM	Tasks
Platform-wide VMs, *i.e.,* one instance per physical machine	
Sdom0	Manages hardware and executes device drivers. (Un)Pauses and schedules client VMs. Manages I/O quotas.
domB	Creates new VMs in response to client requests. Virtualizes the TPM and interacts with physical TPM.
Per-client VMs, *i.e.,* at least one instance per client	
Udom0	Administrative interface for all of the client's VMs. Delegates privileges over UdomUs to SDs. Hosts client's SSL private keys for secure communication with client.
UdomU	The client's work VM, possibly multiple UdomUs per client.
SD	Service domains, each performing a specific administrative task over designated UdomUs. Possibly multiple SDs per client.

Figure 1. Various VMs on an SSC platform.

cloud provider itself. This may include denial of service attacks, *e.g.,* failure to schedule the client's VMs for execution, or explicit monitoring of the client's activities, *e.g.,* via government subpoenas. Protecting against such threats may require heavyweight cryptographic techniques.

2.2 Components of SSC

The hypervisor on an SSC platform uses privilege separation to partition the responsibilities traditionally entrusted with dom0 to two new kinds of administrative domains (see Figure 1). *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. For example, it can map the memory of a UdomU, a feature that can be used to implement memory introspection tools, such as rootkit detectors. Likewise, Udom0 can set I/O backends for the UdomU, thereby intercepting the I/O path of the UdomU. This feature can be used to implement intrusion detection by inspecting I/O traffic.

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. An SD can be started by the Udom0, which delegates specific privileges over a target UdomU (or a set of UdomUs) to the SD. For example, Udom0 can create a rootkit detection SD, and give it privileges to map the memory contents of all the client's UdomUs on that platform.

All physical hardware on the platform is controlled by Sdom0. Thus, all I/O performed by a client's VMs is ultimately relayed to and performed by Sdom0. By our threat model, Sdom0 is potentially malicious and can therefore inspect/modify the client's I/O traffic. However, clients can still protect their I/O traffic via interception. For example, the Udom0 could set a UdomU's storage and network I/O backends to be an SD that encrypts all outgoing traffic. Thus, I/O from the UdomU first traverses the SD, which encrypts the traffic before relaying it to the Sdom0, which is therefore unable to inspect the traffic.

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].

A client session to create VM instances on an SSC platform begins with a request to create a Udom0 instance. As discussed above, Udom0 instances are stateless for the most part, and it is therefore possible to use off-the-shelf OS distributions to create Udom0s. Udom0 creation happens via a protocol (described in the original paper [5]) that also installs the client's SSL private key in its memory. Note that Udom0 memory is inaccessible to Sdom0, thereby protecting the private key from cloud operators. This key enables an SSL handshake with the client, thereby allowing the

creation of an encrypted communication channel. The client then transmits future requests for VM creations (*e.g.,* UdomUs and SDs) via this channel to Udom0, which then creates these VMs for the client on the physical host.

3. The Need for a New Control Plane for SSC

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 2.2.

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.

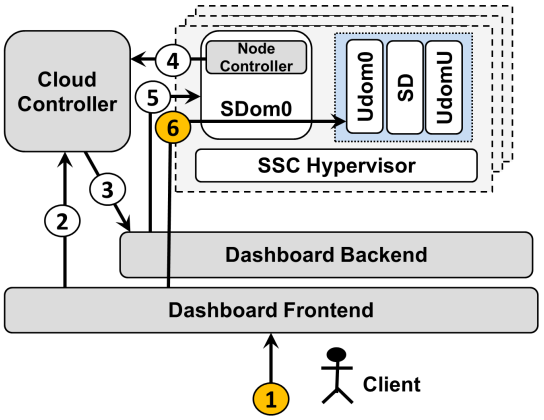
SSC's control plane must provide the illusion of a single administrative interface to the client, while hiding the presence of individual Udom0 instances. The control plane must suitably relay administrative operations from this interface to the corresponding Udom0 instances. It is also responsible for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

4. SSC's 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.

4.1 Components of SSC's Control Plane

SSC's control plane functionally enhances each of the components discussed above 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-



(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 2. Components of the control plane and their interactions. Communications labeled with shaded circles are secured by SSL.

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 2 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 5). These dependencies may imply that certain VMs must be co-located 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 back-end 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 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 2.

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 4.2).

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 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.

4.2 Overall Operation of the 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. 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 ($\mathsf{freshSym}||n_{\text{SSL}}$) encrypted using the AIK public key of the TPM of the dashboard host.[1] Here 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 as $\mathsf{Sig}_{\text{client}}$).

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

Figure 3. Protocol to create a dashboard VM instance.

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 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 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, 28]). 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 propri-

[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.

etary to the cloud provider is in other components of the platform, such as the cloud controller. The dashboard VM simply encodes the protocols discussed in this paper 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 (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, 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 Section 2.2, 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 protocol to start a Udom0 on a host resembles Figure 3, with the major difference that the Udom0 image is also supplied in the first step. The original SSC paper [5] provides the steps of this protocol together with a security analysis, and we elide a description in this paper. 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.

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 6 presents an example).

SSC's control plane provides a language (Figure 4) 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 5, *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

```
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
  Device := STORAGE | NETWORK | ...
Location := MUST_COLOCATE | MAY_COLOCATE
```

Figure 4. 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 5. Example inter-VM dependency specification.

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 colocated (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 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 5 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 6. 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 6. 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 colocation 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 mod-

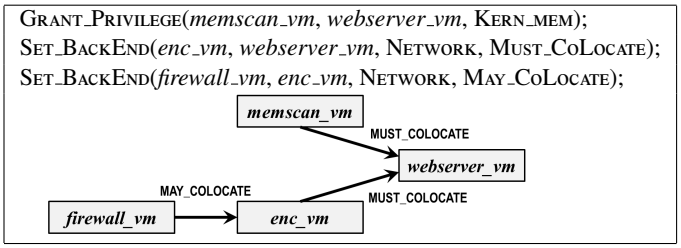


Figure 6. Example showing inter-VM dependencies that require certain VMs to be co-located. We have elided VM declarations and initializations for brevity. Also shown is the VM dependency graph for this example.

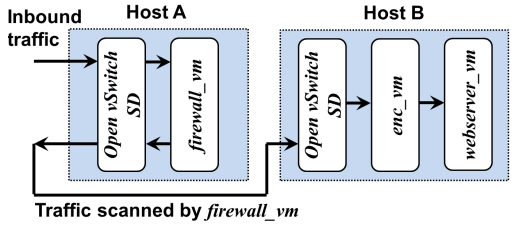


Figure 7. Open vSwitch setup showing the path followed by inbound network traffic to the web server example from Figure 6. Outbound network traffic follows the reverse path. The Udom0 instances and the *memscan_vm* instance on Host B are not shown.

ify 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 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 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 [25] 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 7 illustrates the setup using the VMs for Figure 6 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 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.,* [27, 34]). 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 don't 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 paper.

6. VM Migration

Most cloud platforms employ *live migration* [8] 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 6. 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, …, 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 4.2, 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, …, 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 8. Migrating a group of co-located VMs.

Figure 8 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.

7. Evaluation

The main goal of our experimental evaluation is to understand the performance overhead introduced by the components of SSC's control plane. Other aspects of SSC, such as the overheads involved in creating VMs on an SSC hypervisor and executing system services as SDs (rather than within dom0), were evaluated in the original SSC paper. Those experiments showed that an SSC-based platform was competitive with a traditional Xen-based platform, and focused on evaluating overheads incurred on a *single host*. Performance overheads imposed by SSC were under 5% for system services implemented as SDs in most cases.

We conducted two classes of experiments:

(**1**) ***Networked services.*** SSC's control plane allows networked services to be implemented as SDs. We evaluate the cost of running these SDs both when they are co-located with UdomUs and when they are on a different host.

(**2**) ***Migration.*** We evaluate the cost of migrating VMs and report VM down times for various configurations of the migration algorithm.

7.1 Networked Services

All hosts in our setup execute an SSC-based hypervisor, which we implemented by modifying the privilege model of the Xen-4.3 hypervisor. In all our experiments, we compare the overheads of SSC by comparing the performance of a networked service implemented as an SD against the same network service implemented in a traditional setup (*i.e.,* within dom0 executing on an unmodified Xen-4.3 hypervisor). The hosts in our setup are Dell Poweredge R610 systems equipped with 24GB RAM, eight 2.3GHz Xeon cores with dual threads (16 concurrent executions), FusionMPT SAS drives, and a Broadcom NetXtreme II gigabit NIC. All virtual machines in the experiments (dom0, domU, Sdom0, Udom0, UdomU, SD, domB) were configured to have 2GB RAM and 2 virtual CPUs.

Baseline Overhead. Our first experiment aims to evaluate the baseline overhead of running networked services atop the SSC control plane. 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 9. We evaluate both setups, and use the 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.

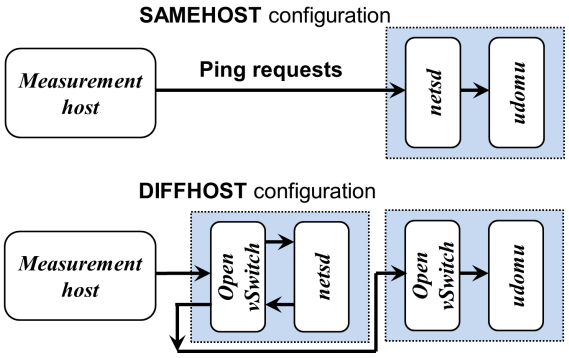


Figure 9. The network topologies used to evaluate the baseline overhead of networked services executing atop SSC. We only show the inbound network path. The outbound path is symmetric.

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 10. Baseline overhead of networked services.

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 [15] 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 10 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-4.3 uses only one kthread in netback to process domU's transmit and receive queues [19]. 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 9). 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.[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 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.

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 11. Network Access Control Service.

Figure 11 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).

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. What is therefore 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 [6] 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 SSC support for *mutually-trusted service domains (MTSDs)*, which were introduced in the original paper on SSC [5]. 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 cloud provider agree upon the software that will be used to account for the client's network bandwidth utilization. This meter-

[2] 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	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 12. Trustworthy Network Metering Service.

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 13. Network intrusion detection (Snort) service.

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

Figure 14. Time to establish a TCP connection in VMWall.

ing 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 [33] 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 12 shows the impact of network metering service on the network throughput of a single work VM (setup similar to Figure 9). As before, the additional overhead imposed over the baseline is minimal.

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[33] 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 has a *May_Colocate* dependency on the UdomU(s) it monitors. Figure 13 presents the results of our experiments and shows that the overhead imposed by an SD implementation of Snort is minimal.

VMWall service. VMWall [31] 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.

Our re-implementation of VMWall uses libvmi [23, 35] for memory introspection. Figure 14 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 that in the traditional setup, TCP connections are established within dom0 itself. In SSC, connection requests are forwarded from Sdom0 to the VMWall SD, resulting in overhead.

7.2 Evaluating VM Migration

We measure the performance of VM migration using two metrics: *VM down time* and *overall migration time*. Recall from Section 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 experi-

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

Figure 15. Total migration time for one virtual machine.

# 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 16. Migrating multiple virtual machines using sequential and parallel migration policies.

ment, 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 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 Section 7.1 except when otherwise mentioned.

Migrating a Single VM. Figure 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 7.2).

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 trade-off is that while parallel migration approach can lead to lower migration times, it can saturate the network link and increase CPU utilization.

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

Figure 17. Down time for migrating VMs.

Figure 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 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 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%.

VM Downtime. Recall from Section 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 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 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.

8. Related Work

Techniques based on Nested Virtualization. Turtles [2] demonstrated that nested virtualization is possible with relatively low overheads. Since then, nesting has been implemented in popular virtualization platforms such as KVM. A number of recent projects have exploited this trend to provide many of the features that SSC does.

CloudVisor [37] uses nesting to protect the contents of user VMs from cloud operators. It does so using a small bare-metal hypervisor atop which the cloud's hypervisor executes. The bare-metal hypervisor only exports an encrypted view of the client's VMs to dom0. CloudVisor's approach has the advantage of presenting a small TCB and using strong cryptographic techniques to protect security. However, it is unclear whether CloudVisor can be extended to implement SD-like services on client VMs.

XenBlanket [36] uses nesting to allow clients to implement their own services on their VMs. XenBlanket is a hypervisor-like layer that executes atop the cloud's hypervisor. The client can deploy SD-like services and administer its VMs, all of which execute atop XenBlanket. However, XenBlanket does not aim to protect the security of client VMs from cloud operators.

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 [17].

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 [26] 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.

Other Work Related to SSC. Aside from work in the two broad areas discussed above, SSC is also related more broadly to work on improving the client VM security in cloud environments. Efforts in this direction include reducing the size of the cloud's TCB (*e.g.,* using tiny hypervisors [21, 30, 32] and hardware-only techniques [18]). SSC takes a practical approach by working with a commodity hypervisor. While this has the implication of having a TCB of relatively large size, it also ensures that popular hypervisor features, often developed in response to customer demand, are supported by SSC. In taking this approach, SSC follows the lead of the work by Murray *et al.* [22] and Xoar [7].

9. Summary

The SSC platform, originally introduced in prior work [5], offers a new hypervisor privilege model that protects client code and data from cloud operators while simultaneously giving clients greater and more flexible control over their VMs running in the cloud. While the original paper on SSC focused on the hypervisor privilege model, it left largely unaddressed the question of how to build a distributed cloud computing platform using such hypervisors.

This paper fills that gap, and presents the design and implementation of SSC's control plane. Novel features of this control plane include giving clients the ability to specify inter-VM dependencies, allowing flexible placement of middleboxes, and new VM migration protocols to support these features. With the control plane in place, SSC can serve as an alternative to current cloud platforms, giving clients greater security and flexibility over their VMs in the cloud setting.

References

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

[2] M. Ben-Yahuda, M. D. Day, Z. Dubitsky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B. Yassour. The Turtles project: Design and implementation of nested virtualization. In *USENIX/ACM OSDI*, 2010.

[3] T. Benson, A. Akella, A. Shaikh, and S. Sahu. Cloudnaas: a cloud networking platform for enterprise applications. In *SoCC*, 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] S. Butt, A. Lagar-Cavilla, A. Srivastava, and V. Ganapathy. Self-service cloud computing. In *ACM CCS*, 2012.

[6] C. Chen, P. Maniatis, A. Perrig, A. Vasudevan, and V.Sekar. Verifiable resource accounting for outsourced computation. In *VEE*, 2013.

[7] 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.

[8] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: High availability via asynchronous virtual machine replication. In *NSDI*, 2008.

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

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

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

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

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

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

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

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

[17] B. Kauer, P. Verissimo, and A. Bessani. Recursive virtual machines for advanced security mechanisms. In *Intl. Wkshp. Dep. of Clouds, Data Centers & Virt. Comp. Env.*, 2011.

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

[19] A URL pointing to our discussion with the Xen developers on lists.xen.org, not provided here for anonymity. Please contact us via the PC chair to obtain the URL.

[20] L. Litty, H. A. Lagar-Cavilla, and D. Lie. Computer Meteorology: Monitoring Compute Clouds. In *HotOS*, 2009.

[21] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. TrustVisor: Efficient TCB reduction and attestation. In *IEEE S&P*, 2010.

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

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

[24] B. Payne, M. Carbone, M. Sharif, and W. Lee. Lares: An architecture for secure active monitoring using virtualization. In *IEEE S&P*, 2008.

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

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

[27] 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.

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

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

[30] 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.

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

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

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

[34] 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.

[35] vmitools. http://code.google.com/p/vmitools/.

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

[37] 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.