Published in Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS'12), October 2012.

Self-service Cloud Computing

Shakeel Butt
Rutgers University

H. Andrés Lagar-Cavilla
GridCentric Inc.

Abhinav Srivastava
AT&T Labs-Research

Vinod Ganapathy
Rutgers University

ABSTRACT

Modern cloud computing infrastructures use virtual machine monitors (VMMs) that often include a large and complex administrative domain with privileges to inspect client VM state. Attacks against or misuse of the administrative domain can *compromise client security and privacy*. Moreover, these VMMs provide clients *inflexible control* over their own VMs, as a result of which clients have to rely on the cloud provider to deploy useful services, such as VM introspection-based security tools.

We introduce a new self-service cloud (SSC) computing model that addresses these two shortcomings. SSC splits administrative privileges between a system-wide domain and per-client administrative domains. Each client can manage and perform privileged system tasks on its own VMs, thereby providing flexibility. The system-wide administrative domain cannot inspect the code, data or computation of client VMs, thereby ensuring security and privacy. SSC also allows providers and clients to establish mutually trusted services that can check regulatory compliance while respecting client privacy. We have implemented SSC by modifying the Xen hypervisor. We demonstrate its utility by building user domains to perform privileged tasks such as memory introspection, storage intrusion detection, and anomaly detection.

Categories and Subject Descriptors. D.4.6 [**Operating Systems**]: Security and Protection
General Terms. Design, Experimentation, Management, Security
Keywords. cloud computing, security, trust, privacy

1. INTRODUCTION

Modern cloud infrastructures rely on virtual machine monitors (VMMs) to flexibly administer and execute client virtual machines (VMs). VMMs 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.

Endowing dom0 with such privileges leads to two problems:

• ***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. This privilege can be misused by attacks against the dom0 software stack (*e.g.,* because of vulnerabilities or misconfigurations) and malicious system administrators. This is a realistic threat [10, 11, 12, 13, 14, 19, 23], 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, 18], 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.

Recent work has investigated the use of nested virtualization [4] to develop solutions for each of the above problems. The CloudVisor project [52] uses nesting to protect client VMs from an untrusted dom0, thereby addressing security and privacy concerns. The XenBlanket project [51] uses nesting to address the problem of inflexible control and cloud provider "lock-in," and allows clients to implement their own services. However, while these projects address the individual concerns above, they do not address both of them simultaneously. In theory, it may be possible to combine the goals of these projects by recursively nesting one within the other (*i.e.,* two levels of nested virtualization). However, on commodity x86 hardware, the use of nested virtualization imposes unwanted performance penalties on client VMs (as compared to the case of non-nested virtualization). These overheads can grow exponentially as client VMs are placed at deeper levels of nesting because of the need to emulate traps within each nested hypervisor (*e.g.,* see [25, Section 3]). Therefore, a straightforward combination of CloudVisor and XenBlanket will likely be unpalatable to clients.

We take a fundamentally different approach to addressing these problems. Our main contribution is a new ***Self-service Cloud (SSC) Computing*** model that simultaneously addresses the problems of security/privacy and inflexbile control. Our main observation is that *both of the above problems are a direct consequence of the way in which commodity hypervisors assign privilege to VMs*. SSC introduces a novel privilege model that reduces the power of the administrative domain and gives clients more flexible control over their own VMs. SSC's privilege model splits the responsibilities traditionally entrusted with dom0 between a new system-wide administrative domain (*Sdom0*) and per-user administrative domains (*Udom0s*), service domains (*SDs*) and mutually-trusted service domains (*MTSDs*) that are described in more detail below. By introducing a new privilege model, SSC addresses both of the above problems *without requiring nested virtualization* and the performance overhead that it entails.

Udom0 (User dom0) is a per-user administrative domain that can monitor and control the set of VMs of a particular client. When

CCS'12, October 16–18, 2012, Raleigh, North Carolina, USA.

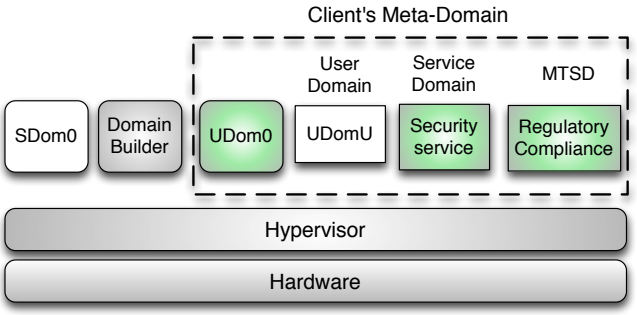

Figure 1. The design of a Self-service Cloud (SSC) computing platform. SSC splits the TCB of the system (indicated using the shaded components) into a *system-level TCB*, with the hardware, the SSC hypervisor, and the domain builder, and a *client-level TCB*, with the Udom0 and service domains.

a client attempts to start a VM in SSC, it is assigned its own Udom0 domain. This domain creates the user VMs that perform the actual work for the client (***UdomUs***). Udom0 can delegate its privileges to ***service domains (SDs)***, which are special-purpose user domains that can perform privileged system services on UdomUs. Clients can leverage SDs to implement services such as memory introspection to verify VM integrity, intrusion detection, and storage encryption. In a traditional cloud, these services would be implemented in dom0, and would have to be deployed by the cloud provider. Thus, SSC allows clients to flexibly deploy new services and control their own VMs using SDs.

Sdom0 (System dom0) is the system-wide administrative domain in SSC. Sdom0 retains the privileges to start/stop Udom0 domains upon request by clients, and to run drivers for virtualized devices. Sdom0 manages resources, including scheduling time-slices and I/O quotas. SSC's privilege model disallows Sdom0 from inspecting the state of the client's domains (Udom0s, SDs, and UdomUs), thereby ensuring the security and privacy of client VMs.

Although this privilege model allows SSC to achieve our stated goals, in practice, cloud providers typically require *some* ability to control client VMs for regulatory compliance. For example, providers may wish to ensure that clients are not misusing their cloud infrastructure to host malicious software [30]. To do so, the cloud provider must have the ability to inspect client VMs, but this 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 resolves this tension by introducing ***mutually-trusted service domains (MTSDs)***. The cloud provider and the client mutually 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 [5, 21, 24] 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 their privacy will not be compromised. Likewise, the cloud provider can ensure liveness of MTSDs for regulatory compliance.

Figure 1 depicts the design of SSC. We use the term ***meta-domain*** to refer to the collection of a client's domains (Udom0, UdomUs, SDs, and MTSDs). Only Udom0 holds privileges over UdomUs in its meta-domain, but can delegate specific privileges to SDs to carryout specialized services. To bootstrap meta-domains, SSC employs a specialized *domain builder (domB)* (akin to the design of Murray *et al.* [33]). DomB is entrusted with the task of creating VMs, a privilege that no longer resides with the system-wide administrative domain (Sdom0). Section 3 presents a detailed overview of the design of SSC.

We have implemented an SSC-compliant VMM by modifying Xen (v3.4.0). We demonstrate SSC's utility by showing that SDs can be used to implement a variety of services (Section 4). To summarize, the main contributions of this paper are:

- ***The SSC model.*** SSC's privilege model allows clients to administer their own VMs, while disallowing the cloud's administrative domain from inspecting client VM state. This new privilege model addresses both client concerns discussed earlier, *i.e.,* security/privacy of VMs, and inflexible control over VMs. Unlike recent work to address these concerns [51, 52], SSC does *not* rely on nested virtualization. Consequently, client VMs are able to enjoy the benefits of SSC with negligible performance overheads.
- ***Service domains.*** SDs allow clients to perform privileged system tasks on their VMs. We demonstrate case studies showing that SDs can be used to implement a variety of system services that are traditionally implemented in dom0. Consequently, SDs provide clients more flexible control over their VMs.
- ***Mutually-trusted service domains.*** MTSDs execute privileged services that check regulatory compliance in a manner that is mutually agreed upon between the cloud provider and the client. They balance the tension between the cloud provider's need to retain control and the client's security and privacy goals.

2. THREAT MODEL

SSC's threat model is similar to those used in recent work on protecting client VMs in the cloud [26, 40, 52], and differentiates between *cloud service providers* and *cloud system administrators*. Cloud providers are entities such as Amazon EC2 and Microsoft Azure, who have a vested interest in protecting their reputations. On the other hand, cloud system administrators are individuals entrusted with system tasks and maintaining the cloud infrastructure. To do so, they have 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 [10, 11, 12, 13, 14, 23] 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-concious clients can employ end-to-end encryption to protect data on the network and storage. We show how clients can achieve this by implementing suitable SDs (Section 4.1). Packet headers need not be encrypted; after all, network middleboxes inspect and mangle packet headers.

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 which 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.

3. THE SSC PLATFORM

We now describe the design and implementation of the SSC platform, focusing on the new abstractions in SSC, their operation, and SSC's privilege model.

3.1 Components

As Figure 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 prevents Sdom0 from inspecting the contents of client meta-domains.

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 2); consequently, Sdom0 is not part of the TCB.

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 and Figure 3). Images of domains to be created are passed by attaching storage volumes containing this information.

3.2 Bootstrapping

Hosts in the cloud infrastructure are assumed to be equipped with TPM and IOMMU hardware, which is available on most modern chipsets. We assume that the TPM is virtualized, as described in prior work [5]. The supporting user-level daemons for the virtualized TPM (vTPM) run within domB, which is in the TCB, and interact with the hardware TPM on the physical host. The protocols described in this section assume client interaction with a vTPM instance. We use the vTPM protocols as described in the original paper [5], although it may also be possible to use recently-proposed variants [15]. The vTPM can cryptographically attest the list of software packages loaded on a system in response to client requests; such attestations are called *measurements* [39].

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 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 allow Sdom0 access to only the pages that it 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.

3.3 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 [5, 39].

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 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 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 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 re-

- 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 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. Summary of new hypercalls introduced to enable SSC. Figure 3 shows their usage.

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

1. client → Sdom0 : n_{TPM}, Udom0_image, $\mathsf{Enc}_{\mathsf{AIK}}(\mathsf{freshSym}||n_{\text{SSL}}||\mathsf{hash}(\text{Udom0\_image}))$, Sig_{client}
2. Sdom0 → domB : CREATE_UDOM0(Udom0_image, n_{TPM}, $\mathsf{Enc}_{\mathsf{AIK}}(\mathsf{freshSym}||n_{\text{SSL}}||\mathsf{hash}(\text{Udom0\_image}))$, Sig_{client}) → ID_{client}
3. domB → client : ID_{client}, $\mathsf{TPMSign}(n_{\text{TPM}}||\mathsf{PCR}), \mathsf{ML}$
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 $\mathsf{freshSym}||n_{\text{SSL}}||\mathsf{hash}(\text{Udom0\_image})||n_{\text{TPM}}$. 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}, $\mathsf{TPMSign}(n_{client}||\mathsf{PCR}), \mathsf{ML}$
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}, $\mathsf{TPMSign}(n_{provider}||\mathsf{PCR}), \mathsf{ML}$
4. domB → Udom0 : ID_{MTSD}, $\mathsf{TPMSign}(n_{client}||\mathsf{PCR}), \mathsf{ML}$
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 3. Protocols used in SSC for the creation of Udom0, UdomUs, SDs and MTSDs.

quests 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 3.2 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 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 [9]. 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 `xend` Python-based toolstack for end-user interaction and to provide an administrative interface to the meta-domain. 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 3.6 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 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.

3.4 SSC Privilege Model

At the heart of SSC 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.

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

Table 1. Actors and operations in the privilege model. Each ✓ in the table denotes that the actor can perform the corresponding operation.

	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**

Table 2. 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 Table 1.

(**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.

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 [28]. 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 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). Table 1 and Table 2 summarize the privilege model of SSC.

We implemented this privilege model in our prototype using the Xen Security Modules (XSM) framework [38]. 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 [38]. 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.

3.5 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). 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 [9], 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.

3.6 Regulatory Compliance using MTSDs

As previously discussed, an MTSD executes within a client meta-domain. The MTSD can request specific privileges over client 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 (see Section 4.2). 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 inadvertantly 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.

We therefore limit the amount of information that an MTSD can transmit outside the meta-domain. MTSDs are not given any persitent storage, and can only communicate with the provider (*i.e.,* Sdom0) via the SSC hypervisor. Further, this communication channel is restricted to be a stream of bits whose semantics is well-understood. That is, each `0` bit in the stream denotes a violation of regulatory compliance, while a `1` bit denotes otherwise.[1] The client can set up a user daemon (*e.g.,* within Udom0) that is awakened by the SSC hypervisor upon every new bit transmitted by the MTSD over this channel. An honest client that does not violate the provider's regulatory compliance policies should therefore only expect to see a stream of `1`s transmitted to Sdom0. Any `0`s

[1] Note that a client cannot modify this stream without tampering with the code of the MTSD. The provider ensures that the MTSD was booted correctly (Figure 3(c)), and SSC's privilege model prevents the client from modifying a running MTSD.

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

Table 3. Cost of building domains.

in the stream either denote an MTSD attempting to steal information, or an inadvertant compliance violation (*e.g.,* due to malware infection). In either case, the client can terminate its meta-domain.

4. EVALUATION

In evaluating our SSC prototype, our main goals were:

(**1**) To demonstrate the flexibility of the SSC model in enabling various virtualization-based services as SDs; and

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

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 configuration of the VM to be created are presented to domB as a virtual disk via the block device interface. Table 3 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 Storage SDs

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 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. Each SD executes as a VM. When Udom0 starts a UdomU that wants to avail the service offered by an SD, it configures the UdomU to advertise the SD as the backend driver for disk operations. The SD itself executes a frontend driver that interfaces within a backend driver running within Sdom0. When UdomU attempts to perform a disk operation, the data first goes to the SD, which is the advertised backend for the UdomU. The SD performs the advertised service,

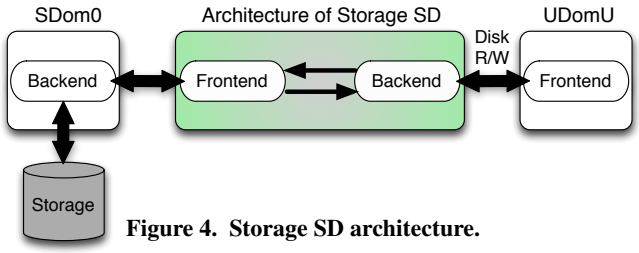

Figure 4. Storage SD architecture.

Platform	Unencrypted (MB/s)	Encrypted (MB/s)
Xen (dom0)	81.72±0.15	71.90±0.19
SSC (SD)	75.88±0.15 (7.1%)	70.64±0.32 (1.5%)

Table 4. Cost incurred by the storage encryption SD. 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.

and passes it to the frontend executing within the SD. In turn, the frontend forwards the (possibly modified) data block to Sdom0's backend, which interacts with the disk to store data persistently.

This setup can also be used to chain SDs, each offering its own service. For example, an encryption SD (see below) can serve as the I/O backend for UdomU. In turn, a checkpointing SD (see Section 4.4) can serve as the I/O backend for the encryption SD. This would allow clients to easily produce disk checkpoints that store encrypted data.

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

Udom0 initiates the storage encryption SD using a key passed as a kernel parameter, and an initialization script that starts the SD with a crypto loopback device. 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, the crypto loopback device in the SD uses 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 extra 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 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. Table 4 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%.

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

Table 5. Cost incurred by the storage integrity checking SD.

Integrity Checking SD. Our integrity checking SD offers a service similar to the one proposed by Payne *et al.* [35]. 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. Table 5 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.

4.2 Memory Introspection SD

Memory introspection tools, such as rootkit detectors (*e.g.,* [2, 29, 36, 45]), 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 [29]. 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, and mark them 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 [29]. 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 databse. We compared the time taken by this SD to a traditional setup where the Patagonix daemon executed within dom0. Table 6 presents the results of our experiment, again showing that using an SD imposes minimal overhead.

Platform	Time (seconds)
Xen (dom0)	6.471±0.067
SSC (SD)	6.487±0.064 (0%)

Table 6. Cost of the memory introspection SD, measured as the time to boot a Linux-based domain.

A memory introspection MTSD. Suppose that a cloud provider wants to ensure that a client is not misusing cloud resources to host and execute malicious software (or that an honest client's VM has not become infected with malware). In today's cloud infrastructure, this is achieved via VM introspection tools that execute in dom0. Such tools can inspect and modify client state, and therefore require dom0 to be trusted.

SSC offers cloud providers unprecedented power and flexibility in verifying client regulatory compliance while respecting client privacy. As an example, cloud providers can implement the Patagonix SD above as an MTSD to ensure that a client VM is free of malware. In this case, the cloud provider would supply the database of code hashes, which is the regulatory compliance policy. The MTSD itself would execute in the client meta-domain; the manifest of this MTSD simply requests privileges to read client memory pages and registers. Because MTSDs resemble SDs in their implementation, and only differ in the privileges assigned to them, the performance of this MTSD is identical to the corresponding SD, as reported in Table 6. The MTSD continuously monitors client UdomUs and reports a violation of regulatory compliance to the cloud provider (*i.e.,* Sdom0) only if the client becomes infected with malware. The cloud provider only learns whether the client has violated regulatory compliance, and cannot otherwise read or modify the content of the client's memory pages.

4.3 System Call Monitoring SD

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 [20] 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 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 [1] as the workload within the target UdomU to evaluate the overhead of this SD. The syscall mi-

Platform	System calls/second
Xen (dom0)	275K ±0.95
SSC (SD)	272K ±0.78 (1%)

Table 7. Cost incurred by the system call monitoring SD, measured using the UnixBench syscall microbenchmark.

Platform	VM size (MB)	No encryption (seconds)	With encryption (seconds)
Xen (dom0)	512	0.764±0.001	5.571±0.004
SSC (SD)	512	0.803±0.006 (5.1%)	5.559±0.005 (-0.2%)
Xen (dom0)	1024	1.840±0.005	11.419 ±0.008
SSC (SD)	1024	1.936±0.001 (5.2%)	11.329 ±0.073 (-0.8%)

Table 8. Cost incurred by the checkpointing SD.

crobenchmark 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. Table 7 presents the result of the experiment and shows that running system call monitor as an SD incurs negligible overhead.

4.4 Other SD-based 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.

Checkpointing SD. 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 [46]. 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). As previously discussed, 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 evaluated it by checkpointing VMs with two memory footprints: 512MB and 1024MB. We also conducted an experiment where we chained this SD with storage encryption SD; 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. Table 8 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.

Memory Deduplication SD. When multiple VMs have memory pages with identical content, one way to conserve physical memory using a mechanism where VMs share memory pages [48]. 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.

Platform	VM size (MB)	Time (seconds)
Xen (dom0)	512	6.948±0.187
SSC (SD)	512	6.941±0.045 (0%)
Xen (dom0)	1024	15.607±0.841
SSC (SD)	1024	15.788±0.659 (1.1%)

Table 9. Cost incurred by the memory deduplication SD.

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 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 9 presents the results, and shows that the performance of the SD is comparable to the traditional approach.

5. IMPLICATIONS OF THE SSC MODEL

The SSC model deviates in a number of ways from the techniques and assumptions used by contemporary cloud services. In this section, we discuss the implications of the SSC model. While the focus of this paper was on the core mechanisms needed to realize the SSC model, the issues discussed in this section are important for the practical deployment of an SSC platform.

5.1 Use of Trusted Computing

SSC relies critically on trusted computing technology in the protocols used to build client domains (Figure 3). We assume that clients interact with a vTPM instance, the supporting daemons for which are implemented in domB. The keys of this vTPM instance (in particular, the attestation identity key (AIK) and the endorsement key (EK)) are bound to the hardware TPM as discussed in prior work [5]. When used in the context of cloud computing, the use of the TPM and associated attestation protocols raises three issues: (1) do TPM/vTPM keys reveal details of the cloud provider's infrastructure? (2) how are keys distributed? and (3) do TPM/vTPM measurements reveal proprietary details of the software platform? We discuss these issues below.

(1) ***Can TPM/vTPM keys reveal physical details of the cloud infrastructure?*** SSC requires each physical machine in the cloud provider's infrastructure to be equipped with a hardware TPM, which serves as a hardware root of trust on that machine. Trusted computing protocols typically require all keys used during attestation to be bound to a specific hardware TPM. This includes the TPM's AIKs, and the AIKs and EKs of vTPM instances hosted on a physical machine. AIKs are distributed to clients, who may include the cloud provider's competitors. Researchers have therefore argued that binding keys to the TPM can expose details of the underlying hardware platform to competitors (*e.g.,* [40]). For example, a competitor may be able to infer the number of physical machines in the cloud infrastructure.

Fortunately, such risks can easily be alleviated. According to specifications released by the Trusted Computing Group [21], each hardware TPM can have arbitrarily many AIKs. However, the TPM's EK is unique, and is burned into the TPM chip by the hardware manufacturer. The public portion of the TPM's EK is distributed to trusted third parties, called *privacy certifying authorities (CAs)*. AIKs are bound to the TPM by signing them using the private portion of the TPM's EK. Likewise, vTPM keys are also bound to the hardware TPM, *e.g.,* by signing them using one of the hardware TPM's AIKs [5]. Given an AIK, the privacy CA can certify that the AIK is genuine, *i.e.,* it was indeed generated by a hardware TPM. Although the association between an AIK and the hardware TPM to which it is bound is known to the privacy CA, this association is never released outside the privacy CA. In SSC, the privacy CA can either be hosted by the cloud provider or a trusted third party.

The protocols in Figure 3 only require the client to be able to verify that an AIK is genuine, and therefore only require the client to interact with the privacy CA. The cloud provider can ensure that the client gets a fresh AIK for each execution of an attestation protocol. Because a single hardware TPM exposes multiple AIKs, it is impossible for an adversarial client to assert whether Udom0s running with different AIKs are executing on the same or different physical hosts, thereby protecting details of the cloud provider's physical infrastructure.

Alternatively, the cloud provider could host a centralized, trusted cloud verification service, as proposed in prior work [41, 42]. This verification service enables indirect verification of hosts by vouching for their integrity. Clients could interact with this verification service to obtain attestations, instead of directly interacting with the vTPM on the execution platform, thereby alleviating the risks discussed above.

(**2**) ***How are keys distributed to clients?*** Before initiating the protocols in Figure 3, clients must first obtain the public key of the vTPM instance assigned to them. While key distribution has historically been a difficult problem, requiring public-key infrastructure (PKI) support, the centralized nature of cloud computing services eases key distribution. The cloud provider, who is trusted in SSC's threat model, can establish trusted services required by PKI, such as a privacy CA and a central directory of AIK public keys. Prior to creating a new meta-domain, a client must leverage the PKI infrastructure to obtain the AIK public key of a vTPM instance assigned to it, and use the privacy CA determine whether the key is genuine.

(**3**) ***Can TPM/vTPM measurements reveal details of proprietary cloud software?*** TPM-based attestation protocols use measurements, typically hashes of software packages loaded for execution, to establish the trustworthiness of a platform. However, this approach may reveal specifics of the cloud provider's software infrastructure to competitors. For example, measurements may reveal the use of a module implementing a particular scheduling algorithm or a performance-enhancing library. The protocols used by SSC are based on measurements and are therefore prone to this risk.

One way to alleviate such risks is to use *property-based* TPM protocols [34, 37, 40, 42, 44]. The main feature of such protocols is that they attest specific *properties* of the software platform. That is, instead of attesting software using low-level measurements (*i.e.,* software hashes), which could reveal proprietary information to competitors, they attest whether the software satisfies certain properties implied by the client's security goals. For example, on SSC, such protocols could attest that the hypervisor implements the SSC privilege model, but not reveal any additional information to clients. Prior research has integrated property-based attestation protocols with the vTPM [37]. We will investigate the applicability of these protocols to SSC in future work.

5.2 VM Hosting and Migration

By its nature, SSC requires co-location of certain VMs on the same platform. A client's UdomU, any SDs and MTSDs associated with it, and the Udom0 of the client's meta-domain must be co-located on the same platform. Such constraints call for research on new algorithms for VM scheduling and migration. For example, if the cloud provider migrates one of the client's UdomUs to another host, it must also migrate SDs that service that UdomU. Some of these SDs may service other UdomUs that are not migrated; in such cases, the SDs (and the Udom0) must be replicated on both hosts. The stateless nature of Udom0 and several SDs (*e.g.,* the storage encryption SD) can potentially ease migration. For such stateless domains, the cloud provider can simply start a fresh instance of the domain on the target platform. A more thorough investigation of the cost and resource implications of these issues requires a deployment of SSC on several hosts. It also requires changes to administrative toolstacks (*e.g.,* VM migration tools, installed in Sdom0) to make them SSC-aware. We plan to investigate these topics in future research.

5.3 Client Technical Knowhow

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* 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. In fact, the dynamics of such an app store model can provide both a revenue generation opportunity to cloud providers (*e.g.,* clients can purchase SDs that they wish to use), as well as a reputation system for clients to judiciously choose SDs for their meta-domains. Of course, technically sophisticated clients can still implement their own SDs without choosing from the app store.

Finally, 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 nullifies this benefit. We feel that this is a fundamental tradeoff, and the price that clients must 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.

6. RELATED WORK

In this section, we compare SSC with prior work in two areas: security and privacy of client VMs in the cloud, and extending the functionality of the cloud.

Security and Privacy of Client VMs. Popular cloud services, such as Amazon's EC2 and Microsoft's Azure rely on hypervisor-based VMMs (Xen [3] and Hyper-V [32], 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 [22]. The relatively small code size of research hypervisors [31, 43, 47], combined with the

recent breakthrough in formally verifying the L4 microkernel [27], 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 [26].

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 [49] 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 [10, 11, 12, 13, 14, 23],

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.* [33] 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 [47]. The Xoar project [9] 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 [28], 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 [52] 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 4.2) 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 [40] 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. Excalibur includes a centralized monitor, as well as new protocols aimed specifically to address the TPM-related issues outlined in Section 5. 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. Future work could investigate a system that integrates concepts from SSC and Excalibur, in an attempt to combine the benefits of both systems.

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.,* [2, 7, 17]), arbitrary rollback [16], 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. At the same time, SSC's MTSDs accommodate the need for cloud providers to ensure regulatory compliance and have some control over client VMs.

The xCloud project [50, 51] also considers the problem of providing clients flexible control over their VMs. The original position paper [50] advocated several approaches to this problem, including by extending hypervisors, which may weaken hypervisor security. The full paper [51] 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.

7. CONCLUSIONS AND FUTURE WORK

SSC is a new cloud computing model that improves client security and privacy, and gives clients the flexibility to deploy privileged services on their own VMs. SSC introduces new abstractions and a supporting privilege model to achieve these goals. We integrated SSC with a commodity hypervisor (Xen), and presented case studies showing SSC's benefits.

In the future, we plan to enhance SSC by factoring devices drivers [28] and XenStore into their own domains [9]. We also plan to explore other novel services enabled by SDs. While our evaluation in Section 4 has primarily focused on SD-based security and systems services, we also plan to build network-based services using SDs. Individual cloud clients can leverage SDs to implement middleboxes, such as NIDS systems, firewalls and traffic

shapers, and to run performance-intensive network monitoring services. Such network-based services are currently under the control of cloud providers, and clients often have no say in configuring them. SDs therefore allow clients to enforce arbitrary network security and auditing policies without having to rely on cloud providers to deploy the corresponding services. Finally, we plan to address several of the issues discussed in Section 5 in an effort to make SSC a practical alternative to current cloud infrastructures.

Acknowledgments. We thank the anonymous reviewers and our shepherd, Andrew D. Gordon, for their comments on the paper. This work was funded in part by NSF grants CNS-0831268, CNS-0915394, and CNS-0952128. Parts of this work were completed when the first two authors were at AT&T Research.

REFERENCES

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

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

[3] 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.

[4] 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.

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

[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] 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.

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

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

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

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

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

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

[16] 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.

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

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

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

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

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

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

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

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

[25] 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.

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

[27] 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.

[28] 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.

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

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

[31] 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.

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

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

[34] A. Nagarajan, V. Varadarajan, M. Hitchens, and E. Gallery. Property-based attestion and trusted computing: Analysis and challenges. In *Intl. Conf. on Network and System Security*, 2009.

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

[36] 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.

[37] A-R. Sadeghi, C. Stuble, and M. Winandy. Property-based TPM virtualization. In *Information Security Conference*, 2008.

[38] 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.

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

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

[41] J. Schiffman, T. Moyer, H. Vijayakumar, T. Jaeger, and P. McDaniel. Seeding clouds with trust anchors. In *ACM Cloud Computing Security Workshop*, 2010.

[42] J. Schiffman, H. Vijayakumar, and T. Jaeger. Verifying system integrity by proxy. In *TRUST*, 2012.

[43] 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.

[44] E. Gun Sirer, W. de Bruijn, P. Reynolds, A. Shieh, K. Walsh, D. Walsh, and F. B Schneider. Logical Attestation: An authorization architecture for trustworthy computing. In *ACM SOSP*, 2011.

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

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

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

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

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

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

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

[52] 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.