Exploring infrastructure support for app-based services on cloud platforms

Hai Nguyen, Vinod Ganapathy, Abhinav Srivastava, Shivaramakrishnan Vaidyanathan

CS Department, Rutgers University, 110 Frelinghuysen Road, Piscataway, NJ 08854, USA
AT&T Labs-Research, 1 AT&T Way, Bedminster, NJ 07921, USA

ABSTRACT

Major infrastructure-as-a-cloud (IaaS) providers have recently been building marketplaces of “cloud apps,” which are VMs pre-installed with a variety of software stacks. Clients of cloud computing leverage such markets by downloading and instantiating the apps that best suit their computing needs, thereby saving the effort needed to configure and build VMs from scratch.

We posit that the notion of cloud apps as defined by these marketplaces is nascent and does not allow apps to leverage the benefits of virtual machine (VM) introspection technology developed over the past decade. We envision a marketplace of apps that can interact with client VMs in a rich set of ways to provide a number of services that are currently supported only by cloud providers. This allows clients to deploy services such as VM introspection-based security tools and network middleboxes on their work VMs without requiring the cloud provider to deploy these services on their behalf.

This paper presents models to support such a marketplace of expressive cloud apps. We present a study of the design space of these models to understand their performance and deployment tradeoffs. We also consider the design of a permissions-based framework to contain untrusted third-party cloud apps. Finally, we demonstrate the utility of our models by building and evaluating a number of security tools built as cloud apps.

1. Introduction

Modern cloud computing platforms such as Amazon EC2 and Windows Azure have become popular over the last several years. These platforms host client computations on a shared computing infrastructure while ensuring that mutually-distrusting clients cannot affect the confidentiality or integrity of each other’s computations. The key enabling technology is virtualization. By running client computations within virtual machines (VMs) and controlling access to the hypervisor layer, cloud providers ensure that clients are isolated from each other.

In parallel with these developments, the security and computer systems research communities more broadly have been investigating the use of virtualization to offer novel services to VMs. For example, prior research has developed a variety of virtual machine introspection (VMI) (Chen and Noble, 2001)...
based security tools such as sophisticated malware detectors using VMI (Garfinkel and Rosenblum, 2003; Payne et al., 2007; Srivastava and Giffin, 2008). For example, using VMI one can build as rootkit detectors for operating systems executing within the VM; rootkit detectors cannot be built using traditional in-VM methods. Similarly, network-function virtualization allows the creation and deployment of networked services such as network intrusion detection systems (NIDS) and firewalls to be deployed on VMs.

In this paper, we explore the possibility of offering such enhanced services as apps on a cloud platform. That is, a client should be able to download an “app” that implements such services (such as a NIDS or a rootkit detector), and use the app on his VMs executing on the cloud. While the motivating use-cases in this paper are all security apps, e.g., VMI-based services that enhance the security of client VMs, the concepts that we develop apply more broadly to any cloud-based service.

Contemporary cloud platforms offer a nascent notion of cloud apps and cloud app marketplaces (NetEx; Managing VMware vApp – VMware vSphere 4 ESX and vCenter Server; The Cloud Market: Complete Catalog of Amazon EC2 Images; Cisco – SourceFire). These “apps” are primarily VMs installed with pre-configured software stacks for a variety of standard workflows, such as Web servers and database servers. They primarily benefit clients who lack the expertise or manpower to perform detailed configurations. For example, Amazon allows publishers to create and publicly offer VM images, known as Amazon Machine Images (AMIs) (AWS MarketPlace – Find and Buy Server Software and Services that Run on the AWS Cloud), that execute on EC2. AMIs that offer a variety of standard software stacks (e.g., LAMP or SQL database software) are now available, and customers can directly simply instantiate them to their specific domains.

Unfortunately, the notion of apps supported by contemporary cloud providers does not support our vision of services as apps. We envision a cloud app market where apps (implemented as VMs) offer standard utilities such as firewalls, NIDS, storage encryption, and VMI-based security tools. Cloud apps can also implement a number of other non-security-related utilities, such as memory and disk deduplication, and network middleboxes such as packet shapers and QoS tools. Clients can leverage these utilities by simply downloading the appropriate cloud apps, and linking them suitably with their work VMs.

The key challenge in realizing this vision on current cloud computing environments is that such interaction between VMs is disallowed because of security considerations. On a cloud platform such as Amazon EC2 or Microsoft Azure, client VMs – even those executing on top of the same hypervisor on a single physical machine – are completely isolated from each other. Thus, one client VM cannot intercept the I/O or memory state of another VM, even if that VM belongs to the same client. The only exception to this isolation is the privileged management VM, a per-node VM controlled by the cloud provider that supervises the execution of client VMs. The management VM oversees all I/O from client VMs and can inspect and modify the memory and CPU state of all VMs executing on that platform. While it is possible to implement VMI-based services or I/O interception within the management VM, it requires the cloud provider to deploy the services on behalf of the client. As a result, the current model fails to realize the full power of an app-based approach – e.g., wherein clients can download and deploy VMI-based services on other VMs that they own.

The primary contribution of this paper is in exploring how we can change the security model in today’s cloud computing infrastructures to support our vision of cloud apps. As we discuss in this paper, a variety of design options are possible, and our goal is to perform a comprehensive exploration of the design space rather than committing to any one model. We therefore make the following contributions:

- In Section 3, we present a number of motivating examples of cloud apps to illustrate our vision. These apps range from standalone apps to ones that involve complex system- and network-level interactions with other VMs. We present these examples to identify the key challenges in enabling such an app model atop a cloud platform.
- In Section 4, we present a detailed exploration of the design space for supporting cloud apps. We consider various models of virtualization that are supported by contemporary cloud platforms, and offer designs for each of these models to support cloud apps. We also develop the notion of cloud app permissions, which clients can use to reason about and control the behavior of apps developed by third parties (just as in mobile app markets).
- We have implemented each of our designs atop the KVM hypervisor (Section 5). In Section 6, we demonstrate the utility of our model by building and evaluating a number of security-related cloud apps and evaluate their performance for various points in the design space.

### 2. Threat model

We assume a standard cloud computing model, where a cloud provider, i.e., an entity such as Amazon or Microsoft, provides computing infrastructure. Clients rent resources from the cloud provider, and run their virtual machines on the cloud provider’s hardware.

To support cloud apps, we assume that the apps are hosted on a marketplace, which is supported either by the cloud provider or a third-party. Clients download these apps and apply them to their VMs to obtain access to various services. The main security problems that we must address are:

1. How do we ensure that a cloud app downloaded by the client can access that client’s VMs alone?
2. How do we confine the cloud apps so that they can access the CPU, memory and I/O state of a client’s VMs but only to the extent that they need to achieve their advertised goals.

To achieve these goals, we assume that the cloud provider is trusted, and does not intentionally violate the security and privacy of clients. Thus the computing platform, consisting of the physical hardware and the hypervisor, is trusted. To a certain extent, clients can verify their trust in the cloud provider using trusted hardware, e.g., using attestation based on the TPM. We also assume that the cloud provider’s infrastructure is equipped with IOMMU units to enable I/O virtualization. The design points
discussed in Section 4 enhance the cloud provider’s infrastructure to achieve our first security goal stated above.

The implication of placing trust in the cloud provider is that we cannot protect client VMs from attacks that involve government subpoenas or even malicious cloud administrators. This setting may therefore be unsuitable for cloud clients with stringent security and privacy requirements, e.g., financial organizations and health providers. However, we believe that trusting the cloud is a reasonable assumption for a large number of cloud clients. Our goal is to explore a number of useful applications that are enabled by placing such trust in the cloud provider. Upcoming hardware advances such as the Intel SGX can potentially allow us to relax the assumption of having to trust the cloud provider (Baumann et al., 2014; Intel).

Clients download and use cloud apps on their VMs. Cloud apps can be written by third-parties and are not trusted. They must be suitably confined by ensuring that they are only given access to the resources that they need to perform their advertised tasks. To achieve this security goal, we present a permission-based approach in which each cloud app explicitly states the permissions that it requires to perform its operations. Our approach also gives clients control over the I/O of cloud apps. We assume that clients examine permissions prior to installing apps and will devise suitable security policies to govern their I/O. Once a client accepts the permissions, the cloud provider enforces these permissions on the cloud apps, and ensures that the app cannot get access to any resource that was not explicitly approved by the client during installation.

In many ways, this situation resembles the one on mobile app markets, where clients download and execute mobile apps written by third-parties. On such markets, it is well understood that malicious apps can make their way into app markets, and that permissions are inadequate to prevent malicious apps from leaking sensitive information that belongs to clients. It is quite likely that the same issues (and much the same kind of research) apply even to the setting of cloud apps. We do not attempt to address those issues in this paper, but rather focus on the infrastructure support necessary to enable cloud apps.

Finally, we do not also explicitly aim to protect against side-channel attacks, in which one client attempts to infer sensitive information about other clients or the cloud provider itself (Pitropakis et al., 2015; Zhang et al., 2011, 2012). We believe that such threats are orthogonal to our work, and that defenses developed for such threats can be incorporated into a cloud platform that supports the app model that we develop.

3. Examples of cloud apps

In this section, we present a taxonomy of cloud apps based on their functionality. We broadly categorize apps into four groups: (1) standalone apps, (2) low-level system apps, (3) I/O interceptors, and (4) bundled apps.

3.1. Standalone apps

Standalone apps (Fig. 1) represent the kind of apps that are currently available on cloud markets (e.g., AMI images available currently on Amazon’s cloud market). These apps could include VMs with new operating system distributions, versions, or drivers supporting new hardware. Clients select cloud apps, instantiate them, and build software and services atop the provided environment. These cloud apps are self-contained and do not interact with the client’s VM.

To create a standalone cloud app, a publisher configures and installs an OS along with one or more user-level tools customized for a specific workflow. For example, a Web server cloud app will likely contain the entire Apache toolchain. Likewise, it is also possible to imagine a security cloud app that is pre-installed with user-space anti-virus tools. Clients that purchase such a VM app will benefit from the pre-installed anti-virus tools, thus saving them the effort of installing and configuring these tools themselves. They can simply install their enterprise software, which will automatically be protected by the anti-virus tools. Note that in the setting of standalone cloud apps, a web-server cloud app does not directly benefit from a security cloud app because these VMs cannot interact with each other.

3.2. Low-level system apps

As noted above, standalone apps do not interact with each other. Our vision of cloud app markets includes apps that can interact with client VMs to provide system-level services. Such low-level system apps contain specialized software to be used on other client VMs (or other cloud apps), and actively interact with the CPU, memory and operating system of client VMs. For example, a checkpointing app will read the CPU and memory state of a client VM to generate a snapshot of that VM.

Low-level system apps can empower clients by providing them with powerful security tools. On contemporary cloud platforms, clients that wish to protect their work VMs from malicious software largely resort to in-VM tools, such as anti-virus software. While such an approach can potentially detect malicious software that operates at the user level, attackers are increasingly exploiting kernel-level vulnerabilities to install rootkits, which in turn allow them to remain stealthy and retain long-term control over infected machines.

To counter kernel-level malware, researchers have proposed a number of solutions (e.g., Garfinkel and Rosenblum, 2003; Litty et al., 2008; Payne et al., 2007; Srivastava and Giffin, 2008) inspired by virtual machine introspection (VMI). In such solutions, the virtual machine monitor acts as the trusted computing base, and helps maintain the integrity of VMs that it executes. The security policy enforcer (e.g., a rootkit detector) itself operates within a privileged VM (e.g., the management...
VM), and checks the state of a target VM. To enable security enforcement, the virtual machine monitor fetches entities from the target VM (e.g., memory pages containing code or data) for inspection by the security policy enforcer. This architecture protects the enforcement engine from being tampered by malicious target VMs.

While there has been extensive research over the past decade on these and other related services enabled by virtualization technology, deploying these solutions on contemporary cloud infrastructures requires cooperation from the cloud providers. For example, VMI-based solutions must perform privileged operations on client VMs, and must therefore be deployed within the management VM. Because the management VM on public cloud infrastructures is only accessible to cloud providers and administrators, clients of the cloud infrastructure cannot avail of these services unless the provider installs and support the service within the management VM. Even if the provider were to support a service within the management VM, clients will likely not have the ability to customize the service to their needs.

In our model, VMI-based intrusion detection tools can be implemented as low-level system apps. To use such an app, a client must be able to download and instantiate the app, and permit the app to inspect the memory of its work VMs to detect attacks. In Section 4, we describe how our cloud app model permits clients to assign specific privileges to its cloud apps, such as mapping memory pages of its work VMs into the cloud app’s address space. Other examples discussed in prior work that can be implemented as low-level system apps include system call interposition engines (e.g., Butt et al., 2012; Pitropakis et al., 2015) and memory deduplication tools. Depending on their functionality, low-level system apps can either require two-way read/write access or one-way read access to the client’s work VM state. For example, a rootkit detection app may send a request to the client VM to access to a particular page, and the client VM would respond by giving it suitable access. In response, the rootkit detector may modify the contents of the page (e.g., to remove the rootkit). On the other hand, a checkpointing app or a forensic analysis app would only require read-only access to client VM memory.

3.3. I/O interceptors

An I/O interceptor is a cloud app that sits on the I/O path (either network or storage or both) of the client’s work VMs (Fig. 3). To use an I/O interception cloud app, clients set up the I/O path so as to direct the output of their work VM.

Such interceptors are already in popular use on many enterprise networks, as middleboxes, although not as cloud apps. For example, enterprises typically deploy a number of network security tools, such as firewalls, intrusion detection systems, and intrusion prevention systems, at the perimeter of networks to defend their in-house networks from attacks such as intrusions, denial-of-service (DoS), spam, and phishing, etc.

While such middleboxes are relatively easy to deploy on networks controlled by the enterprise, this task becomes significantly more complex if the enterprise shifts its operations to public clouds. On contemporary cloud platforms, the enterprise must rely on the cloud provider to implement similar services on its work VMs. This is because I/O interceptors need privileged access to client VMs to intercept their communication with the outside world. On a virtualized system, such communication from VMs is mediated by the privileged management VM, and security considerations demand that one VM’s I/O should not be intercepted by another unprivileged VM. However, I/O interceptors need precisely this ability, and cannot therefore be deployed on contemporary cloud platforms.

In our ecosystem, we endow I/O interceptors with the ability to plumb I/O from the client’s VMs. Thus, I/O interceptor cloud apps provide enterprises the freedom to implement these services without having to rely on cloud providers. The enterprise simply downloads the network security cloud app, instantiates it, and configures it to receive both incoming and outgoing network flows to its work VMs. This would allow the enterprise to define and enforce arbitrary security policies flexibly, without relying on the cloud provider. A key requirement to support such apps is that clients must have the flexibility to create custom I/O channels.

3.4. Bundled VM apps

It is often beneficial to combine the functionality of multiple cloud apps, and obtain a bundled cloud app that composes the functionality of its constituent cloud apps (Fig. 4). Akin to the pipe in primitive operating systems, where the output of one...
command is input to another, in a bundled cloud app, the output of one cloud app is input to another. Individual cloud apps inside a bundle could be a standalone, I/O interceptor or low-level systems apps, or themselves bundled apps.

Bundled cloud apps (Fig. 4) are an ideal strategy for implementing and composing services. For example, a client interested in network security apps, such as intrusion detection systems and firewalls, downloads a network security app bundle. Upon instantiation of the bundle, all the apps inside it will create a chain of services. In Fig. 4, for example, the client VM’s packet will traverse the firewall, and then the network IDS, providing multiple layers of security. To realize the bundled cloud app model, clients must fulfill the requirements of individual apps in the bundle. For example, a bundle consisting of low-level system and I/O interceptor apps will require both necessary privileges and custom communication and I/O channels to perform its task.

In summary, cloud apps can benefit clients of a cloud platform by enabling a wide variety of virtualized services. The cloud app ecosystem enables unprivileged cloud clients to deploy several previously-studied virtualized services. The main contribution of this paper is exploring the design space of infrastructure support to enable such cloud apps. We explore a variety of design options and study the performance characteristics of each of them.

4. Design considerations

In this section, we describe the components of a cloud app market ecosystem, and present a number of ways to design these components. The next section describes our prototype implementation, in which we have incorporated some of these designs.

4.1. Trustworthy launch of cloud apps

The workflow of an app begins with the client downloading it from a cloud app market, configuring and starting the app. To ensure trustworthy operation, the client must have the ability to verify that the app booted correctly, with the same software stack as promised when the app was downloaded. This requirement is particularly important because clients will typically download cloud apps from the cloud market based upon the advertised functionality of the app. Because the cloud app will likely interact with the client’s work VMs, it is important for the client to validate that the operating system and applications running atop the cloud app are as advertised by its creator. Such validation will establish trustworthy launch of cloud apps.

We expect that validation of trustworthy cloud app launch can be performed using attestation protocols developed by the trusted computing community (Sailer et al., 2004). This requires the cloud provider to offer hardware equipped with a trusted platform module (TPM) chip, and possibly virtualize this hardware (Berger et al., 2006). Attestation protocols “measure” software that is started on the system (usually by simply hashing the software), and store these measurements securely within registers of the TPM chip. A client can verify trustworthy launch by verifying the measurement received from the platform using the expected measurements of the cloud app.

4.2. Privileges for cloud apps

A number of useful VM-based services that could be deployed as cloud apps cannot be implemented on contemporary cloud platforms because they require additional privileges. For example, a VMI-based malware detector (a low-level system app) requires the ability to inspect the state of another VM, and therefore map its memory contents and CPU state. On contemporary cloud platforms, such applications can only be deployed with the cooperation of the cloud provider.

In our approach, clients must have the ability to allow apps that they download from app markets to inspect/modify the state of their own VMs. Accomplishing this requires a careful examination of the privileges that cloud apps would need. In particular, privileges granted to cloud apps must satisfy two criteria:

1. Privileges must be fine-grained. Clients must be able to specify that a cloud app be given specific privileges (over its memory pages, or I/O channels) over client VMs.
2. Cloud apps must not leak information. Cloud apps are expected to interact with client VMs, which may store and manipulate sensitive data. Low-level system apps and I/O interceptors can potentially access this information. Apps must not leak sensitive data to their publishers or other malicious third parties.

Cloud app privileges that satisfy these criteria would allow clients to disallow apps from accessing the network, memory or other locations that store sensitive data, thereby preventing apps from sending any sensitive data that it may read from the client’s VMs. It may be possible to implement such defenses using a number of ad hoc hypervisor-specific techniques. For example, on Xen, such control can be implemented using Xen grant tables. A VM that wishes to communicate with the outside world shares pages with the management VM to pass data to hardware. Xen grant tables can be used to specify that a particular VM should never be able to establish shared pages with the management VM, hence restricting I/O.

Instead of relying on such ad hoc methods, we instead introduce a set of cloud app privileges that must be implemented on any platform that supports cloud apps, leaving the specifics of the implementation to what works best on each platform.
As already discussed, our model of a cloud app allows low-level system apps access to the CPU and memory state of a client’s VMs. We additionally require apps that wish to obtain such access must specify their requirements up front as a set of permissions, akin to the concept of permissions currently used on mobile apps in certain platforms (e.g., Android and Windows Phone). These permissions are specified in the app’s manifest. Fig. 5 describes the attributes (and the set of permissible values for these attributes) that are currently supported in our prototype implementation. We assume that as in mobile app markets, specifying permissions in a manifest allows clients to determine whether he wishes to use the app on his VMs.

Once approved by the client, the manifest is used to enforce the set of permissions requested by the app – Section 4.3 describes the design space of hypervisor-level modifications necessary to enforce such permissions. Our approach also allows clients to dynamically revoke permissions that have been granted to cloud apps. In addition to allowing apps access to CPU and memory state, our app model also allows the client to specify how I/O interceptors must reside on the inbound or outbound path of a client VM. We describe the policies used to do so in Section 4.4.

By design, cloud app privileges combined with the policies used to govern I/O interception give a client fine-grained control over app behavior. We also believe that cloud app privileges allow clients to determine how apps access sensitive information from their work VMs. Although careful use of permissions can regulate information flow from work VMs to apps to a certain extent, permissions on their own are not powerful enough to provide detailed, low-level information-flow controls.

The permissions themselves cannot directly provide low-level information-flow tracking (e.g., as available via TaintDroid (Enck et al., 2014) for Android). As a result, it may be difficult to completely eliminate the risk of information leakage. For instance, Bugiel et al. (2011) show that some cloud VMs store sensitive client data and do not erase it even when clients release the VM. To eliminate such threats, cloud infrastructure may have to be enhanced in the future to offer image-cleaning services and control mechanisms, as proposed by Mirage (Wei et al., 2009).

Examples. We now illustrate cloud app privileges by presenting examples of manifest file attributes for the example cloud apps from Section 3:

(1) A standalone app, such as a pre-configured LAMP stack as would be available on Amazon’s cloud market, does not require any special privileges to execute. The manifest for such an app can be empty, except for the version and description of the app. If the “memory,” “network,” and “location” fields of the manifest are left empty, it means that the app should not be given any of these privileges, and therefore runs as an unprivileged app.

(2) A rootkit-detector (low-level system app) must be able to obtain access to the kernel memory, and optionally memory of user-level apps, of the VM that it monitors. For such an app, the “memory” field of the manifest would be assigned to “kernel.” Moreover, because memory analysis only makes sense if the VM containing the app is physically co-located with the VM that it monitors, the “location” field of the manifest is set to “must colocate.” This implies that if the app is used to monitor multiple client VMs, each located on a different physical node in the cloud, a copy of the app has to execute on every node on which the client’s VMs execute.

(3) A network intrusion detection app (I/O interceptor) must be able to intercept all network traffic to and from the VMs that it monitors. Therefore, the “network” field of the manifest of this app is set, indicating that the app needs to access I/O traffic. If set to “on path,” the app will sit on the critical path of all traffic to and from client VMs that it is required to monitor. If set to “off path,” the app gets a copy of all the traffic bound to or from the client VMs.

The underlying cloud infrastructure is responsible for authentication and identifying the client to which a particular app instance belongs. The cloud infrastructure then uses the client’s identity to restrict the app’s access only to other VMs that belong to the same client. The permissions in the manifest are set by the vendor that manufactures the app. This situation is akin to the one in mobile app markets for Android and Windows Phone, where the app vendor declares the permissions needed for the app to operate. While the idea of using permissions is not perfect – in particular, it is well-known that apps often request many more permissions than they actually use (e.g. Felt et al., 2011; Vidas et al., 2011)—it at least provides a declarative approach to security. Orthogonal analyses can determine whether the app’s permission request in its manifest indeed matches its expected functions.

In our proposed approach, the deployment of cloud apps and manifests largely resembles the approach used in mobile app markets today. That is, a third-party app vendor supplies both the app and the manifest to a cloud app market, which publishes the app and the manifest. The app market may optionally conduct analyses to determine whether the manifest corresponds to the functionality of the app, e.g., as is done by Google’s Bouncer for apps submitted to the Play store. Thus, we assume that if a client downloads a cloud app from a trusted source, the app vendor cannot maliciously manipulate the manifest to request more permissions than needed by the app.

4.3. Hypervisor-level support

Low-level system apps must have the ability to perform privileged system tasks on client VMs. Stock virtual machine monitors do not grant user domains such privileges. We

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory</td>
<td>kernel</td>
</tr>
<tr>
<td>network location</td>
<td>on path</td>
</tr>
<tr>
<td>description</td>
<td>must colocate</td>
</tr>
<tr>
<td>version</td>
<td>Functionality of the cloud app version of the cloud app</td>
</tr>
</tbody>
</table>

Fig. 5 – Attributes of a manifest file. (a) Hypervisor modification; (b) Nested virtualization; (c) Paravirtualization.
consider three design options in the following subsections (illustrated in Fig. 6), which require various degrees of modifications to the hypervisor. We evaluate each design with respect to two metrics:

1. **Performance.** What is the runtime performance overhead imposed by the design? Note that cloud apps may intrinsically reduce the performance of client VMs, e.g., the use of an I/O interceptor app will necessarily reduce the throughput of a network-bound client VM because of the additional network element introduced in the I/O path. In evaluating performance, we ignore this intrinsic overhead. Rather, we focus on the overhead introduced by the implementation technique used to build cloud apps.

2. **Deployability.** Does the design require invasive changes to hypervisors, as deployed by cloud providers today?

### 4.3.1. Option 1: hypervisor modifications

Perhaps the most straightforward approach to realizing low-level system apps is to modify the hypervisor (Fig. 6a). These modifications will primarily address the way in which the hypervisor assigns privileges to client VMs. On commodity hypervisors, the management domain runs at the highest privilege level, and orchestrates all communication between VMs running as user domains. In this model, communication between VMs belonging to a single client goes through the management VM.

To support cloud apps, the hypervisor can be modified to allow apps belonging to a particular client to have specific privileges to over other VMs belonging to the same client. This would allow a low-level system app to map the memory pages belonging to the client’s VMs. We expose these changes to the client via a set of hypercalls that allow an app to perform privileged operations on a client VM. These hypercalls allow a cloud app to read the register file, or map the user-space or kernel-space memory pages of the client’s VM into its own address space. We describe our implementation of this design option, and the hypercalls that we added in Section 5.1.

We expect the runtime performance impact of this approach to be minimal. The performance overhead will likely stem from the impact of executing additional checks within the hypervisor to implement and enforce cloud app privileges. Cloud apps and client VMs continue to execute directly atop the modified hypervisor, and their performance will be comparable to VMs executing atop an unmodified hypervisor. However, this design option requires invasive changes to the hypervisor, which a number of cloud providers may be reluctant to deploy.

### 4.3.2. Option 2: nested virtualization

Hardware support for virtualization (e.g., Intel VT), coupled with software-level implementation tricks, have made low-overhead nested virtualization a reality (Ben-Yahuda et al., 2010). Nested virtualization allows clients to execute a full-fledged nested hypervisor, also called an L1 hypervisor, atop a base hypervisor, called the L0 hypervisor, which supports nesting.

In this design option, a cloud app will be distributed as a L1 hypervisor together with a management VM that implements the advertised functionality of the app (Fig. 6b). Clients can execute their work VMs atop the L1 hypervisor, which in turn has its own management VM. Thus, the app can implement its privileged services within the management VM of the L1 hypervisor. Only minimal changes are needed to the cloud provider’s L0 hypervisor (described below).

With nesting, the L0 hypervisor executes at the highest processor privilege level and is the only software layer authorized to perform privileged operations, while the L1 hypervisor and client VMs execute with lower privileges. On the Intel ×86 platform, every trap goes to the L0 hypervisor, which then either handles the trap or forwards it to the right layer. Thus, for instance, any operations by the app to map memory pages of the client’s VM will trap to L0. The L0 hypervisor is modified to enforce the permissions in the cloud app’s manifest; thus, it reads app manifest files, and enforces the permissions when it receives traps to perform sensitive operations.

We expect the runtime performance impact of this approach to be higher than in the case of direct hypervisor modifications. However, this option has the attractive advantage of only requiring minimal changes to the cloud provider’s hypervisor. Our prototype also implements this design option.

### 4.3.3. Option 3: paravirtualization

The third design option is a variant of the nested virtualized-based design described above. If the cloud provider’s platform is based upon a hypervisor that does not have hardware support for efficient nested virtualization, it may still be possible to achieve many of the same benefits using paravirtualization. In particular, the XenBlanket project (Williams et al., 2012) demonstrates that it is possible to achieve nested virtualization atop...
a stock Xen hypervisor (e.g., as deployed on Amazon EC2) that does not support virtualization. XenBlanket achieves this by building a thin software layer (a “blanket”) that emulates privileged operations for a nested VMM executing atop the base Xen hypervisor (Fig. 6c). Cloud apps based upon this design option will resemble those developed for the previous design option. The principal difference is that the software stack of these cloud apps and the client VMs must use a paravirtualized operating system.

We have not implemented this design option, but evaluate its merits using the XenBlanket implementation. XenBlanket was primarily developed to allow nested virtualization atop commodity cloud platforms. Although the reported overhead of the blanket layer is acceptable, the overheads of a cloud app implemented using this approach will likely be higher than if support for nested virtualization was available. Among our design options, we therefore expect this approach to have the highest runtime overheads. However, the primary advantage of this approach is that cloud apps can be deployed today over stock services such as Amazon EC2.

4.3.4. Hybrid designs
It may be possible to create a hybrid design for cloud apps that combines the designs discussed above. For example, suppose that a cloud provider decides to incorporate modifications to allow clients to plumb their I/O paths, in effect allowing clients to place middlebox cloud apps between their work VMs and the platform’s management VM. However, the cloud provider may be reluctant to include VMM modifications that allow cloud apps to map memory pages of client VMs. In such cases, it is still possible to realize cloud apps such as rootkit detectors using nested virtualization using either the nested or paravirtualized designs.

4.4. Plumbing I/O
I/O interceptors need to have the ability to interpose on the traffic to and from the client VMs that they service. We rely on emerging technologies based on software-defined networking (SDN) to perform such interposition, and allow clients to plumb the I/O paths to and from their VMs.

4.4.1. Plumbing network I/O
Like recent work on using SDN to implement network middleboxes (Joseph et al., 2008; Qazi et al., 2013; Sekar et al., 2012), we represent properties of network flows using policy classes. A policy class is a quintuple:<source IP address, destination IP address, source port number, destination port number, protocol type>. Clients use policy classes to specify I/O policies, which are rules of the form $C \rightarrow \{M_1, M_2, \ldots, M_n\}$, where $C$ is a policy class and $M_1, M_2, \ldots, M_n$ is a sequence of cloud apps. The semantics of this rule is that traffic that matches the policy class should traverse the sequence of cloud apps. Because each $M_i$ is itself a VM, they can reside on separate physical machines. Regardless, traffic to and from client VMs must follow the I/O policies for the policy class that they match.

To enforce traffic flow compliant to I/O policies, we rely on the cloud provider’s SDN controller. The cloud infrastructure’s SDN controller tracks the physical placement of each of the middleboxes $M_i$. The SDN controller (which is based on the OpenFlow standard), uses I/O policies specified by the client to configure the software switches on each physical platform that executes a cloud app or a client VM that is affected by the I/O policy. Upon receiving a network packet, each switch in the middlebox app sequence identifies the apps that the packet has traversed. This helps the switch to determine the next hop in its forwarding table.

To make this discussion concrete, consider the network topology shown in Fig. 7. In this example, a client specifies that each network request to the web server VM must go through the firewall VM. The policy class $C$ can be specified as $\langle\text{srcIP} = \text{external prefix}, \text{dstIP} = \text{IP of the web server}, \text{srcPort} = \text{any}, \text{dstPort} = 80, \text{protocol} = \text{TCP} \rangle$. When SW3 receives a packet, it needs to know if the packet has traversed the firewall to decide the next hop to forward the packet. If the packet has not passed through the firewall, SW3 forwards the packet to SW2 (hosting the firewall). Otherwise, SW3 forwards the packet to SW1 (hosting the web server).

In the network topology shown in Fig. 7, this is relatively straightforward. SW3 knows that all packets received on “inbound” have not yet been scanned by the firewall, and so it forwards them to SW2. Similarly, it knows that packets arriving on SW3 from SW2 have already traversed the firewall, and so can be directly forwarded to SW1. The situation is somewhat more complex in Fig. 8 (adapted from Qazi et al. (2013)), where the I/O policy is Firewall→NIDS→Webserver. In this case, the NIDS is connected to SW3 and the Firewall is connected to SW2. When a packet arrives from the Internet, it arrives at SW3, which then forwards it to SW2 for scanning by the Firewall app. Because the NIDS is connected to SW3, SW2 must forward any Firewall-scanned packets back to SW3. Once the packet is scanned by the NIDS, it can be forwarded to SW1 for consumption by the Web server. However, when SW2 receives a packet, it needs to know whether the packet has already traversed the NIDS or not.

To address this challenge, we use packet tagging. In particular, after a packet traverses a middlebox, the switch hosting that middlebox tags the packet using VLAN tags. Each middlebox app is assigned an unique tag ID. In Fig. 8, SW2 uses the tag assigned to the NIDS to know if a packet it receives from SW3 has traversed the NIDS. More generally, we treat each such I/O policy as a state machine. As each network packet traverses middleboxes on the cloud platform, it makes a transition on this state machine. When the state machine reaches an accepting configuration (i.e., all the conditions of the I/O policy are satisfied), the packet can be consumed by the client VM. We keep track of the state that a network packet is in using the packet tags assigned to it. Because each middlebox causes a state transition, we let the middleboxes modify the VLAN tags assigned to the packet in accordance with the I/O policy. Thus, we are able to track network packets and ensure that all packets that are consumed (or emitted) by the client VMs satisfy the I/O policy.

4.4.2. Plumbing disk I/O
In our app model, I/O interceptors that work on storage devices work by intercepting storage streams of the client’s VM to perform various storage related tasks such as storage deduplication and file integrity checking.
One approach to support storage apps is to modify the hypervisor. Commodity hypervisors handle I/O operations from guest VMs by forwarding them to an I/O emulator. The emulator can either be the management VM as in the Xen hypervisor or a user space process as in the KVM hypervisor. The hypervisor can be modified to forward I/O streams of the client’s VM to storage apps before sending them to the I/O emulator. However, this solution only works if storage apps run on the same physical machine as the client’s VM.

We therefore process storage I/O streams as network traffic. In particular, we propose to use network file system (NFS) services to implement storage apps. We run an NFS client service inside the client’s VM and NFS server services inside storage apps. All read/write operations from the client’s VM take place by invoking the NFS services. The NFS client service communicates with the NFS server service to fulfill the operations. With this design, the storage I/O can be intercepted and manipulated in the same manner as network I/O.
4.5. Composing app functionality

In a number of cases, it may be necessary to compose, or “chain,” the effects of multiple cloud apps. Our design allows for such chaining by requiring the developer to specify a composition policy (also called a chaining rule), of the form $C \rightarrow [M_1, M_2, \ldots, M_n]$, where $C$ is the identity of a client’s VM and $M_1, M_2, \ldots, M_n$ is a sequence of cloud apps that either have system-level privileges over $C$ or are I/O interceptors. The chaining policy indicates that the memory pages of the client’s VM that are required by $M_n$ should be processed by $M_1, M_2, \ldots, M_{n-1}$, in order, before being mapped by $M_n$. At any point during this chain, an app can specify that its results must not be passed to the next app, thereby breaking the chain.

We explain this in further detail with an example of a memory checkpointing app and a rootkit detection app as an example. The memory checkpointing app reads memory pages of a client’s VM to create a memory snapshot. The rootkit detector inspects memory pages of the client’s VM to detect malicious code. One benefit of chaining these two apps is that the client can make the memory checkpointing app only take a snapshot of memory pages that are ensured to be benign by the rootkit detector. The hypervisor requests a receive from the memory checkpointing app to read a memory page of the client’s VM. In turn, the hypervisor consults the chaining policy, which specifies client’s VM → Rootkit detector → Memory checkpointing app. Thus, the hypervisor invokes the rootkit detector to scan the memory page, and indicate whether the page is malicious or not. If malicious, the rootkit detector breaks the chain, and the checkpoint is not created. Otherwise, the benign memory page is forwarded to the checkpointing app, which creates the VM image.

Our description of chaining rules is substantially similar to the notion of I/O policies discussed in Section 4.4.1. This similarity is not incidental, and the same concepts discussed there apply here as well (e.g., treating policies as state machines). The primary difference between chaining rules and I/O policies is that chaining rules can specify low-level system apps, I/O interceptors or bundled apps; in contrast, an I/O app only specifies I/O interceptors. While the example discussed above illustrates composition using two low-level system apps, the concept extends in a straightforward fashion to I/O interceptors. In this case, the composition policy is mapped in a natural way to an I/O policy, which is then enforced by the SDN controller by configuring virtual switches on each physical host. We provide an example of such a composition of a low-level system service and an I/O interceptor to provide an application-level traffic firewall in Section 6.3.2.

5. Implementation

We implemented a prototype of the infrastructure to support cloud apps atop KVM version 3.12.8. The implementation adds roughly 600 LOC to the KVM kernel module. In addition, our implementation also comprises a user-space module consisting of roughly 500 LOC of Java code. We describe the components of our prototype implementation in the rest of this section.

5.1. Hypervisor-level support

For low-level system service apps, our implementation supports the two design options discussed in Section 4.3.1 and Section 4.3.2.

The lifecycle of an app begins when the user inspects its manifest and installs the app in his domain. At this point, our infrastructure parses and reads the manifest. This functionality is implemented as a user-space module in the hypervisor’s management domain. This user-space module communicates with the hypervisor to set up the privileges requested by the app. It does so using two new ioctl commands that we introduced in KVM, shown in Fig. 9. Both design option 1 and option 2 incorporate this user-space module and the ioctl interface to control permissions.

Once an app has been installed, in option 1 it interacts with the hypervisor via hypercalls. We modified KVM to add the hypercalls shown in Fig. 9. Cloud apps use the hypercalls to map the memory and register state of client VMs into their

<table>
<thead>
<tr>
<th>New Hypercalls (Reads only; we also added similar calls to write)</th>
</tr>
</thead>
<tbody>
<tr>
<td>vaddr_Read_VM_Memory(id_of_client_vm, virt_addr, bytes, buf)</td>
</tr>
<tr>
<td>Description: Read memory of a client’s VM starting at virtual address virt_addr.</td>
</tr>
<tr>
<td>phyaddr_Read_VM_Memory(id_of_client_vm, phy_addr, bytes, buf)</td>
</tr>
<tr>
<td>Description: Read memory of a client’s VM starting at physical address phy_addr</td>
</tr>
<tr>
<td>PPN_Read_VM_Memory(id_of_client_vm, page_frame_num, buf)</td>
</tr>
<tr>
<td>Description: Read a page frame of a client’s VM</td>
</tr>
<tr>
<td>register_event(id_of_client_vm, virt_or_phy, addr)</td>
</tr>
<tr>
<td>Description: Register to receive a trap from the hypervisor when a client VM address is modified.</td>
</tr>
<tr>
<td>read_Register(id_of_client_vm, register_id, buf)</td>
</tr>
<tr>
<td>Description: Read a register’s content of a client’s VM</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IOCTL Commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>kvm_ioctl_set_register_privilege(VM1_Name, VM2_Name)</td>
</tr>
<tr>
<td>Description: Grant VM1 a permission to have access to VM2’s registers.</td>
</tr>
<tr>
<td>kvm_ioctl_set_memory_privilege(VM1_Name, VM2_Name, Memory_Type)</td>
</tr>
<tr>
<td>Description: Grant VM1 a permission to have access to VM2’s memory. Memory_Type indicates user space memory or kernel space memory</td>
</tr>
</tbody>
</table>

Fig. 9 – Description of hypercalls and ioctl commands added to KVM to support Option 1.
address VMs. To map a physical page belonging to a client's VM into a virtual address space of a cloud app, a hypercall handler translates the client VM's physical address in memory to a virtual address via the extended page tables (EPTs) maintained by KVM. Once an app has mapped a client VM's pages into memory, it can modify the client VM's page tables and register with the hypervisor to receive traps on events of interest (register_event). For example, it can modify the page tables to receive a trap each time a page of the client VM is executed. This enables the development of cloud apps that can check code pages as they execute (e.g., as was done in the Patagonix system (Litty et al., 2008)).

It is straightforward to support option 2 in KVM because it supports nested virtualization. The above hypercalls can be supported within the L1 hypervisor (which is part of the cloud app), and no changes are necessary on the L0 hypervisor (which is controlled by the cloud provider).

5.2. Support for I/O plumbing

To intercept I/O and allow clients to plumb I/O paths, our prototype uses a daemon within the management domain to process client I/O and composition policies. As discussed in Section 4.4.1, an I/O policy can be thought of as a state machine that the packet must traverse from the start state to the finish state. Thus, our goal is to tag each packet with the “state” that it currently is in.

We assume that the cloud provider supports a centralized controller (as is standard in SDN deployments), which stores the identities of the switches hosting the cloud app/client VM. This controller accepts the client's I/O and composition policies and distributes the corresponding state machines to all the software switches. When a packet is processed by the cloud app running on a physical host, the software switch that executes within the management domain of that host tags the packet with the identity of the cloud app that just processed the packet. Once the packet reaches the “final state,” of the state machine, it is forwarded to the client VM.

Our implementation incorporates the above tagging and forwarding support within the software switches using the Floodlight (Project Floodlight) Openflow controller. We communicate with Floodlight’s static flow pusher via its REST API (Floodlight REST API – Floodlight Controller – Project Floodlight) to insert flows. We use the mod_vlan_id and strip_vlan actions to respectively tag and untag packets.

6. Evaluation

The main goal of our experimental evaluation is to understand the performance overhead introduced by the cloud app model. We do not focus on the effectiveness of the apps (e.g., the effectiveness of a rootkit detection app at identifying real rootkits) because we believe that is orthogonal to the goal of this paper. Rather, we show how apps that have been proposed in prior work can be implemented and deployed easily as cloud apps, and measure their performance overheads.

We conducted experiments for both low-level system apps and I/O interceptors. Our setup consisted of three physical machines in the same physical LAN. The throughput between each machine is 100 Mbps. The first machine is equipped with an Intel Core i7 CPU, 8 GB of memory and an Intel Gigabit NIC. The second machine has an Intel Xeon(R) CPU, 24 GB of memory and two NetXtreme BCM5722 Gigabit Ethernet cards. We dedicate the third machine as the measurement host; this machine has an Intel Core i5 CPU and 4 GB of memory. The first two machines run Ubuntu version 12.04 with kernel version 12.08. The measurement host runs CentOS with kernel version 2.6.32. Guest VMs run Ubuntu version 12.04 with kernel version 3.5. Both L0 and L1 uses KVM kernel module version 12.08 as their hypervisors.

6.1. Low-level system apps

In this section, we evaluate the impact of running low-level system apps. In particular, we implemented two system service apps that map memory pages of a client’s VM for detecting rootkits and checkpointing memory. We use SPEC CINT2006 benchmark to measure the utilization of the CPU and memory of VMs. We run the benchmark in three setups: on the host OS, on the single-level guest and on the nested guest.

6.1.1. Rootkit detection

We implemented a rootkit detection app that implements the same functionality as Patagonix (Litty et al., 2008), and detects rootkits by checking code integrity. It ensures that all code executing on the client’s VM belongs to a whitelist. The KVM hypervisor traps any executing code pages of the client’s VM and allows the rootkit detector to map those pages for checking. We execute the benchmark inside the client’s VM to measure the impact of the rootkit detector on the client’s VM.

We implemented this app using both option 1 (hypervisor modifications) and option 2 (nesting). We compared the performance of both options to the traditional approach of running the service as a daemon within the management domain, as would be done on contemporary cloud platforms that do not support a cloud app model. This traditional approach serves as our baseline for performance measurements.

Fig. 10 shows the results under the three setups. We observed a slowdown of 15.3% in the nested virtualization setup and an overhead of 8.5% in the hypervisor modification setup. As explained in prior work (Ben-Yahuda et al., 2010), most of the overhead in the nested case is caused by VM exits from the guests to the L0 hypervisor due to privileged instructions. Fig. 11 shows the CPU utilization in each setup.

6.1.2. Memory checkpointing

Our memory checkpointing app maps memory pages of the client’s VM and checkpoints them for various purposes such as debugging or crash recovery. We evaluated this app by checkpointing three client’s VMs with different memory footprints: 512MB, 1024MB and 2048MB.

Fig. 12 presents the results of our experiments, comparing the costs of implementing the memory checkpointing service within a cloud app (both design options) with the traditional setup. Our results show that the overhead of implementing the checkpointing service within a cloud app using hypervisor modification is 7.95%. The slowdown caused by implementing the same service using nested virtualization is 12.38%.
6.2. I/O interceptors

To evaluate the performance of I/O interceptors, we compare the overhead of implementing the service as a cloud app, which runs as a separate VM, with the traditional approach of implementing the I/O interception service as a daemon within the management VM. Each cloud app in our experiment is assigned one virtual CPU and 2 GB of memory. We performed each experiment under two settings, a same host setting where the cloud app is co-located with the client VM that it services, and a diffhost setting, where the cloud app and the client VM are on different physical hosts, as shown in Fig. 13.

6.2.1. Baseline overhead

Before measuring the overhead of an I/O interceptor we measure the overhead when a cloud app does no additional processing on the packets that it receives. This overhead is the
best performance achievable for an I/O interceptor. The overhead of any additional services that the interceptor provides must be compared against this baseline overhead.

The measurement host is used to transmit and receive network packets to the client’s VM. We measured the network throughput to the client’s VM using iperf3 ([iPerf – The TCP, UDP and SCTP network bandwidth measurement tool](https://iperf.fr/)), and used ping to measure the round-trip time (RTT) of network traffic to and from the client’s VM. Our results report averages over ten executions along with the standard deviations. Fig. 14 presents the results of our experiment. If the cloud app is co-located with the client’s VM, the throughput drops to 3.44% compared to the traditional setup. The RTT overhead also increases 2.29× as a result of having to traverse the cloud app on the path to the client’s VM. When the cloud app and the client’s VM are on different hosts, the RTT overhead increases 1.36× and the throughput reduces to 4.08% compared to the traditional setup.

6.2.2. Network intrusion detection

In the cloud app model, clients can deploy and configure NIDS apps as network middleboxes. As an example, we used Snort ([Roesch, 1999](https://n FIELDID-104)) to build an NIDS app. Snort uses libpcap ([The libpcap project – a system-independent interface for user-level packet capture](https://www.tcpdump.org/libpcap.html)) 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. Fig. 15 shows the results of our experiments.

6.2.3. Storage services

We implemented two storage I/O interceptors that intercept storage streams from the client’s VM to respectively detect intrusion and check the integrity of the file system. We conducted the experiments by writing a file of size 500 MB from the client’s VM. We measured the throughput of the write operations and the total time to write the file. We compared the results against the traditional setup where the storage service is implemented within the host OS. As before, we compared the results against a baseline overhead when a storage app does no additional processing on the storage streams it receives.

6.2.3.1. Storage intrusion detection system (SIDS). We implemented the SIDS by running an NFS server service and using Snort as an intrusion detection system (IDS). We use the same configuration for Snort as described earlier, with additional rules for storage packets. Any write operations from the client’s VM are sent to the NFS server service in form of network packets. The IDS discards any packets that violate its rules. The performance results are shown in Fig. 16.

6.2.3.2. File integrity checking. Our file integrity checking app checks the integrity of files of the client’s VM at block level. We implemented this app by fetching inode information of the file that the client wants to check. We then collect the number of blocks of the file and hash them to form a whitelist. The app can check the integrity of the file by re-hashing it and compare the results with the whitelist. Fig. 17 presents the evaluation results.

6.3. Composing cloud services

In this section we evaluate the impact of composing cloud apps. In particular, our experiments examine the composition of low-level system apps and I/O interceptors.

6.3.1. Composing system-level service apps

In this experiment, we composed the memory checkpointing app with a rootkit detection app so that the checkpointed memory pages are ensured to be benign. Fig. 18 shows the results. The total overhead incurred by the bundled app is about 16%. This is because memory pages have to be checked by the
rootkit detection app before being checkpointed by the memory checkpointing app.

We also composed a firewall app with a NIDS app to form a policy chain Firewall→NIDS→Client’s VM. The firewall app contains a set of rules that include a list of IP addresses and open ports that are accepted. The two apps run on the same physical machine while the client’s VM run on the other. The results shown in Fig. 19 indicate that the throughput drops to 9.3%. The RTT also increases 1.75x due to having two middleboxes in front of the client’s VM.

Finally, we composed storage I/O interceptors by chaining our SIDS app with a storage deduplication app. In Unix file systems, inodes hold the block numbers of disk blocks. Our storage deduplication app works by having ino dereference a single block rather than duplicate blocks. We use the MD5 checksum of the block’s contents as its fingerprint.

We conducted the experiments by writing two files of size 500 MB from the client’s VM. The first file is written two times while the second file is written one time. We measured the throughput of the write operations as well as the disk usage at the storage deduplication app either when both cloud apps are on the same host or on a different host with the client’s VM. We compared the results against baseline overheads when storage apps do not perform additional processing on the storage streams they receive.

The results are shown in Fig. 20. We consider both the samehost case, where the bundled app and the client VM are on the same host, and the diffhost case, where the bundle executes on one machine and the client VM on the other. We observe that the throughput reduces to 10% and the deduplication app saves approximately 4.3% of disk space.

### 6.3.2. Hybrid apps

A hybrid app bundles an I/O interceptor with a system-level service. We evaluated an app that implements an application-level firewall (based on prior work by Srivastava and Giffin, 2008)

<table>
<thead>
<tr>
<th>File Name</th>
<th>Throughput (Mbps)</th>
<th>Disk Usage(MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>No File</td>
<td>NA</td>
<td>280</td>
</tr>
<tr>
<td>File 1 (500M)</td>
<td>89.1±0.328</td>
<td>780</td>
</tr>
<tr>
<td>File 1 (Copy)</td>
<td>89.9±0.247</td>
<td>1330</td>
</tr>
<tr>
<td>File 2 (500M)</td>
<td>89±0.153</td>
<td>1380</td>
</tr>
</tbody>
</table>

**SAMEHOST configuration**

<table>
<thead>
<tr>
<th>File Name</th>
<th>Throughput (Mbps)</th>
<th>Disk Usage(MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>No File</td>
<td>NA</td>
<td>280</td>
</tr>
<tr>
<td>File 1 (500M)</td>
<td>85.2±0.146</td>
<td>780</td>
</tr>
<tr>
<td>File 1 (Copy)</td>
<td>85.6±0.483</td>
<td>820</td>
</tr>
<tr>
<td>File 2 (500M)</td>
<td>84.9±0.398</td>
<td>1320</td>
</tr>
</tbody>
</table>

**DIFFHOST configuration**

<table>
<thead>
<tr>
<th>File Name</th>
<th>Throughput (Mbps)</th>
<th>Disk Usage(MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>No File</td>
<td>NA</td>
<td>280</td>
</tr>
<tr>
<td>File 1 (500M)</td>
<td>79.4±0.216</td>
<td>780</td>
</tr>
<tr>
<td>File 1 (Copy)</td>
<td>79.7±0.387</td>
<td>820</td>
</tr>
<tr>
<td>File 2 (500M)</td>
<td>79.1±0.178</td>
<td>1320</td>
</tr>
</tbody>
</table>

Fig. 20 – Composition of storage based service.

by intercepting network packets from the client’s VM. It then performs memory introspection of the client’s VM to identify the process that is bound to the network connection. If the process belongs to a whitelist, then the app permits the network flow; otherwise it blocks the network connection. This app therefore provides both a system-level service and a networked service and must be co-locate with the client’s VM.

Fig. 21 presents the results of our experimental evaluation that compares the implementation of this firewall as an app versus a traditional setting, where it is implemented as a daemon within the management VM. The app’s throughput reduces by 2.93% when the firewall service is implemented as a cloud app. The RTT increases 1.53x due to having network packets going through the app.

### 7. Related work

Prior work by virtualization and cloud vendors has attempted to implement some of the features that we have described in our cloud app model. In an effort to manage multi-tiered applications, VMware introduced the concept of a virtual appliance (vApp) (Managing VMWare vApp – VMWare vSphere 4 ESX and vCenter Server). A vApp is a container that contains one or more VMs. It allows resource control and management of VMs. In particular, VMWare vApp offers users different options to allocate IP addresses for VMs. IP addresses can be fixed, allocated by a DHCP server or transient (allocated at power-on and released at power-off). Also, entire vApps can be cloned, powered on and powered off. Users also have the capability to specify startup as well as shutdown order among VMs. For example, in a vApp that holds a web server VM and a database server VM, a user can require that the database server to start before and shutdown after the web server. VMWare vApp, however, provides no explicit feature for composing middleboxes among a group of VMs. To our knowledge, it also does not give clients an API to implement low-level system services as apps.

The Amazon virtual private cloud (Amazon Virtual Private Cloud (VPC) – Secure Private Cloud VPN) lets users define a virtual network for their virtual machines. Users can select IP address range, create subnets as well as configure network gateways. With Amazon VPC, users can specify simple middlebox policies that involve only one middlebox. For example, a user can deploy a firewall middlebox in her network by configuring the firewall VM to function as the gateway of her network. Complex middlebox policies that entail the traversal of network packets among a sequence of middleboxes are hard to achieve with Amazon VPC.

While both of these techniques implement support for grouping and controlling collections of VMs, to our knowledge, our work is the first to provide a unified framework for
implementing system and network-level services as cloud apps, consider various design options, and develop the framework to support third-party cloud apps (e.g., permissions, I/O policies, composition policies).

There is nearly a decade of prior work on virtual machine introspection-based services, starting with the seminal work of Chen and Noble (2001) and Garfinkel and Rosenblum (2003). This body of work (references too numerous to list here) has developed a number of innovative approaches to building security services using virtual machines. While these prior techniques have focused on building the capabilities and precision of the security tools, they have largely assumed that the tools themselves will be implemented in a separate virtual machine (e.g., as a daemon in the management domain).

On traditional cloud platforms, implementing these techniques requires the cloud provider to deploy the services within the management domain for each client. In contrast, our work considers a baseline infrastructure that the cloud provider must offer (e.g., hypervisor-level changes, modifications to software switches, and supporting cloud app privileges) that would empower clients to independently deploy these services as apps on their own VMs. We laid some of the foundations for these infrastructural changes in our prior work on self-service cloud computing (Butt et al., 2012, 2014), where we developed the basic ideas to support low-level system service apps. This paper significantly extends those ideas, and introduces the notions of and provides supporting infrastructure for I/O interceptors, app permissions and composition policies.

8. Summary and conclusions

This paper presents the design and implementation of a rich model that allows cloud apps developed by third-parties to perform privileged tasks on a client’s VMs. Clients can implement a variety of services, such as low-level system services, I/O interception, and even bundle several services into a single app. We discussed the infrastructure support needed to support cloud apps, explored various design options to implement cloud apps, demonstrated and evaluated the practicality of cloud apps using various examples. We believe that our work demonstrates the potential of and a practical way to realize the vision of security as a service, where security services are implemented as apps that can be paid for, downloaded and used by clients on their cloud-based VMs.

Acknowledgments

This work was funded in part by the NSF grant CNS-1420815 and a gift from Microsoft Research.

References


Pitropakis N, Lambrinoudakis C, Geneiatakis D. Till all are one: towards a unified cloud IDS. In: 12th International Conference on Trust, Privacy and Security in Digital Business. 2015.


Amazon Virtual Private Cloud (VPC) – secure private cloud VPN. [http://aws.amazon.com/vpc/] [accessed 29.06.16].

The libpcap project – a system-independent interface for user-level packet capture. [http://sourceforge.net/projects/libpcap/] [accessed 29.06.16].

Managing VMWare vApp – VMWare vSphere 4 ESX and vCenter Server. [https://pubs.vmware.com/vsphere-4-esx-vcenter/index.jsp?topic=/com.vmware.vsphere.bsa.doc_40/vc_admin_guide/managing_vmware_vapp/c_managing_vmware_vapp.html] [accessed 29.06.16].

Project Floodlight. [http://www.projectfloodlight.org/projects/].

The Cloud Market: complete catalog of Amazon EC2 images. [http://thecloudmarket.com] [accessed 29.06.16].

AWS MarketPlace – find and buy server software and services that run on the AWS Cloud. [http://aws.amazon.com/amis] [accessed 29.06.16].

Cisco – SourceFire. [http://www.sourceforge.com] [accessed 29.06.16].

Floodlight REST API – Floodlight Controller – Project Floodlight. [http://www.openflowhub.org/display/floodlightcontroller/REST-API] [accessed 29.06.16].

iPerf – The TCP, UDP and SCTP network bandwidth measurement tool. [https://github.com/esnet/iperf] [accessed 29.06.16].

Hai Nguyen is a Ph.D. candidate in the Department of Computer Science at Rutgers University. His research is on cloud computing, operating systems and virtualization.

Vinod Ganapathy is an associate professor of Computer Science at Rutgers University. He earned a B.Tech. degree in Computer Science & Engineering from IIT Bombay in 2001 and a Ph.D. degree in Computer Science from the University of Wisconsin-Madison in 2007. His research interests are in computer security and privacy, software engineering, mobile systems and virtualization.

Abhinav Srivastava is research scientist at AT&T Labs-Research in Bedminster, NJ. He earned a Ph.D. degree in Computer Science from Georgia Institute of Technology in 2011. His research interests are in operating systems, distributed systems, systems security, and cloud computing.

Shivaramakrishnan Vaidyanathan is a Ph.D. candidate in the Department of Computer Science at Rutgers University. He previously earned an MS degree from NYU Polytechnic in 2011 and worked at Barracuda Networks. His research is on cloud computing.