CQSTR: Securing Cross-Tenant Applications with Cloud Containers

Yan Zhai
University of Wisconsin Madison
yanzhai@cs.wisc.edu

Lichao Yin
Google, Inc.
lichaoy@google.com

Jeffrey Chase
Duke University
chase@cs.duke.edu

Thomas Ristenpart
Cornell Tech
ristenpart@cornell.edu

Michael Swift
University of Wisconsin Madison
swift@cs.wisc.edu

Abstract

Cloud providers are in a position to greatly improve the trust clients have in network services: IaaS platforms can isolate services so they cannot leak data, and can help verify that they are securely deployed. We describe a new system called CQSTR that allows clients to verify a service's security properties. CQSTR provides a new *cloud container* abstraction similar to Linux containers but for VM clusters within IaaS clouds. Cloud containers enforce constraints on what software can run, and control where and how much data can be communicated across service boundaries. With CQSTR, IaaS providers can make assertions about the security properties of a service running in the cloud.

We investigate implementations of CQSTR on both Amazon AWS and OpenStack. With AWS, we build on virtual private clouds to limit network access and on authorization mechanisms to limit storage access. However, with AWS certain security properties can be checked only by monitoring audit logs for violations after the fact. We modified OpenStack to implement the full CQSTR model with only modest code changes. We show how to use CQSTR to build more secure deployments of the data analytics frameworks PredictionIO, PacketPig, and SpamAssassin. In experiments on CloudLab we found that the performance impact of CQSTR on applications is near zero.

SoCC '16, October 05 - 07, 2016, Santa Clara, CA, USA.

ISBN 978-1-4503-4525-5/16/10. . . $15.00.
DOI: http://dx.doi.org/10.1145/2987550.2987558

* Categories and Subject Descriptors: D.4.6 [*Operating System*]: Security and Protection—Access Control
* General Terms: Design, Security
* Keywords: attestation, cloud containers

1. Introduction

An increasing fraction of computing services run on managed infrastructure-as-a-service (IaaS) systems such as Amazon AWS, Google GCE, or Microsoft Azure [36, 56]. They form an ecosystem of inter-connected tenant services run on a common virtualized platform, sometimes sharing data [33]. Despite a common underlying platform, *safe cross-tenant sharing* generally requires blind trust: clients with sensitive data must trust their partners to protect it.

Consider a motivating example from *outsourced analytics*: a mail provider contracts out spam filtering to a third-party service. The mail provider would like some confidence that its private email will not be released or abused. More generally, any client may want specific assurance that a service it uses is configured to protect its data.

We argue that cross-tenant *security attestations* about the security properties of service deployments are critical elements to build this trust. Attesting to the properties of a single program, VM instance, or physical machine is not sufficient for modern cloud-based services; vulnerability in any component of a service can compromise security of the entire service. Rather, customers must be able to check the security properties of a distributed service as a whole.

We make three observations about IaaS platforms:

(1) Cloud providers already serve as a root of trust for their tenants, as they provide trusted services such as authentication and access control.

(2) A provider's software can attest to a wide array of security properties of their tenants from information already available to the platform.

(3) IaaS platforms are managed by a single entity and provide a trusted communication and distribution infrastructure.

Thus, *IaaS cloud platforms already provide most of what is needed to enable safe cross-tenant sharing.* In this work we explore how to take advantage of this potential. We propose an extended IaaS-layer framework—called CQSTR[1]—for managing *contained* execution, in which a group of tenant instances (VMs) have their external connectivity restricted according to a declared policy as a defense against information leakage.

CQSTR introduces a new *cloud container* abstraction, which extends the notion of a container on a local operating system [7] to cloud-scale applications consisting of many VM instances and other resources. A cloud container specifies immutable confinement properties that limit network and storage access for computations in the container. In addition, the policy may specify a set of images that are allowable to boot VM instances into the cloud container. The IaaS provider issues attestations of these properties to clients of the service. For example, CQSTR can attest that a service runs from known disk images containing a locked-down operating system and a trusted application framework, and that network access is limited.

Cloud containers provide a data owner with a verifiable set of controls over outsourced private data. A client can verify that a service meets the its required security policies to protect data from leakage and misuse. Security policy is designed and specified separately from the application and is enforced by the IaaS platform, which allows CQSTR to run existing operating systems and applications without modification. Further, CQSTR provides a foundation for more flexible control based on security mechanisms in the attested VM images, such as privilege separation, least privilege or defense-in-depth policies.

To explore how existing IaaS APIs support secure cross-tenant sharing, we implemented CQSTR abstractions as a layer above AWS. The implementation builds on the rich support in AWS for role-based access controls [15], fine-grained control over the network with Virtual Private Clouds [20], and security logging. However, due to various limitations (see Section 4), we cannot use AWS authorization mechanisms alone to control all output paths available to a service on the AWS platform at this time. As a result, our AWS-based implementation enforces compliance with a cloud container specification by monitoring audit logs to detect violations after the fact.

To provide stronger assurances, we separately implemented CQSTR as an extension to OpenStack. We made modest changes to the OpenStack platform to add new functions missing from existing IaaS platforms, including the ability to freeze container configurations, enforce access control based on container security properties, and prevent extraction of data from a closed container by abuse of IaaS-level management services such as backups [18] and log monitoring [17]. We believe that these extensions are practical for broader adoption.

We exercise CQSTR on OpenStack with outsourced analytics scenarios using SpamAssassin, PacketPig, and PredictionIO. No changes to the application source code are needed, although we did write small proxy services to enforce additional access controls. Our performance measurements show that CQSTR has essentially *zero overhead* in these data analytics scenarios.

[1] Read as *sequester*: verb, meaning "to isolate or hide away".

2. Background and Challenges

Our work focuses on infrastructure-as-a-service (IaaS) clouds that provide customers with virtual machine instances and other resources. Customers specify a disk image stored in the cloud, and the IaaS cloud provider selects a physical machine and boots a virtual machine from the chosen image. IaaS systems also provide customers with virtual disks, blob storage and other services whose use are mediated by the cloud provider.

Cloud applications are increasingly constructed as a mix of IaaS-provided services and third-party services provided by other tenants running on the same IaaS platform. The goal of our work is to improve trust between components of such systems. To set context, we outline an exemplary scenario for which current solutions prove unsatisfactory.

An example problem: secure data analytics. Suppose a cloud tenant Alice has shopping-cart data, and would like to use an analytics service run by another cloud tenant Bob to make shopping recommendations from the data. However, the data has competitive value, so she would like to make sure that Bob cannot use the data for anything but generating recommendations.

The ideal workflow for this scenario would be:

(1) Alice pushes her data to Bob's service every day.

(2) Bob's service trains a prediction model based on the data each day, which might require tens or even hundreds of VM instances.

(3) The prediction service deploys the model to a web service, to which Alice submits requests for shopping predictions. The web service may itself consist of many instances and tiers.

Alice's security requirement is that her history data and queries for prediction are kept secret from Bob and any third parties. Bob may have proprietary algorithms running in his service, and will not expose his code to Alice, so Alice cannot run the prediction service herself.

Note that the scenario above arises in practice already: companies like Google, Amazon, and BigML offer such machine-learning services and in many cases keep their al-

gorithms secret [19, 35]. This example represents a *client/server* setting, for which containment is sufficient: Alice, the client, provides data to Bob's contained computation service, and the output is delivered only to Alice. In a *delegated* setting, the output of the service may instead go to Bob or an unrelated third party; in essence Alice delegates control over the data to the service, so some trust in the service code is required. For example, a hospital may allow researchers to use the results of a data analysis. Beyond machine learning, many other applications fall into this setting of cross-tenant sharing between a data provider (Alice) and a compute provider (Bob). More examples are in Section 6.

These scenarios all prompt the key question faced in our work: ***How can a trusted IaaS provider safely assure Alice of Bob's secure handling of her data?***

On using existing IaaS features. Leading IaaS clouds such as Amazon Web Services (AWS) are feature-rich, and one might expect that such cross-tenant sharing scenarios can be handled with existing functionality. In particular, AWS offers virtual private clouds (VPCs) [20], which support a logically isolated network for a set of instances, and AWS Identity and Access Management (IAM) provides role-based access control over data and services. Bob could launch his service within a VPC, and Alice could grant Bob via IAM the permission to fetch her customer data (e.g., from an S3 bucket) after checking the service properties for compliance with her policy.

However, there is no agreed-upon mechanism in current IaaS clouds for services to expose security information to clients. And, there are numerous individual services that must be separately managed to ensure security. Section 4 discusses the challenges of implementing cloud containers on AWS, and Section 8 discusses other approaches, such as cryptographic multiparty computation, information flow control, or trusted hardware.

Threat model. We view the IaaS provider as trusted, a reasonable presumption for users of cloud systems today. We therefore consider threats such as insider attacks by IaaS employees, cross-VM isolation boundary violations [39, 45, 55, 57, 60], or compromise of the IaaS control plane (e.g., exploits against hypervisors) to be out of scope. These threats are important, but are not addressed by CQSTR.

Instead, we focus primarily on threats arising in applications running on top of the IaaS platform. We consider for simplicity settings with two tenants, e.g., Alice and Bob in our example above. We consider situations in which either Alice or Bob behaves maliciously, because of (accidental) misconfigurations or insider attacks, e.g., by application developers. We also consider remote attackers: one goal is to attest to Alice that Bob is using best practices for defense.

3. CQSTR Abstractions

Our central hypothesis is that carefully chosen, but relatively modest, extensions to IaaS platforms can provide a foundation for constructing trustworthy cloud-hosted services. To this end, we describe the design of CQSTR, which extends IaaS cloud management platforms to enable tenants to share data safely and assert more control over use of their data.

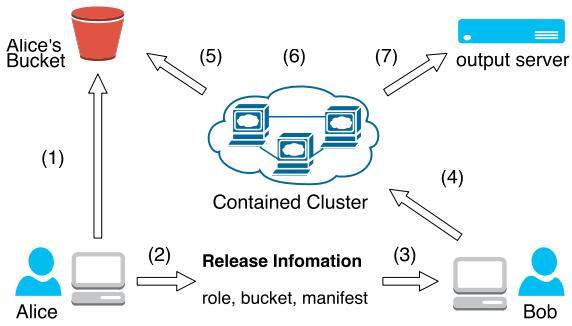

Figure 1. The logical view of CQSTR. (1) Alice places data in bucket, and (2) publishes location of data and security properties required for access. (3) Bob retrieves information, and (4) launches service in cloud container with required restrictions. Service (5) fetches data, (6) computes over it, and (7) publishes results to allowed output server.

Overview. CQSTR comprises three central elements:

(1) *Cloud containers* are groupings of VM instances and are the units of security policy management.

(2) *State assertions* are authenticated statements from the cloud provider about the security properties of a service's container.

(3) *API restrictions* ensure that IaaS management APIs cannot be abused to violate containment policies.

Cloud containers group the VM instances and related resources comprising a service and allow attesting to their security properties collectively. State assertions provide clients a mechanism to verify the configuration of a service, such as the isolation policies of its container or the set of VM images used to boot instances in the service (i.e., a form of remote attestation). Finally, API restrictions limit the use of provider APIs that might compromise a container's asserted security properties. CQSTR leverages the IaaS provider's trusted network for secure cross-tenant communication; IaaS providers already control their networks to prevent spoofing and snooping by their tenants [14].

In client/server settings, the service is launched into a cloud container. A client consumes state assertions to verify that the container's security properties meet its policy requirements for protection against data leakage. In delegated settings, the client can pass the service provider a description of the required security properties, and then verify that the service meets those properties before handing it data. A data owner can also place security conditions on stored data (e.g., storage buckets or objects) to limit access to only those containers whose security properties comply with the conditions. Figure 1 illustrates the use of the system.

3.1 Cloud Containers

The key new abstraction provided by CQSTR is the cloud container, which is a set of VM instances sharing access to

a private network and cloud resources, with a common set of networking, storage, and boot image restrictions. Thus, a cloud container logically extends operating system containers (e.g., Linux containers [8]) to a cluster. Abstractly, a container is described with a *manifest* that specifies a set of *security properties* detailing explicitly what instances in the container can do; anything not listed is prevented. Launching an instance into a cloud container restricts what code it can run (described below), to whom it can communicate, and where it can store and retrieve data.

Cloud containers build on existing virtual private network functionality already available on AWS/EC2 and OpenStack to limit the set of reachable hosts and to enforce firewall rules. A cloud container policy may also enforce traffic controls, such as limiting the ingress or egress bandwidth of a container, and the total amount of data that can be transferred.

A service may consist of one or more cloud containers. When a service has multiple instance types internally, such as master and worker in Hadoop, cloud containers can further secure a system by placing each class of VM in a separate cloud container that has its own set of boot images and accessible network/storage. We term this a *cloud container group*.

Storage containment. Storage protection is enforced by extending the mechanisms to control the resources accessible to a cloud container with existing IaaS services. In addition to access control lists on storage objects, CQSTR adds sets of reachable objects to a cloud container description, providing complete control over the storage and other objects accessible to a service. CQSTR allows controlling both access to existing storage objects as well as what new storage objects, such as disk images or blob storage buckets, can be created from a container.

Trusted images. In delegated scenarios, a client may require trust in the code executing in the service before releasing information to it. For example, a hospital may require that a data-mining service uses a known and endorsed software image that enforces an information disclosure policy. A container manifest can therefore specify a set of *trusted images* eligible to run in the container. In this scenario, the client must trust the container's images to behave correctly, either through direct knowledge or endorsement by a trusted third party. All other images are prevented from being used to boot VMs within the container.

Cloud container API. A cloud container is created with a *manifest* (see Figure 2 for an example) that specifies the security properties of the container, or by creating a default container and explicitly setting the properties. CQSTR provides APIs to create, read, update, and delete cloud containers. Once created, the owner of the container can launch VM instances into the container from trusted images. However, once an instance has been launched into a container, the container properties are *frozen*, meaning they can no longer be modified and clients are assured they will remain in force.

3.2 State Assertions

A client of a service using CQSTR can make access control or usage decisions based on the security properties of the service's cloud container. There are two mechanisms for verifying assertions over a cloud container's state. First, a client can directly query the provider for assertions about a service, such as how the network is configured or what storage objects are accessible. Second, a client can place access control lists on storage objects that specify the desired properties of services accessing those objects. We describe in Sections 4 and 5 how to do this with small extensions to existing IaaS capabilities.

Properties covered by state assertions are subject to two conditions. First, a property must be *knowable* to the IaaS provider: it is easily observed by the cloud management plane. For example, CQSTR cannot assert properties that would require introspection within VM images or inspection of network traffic data contents. Second, the property must be *stable*, meaning that once it is true it remains true. As an example, "has an instance stored more than 1 MB of data?" is a stable property, whereas the amount of data stored is not stable, as it changes whenever an instance writes more data. The reason for targeting stable properties is to avoid TOCTOU bugs [28] and for efficiency: clients can cache the results of most assertions.

Immutable configurations. Many useful properties already satisfy both conditions. One is "what image did an instance boot from?" The provider knows this easily, since the image was specified in the request to the provider, and it is stable. But other useful security properties are not stable. In AWS and OpenStack, configuration properties of an instance, such as the network security group, can be modified by the owner at any time. If a data owner granted access based on one configuration (e.g., when the security settings were compliant) but the configuration changed later (on purpose or accidentally), then data could be released.

CQSTR addresses this concern by restricting the set of management operations allowable on cloud containers once they are frozen. Specifically, operations cannot change the outcome of any state assertion. For example, an owner cannot reconfigure the network to allow more access, or add more trusted images. We call the properties of a cloud container an *immutable configuration*.

Manifest templates. It may be expensive to check that the manifest for a container meets all required restrictions. CQSTR therefore supports *manifest templates*, which are parameterized manifests. A tenant specifies the template and its parameters when launching an instance. For example, a common policy is to allow network traffic from a single client to a single port; the manifest template specifies complete isolation of network and storage except for this one port, with the client's IP address as the port parameter.

```
network:
  container: container2,
  direction: ingress,
  protocol: tcp,
  local_port: 8080
traffic:
  container: container2,
  out_quota: 100MB,
  bandwidth: 100KB/sec
volume:
  ro: public,
  rw: container
objectstore:
  bucket_name: dataprovider.hadoop
     visibility: public,
     write_quota: 100KB
images:
  hadoop_master, hadoop_slave
```

Figure 2. Sample manifest allowing *container2* to receive TCP packets on port 8080 with a download limit and bandwidth cap, use two images, access public volumes read-only, and write only to specified private volumes, with up to 100KB writes to a specified public object store bucket.

To check that a cloud container complies with a given parameterized template, a client first checks that the template matches the expected template, and that the parameter matches the expected parameter; this is much simpler than checking that an arbitrary manifest enforces a superset of required restrictions.

3.3 API Restrictions

The final element of CQSTR is a set of restrictions on the allowed management operations. As noted above, some operations are prohibited to ensure that configurations are immutable. In addition, IaaS APIs may leak information from a running instance, or to modify its behavior. API restrictions on a cloud container prohibit use of APIs that violate containment, such as creating a VPN tunnel or, more insidiously, writing to a log monitored by the IaaS provider, such as Amazon's CloudWatch service [17]: log entries could expose private data outside the cloud container.

API restrictions limit what the owner of a container can do. Any action that copies or transfers storage must apply the same controls on the data after copy or transfer. Thus, snapshots are given the same ACL as the source virtual disk. This also applies to booting a VM instance: an image generated within a cloud container cannot be used to boot instances outside the container.

These restrictions cannot fully prevent covert channels. For example, a service could intentionally vary its resource usage to encode information through fine-grained billing statements, or through monitoring APIs like AWS CloudTrail that record every provider API call [16]. In our implementation we strive to limit high-bandwidth covert channels.

3.4 Security Analysis

We briefly discuss some of the security benefits of CQSTR. We assume that an attacker cannot subvert the cloud provider's code, and hence cannot subvert network and storage restrictions. In addition, an attacker cannot spoof packets within the cloud network. We address three threat cases: malicious computation owners, malicious code, and remote attackers.

Malicious owners. A service owner may attempt to force trusted code to leak data. This can be done by SSH'ing into the VM, inspecting system logs, or using cloud APIs. Here, firewall rules can restrict the ability of the owner to access instances within the cloud container, and API restrictions prevent bulk channels such as logs or volume snapshots. Finally, a client may choose to trust only images with locked-down configurations that prohibit remote console access, or images endorsed by trusted third parties.

Malicious code. The worst case of attack is when VM instances run malicious code. Here, cloud containers can prevent leaking through explicit network and storage channels, but cannot prevent all covert channels through the IaaS API, as in the billing and logging examples.

In a client/server setting, output is accessible only to the client. In a delegated setting, though, output may go elsewhere. For service-generated data, such as the program output, logging, or performance metrics, traffic control on cloud containers mitigates the risk by limiting the output size and bandwidth. We show in Section 6 how an application-specific proxy can be used to further check or sanitize output.

Remote attackers. For third-party attackers, the primary defense is the network isolation and firewall rules. These network defenses can largely prohibit public access to instances within a cloud container.

4. Implementing Cloud Containers on AWS

To explore the potential for secure cross-tenant sharing with current IaaS cloud APIs, we implemented elements of the cloud container abstraction as a library above Amazon AWS. AWS provides key building blocks for the cloud container abstraction. Tenants may group instances in a virtual private cloud (VPC), and use VPC primitives to restrict network connectivity and access to some types of storage objects (e.g., S3 buckets). Role-based access control (through AWS IAM) can also limit access to storage objects. However, AWS only provides partial support to control access to all resources from a VPC, and it cannot freeze a VPC to prevent configuration changes. For these reasons, the AWS cloud container library uses *auditing* of VPC configurations and data access.

The audit-based implementation for AWS monitors compliance with a declarative policy description for a cloud container. It serves as a case study of how to use AWS security mechanisms for cross-tenant cooperation. However, the reliance on auditing security logs necessarily implies that the library cannot *prevent* non-compliant operations, but can only *detect* abuses afterwards. In addition, the library has shortcomings stemming from the limited control over logging granularity in today's AWS API. Most significantly, the

logs reveal unrelated information about the service owner's account to the client, and collusion with a third account can open hidden channels for data leakage, as described below.

4.1 Implementing Containers over AWS

VPCs form the basis of a cloud container implementation on AWS: CQSTR launches the instances comprising a container within a VPC, which controls network reachability and firewall rules. AWS does not yet support traffic shaping, so there are no bandwidth limits or traffic caps.

To restrict access to storage, the VPC specifies in advance any S3 buckets that the VPC writes to, so a client can check compliance with its policy. The service owner configures a *VPC endpoint* [23] that specifies a set of pre-existing S3 buckets accessible from the VPC; the endpoint denies write access to all other S3 buckets. Contained services cannot create new buckets.

When a client wants to grant a service access to its stored data, additional VPC setup steps are necessary. The service owner creates a VPC, and passes its ID to the client. The client creates a new role with access to its data, restricts the role for use only by the VPC, grants the service owner access to the role, and passes the role's name (ARN) to the service owner. The service owner can then configure instances to assume the role when accessing the data.

For other storage resources, such as EBS volumes or message queues, AWS does not provide mechanisms to restrict a VPC to access only a predefined set of objects: instances within the VPC can reach a resource if any party grants the VPC access to the resource. Restricting destination IP addresses is not enough, as AWS services may share a front-end server. As we discuss below, CQSTR uses auditing to verify that all resource accesses from the VPC are compliant with the policy. Similarly, AWS does not have a mechanism to limit the set of images bootable within a VPC, so CQSTR checks audit logs to verify that all instances in a VPC were booted with compliant images.

State Assertions. A client-side library checks compliance with security properties using two mechanisms. First, the client can contact the *config service* [22] to determine the properties of a VPC and its service; the service owner must grant clients access for this. In advance of contacting a service, clients query the configuration service for relevant security properties and verify that they meet requirements (e.g., restricted networking).

Clients must also verify that instances within the VPC do not transmit data to unapproved objects, such as EBS volumes or SQS message queues. However, the config service does not provide configurations of all objects, such as volume snapshots, S3 buckets, and SQS queues. Therefore, clients must retrieve audit logs for these resources to verify that past operations did not violate container security policies.

API restrictions. Cloud containers require immutable security properties that prevent TOCTOU races. However, AWS does not provide mechanisms to prevent management operations from changing the configuration of a VPC. Thus, it is not possible to freeze a configuration to make its properties immutable. As with storage access and attestations above, CQSTR on AWS relies on auditing as a substitute for the freeze operation: clients of a service must use the library to scan audit logs both before and after using the service to verify that the configuration never violated the client's desired policy. For example, the client must verify that VPC configurations do not change.

4.2 Checking Audit Logs

AWS provides powerful features for auditing VPCs and their configurations. The CloudTrail security event service [16] can log operations within a tenant account on an ongoing basis. We observed that the latency between an event and a record showing up in CloudTrail was about 5 minutes. While we were writing this paper, Amazon updated CloudWatch [17] with sufficient functionality to implement auditing at lower latency. As CloudWatch audits are functionally equivalent to CloudTrail, we describe our implementation in terms of CloudTrail.

We found that even when complete audit records are available for AWS objects, such as VPCs, the security logs do not always have complete information. For example, some objects have default settings that are not provided in CloudTrail logs, such as routing tables for VPCs. In these cases, we augmented log processing with calls to the AWS configuration service and use configuration histories to retrieve the complete settings.

Our audit log processing tool is about 700 lines of Python code and monitors 24 types of AWS objects.

Auditing handshake. Normally, audit logs are available to the account running a service. To allow clients of a service to check compliance with CQSTR security properties, the clients must be able to access the logs. Setting up a monitored cross-tenant VPC takes a number of steps, as, the service owner must make log events accessible to clients. The client creates a log bucket, creates a role, grants the role access to the log bucket, and grants the service owner's account access to the role. To make the log tamper-evident, the client creates an SSE-KMS key and grants the service owner's account permission to encrypt CloudTrail logs with the key. It passes the names (ARNs) of the bucket, role, and key to the service owner. The service owner enables CloudTrail logging on its account using the received ARNs.

Limitations of AWS auditing. We found that the AWS support for security logging is not well-matched to the needs of secure cross-tenant sharing. Generally, CloudTrail logging is designed to enable an account owner to monitor operations taken by identities associated with the account, rather than all operations associated with a given group of instances (e.g., a VPC). While a VPC is bound to account, its instances can issue operations under roles from a different account.

Operations issued under a role are logged to the role owner and not the VPC owner. If the service uses a role from a third account other than the client or service owner, the audit logs go to that third account and are not visible to the client.

CloudTrail logs also compromise privacy when used for cross-tenant monitoring. The security log includes *all events* in the service owner's account, and not just the events pertaining to a particular VPC of interest. If the service owner regards its logs to be confidential, then this confidentiality is violated. CloudWatch events can, in theory, filter out unrelated events involving non-target VPCs. However, in our experiments we found that filters must be updated when objects are created or deleted. In an asynchronous setting such as AWS, this can lead to lost audit records. Moreover, with multiple clients, events from one client are exposed to all others. One alternative is to use a third-party trusted auditor to check a service's compliance with the policies of its clients; ideally the IaaS provider would play this role. Furthermore, audit logs are a high-bandwidth covert channel and can allow a service to export sensitive data out of the VPC.

These limitations could be resolved by providing auditing on the granularity of an instance or VPC, rather than per account. In any case, the integrity of the auditing approach depends on comprehensive logging and monitoring of *all* data channels out of a VPC. As the default it so allow access to unaudited resources, security depends on extending the audit tool for each new resource introduced by AWS.

More generally, we conclude that adequate control of cross-tenant sharing requires specific support within the cloud provider API. Furthermore, it requires continued logging *after* using a service to ensure there are not data subsequent breaches. While security logging and auditing can be a basis for secure cross-tenant sharing, our premise is that an authorization model is a stronger basis for secure sharing. We therefore turn to an implementation of CQSTR on OpenStack.

5. Implementing CQSTR on OpenStack

We implemented CQSTR as an extension to OpenStack. Unlike the auditing based approach that is possible with current IaaS APIs, our OpenStack CQSTR implementation provides immutable cloud containers that allow only those resource accesses that are enumerated explicitly as compliant with a policy. This provides stronger guarantees.

We implemented CQSTR as an extension to OpenStack Kilo (2015.1) [9]. In total, we added 15 new files with 3,096 lines of code and modified 2,386 lines of code across 52 files. For comparison, the original components we modify comprise 455,247 lines of Python code. The added API restrictions comprise less than 500 lines.

OpenStack overview. OpenStack provides a complete set of software to implement a public or private cloud. It is widely used both within enterprises and as the basis for several public clouds including Rackspace and DreamHost. Table 1 describes the key components we modify.

Components	Functionality	Modification
Nova	Compute Service	Container manager, metadata service and API Restrictions
KeyStone	Authentication Service	Attestation-based access control
Neutron	Network Service	Network reachability and traffic control
Swift	Object Storage	Storage containment
Cinder	Volume Storage	Storage containment

Table 1. Modified OpenStack components.

Recent versions of OpenStack provide a *trusted computing pool* built on TPMs and an attestation server [10]. The server verifies measurements made by the TPMs, and allows clients to query the attestations for specific machines. This provides assurance about the code booted on a single machine, but not about the network configuration or use of management APIs. Furthermore, the goal of trusted computing pools is to verify trust in the hypervisor, which is complementary to our focus.

Our implementation comprises several components:

- *Container Manager*: stores container information and implements container APIs.
- *Network Agent*: enforces network control in *Neutron*.
- *Storage Agent*: isolates object namespace on *Swift*.
- *Metadata Service*: provide state assertions on *Nova*.

The crux of our implementation is to show how small modifications and extensions to existing services can provide a much richer security platform.

5.1 Containers and Their Management

CQSTR implements the container manager as an extension to the Nova compute service. All container and container group management APIs (Create, Read, Update, Delete) are provided as HTTP-based Nova APIs. Currently we require that a container be a member of exactly one container group. Containers are mutable until the first instance is launched into it; at that point they are frozen and become immutable. CQSTR supports dynamically adding and removing resources to/from a container, which can be important for allowing the container to scale with the computation dynamically.

For debugging purposes, CQSTR also provides a *debug* mode that allows configuration changes after instance launch. This allows an owner to extract debug logs or adjust security properties while instances are running. We found this to be critical when setting up containers. Debug mode can only be enabled before a container is frozen, though, and prevents the use of state assertions. When a container is frozen, running instances are terminated, as their configurations may have changed. Instances can then be (re-)launched.

Network isolation is built on OpenStack's tenant network. Each container group is put in a shared layer-2 network, and cloud containers have their own subnets. All these networks are fully managed by IaaS infrastructure, and cannot be modified administratively. We built traffic control in *Neutron*'s gateway to limit the aggregate bytes transferred using IPTable's *quota* module. We have not yet implemented bandwidth limits.

CQSTR provides containers with private volume storage using Cinder. Volumes that are used inside a cloud container are marked as container-local unless they are explicitly allowed to escape as listed in the manifest. Local volumes and their copies/snapshots cannot be used outside the container.

The Swift object store lacks a few features of AWS S3, such as VPC endpoints to restrict the set of accessible buckets. To control storage access, we built a storage agent inside the Swift server that implements a private namespace for cloud containers. Object accesses are directed to this namespace unless the bucket name is explicitly listed on the container manifest. The namespace is derived from the tenant account ID and the container ID. This design allows containers in a container group to share objects through Swift without exposing them outside the container. We use this feature in Prediction IO (see Section 6).

5.2 Attestation-based Access Control on Sharing

We extend the OpenStack *Metadata Service* to provide state assertions about cloud containers. A tenant using a cloud container can explicitly grant access to designated services for specific metadata properties. A client can contact the metadata service with a container ID or its IP address for state assertions about a service before using it.

CQSTR also introduces attestation-based access control, which allows embedding state assertions on resources such as Swift buckets: only services meeting required assertions can access the resource. Here, the resource owner (e.g., client of a service) creates a role, grants it the desired access, and specifies the required state assertions, which can be either a container ID or a manifest template and parameter. Upon receiving a request, the resource server (e.g., Swift) consults the metadata service for the information about both the requesting instance and its cloud container. Only if the requester meets the security requirements is access granted. We modified OpenStack to support dynamic creation and sharing of such roles (similar to what AWS's IAM mechanism already provides).

These requests to the metadata service for state assertions provide an IP address for the service. As IaaS networks prevent spoofing, the IP address identifies the client of a resource server. To ensure uniqueness, requests to a resource server must use a globally unique IP address. These addresses can float between instances, so CQSTR imposes a configurable waiting period on reusing these addresses so that assertions can be cached.

A tenant can implement their own storage service that uses CQSTR services. We implemented a library for external services to request and verify container manifests, so that any tenant-implemented service can enforce the same types of policies as platform services.

5.3 API Restrictions

We implement API restrictions by adding hooks in existing OpenStack services to change behavior when cloud containers are in use. We modified storage APIs to implement mandatory access controls (i.e., private volumes or storage objects cannot be made public) by setting initial ACLs on objects or volumes created by a container to prevent access from outside the container. OpenStack allows configuring a VPN with an "allowed address pair" that bypasses network security, so we disable the APIs for that feature on instances in cloud containers.

We also patched many management APIs to prevent side channels. As an example, we prevent snapshots on instances in containers, prevent moving IP addresses between instances, and prohibit most calls to management APIs from within a container. We strove to limit covert channels that would enable malicious instances within a container to violate bandwidth limitations. A few such covert channels still exist, such as the previously mentioned billing example.

6. Application Case Studies

We have applied CQSTR to three real applications: SpamAssassin [13], a spam filter service; PacketPig [11], a network trace analyzer; and PredictionIO [12], a machine learning service. For each application, we describe how we configure the container to maximally restrict the service from releasing data. In this section, an operator is the one who sets up the target containers, and a client is the one who receives the output data from the service. Note that a client may or may not be the data owner, and a client can also be an operator in some application scenarios.

6.1 Basic Pattern

The basic pattern for deploying a service with CQSTR is to identify the inputs, outputs, and storage requirements, and instantiate a cloud container that restricts networking and storage to just what is required. Two of the applications we investigated follow a simple pattern we call "single output, local storage," meaning that they produce output to a single endpoint, and require local storage for intermediate results. The manifest template for this pattern is shown Figure 3. Note in this context, the word "public" means external to the container, rather than public to the world.

This template enables the service to receive connections by network hosts parameter1 on TCP port parameter2, which can be another container or a specific address, and to read and write private local storage and read public read-only data from outside the container. This template does not restrict which images can be used to boot instances: the

```
network:
  public: <parameter1>,
  direction: ingress,
  protocol: tcp,
  local_port: <parameter2>

volume:
  ro: public,
  rw: container

objectstore:
  default: local

image:
  any
```

Figure 3. Basic Container Pattern

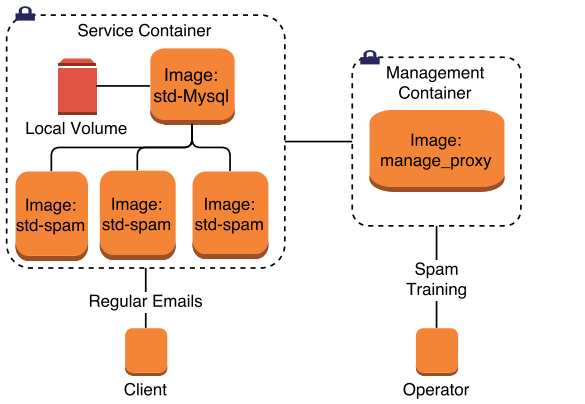

Figure 4. Configuration for SpamAssassin. Image name with prefix "std-" means unmodified applications and OS from official sources. Other images contain code that implements management functions.

output and input are the same endpoint, so there is no need to trust the service to make delegated access control decisions. These restrictions are a limited form of mandatory access control over the data passed to the target cloud container. Once a contained service is launched, the service owner cannot change these controls, so data inside will always be contained as the manifest describes.

We will extend this basic pattern with additional containers in order to implement richer multi-container functionalities. With CQSTR alone at least one separate container is needed for each customer, which can be inefficient for lightweight services. But, CQSTR can act as a platform for building higher layers of trust, so finer-grained containment is possible. For example, a trusted OS running within CQSTR could isolate instances of a service with containers.

6.2 SpamAssassin

SpamAssassin has a simple work flow: a client sends the service a stream of emails, and the service scans each email for spam. It adds an email header indicating the likelihood that it is spam and then returns the message back to the client. Our security requirement is that SpamAssassin should not disclose emails to other parties, including the service's operator.

This workflow fits the pattern shown in Figure 3, with the client's address as parameter1. The client of SpamAssassin queries the metadata server with the service's address, and verifies that the returned manifest uses the template with the correct parameters.

We extend this basic design to allow the operator to pass labeled training data to the service enabling it to learn additional spam patterns. Here, we ensure that the act of providing training sets does not leak any information: even TCP ack packets from the service could be used to leak client email content.

We solve this by extending SpamAssassin with a second container including a single *management proxy*. A management proxy is a small VM instance that contains only trusted code and provides a limited set of management operations to an isolated service, much narrower than the service interface. This proxy is comparable to AWS API Gateways [21]. Each proxy in our deployments use a slightly modified Ubuntu 14.10 installation that includes a script to export desired management functionality through an HTTP interface. The functionality of a proxy should be simple and open source, so that it can be trusted.

The SpamAssassin proxy allows a single management operation, which is to accept training data from the operator. The service passes the data to SpamAssassin, and returns a fixed acknowledgment to the operator. It runs in a separate container from the service, so the proxy cannot access or return any of the client's email to the operator. The proxy's container allows access to the operator and to the service's container, and we allow the service's container to communicate with the proxy, as shown in Figure 4. Because this proxy could be a channel to leak information, we require a locked-down trusted image for the proxy.

6.3 PacketPig

PacketPig is a distributed network trace analysis tool [11] built on Apache Pig [25] and Hadoop. We use CQSTR to allow untrusted users to submit scripts that analyze sensitive network traces stored in a Swift bucket by ensuring that the service cannot reveal much about the trace. Unlike SpamAssassin, the data is not owned by the client; instead this is a delegated setting and CQSTR is used to limit the information clients extract rather than prevent release completely.

Our basic configuration follows the pattern described above: PacketPig runs in a container that only allows access to a single port to submit scripts and receive results. To limit the amount of data released, we enforce a total download limit on the container that is a small fraction of the total trace size so the complete trace cannot be exposed. The trace data is stored in a Swift bucket. It is not possible to directly reference a Swift bucket without modifying the ap-

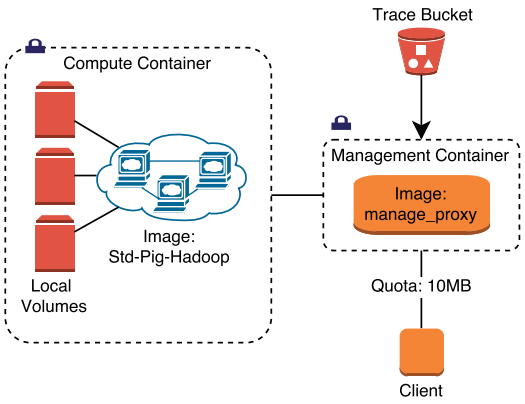

Figure 5. Configuration for PacketPig.

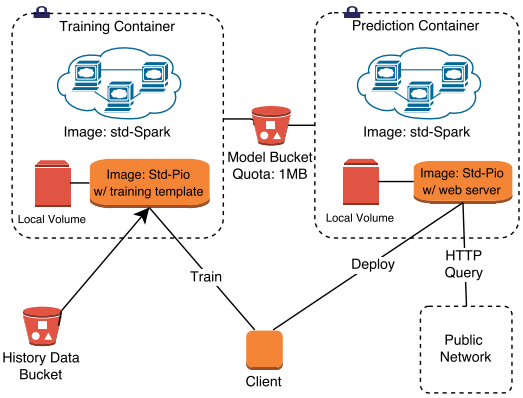

Figure 6. Configuration for PredictionIO.

plication. Instead, we integrate the data importing function into a management proxy described below. The data owner sets an ACL on a trace bucket that only grants access to a role from the management container.

We implemented a management proxy that, in addition to providing network traces, allows dynamic addition and removal of nodes in the service, and modifies query scripts to add random noise. PacketPig can only communicate with the proxy, and clients submit scripts and receive result from the proxy. This configuration is shown in Figure 5. The set of nodes is changed by providing the address of the new instance to add or the existing instance to shut down; the proxy modifies Hadoop's configuration and restarts the cluster with the new configuration. The script rewriter illustrates use of a proxy to sanitize inputs and outputs for better security and privacy. It adds uniformly distributed noise to the *SUM* function in scripts, with respect to the average and variance of the data. More complex mechanisms like differential privacy could be implemented [30], but are out of scope for this paper. This use of a proxy demonstrates the more general idea of an *output proxy* that checks or sanitizes results before returning them to the client.

This configuration can also be used more generally as a contained Hadoop/Pig cluster, where clients submit jobs without worrying that the service owner may leak scripts or data. The only modification to the above design is to change the management to allow all scripts.

PacketPig demonstrates how CQSTR can restrict and isolate data flows. However, service-generated data, such as program outputs and logs, might still carry sensitive information. Hence, in deployment one would need to carefully whitelist which scripts and supporting service code can be allowed to run within the cloud container.

6.4 PredictionIO

PredictionIO is an open-source prediction service, and demonstrates how CQSTR solves the example problem in Section 2. PredictionIO reads a client's data, with which it trains a model based on a code template from the client. With the model, PredictionIO deploys a webserver to make online predictions. Our security goal is that the training service does not leak the client's data; the data owner allows the PredictionIO service to train on its data and make public predictions from the resulting model but the training data must stay private.

We created an image using the standard Prediction IO codebase with a trusted prediction template. We install this on a locked-down configuration of Linux with unnecessary services (including sshd) disabled. In addition to the PredictionIO code, the service also runs a standard Apache Spark cluster [24] to provide the compute power needed to generate the prediction model.

As shown in Figure 6, we deploy the training service in a cloud container with access to the client's data source. The web server for prediction queries runs in a separate container that shares a Swift bucket with the training container, and allows public access. We limit the quota of data that can be written into the bucket to the model's estimated size (1MB).

This configuration ensures that the training data is limited to the training service, and the publicly accessible web server has no direct access to the data. Even if the web service container is compromised due to an application or OS vulnerability, the attacker cannot access the training data.

Our deployment of Prediction IO with CQSTR demonstrates how an application can be decomposed into components, each in a separate container. Using a similar decomposition with rules allowing access to specific endpoints, an application can contact trusted third-party services. If they are deployed in cloud containers, CQSTR can also verify their trustworthiness.

6.5 Experience

Porting the three applications to operate in CQSTR was straightforward, requiring no source modification. The management proxies are small, simple services to code. We ran into a few issues with services that assume open Internet access: SpamAssassin looks up the hostname in all messages using DNS, but access to DNS can leak information

about what messages are being scanned. Some prediction templates for PredictionIO access external services, which may not be allowed in a cloud container. In both cases, we disabled code that accessed external services without hindering the desired functionality.

7. Performance Evaluation

We evaluate the performance cost of CQSTR's modifications to OpenStack on our three applications.

7.1 Configuration

We run CQSTR with nine physical hosts on the Wisconsin cluster of CloudLab [1]. One node is the cloud controller, one is the network gateway, two serve as both volume and object storage, and five are compute servers.

All nodes have two 8-core 2.40 GHz Intel E5-2630 CPUs with hyperthreading enabled and 128GB memory. The code is stored on a local SAS disk, while storage services (Swift and Cinder) use an SSD. User VMs are connected through a GRE tunnel over 10Gb Ethernet, with a separate 10Gb Ethernet for storage requests. We multiplex the management network with the data network, with address spaces isolated.

We used Ubuntu 14.04 LTS image for guests and services. Both guests and services run in m1.large instances, which have 4 virtual CPUs without usage caps, 8GB memory and 80GB file-backed storage on a SAS disk. Management proxies are lightweight and run in m1.small instances with 1 VCPU and 2GB memory.

7.2 Application Performance

Table 2 shows the workloads and average completion time over 10 repetitions for PredictionIO, PacketPig, and SpamAssassin. The Email dataset is the EDRM Email DataSet V2 [2], the network trace comes from ISTS12 [5], and both training and predictions use data from the Million Song Dataset [27]. All results have standard deviations below 3%; the prediction workload has a 1% deviation.

The performance of PredictionIO predicting and PacketPig on CQSTR are 1.5% slower than the native versions. The latency of predictions is unchanged. For SpamAssassin CQSTR is 1% slower.

7.3 Microbenchmarks

We created a set of microbenchmarks to better understand why application performances was unchanged.

Container management. We measured the cost of management operations to create and launch a service in a container. Creating a container takes 1.5 seconds and configuring its security properties takes 3 seconds, which is a fraction of the time to instantiate and boot an instance (11 seconds).

Existing management APIs. CQSTR imposes additional checks on existing IaaS management APIs, e.g., attaching a volume or IP address to an instance. In measurements, we found these checks add less than 2% overhead.

Network performance. CQSTR adds extra firewall and traffic rules compared to normal instances. Using iperf to test bandwidth and latency, we found no measurable difference.

Storage performance. For volume storage, additional checks are only imposed when a volume is attached, so there is no performance impact on data access. For object storage, Swift must call the metadata service with the requester's IP address to verify state assertions, which takes 30ms on average. The result is cached for the minimum IP address reuse time (10 minutes). In our experiments, the average latency of object storage requests is unchanged at 12ms, but the 99th percentile latency increases from 250ms to 750ms, reflecting the time spent verifying the requester's capabilities. Bandwidth is unchanged.

External storage performance. External services must fetch state assertions before granting access to data, which takes approximately 700ms. However, the result of checking assertions can be cached as with Swift, and hence has little impact on overall performance.

These results explain why application performance is largely unchanged: containment operations rely on the existing behavior of IaaS services, or are cacheable, so there is effectively no performance overhead at runtime.

8. Related Work

A large number of works seek to improve control over sensitive data. One solution for the data owners to run analysis code only in instances (VMs) under their direct control. Mittal et al. [43] describe such an approach for packet trace analysis. It precludes instances that run third-party proprietary code or are managed by any party other than the data owner; CQSTR allows both. Secure multiparty computation [38] and fully homomorphic encryption [34] are promising solutions for data analytics, but are not yet general and practical enough for most computing problems.

Another approach is to build limited interfaces to code that is trusted not to leak data. Bunker [42] creates a closed-box for the specific case of network trace analysis on a single node. Airavat [46] ensures privacy-preserving outputs for MapReduce workloads by confining untrusted mappers and running only trusted reducers. PINQ [41] implements differential privacy mechanisms based on C# LINQ [6], and provides a restricted programming language to compute on sensitive information. These works are complementary to CQSTR: one could for example set up the trusted code (e.g., Bunker, Airavat, or PINQ) in a CQSTR cloud container, and obtain the benefits of both systems. Here the role of CQSTR is to attest to clients of the service that it runs the correct (trusted) code and is properly locked down. Brown and Chase [29] proposed a similar idea for a PaaS service that attests to the code running in an instance.

The cloud container abstraction can be viewed as a limited form of mandatory access control using Information Flow Control, including Decentralized IFC (DIFC [44]). IFC

Application	Workload	Native	CQSTR	% diff.
SpamAssassin	Filter 100,000 emails	598 sec	604 sec	1.0%
PacketPig	Fingerprint 146GB trace	856 sec	869 sec	1.5%
PredictionIO Training	Train on 10,000 songs (1.8GB)	26 sec	27 sec	3.1%
PredictionIO Prediction	Perform 1,000,000 predictions	914 sec	925 sec	1.2%

Table 2. Application workloads and execution time.

systems track labels on data (a program's inputs "taint" outputs they may have affected) and confine untrusted code by blocking unsafe data flows to incompatible channels. DIFC systems also provide a natural means to grant specific authority to trusted code, e.g., to declassify its output data by changing the label. Examples include Airavat and OS-level systems including Histar [58], Flume [37], and Asbestos [31]. DIFC has also been implemented for a distributed environment (e.g., [59]). CQSTR provides a less flexible but simpler abstraction that meets the goals of specific uses of IFC (access control based on code identity, confinement of untrusted code) without the complexities and overheads of data labeling, taint tracking, and label-based access checks. At the IaaS layer what we seek is to provide a practical and deployable subset of IFC's power and flexibility—strong containment and attestation—as a foundation for safer data sharing among tenants and richer trust mechanisms (such as IFC) at higher layers. Because CQSTR applies checks at the cloud container boundary, it is independent of the guest operating systems and applications, and extends easily to containers with many instances: CQSTR can attest that all instances within a cloud container are running IFC software trusted by the client, to obtain strong defense-in-depth and controlled data release.

CQSTR is complementary to trusted computing mechanisms (e.g., TPM, TXT) that enable trusted execution based on a hardware root of trust [32, 40, 47, 48, 52, 54]. CQSTR is similar to these in that it attests code identity and allows clients to verify that a service is running the proper version of code in a proper configuration. However, in CQSTR the IaaS cloud provider is the root of trust. We believe that this trust assumption is reasonable and matches the practical realities of cloud computing today; the goal of our work is to leverage this trust to bootstrap richer security controls for safe cross-tenant data sharing in large-scale cloud services [33], and not just in individual computers. In principle, future advances in trusted execution technology could extend the trust chain down to the hardware by attesting the trusted cloud stack itself.

Some systems advocate property-based attestation [47]. These are more convenient compared to the above TPM-style interfaces. For example, Santos [49] builds a trusted-computing-based runtime that seals data from malicious cloud operators based on instance properties. However, these do not protect one tenant from another.

To further protect trusted software, Intel's SGX extension creates hardware-isolated environments (enclaves) for trusted code and private data [4]. VC3 [51] and Haven [26] use SGX to ensure private data is not released to untrusted components (i.e., the programming framework, hypervisor or OS). Enclaves enable protection against untrusted cloud providers [50, 53]. These works view the cloud provider as a threat to the guest rather than a root of trust that can facilitate secure sharing among mutually distrusting tenants. But, the security assumption behind these designs may be too restrictive for cross-tenant services. Without IaaS-level network restrictions, all enclaves in a service must be trusted to not leak information. We see SGX-based enclaves as complementary to the cloud container abstraction, where enclaves are used for code confidentiality, and cloud containers are used to organize and enforce the high-level properties of a service.

Existing cloud providers offer rich authentication and authorization to protect user data. AWS controls access based on users, groups, and roles, (IAM [14]) and can assign a role to an instance based on its boot image. CQSTR adds new container-level mechanisms to control access by instances (rather than by users) based on their code identity and/or configuration, and to block data leakage even after access is granted.

Platforms-as-a-Service(PaaS) also provide containment: Heroku recently announced its private application namespace [3]. CQSTR can be seen as moving this viewpoint to the IaaS level. As with other application frameworks, CQSTR can also be used to establish trust in layered PaaS systems and PaaS applications.

9. Conclusion

Controlling use of data is central to secure computing. CQSTR is the first system to leverage existing IaaS infrastructure to provide control over data usage to clients of a service, and to do so without TPMs or cryptographic protocols. CQSTR is directly applicable to a wide variety of data-analysis applications, and can also be used as a general application-structuring technique. It provides a basis for OS and application-level mechanisms to further confine code. However, CQSTR currently works for a single provider, as it relies on IP addresses for authentication. How to extend across multiple cloud platforms remains an open question.

Acknowledgements

This work was supported in part by NSF grant CNS-1330308 and CNS-1330659.

References

[1] Cloudlab. https://www.cloudlab.us.

[2] Email data set. https://aws.amazon.com/datasets/917205.

[3] Heroku private namespace. http://blog.heroku.com/archives/2015/9/10/heroku_private_spaces_private_paas_delivered_as_a_service.

[4] Intel sgx. https://software.intel.com/en-us/isa-extensions/intel-sgx.

[5] Ists competition 12. http://www.netresec.com/?page=ISTS.

[6] Linq (language-integrated query). https://msdn.microsoft.com/en-us/library/bb397926.aspx.

[7] Linux containers. https://linuxcontainers.org/.

[8] Linux containers. http://linuxcontainers.org/.

[9] Openstack. http://www.OpenStack.org/.

[10] Openstack trusted computing pool. https://wiki.OpenStack.org/wiki/TrustedComputingPools.

[11] Packetpig. http://blog.packetloop.com/search/label/packetpig.

[12] Prediction io. http://prediction.io.

[13] Spamassassin. http://spamassassin.apache.org/.

[14] Amazon Web Services. Amazon web services: Overview of security processes, 2014. available at http://www.utdallas.edu/~muratk/courses/cloud11f_files/AWS_Security_Whitepaper.pdf.

[15] I. Amazon Web Services. Aws identity and access management (iam). https://aws.amazon.com/iam/.

[16] Amazon Web Services, Inc. Amazon cloudtrail. https://aws.amazon.com/cloudtrail/.

[17] Amazon Web Services, Inc. Amazon cloudwatch. https://aws.amazon.com/cloudwatch/.

[18] Amazon Web Services, Inc. Amazon glacier. https://aws.amazon.com/glacier/.

[19] Amazon Web Services, Inc. Amazon machine learning. https://aws.amazon.com/machine-learning/.

[20] Amazon Web Services, Inc. Amazon virtual private cloud. https://aws.amazon.com/vpc/.

[21] Amazon Web Services, Inc. AWS api gateway. https://aws.amazon.com/api-gateway/.

[22] Amazon Web Services, Inc. AWS config. https://aws.amazon.com/config/.

[23] Amazon Web Services, Inc. AWS vpc endpoint. http://docs.aws.amazon.com/AmazonVPC/latest/UserGuide/vpc-endpoints.html.

[24] Apache Foundation. Spark. https://spark.apache.org/.

[25] Apache Foundation. Welcome to apache pig. https://pig.apache.org/.

[26] A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with haven. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[27] T. Bertin-Mahieux, D. P. Ellis, B. Whitman, and P. Lamere. The million song dataset. In *Proceedings of the 12th International Conference on Music Information Retrieval (ISMIR 2011)*, 2011.

[28] N. Borisov, R. Johnson, N. Sastry, and D. Wagner. Fixing races for fun and profit: How to abuse atime. In *Proceedings of the 14th Conference on USENIX Security Symposium*, pages 20–20, 2005.

[29] A. Brown and J. S. Chase. Trusted platform-as-a-service: a foundation for trustworthy cloud-hosted applications. In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, pages 15–20. ACM, 2011.

[30] C. Dwork, F. McSherry, K. Nissim, and A. Smith. Calibrating noise to sensitivity in private data analysis. In *Proceedings of the Third Conference on Theory of Cryptography*, pages 265–284, 2006.

[31] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazieres, F. Kaashoek, and R. Morris. Labels and event processes in the asbestos operating system. *ACM SIGOPS Operating Systems Review*, 39(5):17–30, 2005.

[32] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proc. Symposium on Operating Systems Principles*, 2003.

[33] R. Geambasu, S. D. Gribble, and H. M. Levy. Cloudviews: Communal data sharing in public clouds. In *Proceedings of the 2009 Conference on Hot Topics in Cloud Computing*, 2009.

[34] C. Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing*, STOC '09, pages 169–178, New York, NY, USA, 2009. ACM.

[35] Google, Inc. Google prediction api. https://cloud.google.com/prediction/.

[36] K. He, A. Fisher, L. Wang, A. Gember, A. Akella, and T. Ristenpart. Next stop, the cloud: Understanding modern web service deployment in ec2 and azure. In *Proceedings of the 2013 conference on Internet measurement conference*, pages 177–190. ACM, 2013.

[37] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard os abstractions. *ACM SIGOPS Operating Systems Review*, 41(6):321–334, 2007.

[38] Y. Lindell and B. Pinkas. Secure multiparty computation for privacy-preserving data mining. *Journal of Privacy and Confidentiality*, 1(1):5, 2009.

[39] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-level cache side-channel attacks are practical. In *IEEE Symposium on Security and Privacy*, pages 605–622, 2015.

[40] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for tcb min-

imization. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, pages 315–328, 2008.

[41] F. Mcsherry. Privacy integrated queries: an extensible platform for privacy-preserving data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, pages 19–30. acm, 2009.

[42] A. G. Miklas, S. Saroiu, A. Wolman, and A. D. Brown. Bunker: A privacy-oriented platform for network tracing. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, pages 29–42, 2009.

[43] P. Mittal, V. Paxson, R. Sommer, and M. Winterrowd. Securing mediated trace access using black-box permutation analysis. In *HotNets*. Citeseer, 2009.

[44] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *Proc. Symposium on Operating Systems Principles (SOSP)*, 1997.

[45] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 199–212. ACM, 2009.

[46] I. Roy, S. T. Setty, A. Kilzer, V. Shmatikov, and E. Witchel. Airavat: Security and privacy for mapreduce. In *Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation*, pages 297–312, 2010.

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

[48] R. Sailer, T. Jaeger, X. Zhang, and L. Van Doorn. Attestation-based policy enforcement for remote access. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 308–317. ACM, 2004.

[49] N. Santos, R. Rodrigues, K. P. Gummadi, and S. Saroiu. Policy-sealed data: A new abstraction for building trusted cloud services. In *USENIX Security Symposium*, pages 175–188, 2012.

[50] J. Schiffman, Y. Sun, H. Vijayakumar, and T. Jaeger. Cloud verifier: Verifiable auditing service for iaas clouds. In *IEEE Ninth World Congress on Services*, pages 239–246, 2013.

[51] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. Vc3: Trustworthy data analytics in the cloud using sgx. In *36th IEEE Symposium on Security and Privacy*, May 2015.

[52] E. Shi, A. Perrig, and L. Van Doorn. Bind: A fine-grained attestation service for secure distributed systems. In *2005 IEEE Symposium on Security and Privacy*, pages 154–168, 2005.

[53] R. Sinha, S. Rajamani, S. A. Seshia, and K. Vaswani. Moat: Verifying confidentiality of enclave programs. In *The ACM Conference on Computer and Communications Security (CCS)*, October 2015.

[54] E. G. Sirer, W. de Bruijn, P. Reynolds, A. Shieh, K. Walsh, D. Williams, and F. B. Schneider. Logical attestation: an authorization architecture for trustworthy computing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 249–264, 2011.

[55] V. Varadarajan, T. Kooburat, B. Farley, T. Ristenpart, and M. M. Swift. Resource-freeing attacks: improve your cloud performance (at your neighbor's expense). In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 281–292. ACM, 2012.

[56] L. Wang, A. Nappa, J. Caballero, T. Ristenpart, and A. Akella. Whowas: A platform for measuring web deployments on iaas clouds. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, pages 101–114. ACM, 2014.

[57] Y. Yarom and K. Falkner. Flush+ reload: a high resolution, low noise, l3 cache side-channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732, 2014.

[58] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *Proc. 7th USENIX Symposium on Operating Systems Design and Implementation*, pages 263–278, 2006.

[59] N. Zeldovich, S. Boyd-Wickizer, and D. Mazieres. Securing distributed systems with information flow control. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, volume 8, pages 293–308, 2008.

[60] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-vm side channels and their use to extract private keys. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 305–316. ACM, 2012.