Cloud Verifier: Verifiable Auditing Service for IaaS Clouds

Joshua Schiffman
Security Architecture R&D
Advanced Micro Devices. Inc,
Austin, TX, USA
Josh.Schiffman@amd.com

Yuqiong Sun, Hayawardh Vijayakumar, and Trent Jaeger
Department of Computer Science and Engineering
Pennsylvania State University
University Park, PA, USA
yus138,hvijay,tjaeger@cse.psu.edu

***Abstract*—Cloud computing has commoditized compute, storage, and networking resources creating an on-demand utility. Despite the attractiveness of this new paradigm, its adoption has been stymied by cloud platform's lack of transparency, which leaves customers unsure if their sensitive data and computation can be entrusted to the cloud. While techniques like encryption can protect customers' data at rest, clouds still lack mechanisms for customers to verify that their computations are being executed as expected, a guarantee one could obtain if they were running the computation in their own data center.**

In this paper, we present the *cloud verifier* (CV), a flexible framework that cloud vendors can configure to provide cloud monitoring services for customers to validate that their computations are configured and being run as expected in Infrastructure as a Service (IaaS) clouds. The CV builds a chain of trust from the customer to their hosted virtual machine (VM) instances through the cloud platform, enabling it to check customer-specified requirements against a comprehensive view of both the VM's load-time and run-time properties. In addition, the CV enables cloud vendors to provide more responsive remediation techniques than traditional attestation mechanisms. We built a proof of concept CV for the OpenStack cloud platform whose evaluation demonstrates that a single CV enables over 20,000 simultaneous customers to verify numerous properties with little impact on cloud application performance. As a result, the CV gives cloud customers a low-overhead method for assuring that their instances are running according to their requirements.

***Keywords*-cloud; integrity; monitoring;**

I. Introduction

Cloud computing has revolutionized they way we consume computing resources. Instead of maintaining a locally administered data center, resources can be obtained on demand from a public cloud utility. Clouds come in a variety of models ranging from virtual machine (VM) hosting to fully managed web services. For example, Infrastructure as a Service (IaaS) clouds like Amazon's EC2 [1] and Rackspace Cloud Servers [2] provide fully customizable virtualized computing infrastructures.

While this new model has increased access to affordable resources, cloud computing makes it more difficult for customers to track their own computations on *cloud instances*. When customers deploy computations in their own data centers, they have a variety of tools for monitoring the health of those computations [3], [4]. However, by using remotely-administered cloud systems, cloud customers are no longer in able to maintain *visibility* into their computing infrastructure. Little assurance is provided that the security settings they specify are properly enforced or that their instances are protected from accidental misconfiguration, malicious insiders [5], and traditional external threats. The problem is only compounded when considering the risks that clients of these customers' cloud-hosted services must blindly accept when trusting both the cloud and the customers (service providers) with their sensitive data.

Researchers have explored methods that enable cloud customers to obtain greater control over the conditions under which they will run their instances on a particular cloud. The *Excalibur system* [6], for example, allows customers to select the attributes of cloud platforms that must be met before releasing their data to be run on those clouds. However, it only focuses on the platform, so customers cannot validate whether their instance was launched as expected nor evaluate runtime properties of the instance. *Self-service clouds* [7] (SSCs) extend the cloud infrastructure by providing per-customer administrative domains, enabling customers to manage their own instances. However, the SSC approach wrests too much control away from the cloud vendors without providing enough information to the customers. Using SSC, the cloud vendor is forced to reveal a great deal of information about its own, possibly confidential cloud node configuration to the customers, and the customer does not obtain any justification for the rest of the cloud services upon which the instance depends, such as the cloud controllers, which have had vulnerabilities reported [8].

In this paper, we present a framework for cloud vendors to build auditing services that enable customers to obtain a verifiable chain of trust to a monitor that tracks the runtime health of their cloud instances. The framework consists of two parts. First, the *Cloud Verifier* (CV) service enables cloud vendors to monitor the health of their own clouds, produce proofs of cloud health for their customers, and build verifiable chains of trust between customers and their instances. Cloud vendors configure CVs to leverage the cloud's hierarchical structure to build transitive trust from the cloud platform to the instances themselves, which customers

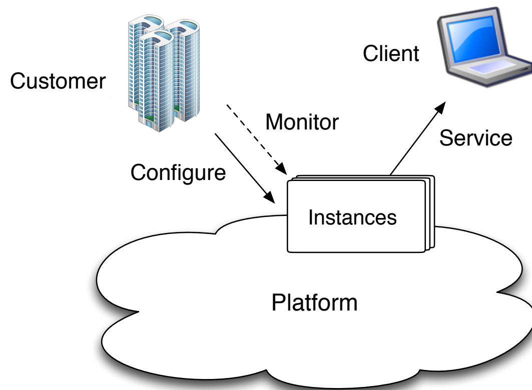

Figure 1. Host-based Monitoring.

can then verify to obtain visibility to their instances. Second, the *Instance Monitor* (IM) service monitors each instance's state to detect violations from its customer's requirements, which we call *integrity criteria*[1]. Like the SSC approach [7], we find that by verifying integrity requirements from within the cloud instead of remotely, more comprehensive and efficient enforcement can be achieved. Unlike the SSC approach, we use the CV to build a chain of trust from the customer to the IM service to monitor instances. In addition, cloud vendors can provide monitoring modules for IM services to measure various load-time and runtime properties of customer instances, enabling customers to write integrity criteria in terms of queries on those measurements. The IM service is designed so that customers can take remediative actions as soon as any integrity criteria are violated to prevent themselves or clients of their services from using data from potentially compromised instances or sending data to those instances.

The rest of the paper is organized as follows: Section II describes the problems of instance monitoring in IaaS clouds. Section III introduces two techniques our monitoring framework builds upon. We present the design overview of our Cloud Verifier framework and its integration with OpenStack in Sections IV and V, respectively. Section VI evaluates the framework. Section VII discusses related work and Section VIII concludes the paper.

II. Problem Definition

The goal of host-based monitoring is to evaluate properties of an individual host, such as its performance, resource usage, and integrity. Figure 1 shows the general approach for host-based monitoring that we will use in this paper. First, a *host instance* is deployed in the context of a *platform*, such as a data center or cloud, run by a *platform owner* on behalf of a *customer*. The host instance deployed by the customer may, in turn, be used by *clients*, who utilize the services the customers provide through their instances. *Host-based monitors* inspect the instance throughout its lifetime, from its load time through the duration of its execution. Traditionally, host-based monitoring mechanisms consist of a daemon running on the host instance itself and a monitoring server that runs within the platform, protected from the host instance, that retrieves input from the monitoring daemon. Commercial monitoring products utilize this architecture [3], [4].

[1]Integrity criteria may express a broad range of requirements that customers may want to monitor and enforce upon their cloud instances.

A. Host-Based Monitoring

The problem with this approach is that the host instance may not be fully trusted, such that the host instance may tamper with the monitoring daemon to change the results it reports. This is particularly a concern when the daemon's job is to monitor host integrity. If an adversary is able to compromise the host, she may be capable of compromising the monitoring daemon as well to report falsely that the host instance is operating as expected.

To counteract this problem, researchers have explored monitor designs that remove the monitor daemon from the host instance, yet are still able to provide the desired monitoring information. One particular way is to utilize virtualization features of commodity (x86) processors. Many solutions [9], [10], [11] have been proposed since then for monitoring VM integrity via *virtual machine introspection*. The main challenge in VMI is to provide efficient measurement mechanisms that will monitor the properties of interest to the customer accurately from outside the host.

B. Host-Based Monitoring in IaaS Clouds

Though aforementioned host-based monitoring mechanisms have proven effective in traditional data centers, they face serious challenges when deployed in a cloud setting.

First, the trust relationship between customer and platform changes when the customer moves computation into the cloud. In traditional data centers, the customer is also the platform owner and the platform is visible and directly configured and administered by customer. Consequently, they can implicitly trust the monitoring services in the hosting platforms.

However, when moving to a cloud, the owner becomes a detached customer that no longer maintains such visibility. Because the cloud platform is configured by external administrators, threats may arise that prevent customers from accurately monitoring their instances. One significant concern, for example, is the fear that privileged insiders may access the cloud's management interface and corrupt a customer's computing environment. Several studies have shown that this internal threat does exist and current cloud platforms lack effective countermeasures to address it [12], [13].

Second, verification of platforms becomes much harder in the cloud, as cloud vendors usually hesitate to expose their infrastructure to outside parties. Customers are left unsure whether their monitor is functioning correctly and that the results returned are trustworthy. As administration of cloud infrastructure may be error-prone, it is likely that accidental mismanagement, such as an unpatched *cloud node* or a misconfigured cloud service, could result in an incorrect or even vulnerable monitoring facility that would render an incorrect view over customers' instances. Moreover, failing to provide verifiability could prevent adoption of cloud in many scenarios that require external auditing, such as bank services or public health databases.

Finally, host-based monitoring mechanisms in general have an inherent weakness because they only focus on collecting and reporting instance's state, relying on customer to later perform mediation based on the state. Consequently, a vulnerability window exists between the detection of an anomaly and the notification of customer, and during such a window the instance will run in an incorrect state. Such problems become even more severe in cloud, as remotely administered platforms inevitably prolong such anomaly notification delay. Moreover, clouds only export limited API to customers for controlling their instances, thus limiting the remediation options that are possible.

We generalize the above challenges into three basic requirements that motivate our design of a verifiable cloud monitoring service:

Correctness. The cloud monitoring service should gather and report properties of instances faithfully, despite the existence of cloud insiders, co-resident VMs, and other external threats.

Verifiability. Customers should be able to verify that the monitoring service is configured and running correctly.

Timely Control. As soon as an instance violates its integrity criteria, the cloud monitoring service blocks the violating instance from sending or receiving requests and initiates rememdiation.

III. Building Blocks

In this section, we introduce two techniques, Trusted Computing (TC) and Integrity Verification Proxy (IVP), that serve as building blocks for our verifiable cloud monitoring framework.

A. Trusted Computing

TC techniques enable systems to report their configurations (e.g., loaded code and data) to relying parties through a technique called *remote attestation*. The Trusted Platform Module (TPM) [14] is an example of a widely deployed TC co-processor available in many commodity systems like servers and laptops. The TPM possesses several *platform configuration registers (PCRs)* that store measurements (e.g., SHA1 hash chain values) of integrity-relevant events. Various measurement frameworks like the BIOS, LIM in the Linux kernel [15], and the Trousers userspace daemon [16] store sensitive events like code loading in the TPM's PCRs. These PCRs are append-only and reset at boot-time to prevent falsified measurements. The TPM also possesses an RSA key pair, called an *endorsement key (EK)*, that is burned into the chip to uniquely identify the physical machine. The EK certifies RSA *attestation identity keys (AIKs)* that are used to sign the PCRs to link the measurements to the platform.

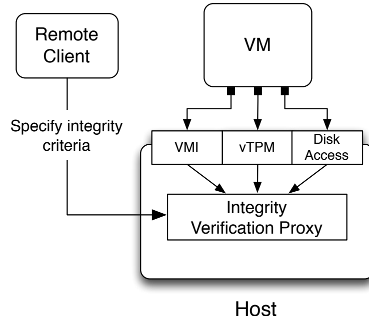

Figure 2. Integrity Verification Proxy Overview

Definition 1 (Quote): A proving system P reports its configuration as a signature over the TPM's current PCR values with a nonce N for freshness. Formally, $\text{QUOTE}_N = \{PCR||N\}_{AIK_P^{-1}}$.

Definition 2 (Attestation): Given a nonce N, an *attestation* is $\text{ATT}_N = \{\text{QUOTE}_N, M\}$, where M is the measurement list of events corresponding to the PCR values of QUOTE_N.

Researchers have leveraged trusted computing hardware to verify the *integrity* of remote systems [17]. While initial efforts aimed for loadtime verification of individual systems [18] and isolated execution environments [19], recent projects have focused on verifying distributed systems [20] and even cloud platforms [21]. However, such solutions are insufficient for providing a complete view of the cloud's integrity. In particular, their reliance on inefficient and incomplete attestation protocols limits their utility for monitoring cloud integrity [22].

B. Integrity Verification Proxy

Integrity Verification Proxy (IVP) [22] is a modular service that enforces integrity requirements over connections between clients and remote systems. It monitors any changes to remote systems that violate clients' integrity requirements and immediately terminates clients' connections. Figure 2 illustrates an overview of IVP. It is designed as a modular service that collects the evolving state of VMs it hosts through the available integrity measurement (IM) interfaces

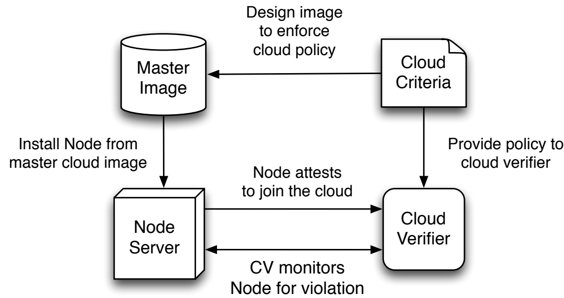

Figure 3. Cloud Join Cycle.

(e.g., VM introspection, vTPM, etc.) used for reporting integrity-relevant events within the VM. Such accumulated state is then used by the IVP to determine whether a client supplied integrity requirement, called as integrity criteria, is satisfied.

The advantage of leveraging IVP as the basis of our IM service is twofold. First, IVP resides in the VM host thus always has the most up-to-date view with which to assess VM state, by continually gathering information about the VM's changing configuration. Second, IVP can be leveraged to perform timely remediation. Current monitoring approaches rely on the remote party to make remediation decisions when a violation is detected by the monitor. This often introduces windows where malicious or undesirable behavior may occur because the remote party is notified too late. IVP can enforce integrity criteria at VM host, consequently performing remediation right upon detection of anomalies.

Designing an IM service based on the IVP and incorporating it into a cloud platform incurs many challenges. First, clouds are opaque to customers. The IVP design assumes that customers can find and verify that a host is running an IVP. In cloud platforms, customers often cannot identify the specific cloud nodes on which their instances run nor can they communicate directly with the host software of the node to determine if it runs a valid IM service (e.g., IVP). Second, IVPs only focus on the monitoring of a VM on a host platform. On the other hand, cloud platforms are complex, comprised of multiple services (e.g. network service, API service, etc.) that support the execution of instances. Merely verifying the compute service does not provide sufficient guarantees that the IM service will monitor customers' requirements comprehensively.

IV. Cloud Verification Architecture Overview

The goal of the CV is to construct a verifiable cloud monitoring service. That is, we want to provide customers with a trustworthy view of their running instances in a manner that the correctness of the services can be verified by customers themselves or third parties. To that end, the CV is designed to enable customers to establish trust through a cloud platform to the IM service that will monitor and enforce customers' requirements.

Enforcing Cloud Platform Integrity. Enabling remote customers to monitor their instances through a hosting cloud platform requires the cloud platform to function correctly on its behalf. That is, the monitoring service should faithfully report the state of instance, despite potential existence of cloud insiders, mismanagement and possible network attacks within cloud infrastructure. To that end, the CV ensures that all cloud components satisfy an integrity criteria specified by the cloud administrator, which customers can then compare to their own requirements. The CV does this by verifying services like the node controllers, network controllers, and API endpoints before they can join the cloud and ejects systems that violate the cloud-wide criteria.

Figure 3 illustrates the high-level protocol for a generic IaaS cloud node. First, the cloud administrators formulate an integrity criteria for a trustworthy cloud node. Next, the administrators design a master disk image for all nodes in the cloud that satisfies the criteria. The disk image is then pushed out to all of the node machines through network installation. When a node boots, it must request to "join" the cloud in order to host instances. To do this, nodes send attestations [23] to the CV, which evaluates the node against the cloud criteria. If successful, the node joins the cloud, and the CV monitors the node for changes, "ejecting" the cloud node if it violates the cloud criteria.

This protocol ensures all monitored components belonging to cloud platform currently satisfy the cloud criteria. As a result, the CV effectively speaks for the integrity of the cloud platform. A remote customer can then assess the integrity of the cloud by verifying the CV is trustworthy and then comparing the enforced criteria to the customer's requirements[2]. If they are acceptable, the client monitors the CV's runtime integrity and ensure it continues to speak accurately for the cloud platform. If the CV later violates the customer's requirements, then the customer can no longer trust the CV and must discontinue using it.

Monitoring Instance. The second layer of the CV framework provides a monitoring service over cloud instances on behalf of remote customers. This is done through the IM service resident in cloud node. Unlike the Excalibur system [6] which mainly focuses on measuring host properties such as hypervisor version, the IM service hooks into the instance control interface of cloud node, enabling it to measure load-time properties of instance, and attaches itself to a running instance through the VMI interface to collect runtime properties as well.

Remediation. In addition to verifying the integrity of the cloud platform and monitoring running instances, the CV enables remote customers to perform remediation in response to violations of their criteria. However, the degree

[2]The cloud vendors can choose what requirements about their cloud can be verified by their customers to balance secrecy of cloud configurations with customer-relevant requirements.

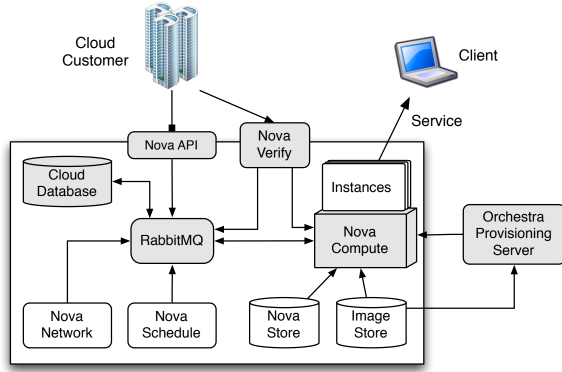

Figure 4. Integrating the CV Approach into OpenStack. The shaded elements required modifications.

of remediation is dependent on the authority of each remote party over the monitored component. For the cloud administrator, when the CV detects violating cloud components, it protects the cloud by ejecting that component from the cloud. This involves re-imaging the node back to the master disk image. For the cloud customers like service providers that host instances on the cloud, when the IM service detects a violating instance, it can shutdown the image and rollback the attached storage volume to a snapshot before the VM started. Finally, the IM service automatically disconnects clients from cloud instances that violate their integrity requirements. For each of these remediations, the IM service is able to perform these actions sooner than any remote party could by avoiding the delay introduced by attestation. As a result, the window for malicious behavior by the monitored system is reduced using the IM service.

V. OpenStack Integration

We now describe our implementation of a proof-of-concept CV framework for the OpenStack open-source IaaS cloud platform [24] with KVM virtualization on Linux 3.0 kernels. Figure 4 illustrates a simplified view of the OpenStack integration. The main changes are the addition of the *nova-verify* service to implement the CV and modification of *nova-compute* nodes to support the IM service. Other OpenStack services have been modified to use the CV and IM service as described below. Overall, our changes comprised roughly 2,600 SLOC in Python. Additional code for implementing modules and measurement interfaces was under 1,000 SLOC of Python.

A. Building a Verifiable Cloud Node

We designed a single disk image for each component and installed it on cloud nodes over the network using the open source Orchestra Provisioning Server. The disk image in installed using the network-based ROTI installer (netROTI) [23]. The netROTI uses hardware support to establish a dynamic root of trust at system reboot [25] to enable verifiable installation over a hostile network. The installer generates ROTI proofs by taking measurements of the installed file system, the Orchestra installer, and the disk image. Such proofs are then sent to nova-verify service for authenticating and authorizing the cloud node which we will discuss later in SectionV-B.

Each component maintains a static filesystem across reboots. To do this, we used AUFS to overlay filesystem branches into a single virtual file system. This allows us to set the installed filesystem to be read-only while layering a temporary, RAM file system on top. In this way, modifications to the installed filesystem will be discarded at reboot, so the node can return to the installed state on each reboot.

To prevent runtime modifications from exceeding the allocated temporary filesystem (e.g., caching of VM disk images) we overlay a writable disk partition over the cache location so that modifications to this directory will be preserved. This cache is partitioned from the installed distribution and is measured only on first access instead of at boot time.

B. Admitting Compute Nodes

The nova-verify service is responsible for admitting and evicting nova-compute nodes into and from the cloud.

Registering Compute Nodes. When a nova-compute node boots, it parses a locally-saved configuration file to locate the RabbitMQ message queue server and the cloud database. Normally, it attempts to register itself as an available compute service in the database and RabbitMQ. To mediate the registration process, we block all nodes except those that connect through an SSL tunnel (*stunnel*) proxy service, which provides SSL functionality without modification to application. We then provide a port that redirects all non-stunnel traffic to the nova-verify service to begin the cloud join protocol (see Section IV), where nodes use the ROTI proofs above as their attestations. Upon a successful join, the nova-verify node places the node's certificate in the directory of valid certificates on the message queue and database.

Monitoring Compute Nodes. After a nova-compute node registers itself to the cloud, the nova-verify service monitors that compute node for reboots. We implement node monitoring by having the nova-compute nodes periodically query the database per their normal behavior. This is done to look for the pending requests they should service. This automatically updates a last-accessed time for the node. However, if the status of a nova-compute node is not updated for a certain duration[3], the nova-verify service will notice the missing update and remove the node's identity certificate from the message queue. This, in turn, invalidates the stunnel connection, which isolates the node from the cloud.

[3] In our case since we believe it is unlikely for a server to fully reboot into a malicious state from a benign state in less than 30 seconds.

C. Starting an Instance

When loading an instance on the cloud, customer can choose whether to initiate the IM service at the same time. We implemented the IM service as a separate daemon spawned by the *nova-manage* service running in the nova-compute node. It is initiated prior to the customer's instance, allowing it to collect load-time properties of the instance using its registered modules.

We modified the `euca-run-instances` command in the Eucalyptus command suite to spawn a new IM service as part of instance initiation. For example, the `euca-run-instances Image_ID -v` command starts up a new instance with `Image_ID`. We extended this command to start an IM service as well. When an IM service is spawned, it obtains handles to the available measurement modules. These handles are passed to the modules, which select the monitoring interface (e.g., VM introspection, vTPM, etc.) they intend to use.

D. Registering Integrity Criteria

After the IM service is initiated, customers connect to it through cloud by verifying the nova-verify service, which then forwards the request to the nova-compute node hosting the IM services. Our prototype is built under OpenStack FlatDHCP network mode, which places all instances on a private network whose gateway is the network controller (*nova-network* in Figure 4).

Connect to the IM Service. A customer starts the connection protocol by first communicating with the instance using a well-known IM service port. The network controller then redirects the request to nova-verify by responding with the IP address of the nova-verify service. This tells the customer that the instance is being hosted on a cloud and that a CV is available.

Verifying CVs. The customer verifies the ROTI proof and the current boot cycle of the nova-verify node before proceeding to use the cloud's services. Since current commodity TPMs can only generate one attestation per second, methods are needed for nova-verify to produce attestations for several customers concurrently. We addressed this problem by using the batch attestation approach described by Santos et al. [6]. Using this method, nova-verify server batches multiple customer attestation requests into a Merkle hash tree and produces a single attestation that all customers can verify.

Monitoring CVs. If the verification succeeds, the customer then monitors nova-verify to detect when it reboots. This can be done through heartbeat signals over a SSL tunnel between nova-verify node and the customer. But in practice, we found that maintaining multiple tunnels is expensive, as it greatly reduces the number of simultaneous customer connections that a nova-verify node can support. As an alternative, we explored the solution of using attestation as the heart beat signal. Customers will continue requesting nova-verify node to generate attestations and verify the attestations. Since such attestations can only be produced by the TPM on nova-verify node, the liveness of nova-verify node can thus be guaranteed. We investigated the number of simultaneous connections supported by this scheme. Results show that a single nova-verify node can support more than 20,000 concurrent customers with a heartbeat window setup to be 10 seconds, in which case the TPM produces one attestation every 10 seconds.

Request Connection to IM service. The customer then requests to connect to the IM service through a new API command `euca-register-criteria`. The arguments for the command are: 1) the public IP of the intended instance, 2) a port on the instance, and 3) the criteria the customer wishes to have monitored and enforced. We added a function to the *nova-api* service to forward the request through the message queue to the compute node hosting the instance with the specified IP. We also added a special handler in the nova-compute service to respond to the criteria registration RPC call to interact with IM service, as described below.

Register Integrity Criteria. When the IM service receives a customer's request, it first registers the customer's integrity criteria with each measurement module. If the modules report that the criteria are satisfied, the IM service unblocks a randomly assigned port $P_{criteria}$ for $IP_{instance}$ in the compute node (the VM host) firewall. This port then redirects requests to $P_{service}$ on the instance. For example, the IM service will add a firewall rule to redirect HTTP traffic from port 9000 to port 80 for that nova-compute node using iptables. By using this "criteria" port (port 9000 in this case), the customer obtains a guarantee that that instance will only send or receive data using that port if it satisfies the associated integrity criteria. That is, if a violation of that criteria is later detected by IM service, this rule is deleted and all traffic to port 9000 will be denied. Considering the limited number of ports on compute node, we aggregate the connections based on the $\langle C_{criteria}, P_{criteria} \rangle$ tuple. If more than one set of criteria are registered, different ports will be assigned to represent different sets of criteria.

E. Serving Clients

Upon successful registration of integrity criteria for an instance, the customer obtains an instance certificate and a node certificate from the nova-verify service. The instance certificate is signed by compute node, containing the instance's public key generated by the IM service and the details of the connection request. Upon receipt of an instance certificate from a registered nova-compute node, the nova-verify service produces a node certificate containing the public key of the compute node signed by the nova-

verify service. Customer will then announce $IP_{instance}$ and $P_{criteria}$ to clients. Clients who wish to use such security enhanced service can then establish a TLS connection over $IP_{instance}$:$P_{criteria}$ to the service, using the certificate chain provided by customer. Note that $P_{criteria}$ is associated with a set of integrity criteria, violation of which will disable all connections to $P_{criteria}$ immediately. Clients also have the flexibility to choose different sets of criteria, if registered by customer, or even no criteria, in which case clients will connect over the original service port.

Accessing the same service using different ports reflects groups of clients with different security requirements. Clients who only read public information, may be more interested in using a non-disruptive service through accessing the original service port. Clients who use such cloud-hosted services to process their sensitive data would be more willing to immediately cut their connections once anomalies are detected in the service. This is a trade-off between security enhanced service and an "always-on" service that clients need to evaluate.

VI. Evaluation

We evaluated performance overhead of our OpenStack CV prototype for cloud applications. Results show that the CV framework poses less 3% overhead for cloud hosted web servers. We attribute such overhead to the VMI interface that IM service uses to collect runtime properties within instances. We refer the reader to the IVP paper by Schiffman *et al.* [22] for more details.

We also evaluated our CV framework's ability to monitor and enforce customers' requirements by designing various monitoring modules that collect a variety of instance properties. Several example interfaces are implemented for these modules to demonstrate the framework's flexibility and tested their correctness using example criteria that checked for example properties.

Monitoring VMinfo. The VMinfo interface exposes properties of the instance profile such as number of vcpus, memory size, boot parameters, network devices, etc. This information can be compared against the customer's requirements to ensure the instances are given the correct resources and are not being cheated. The interface uses the libvirt Python API directly to query the VM instance object.

Monitoring Network and Host Security Policies. Interfaces are provided to monitor iptables and SELinux policies and authorized operations. For networks, iptables conntrack interface is monitored via `ulogd` to obtain the network flows connecting to the monitored instance. In other words, Netflow modules can track the IPs of remote systems that connect to the instance. The flows are stored in a local `sqlite3` flat file database. Customer criteria can use this information to detect if unwanted hosts are using the instance, such as known botnet C&C networks. In addition, the customer can use these flows to check that the firewall rules specified to the cloud are actually enforced.

Monitoring Memory. A proof-of-concept VMI interface lets modules check for writes to specific memory locations in the instance. When an instance spawns, we attach a gdb thread to it and set the hardware debug registers to watch specific kernel data structures defined in the kernel symbol table. Each module can set a hardware watchpoint[4] and when a write is performed, the module is notified of the change. For example, we use this to verify that all security policy changes meet integrity requirements and that each file executed satisfies a white-list of programs. By checking these properties, customer can verify that instances are enforcing the expected security constraints at runtime. Since changes to these values are infrequent, so are interrupts that pause the VM.

VII. Related Work

We now present a summary of work related to verifying cloud computing platforms.

Virtual Machine Security. Research in hardening hypervisors and protecting instances has become popular with the rise of cloud platforms. Techniques like SecVisor [26] and XenAccess [11] introduce VMI interfaces for monitoring VM integrity. However, these approaches are designed to monitor specific integrity requirements at install-time and are not flexible enough to support various customer specified criteria. Moreover, the power of these tools are unmitigated in their deployment and give administrators far more access to sensitive VM data than should be permitted. The CV framework can benefit from the inclusion of these VMI tools that targets more integrity-relevant memory locations.

Cloud Security. Another research direction is to enable customers to obtain greater control over the conditions under which they will run their instances on a cloud. Excalibur [6] and self-service clouds [7] are examples. However, Excalibur [6] focuses only on properties of compute node, failing to provide more meaningful monitoring such as runtime properties of instances. Self-service clouds [7], on the other hand, allows customers to manage their own instances, but fails to protect the rest of the cloud services, such as network service and api service, upon which the instances depend. Moreover, they cannot provide responsive yet flexible control over customers' instances upon detection of anomalies, as our CV does.

Other work looks at hardening the hypervisor from both co-resident VMs and insiders. Approaches like NOVA [27] and NoHype [28] minimize the VMM's attack surface by essentially eliminating all but a small resource management

[4]The Intel x86 hardware has four hardware debug registers, supporting four hardware watchpoints per instance. If more watchpoints are required, previous work has demonstrated the feasibility of adding software-managed VMI watchpoints in KVM [10].

kernel. Other techniques such as HyperSafe [29] use the protected System Management Mode memory region to monitor and enforce hypervisor integrity at runtime. The CV can use these to provide strong guarantee that the runtime integrity of the cloud components are maintained.

VIII. Conclusion

In this paper, we presented the Cloud Verifier, a framework that cloud vendor can configure to provide cloud monitoring service in IaaS clouds. The CV is an independent and verifiable service in the cloud that enforces a cloud administrator's integrity criteria over the cloud components. Customers can then build transitive trust through the CV and cloud platform to the Instance Monitor (IM) service which monitors various loadtime and runtime properties of their instances. We constructed a proof of concept CV for the open source OpenStack cloud platform and demonstrated several monitoring options that customers can utilize to enforce the healthy state of their instances. We further evaluated the framework on two of the most popular application instances used in Amazon's EC2 cloud. Our experiments show negligible overhead is incurred by the instances due to monitoring and that the CV is capable of supporting more than 20,000 concurrent customers. We believe that, by adopting the CV framework into an IaaS cloud platform, cloud providers can provide a both effective and efficient way for assuring customers that their instances are running according to their requirements.

References

[1] "Amazon EC2," http://aws.amazon.com/ec2.

[2] "Rackspace Cloud Servers," http://www.rackspace.com/cloud/.

[3] "Nagios IT Infrastructure Monitoring," http://www.nagios.org/.

[4] "Ganglia Monitoring System," http://ganglia.sourceforge.net/.

[5] F. Rocha and M. Correia, "Lucy in the sky without diamonds: Stealing confidential data in the cloud," in *DSNW '11*. IEEE Computer Society, 2011.

[6] 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*, 2012.

[7] S. Butt, H. A. Lagar-Cavilla, A. Srivastava, and V. Ganapathy, "Self-service cloud computing," in *Proceedings of the 2012 ACM conference on Computer and communications security*, ser. CCS '12. New York, NY, USA: ACM, 2012, pp. 253–264.

[8] S. et al, "All your clouds are belong to us: security analysis of cloud management interfaces," in *ACM CCSW '11*.

[9] T. Garfinkel and M. Rosenblum, "A Virtual Machine Introspection Based Architecture for Intrusion Detection," in *Proc. NDSS*, 2003.

[10] M. I. Sharif, W. Lee, W. Cui, and A. Lanzi, "Secure in-vm monitoring using hardware virtualization," in *CCS '09*. ACM, 2009.

[11] B. D. Payne, M. Carbone, and W. Lee, "Secure and Flexible Monitoring of Virtual Machines," in *ACSAC*, 2007.

[12] E. Kowalski, D. Cappelli, and A. Moore, "Insider Threat Study: Illicit Cyber Activity in the Information Technology and Telecommunication Sector," U.S. Secret Service and CMU, Tech. Rep., 2008.

[13] M. Keeney, "Insider Threat Study: Computer System Sabotage in Critical Infrastructure Sectors," U.S. Secret Service and CMU, Tech. Rep., 2005.

[14] TCG, "Trusted Platform Module," https://www.trustedcomputinggroup.org/specs/TPM/, 2005.

[15] "Integrity: Linux Integrity Module(LIM)," http://lwn.net/Articles/287790/.

[16] "Trousers," http://trousers.sourceforge.net.

[17] B. Parno, J. M. McCune, and A. Perrig, "Bootstrapping Trust in Commodity Computers," in *IEEE SP '10*, 2010.

[18] T. Jaeger, R. Sailer, and U. Shankar, "PRIMA: Policy-Reduced Integrity Measurement Architecture," in *Proc. 11th ACM SACMAT*, 2006.

[19] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki, "Flicker: An Execution Infrastructure for TCB Minimization," in *Proc. 3rd ACM SIGOPS/EuroSys*, 2008.

[20] J. McCune, S. Berger, R. Caceres, T. Jaeger, and R. Sailer, "Shamon: A System for Distributed Mandatory Access Control," in *Proc. ACSAC*, 2006.

[21] S. Bleikertz, T. Groß, and S. Mödersheim, "Automated verification of virtualized infrastructures," in *ACM CCSW '11*.

[22] J. Schiffman, H. Vijayakumar, and T. Jaeger, "Verifying system integrity by proxy," in *TRUST '12*, 2012.

[23] J. Schiffman, T. Moyer, T. Jaeger, and P. McDaniel, "Network-based Root of Trust for Installation," *IEEE Security & Privacy*, 2011.

[24] "OpenStack," http://www.openstack.org/.

[25] "Trusted Execution Technology," http://www.intel.com/technology/security/.

[26] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "Secvisor: A Tiny Hypervisor To Provide Lifetime Kernel Code Integrity For Commodity Oses," in *SOSP '07*. ACM, 2007.

[27] U. Steinberg and B. Kauer, "Nova: a microhypervisor-based secure virtualization architecture," in *EuroSys '10*. ACM, 2010.

[28] J. Szefer, E. Keller, R. B. Lee, and J. Rexford, "Eliminating the hypervisor attack surface for a more secure cloud," in *CCS '11*. ACM, 2011.

[29] Z. Wang and X. Jiang, "Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity," in *SOSP '10*. IEEE Computer Society, 2010.