Seeding Clouds with Trust Anchors

Joshua Schiffman, Thomas Moyer, Hayawardh Vijayakumar
Trent Jaeger and Patrick McDaniel
Systems and Internet Infrastructure Security Laboratory
Pennsylvania State University
jschiffm, tmmoyer, huv101, tjaeger, mcdaniel@cse.psu.edu

ABSTRACT

Customers with security-critical data processing needs are beginning to push back strongly against using cloud computing. Cloud vendors run their computations upon cloud provided VM systems, but customers are worried such host systems may not be able to protect themselves from attack, ensure isolation of customer processing, or load customer processing correctly. To provide assurance of data processing protection in clouds to customers, we advocate methods to improve cloud *transparency* using hardware-based attestation mechanisms. We find that the centralized management of cloud data centers is ideal for attestation frameworks, enabling the development of a practical approach for customers to trust in the cloud platform. Specifically, we propose a *cloud verifier* service that generates integrity proofs for customers to verify the integrity and access control enforcement abilities of the cloud platform that protect the integrity of customer's application VMs in IaaS clouds. While a cloud-wide verifier service could present a significant system bottleneck, we demonstrate that aggregating proofs enables significant overhead reductions. As a result, transparency of data security protection can be verified at cloud-scale.

Categories and Subject Descriptors

C.2.4 [**Computer Systems Organization**]: Distributed Systems

General Terms

Security

1. INTRODUCTION

Cloud computing has become the buzzword *du jour* for IT professionals and researchers alike. The cloud's on-demand provisioning of computing resources has ushered a shift in application deployment. Developers, no longer required to build and maintain applications on expensive in-house infrastructures, may now leverage publicly accessible, multi-tenant systems administered by third parties to host their code and data. The promise of reduced cost and maintenance has already excited many companies, as a recent poll found over 25% of surveyed business are interested in moving their services to public cloud platforms [8]. While several cloud deployment models have evolved into the marketplace, Infrastructure as a Service (IaaS) clouds appear to be the most interesting to large corporations, as they offer VM hosting as a primary service enabling hosting of complex application designs.

However, recent scares of massive data loss have discouraged more rapid adoption as indicated by the fact many companies believe security and privacy are the top concerns when deciding to adopt clouds [10]. What is needed is a way to increase the *transparency* of clouds so that users can build trust in these public services. The most obvious issue cloud customers face is ensuring that their data security requirements are satisfied while using the cloud for simple tasks like storage or more complex cloud processing. To ensure these requirements are met, a cloud customer must be able to verify that the cloud's integrity has not been compromised and that it is functioning within the parameters necessary to satisfy the customers security needs. By integrity, we mean the classical notions of integrity used in models like Biba [2] and Clark-Wilson [5], where trusted entities and data must come from high integrity sources and depend only on similar integrity level inputs or higher (discussed further in Section 3.1). In this paper, we identify three main challenges that cloud providers face when generating proofs that can placate a user's concerns: 1) that cloud vendors provide a proof of data security protection of their hosts and customer processing; 2) that such proofs have a clear meaning to cloud customers; and 3) that such proofs can be generated effectively and efficiently in a cloud computing environment.

First, the lack of physical control (by customers) necessitates greater scrutiny of cloud hosts than traditional in-house deployments. While cloud providers use SLAs, armed guards around their data centers, and non-specific claims of "hardened systems", no cloud offers clear proofs of their hosting platform's runtime integrity. Second, clouds by their nature support multi-tenancy, which allows other customers to run their code on the same physical machine. Since cloud applications are hosted on a complex software stack consisting of a virtual machine monitor (VMM), host OS, and potentially several guest VMs, a host incapable of enforcing noninterference between VMs opens an avenue for malicious VMs to obtain leaked data or compromise the entire system. It is for these reasons that cloud customers require a proof that their VMs are hosted on high integrity systems and that their VMs have been loaded in a secure manner.

The second challenge is ensuring cloud customers can correctly assess integrity via attestations produced by the cloud. Approaches like integrity measurement [25, 23, 19] allow systems to record measurement of the code and data on a system, but it is up to the remote verifier to determine if that combination of measurements results in a secure system. This inherently difficult challenge is

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CCSW'10, October 8, 2010, Chicago, Illinois, USA.
Copyright 2010 ACM 978-1-4503-0089-6/10/10 ...$10.00.

further complicated by a cloud that may use proprietary or unverified code, configure systems in unusual ways, and lack necessary access control mechanisms. Instead of complex low-level measurements, we propose measuring *enforcement of properties*. In the past, researchers have proposed property-based attestation [16], which convert a set of measurements into higher-level properties. However, this does not obviate the need for fine-grained measurements. We instead advocate measuring enforcement of properties by measuring an enforcer and policy defining the property, where possible. A variety of enforcement is used in integrity measurement already, from data [23] and access control [11], to code loading [24], but we make this a first class concept.

Finally, deploying a practical cloud verification system is key to the adoption of current cloud frameworks. Current cloud architectures provide some benefits for integrity measurement, such as a centralized deployment for a PKI, integrity measurement devices, and physical security over machines, but there are some key challenges as well. First, cloud architectures are largely closed to external inspection. Recent work has examined the use of fully trusted third parties to audit clouds [20], but this still limits the cloud's transparency to the user. Second, a cloud may have many customers requesting attestations, so the benefits of centralization may be offset by the bottleneck of responding to attestation requests from an inherently slow (about 1 second per request) trusted computing devices. Any cloud-based integrity measurement mechanism must enable the generation of proofs indirectly with good scalability.

In this paper, we propose a centralized verification service called the *cloud verifier* (CV), which produces attestations of a customer's cloud instances that can be used to verify both the integrity of a customer's VM and the system hosting it. A CV is a service residing in the cloud that monitors the state of the cloud platform and user VMs. The cloud verifier enables indirect verification of its hosts by *vouching* for the enforcement of properties on them. Using this approach, cloud customers can verify that the cloud verifier satisfies their integrity property requirements and that the properties the cloud verifier vouches for being enforced on its hosts satisfies the customer's properties for those components as well. Customers can then leverage their trust in the cloud hosts to verify their own application VMs' data security.

In Section 2, we outline the expectations that customers have of cloud systems that run their security-critical applications. In Section 3, we detail what is required to verify the integrity of a remote component in general and extend this approach into a protocol that enables a cloud verifier to generate such integrity proofs for cloud applications. In Section 4, we outline a prototype implementation of our cloud verifier, and in Section 5, we examine performance issues in building a scalable cloud verifier service.

2. TRUSTED CLOUD EXPECTATIONS

We now consider a scenario to explore the requirements that customers have for running their security-critical applications on the cloud. Consider the typical computer science graduate student, Alice, who uses an integrity-verifying cloud. She uses the cloud for a variety of applications including a research project distributed across several VMs, a personal web server VM to host her homepage, and a code repository VM to store and compile her coding projects. Alice is computer-savvy and well informed, so she is concerned about protecting the secrecy of her data on the cloud as well as ensuring her VMs are not compromised by viruses or rootkits. She expects the cloud hosting her code and data to prove its own integrity and that her security requirements are being enforced over the lifetime of her applications' execution. Moreover, Alice may want to perform sanity checks on her VMs and data, but is concerned these checks are futile if the hosting system is compromised. To satisfy Alice, the cloud must be configured to clearly demonstrate its VMMs are running as expected and protecting her applications from compromise by internal cloud components.

In demonstrating these qualities, Alice first expects the cloud platform (VMMs) to be trustworthy. Proving general system security is difficult, but a hardened, verifiable, and consistent deployment of VM hosting systems simplifies the task of proving the cloud's code and data. Clouds offer a unique opportunity for the reuse of expert administration across the data center to deploy a concrete foundation for integrity. For example, the cloud's economy of scale may encourage the use or development of proven components, where possible (e.g., trusted microkernels for VMMs like SEL4 [13]), and the careful configuration of cloud platforms whose code and data are largely fixed once installed [4]. Another key issue is the management of the cloud's attack surface [9]. Currently, we know that network-facing processes are an obvious attack vector, but we do not have a clear understanding of how compromises may propagate [3, 26] throughout arbitrary systems. Cloud architectures may provide a motivation for developing a concrete understanding of attack surfaces.

Once configured, the cloud must generate proofs of integrity for customers like Alice. One approach is to use integrity measurement techniques that leverage secure hardware like the Trusted Platform Module (TPM) to produce attestations, measurements of integrity-sensitive code and data, signed by a key tied to the physical hardware. While nearly ten year of research has been done on leveraging these trust building primitives, practical deployments are limited at present. Nonetheless, the cloud environment provides one of the most attractive opportunities for broad use of this technology. Integrity measurement requires physical security of hardware, a PKI to manage TPM keys, and a means of interpreting attestations. First, clouds providers often advertise their armed guards and internal auditing practices, which provides reasonable protection from physical attacks. Second, the central administration of clouds serves as an ideal scenario for an internal PKI. Third, as the cloud platform is also controlled by one party, they can define a methodology for verifying the integrity of their cloud. While it is possible that a cloud provider will generate vacuous proofs (i.e., ones that do not imply strong integrity or security guarantees), certainly the research community will evaluate the meaning of these proofs. The key idea is that such proofs will enable a more transparent and verifiable operating environment.

With a proof of the integrity of this cloud platform, Alice would also want to determine whether the data security of her application is protected. This includes assessing both data secrecy issues caused by multi-tenancy (e.g., [18]) and the use of shared cloud resources and data integrity issues caused by customer configuration of access control policies over multiple services. Using integrity measurement methods to prove data security in general purpose applications has not been practical to date, but again, the structure of cloud systems provides a foundation for building data security proofs that are accurate and meaningful to Alice. In general, a cloud system provides well-defined loading and execution phases, where much of the execution may be governed by the cloud (networking and storage). Successful integrity measurement methods have been developed for the loading of code [19, 25] and the loading of data from prior computations [23]. However, tracking the runtime integrity of general systems has been incomplete. Early methods implemented *authenticated boot* [19], where a system would be allowed to run regardless of whether it was secure or not, and the verifier would have to determine this from the integrity proof. Over time, methods have adopted a *secure boot* [1] posture,

where enforcement of code loaded [14] and accesses authorized once loaded [11], is used to make it more likely that the system will remain high integrity. In the cloud, effective secure boot is practical because Alice defines the secure boot conditions for her applications and the cloud simply enforces what Alice requests.

3. VERIFYING CLOUD INTEGRITY

In order to prove its integrity, a remote entity generates an attestation of its configuration, but a verifier must have some methodology for assessing this attestation. While several methods have been developed to validate the integrity of a running application (see Parno *et al*'s survey for details [17]), we lack a model for comparing the integrity guarantees offered by such methods [1]. In this section, we develop a preliminary model for reasoning about the integrity of systems at a more abstract level than the low-level code and data measurements employed in many integrity measurement techniques [19, 14, 24]. To do this, we adapt the Outbound Authentication [25] (OA) model, which authenticates processes running on a secure co-processor, to a model for verifying higher level integrity guarantees of a running application on a complex system like a VM. We start by discussing the integrity properties that must be verified in a single system case and then extend these notions for verifying applications in a cloud scenario. We describe the challenges of using this model and possible directions for solving them.

3.1 Integrity Measurement

Before discussing our model, we first provide some terminology and background on the aspects of integrity measurement germane to it. The TPM is a secure coprocessor on the motherboard that exposes several key functions. It is capable of producing cryptographic keys, perform digital signing, encryption, and hashing. A signing key pair called the Endorsement Key (EK) that uniquely identifies the TPM device and associated client is burned into the TPM. The TPM uses the EK to certify other keys including attestation identity keys (AIKs). Measurements are recorded with the *TPM Extend* function that takes a SHA1 hash of arbitrary data and updates one of the TPM's platform configuration registers (PCRs) with a hash of the measurement and current PCR value. This forms a hash chain a verifier can recreate given the measurement list that formed it. Typically, a system will measure security critical code and data starting from the BIOS followed by each stage of the boot process up to runtime operations at the application level. These measurements are reported using the *TPM Quote* operation that takes a nonce from a remote party and forms a statement containing the PCRs and the nonce signed by an AIK. This quote along with the measurement list forms an attestation of the system's state.

A criteria is a set of properties P that must be satisfied for a verifier to consider a system high integrity. Such properties are arbitrary statements mapped to a set of measurements M, which we denote $M \times P$. This representation is similar to previous research in property-based attestation [16]. In order to obtain these mappings, the verifier chooses a *trust set* of authorities A that speak for the validity of a measurement to property mapping. Thus, we can consider a verifier's trust function $Trust : A \rightarrow M \times P$, which takes a set of authorities and returns the mapping of measurements to properties. How these mappings are formed in current approaches is largely an ad hoc endeavor. Typically, an administrator is expected to choose a set of software distributions to trust. Using that set, the administrator might chose a simple property that requires the attesting systems to run only code from the distribution. Ideally, a more sophisticated criteria based on abstract security properties like Biba [2] integrity is desirable so that the underlying system configuration is not tied so directly to the verifier's requirements. Such properties could require a combination of access control policy and code measurements that ensure high integrity labeled processes only depend on high integrity inputs. Such properties allow a verifier to compare different criteria to one another. We plan to investigate criteria formats and how they can be compared in future work. Once the verifier has chosen its criteria and trust set, it can then assess the integrity of an attestation.

[1]Datta *et al*'s logic [6] enables reasoning about TPM-level operations, but not how the composition of code in systems impacts overall integrity.

3.2 Verifying a single entity

We choose the OA model for verifying systems as it was one of the first systems for reasoning about the authenticity of remote secure coprocessor systems. In OA, a system is organized into *layers*, which are segregated along divisions of function, storage, and control. An application is a layer running on an OS layer, which may be run on hardware virtualized by a VMM layer. We focus now on what it takes to verify a single layer. A layer's *configuration* is initialized by a finite sequence of operations called a *history* H that defines how a set of software entities E (e.g., code, inputs) and an enforcement policy π were loaded into the layer. A history shows how an initial configuration was produced prior to this execution, which could be a hash of the layer's files signed by some authority. A *run* R is an unbounded sequence of operations prefixed by H, written $H \prec R$. Each operation σ in R causes a change in the layer's configuration. We can express a run for a layer after i operations were executed at time τ as $R_\tau = (\sigma_0, \ldots, \sigma_i)$. Such operations include code loading and interactions with other software entities that cause a change in the configuration. A run is analogous to the measurement log included with an attestation and represents how the current configuration was produced.

We modify the OA construction by considering the operations that affect integrity (i.e., *integrity-relevant operations*) in a run and the history that initialized it. The set of all integrity-relevant operations that can occur at a layer is Σ. Furthermore, a system's enforcement mechanism (e.g., access control) uses a policy π to mediate a set of operations Σ_π, which is a subset of Σ. These mediated operations are not recorded in the run if the subjects performing the operations are allowed by the policy and thus are trusted not to affect the configuration's integrity. One result of this extension is that strict enforcement policies reduce the size of a layer's run because there are fewer possible operations to assess. Thus, a layer N_τ's configuration at time τ is defined as a tuple $\langle E, \pi, H, R_\tau \rangle$, where E and π are the current entities (code or programs running in the layer) and policies, respectively.

A verifier evaluates a layer's integrity by: (1) verifying all entities loaded during H are trusted; (2) the sequence of operations in R_τ do not alter the current configuration in an untrusted way (e.g., load untrusted entities); and (3) π does not allow mediated operations Σ_π to affect the configuration's integrity. We now examine how these three conditions are verified.

Verifying the History.

Verifying a layer's history H requires checking that all entities installed into the layer are trusted. To do this, the verifier examines the history of measurements that produced the layer's initial state. If these measurements M map to the properties P required by the verifier, as dictated by the verifier's trusted authorities A, then the initial state can be considered trusted. Approaches used by existing integrity measurement systems like IMA [19] match hashes of loaded code to a trusted software distribution. While IMA is useful

for examining the identity of code, data is more difficult to verify as arbitrary files may have no well-known hash. The Root of Trust Installer (ROTI) approach [4], addresses this problem by generating a ROTI proof that associates the system's installed state to a trusted installer. This enables a verifier to simply check that files have not been modified from the installed state and the installer is trustworthy. Dynamic files need an ad hoc verification, analogous to a Clark-Wilson integrity verification procedure [5].

Verifying the Run.

After a layer is initialized by H, its configuration undergoes modification due to a sequence of operations that may affect E in the layer. Verifying that the operations in R have not degraded the configuration's integrity requires checking that all operations satisfy the verifier's criteria. For code loads, external inputs, or other modifications to entities, the run contains measurements M of these changes and each must map to the properties P satisfying the criteria. The run differs from the history in that it contains operations occurring after installation and are dynamic for each run.

Verifying Enforcement.

Since a run is unbounded, verifying the totality of R can be quite extensive, which we imagine has limited its utility. Recording and verifying massive logs of operations like server transactions or writes to memory is not practical. However, enforcement can reduce the set of operations that must be recorded. Researchers have built integrity measurement methods that enable an enforcement mechanism to speak for the integrity impact of an operation like limiting code execution to trusted sources [24], limiting external input to trusted processes [11], or vouching for integrity of remotely-generated data [23]. A verifier checks whether this enforcement is trusted to enforce an integrity property by assessing the integrity of the entities enforcing the policy π and the policy itself against the verifier's criteria. That is, the measurements M consist of the enforcing entities and policy enforced to satisfy the property P, where an authority A dictate the requirements.

3.3 Verifying a Multi-layer System

In cloud systems, an application VM does not run as a single layer. The VM often runs in a system containing a VMM, and one or more VMs privileged VMs that provide services for running application VMs. We refer to this VM host using the term *node controller* (NC), which is a term used by the open source cloud framework Eucalyptus [7]. An NC has a direct impact on the integrity of the VMs it runs. An untrusted NC could compromise the VM's configuration, leak secrets, or worse. Thus, it is crucial to verify the integrity of all layers when assessing a particular layer's integrity. Verifying multiple layers involves single layer verification as above plus the ability to verify that lower layer provided enforcement speaks for the integrity of higher layers.

A lower layer can provide enforcement on the operations affecting the layers above. For example, a NC can restrict the number of untrusted external network inputs to a VM or introspect into its memory space to verify runtime integrity of code. Thus, a layer's enforcement policy π can be considered the composition of policies below it. Another consideration is the ability of a layer to verify the integrity of a higher layer, and speak for that integrity to others. Suppose that one layer proves that the integrity of a higher layer satisfies a criteria that is compliant with a verifier's criteria. If the verifier accepts the integrity of the lower layer and its ability to judge other layers based on that criteria, then the verifier trusts the lower layer to advocate for a higher layer's integrity. For example, if a verifier's criteria C_A has the same or greater number

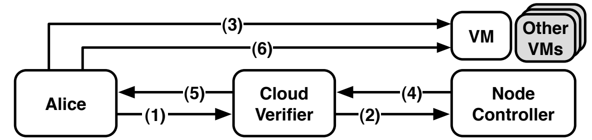

Figure 1: Alice (1) requests an attestation of the CV and its criteria. The CV (2) verifies the cloud's NCs while Alice (3) starts her VM. The NC (4) sends an attestation and identity key of Alice's VM to the CV, which CV (5) forwards to Alice. Finally, Alice (6) uses the key to authorize her VM to access her data after verifying the VM attestation and key signature.

of properties and trusts the same or fewer set of authorities as C_B, then criteria C_B should enable a verifier to trust only a subset of systems that a verifier using C_A would trust, if the layer applying CB is trusted to evaluate that criteria. This is similar to BIND's notion of transitive trust [23].

As an example, consider an NC running the Virtual Machine Verifier (VMV) enforcement and attestation framework [22] that enforces criteria C_{NC} on its VMs. The VMV ensures a VM satisfies C_{NC} by enforcing a particular runtime enforcement policy on the VM and inspecting measurements of its initial state. A verifier with criteria C_V can then inspect an attestation of the NC against its criteria to establish its integrity. It can then check that C_{NC} satisfies the same properties as C_V. If so, then the NC can speak for the integrity of the VM. We employ this concept in the following subsection to design a protocol for verifying application VM integrity using a verification proxy for the cloud.

3.4 Verification Protocol

We now describe a protocol to deploy integrity-verifiable applications on a cloud system. First, we introduce the protocol's components and their purpose and then show how the protocol is used by a customer Alice to deploy and verify her application on the cloud. Recall that Alice wants to determine whether the cloud platform satisfies her integrity criteria and whether her VM's runtime data security is enforced. We assume the cloud provides physical security and public key certificates for the hardware TPMs of the nodes.

A cloud application can be viewed as a collection of specialized VMs running on the cloud that interact with data hosted on the cloud and input from external parties. The goal of our verification protocol is to prove to Alice that her application VMs are running on a high integrity platform and that only high integrity VMs can access the application's data. Figure 1 illustrates our protocol for launching and verifying such application VMs. Since direct access to the NC by Alice is not possible as clouds do not want to open an avenue for attack and denial of service, we introduce the *cloud verifier* (CV), a verifiable and minimal service within and operated by the cloud to verify the integrity of the NCs within the cloud using the cloud's own criteria C_{CV}. To use the CV, users first verify the CV's integrity and the criteria it uses to verify the cloud's NCs. If the user finds the CV is correct and that its criteria satisfies the user's own criteria, the CV can act as a verified proxy, enabling the CV to speak for NC integrity in the cloud.

For our protocol, we assume Alice's criteria C_A contains her requirements for both cloud components (CV and NCs) as well as her VMs. In step (1), Alice requests an attestation of the CV's configuration, $Attest(CV) = Sign(E, \pi, H, R)_{K^-_{CV}}$ and its criteria C_{CV}, where K^-_{CV} is the private portion of the CV's AIK. Alice verifies $Attest(CV)$ as described in Section 3.2, checks the signing key is certified by the cloud's signing key, and examines if C_{CV}

is compliant with C_A. In step (2), the CV requests $Attest(NC)$ for NCs in the cloud and verifies them against C_{CV}. If an NC's proof fails verification, the cloud is alerted and action can be taken to correct the problem. Since other VMs are most likely running on the NC, the CV may be already monitoring the NCs. For larger clouds where a single CV may not be sufficient to verify all NCs, we envision using a tiered structure where multiple CVs are delegated different sets of NCs. Once the cloud NCs are verified, Alice (3) sends a request to start a VM on the cloud and loads a VM image on a NC directed by the CV. The NC then creates the VM's executing environment and provides any requested resources. The NC generates the VM's identity key pair and signs it. In step (4), the NC sends an attestation of the VM $Attest(VM)$ and $Sign(K^{+}_{VM})_{K^{-}_{NC}}$ of the VM to the CV, where K^{+}_{VM} is the identity public key and K^{-}_{NC} is private AIK portion. The CV also signs the identity key with the cloud's AIK, signifying to Alice that the key is from the cloud. The CV then delivers the VM attestation and signed key to Alice in (5). Alice verifies the VM's attestation against C_A's VM integrity properties and validates that the VM identity key's signature chain comes from the CV. Finally, she uses the identity key to establish an authenticated connection with the VM in step (6) and sends an authorization for the VM to access application data hosted on the cloud. Without this authorization, no VM is able to access the data.

4. IMPLEMENTING VERIFICATION

We now describe our preliminary implementation of our cloud verification protocol. Figure 2 illustrates our design deployed in a private Eucalyptus [7] cloud. The cloud consists of several components. The *cloud controller* is a web management front-end that authenticates user requests and allows basic administration of the application provider's VMs. Since the cloud controller is a publicly accessible interface, we also implement the cloud verifier in it. While the verifier service is small, placing it in the cloud controller increases its potential attack surface. In future work, we plan to design the CV as a separate, minimal system from the cloud controller. The NCs run VMs from a cloud storage, which holds user data volumes and VM images. Each VM contains a volume encryption key to access encrypted data volumes in the cloud store. This key is further encrypted by the customer and is only decrypted when the VM is verified as high integrity.

Our implementation runs as follows. The cloud customer Alice (1) requests an attestation of the cloud controller. After verifying it, she (2a) sends a request to start a new VM, which is (2b) forwarded to an NC. The NC (3) downloads the specific VM image from the cloud storage and runs the VM using Linux KVM. The cloud controller's CV (4) collects attestations of every NC at regular intervals. Included in the attestation is a proof for each VM on the NC. These proofs contain a signed hash of the VM image and a fresh IPsec key pair loaded into the VM before booting it. The cloud controller first verifies the NC's attestation satisfies its criteria and then (5) sends the VM proof and key signed by the CV to Alice. After verifying her VM satisfies her criteria for her VM, Alice (6) connects to her VM using the signed IPsec key and sends a key to decrypt the VM's volume encryption key used to (7) access the application data stored in the cloud storage.

5. PRELIMINARY EVALUATION

We now turn to our preliminary evaluation of our trusted cloud environment. Our goal is to determine: 1) whether it is practical to deploy such a framework and 2) how much of an impact verifying the individual cloud components will have on performance. To do this, we constructed a proof of concept trusted cloud testbed using

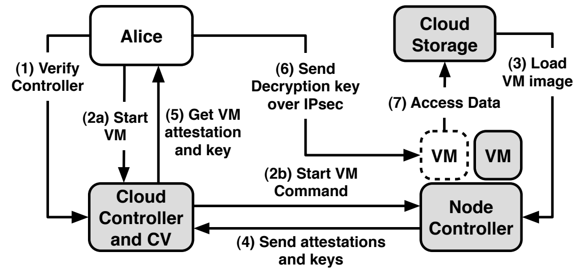

Figure 2: Implementation of our verification protocol.

a Eucalyptus cloud on Dell PowerEdge M605 blades with 8-core 2.3GHz Opteron CPUs with 16GB RAM on a quiescent gigabit network. Our evaluation was broken down into three distinct tests, which speak to the primary concerns we are evaluating: configuring the cloud hosts, generating attestations for those components and VMs, and serving proofs to cloud customers.

5.1 Configuring Hosts

We first evaluated the practicality of deploying NCs in a trusted cloud. First, we created a 368MB compressed disk image of the NC's 100GB hard disk and stored it on a partimage server. Next, we built a network boot installer to write and configure the disk image on each NC over the network. The installer followed the ROTI design [4], where a hash of the resulting filesystem was bound to the network installer by the TPM. One design challenge of using a network boot installer is the threat of malicious attackers on the local network that could modify data on the wire or pose as a rogue network boot or partimage server. We address this issue by employing the OSLO bootloader [12], which uses the SKINIT instruction in AMD CPUs to setup a secure execution environment for the installer and measures the code before executing it. This allows us to establish a root of trust for the installation procedure, even though the code was obtained over the network. We detail the design of our installer further in our techreport [21].

The installer consisted of a Linux 2.6.18.8 kernel, an initial ramdisk containing the installer environment, and the OSLO [12] bootloader. The ramdisk was 11MBs and contained 370 lines of custom shell scripts to automate the process. Finally, we setup ProxyDHCP and TFTP servers to host the installer and `pxelinux` network boot client. After the NC's machine is directed by the ProxyDHCP server to download the installer from the TFTP boot server, the OSLO bootloader performs the SKINIT instruction to measure and launch the installer kernel. From that point, the installer took approximate 150 seconds to perform the full installation, which includes downloading the disk image, making system specific configurations, and measuring the filesystem. Of that time, about 8 seconds were necessary to configure the netROTI software and 2 seconds were spent taking a hash of the local filesystem and downloaded disk image, which is a function of the size of the disk and hashing algorithm (3% of the download time). Configuring the TPM to generate an RSA signing key pair takes approximately one minute, but need not be performed for each cloud host install.

5.2 Attestation of Cloud Components and VMs

Once the hosts are deployed, we evaluated the performance of generating attestations from the CV for customers to verify hosts and VMs. For the CV to generate an attestation for its hosts, the host must generate an attestation, the CV must verify it satisfies the CV's properties for hosts, and the CV must generate an attestation of this fact for the clients. The performance cost is dominated by

the time to build attestations, unsurprisingly, as the TPM hardware is slow, at a little over one second per attestation [22]. Verification costs are negligible in comparison. We use IPsec between the hosts and the CV also, so the CV can maintain the integrity of the hosts, first done for the ROTI work [4], which add about one second before the first attestation.

We then evaluated the time to gather attestations and monitor the integrity of each cloud guest VM. After receiving a request from the CV that we integrated into our Cloud Controller, the NC took approximately 1 second to generate attestations. Verification of the attestations for each NC is performed in parallel with negligible overhead. Along with each attestation, the NCs send attestations and signed identity keys of each VM they are hosting. In this way, we batch the VM attestation process into a single step to reduce the number of requests to each NC for a VM attestation.

5.3 Scalable Proof Generation

Besides verifying attestations, the CV must also service requests for VM attestations made by the cloud users. Since a CV could receive a large number of request and generating attestations of the CV itself from a TPM is slow (around 900 msecs per attestation), we use *asynchronous attestations* to serve up these requests. We developed a mechanism to provide asynchronous attestations that was able to scale to very large client loads while still providing integrity proofs. By utilizing the proof constructions proposed, the cloud verifier can support over $7,000$ requests per second, as previously shown [15]. This demonstrates the CV can handle a large number of requests without degrading in performance and can be further improved with load balancing between multiple CVs.

6. FUTURE WORK

In this paper, we presented a method of deploying and verifying cloud applications while minimizing the amount of implicitly trusted cloud components. We introduced the notion of a cloud verifier that assesses the integrity of the cloud's VMMs for an application provider. We evaluated our system with a Tor cloud application and found verification caused minimal impact on performance. In future work, we plan to investigate automated integrity criteria generation and a more expressive model of integrity verification.

7. REFERENCES

[1] Arbaugh, W. A., Farber, D. J., and Smith, J. M. A Secure and Reliable Bootstrap Architecture. In *IEEE SP '97* (1997), IEEE Computer Society, p. 65.

[2] Biba, K. J. Integrity Considerations for Secure Computer Systems. Tech. Rep. MTR-3153, MITRE, April 1977.

[3] Chen, H., Li, N., and Mao, Z. Analyzing and Comparing the Protection Quality of Security Enhanced Operating Systems. In *NDSS* (2009).

[4] Clair, L. S., Schiffman, J., Jaeger, T., and McDaniel, P. Establishing and sustaining system integrity via root of trust installation. In *ACSAC* (Dec. 2007).

[5] Clark, D. D., and Wilson, D. R. A Comparison of Commercial and Military Computer Security Policies. *Security and Privacy 00* (1987), 184.

[6] Datta, A., Franklin, J., Garg, D., and Kaynar, D. A Logic of Secure Systems and its Application to Trusted Computing. In *IEEE SP '09* (May 2009).

[7] Eucalyptus. http://www.eucalyptus.com/.

[8] Forrester Research. Conventional Wisdom Is Wrong About Cloud IaaS. http://www.forrester.com.

[9] Howard, M., Pincus, J., and Wing, J. M. Measuring Relative Attack Surfaces. In *Proceedings of Workshop on Advanced Developments in Software and Systems Security* (2003).

[10] IDC. The Single Biggest Reason Public Clouds Will Dominate the Next Era of IT. http://blogs.idc.com/ie/?p=345.

[11] Jaeger, T., Sailer, R., and Shankar, U. PRIMA: Policy-Reduced Integrity Measurement Architecture. In *Proceedings of the 11th ACM SACMAT* (2006).

[12] Kauer, B. Oslo: improving the security of trusted computing. In *16th USENIX Security Symposium* (Berkeley, CA, USA, 2007), USENIX Association, pp. 1–9.

[13] Klein, G., et al. seL4: Formal Verification of an OS Kernel. In *SOSP '09* (2009), ACM, pp. 207–220.

[14] McCune, J. M., Perrig, A., and Reiter, M. K. Safe Passage for Passwords and Other Sensitive Data. In *NDSS* (Feb. 2009).

[15] Moyer, T., Butler, K., Schiffman, J., McDaniel, P., and Jaeger, T. Scalable Web Content Attestation. In *ACSAC '09* (2009).

[16] Nagarajan, A., Varadharajan, V., Hitchens, M., and Gallery, E. Property based attestation and trusted computing: Analysis and challenges. In *International Conference on Network and System Security* (Washington, DC, USA, 2009), IEEE Computer Society, pp. 278–285.

[17] Parno, B., McCune, J. M., and Perrig, A. Bootstrapping trust in commodity computers. In *IEEE SP '10* (May 2010).

[18] Ristenpart, T., Tromer, E., Shacham, H., and Savage, S. Hey, You, Get Off of my Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In *CCS '09*, ACM.

[19] Sailer, R., Zhang, X., Jaeger, T., and van Doorn, L. Design and Implementation of a TCG-based Integrity Measurement Architecture. In *USENIX-SS '04* (Aug. 2004), USENIX Association.

[20] Santos, N., Gummadi, K. P., and Rodrigues, R. Towards trusted cloud computing. In *Proceedings of USENIX HotCloud'09: Workshop on Hot Topics in Cloud Computing* (San Diego, CA, June 2009).

[21] Schiffman, J., Jaeger, T., and McDaniel, P. Network-based Root of Trust for Installation. Tech. Rep. Technical Report NAS-TR-0135-2010, Network and Security Research Center, June 2010.

[22] Schiffman, J., Moyer, T., Shal, C., Jaeger, T., and McDaniel, P. Justifying integrity using a Virtual Machine Verifier. In *ACSAC '09* (2009), ACSA.

[23] Shi, E., Perrig, A., and van Doorn, L. BIND: A Fine-Grained Attestation Service for Secure Distributed Systems. In *IEEE SP '05* (2005), IEEE Computer Society.

[24] Shrivastava, S. Satem: Trusted Service Code Execution across Transactions. In *SRDS '06* (2006), IEEE Computer Society, pp. 337–338.

[25] Smith, S. W. Outbound Authentication for Programmable Secure Coprocessors. In *ESORICS* (Oct. 2002).

[26] Vijayakumar, H., Jakka, G., Rueda, S., Schiffman, J., and Jaeger, T. Integrity Walls: Finding attack surfaces from mandatory access control policies. Tech. Rep. NAS-TR-0124-2010, NSRC, Feb. 2010.