Bootstrapping and Maintaining Trust in the Cloud *

Nabil Schear
MIT Lincoln Laboratory
nabil@ll.mit.edu

Patrick T. Cable II
Threat Stack, Inc.
pat@threatstack.com

Thomas M. Moyer
MIT Lincoln Laboratory
tmoyer@ll.mit.edu

Bryan Richard
MIT Lincoln Laboratory
bryan.richard@ll.mit.edu

Robert Rudd
MIT Lincoln Laboratory
robert.rudd@ll.mit.edu

ABSTRACT

Today's infrastructure as a service (IaaS) cloud environments rely upon full trust in the provider to secure applications and data. Cloud providers do not offer the ability to create hardware-rooted cryptographic identities for IaaS cloud resources or sufficient information to verify the integrity of systems. Trusted computing protocols and hardware like the TPM have long promised a solution to this problem. However, these technologies have not seen broad adoption because of their complexity of implementation, low performance, and lack of compatibility with virtualized environments. In this paper we introduce `keylime`, a scalable trusted cloud key management system. `keylime` provides an end-to-end solution for both bootstrapping hardware rooted cryptographic identities for IaaS nodes and for system integrity monitoring of those nodes via periodic attestation. We support these functions in both bare-metal and virtualized IaaS environments using a virtual TPM. `keylime` provides a clean interface that allows higher level security services like disk encryption or configuration management to leverage trusted computing without being trusted computing aware. We show that our bootstrapping protocol can derive a key in less than two seconds, we can detect system integrity violations in as little as 110ms, and that `keylime` can scale to thousands of IaaS cloud nodes.

*This material is based upon work supported by the Assistant Secretary of Defense for Research and Engineering under Air Force Contract No. FA8721-05-C-0002 and/or FA8702-15-D-0001. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Assistant Secretary of Defense for Research and Engineering.
© 2016 Massachusetts Institute of Technology. Delivered to the U.S. Government with Unlimited Rights, as defined in DFARS Part 252.227-7013 or 7014 (Feb 2014). Notwithstanding any copyright notice, U.S. Government rights in this work are defined by DFARS 252.227-7013 or DFARS 252.227-7014 as detailed above. Use of this work other than as specifically authorized by the U.S. Government may violate any copyrights that exist in this work.

ACM acknowledges that this contribution was authored or co-authored by an employee, or contractor of the national government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only. Permission to make digital or hard copies for personal or classroom use is granted. Copies must bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. To copy otherwise, distribute, republish, or post, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ACSAC '16, December 05-09, 2016, Los Angeles, CA, USA
© 2016 ACM. ISBN 978-1-4503-4771-6/16/12...$15.00
DOI: http://dx.doi.org/10.1145/2991079.2991104

1. INTRODUCTION

The proliferation and popularity of infrastructure-as-a-service (IaaS) cloud computing services such as Amazon Web Services and Google Compute Engine means more cloud tenants are hosting sensitive, private, and business critical data and applications in the cloud. Unfortunately, IaaS cloud service providers do not currently furnish the building blocks necessary to establish a trusted environment for hosting these sensitive resources. Tenants have limited ability to verify the underlying platform when they deploy to the cloud and to ensure that the platform remains in a good state for the duration of their computation. Additionally, current practices restrict tenants' ability to establish unique, unforgeable identities for individual nodes that are tied to a hardware root of trust. Often, identity is based solely on a software-based cryptographic solution or unverifiable trust in the provider. For example, tenants often pass unprotected secrets to their IaaS nodes via the cloud provider.

Commodity trusted hardware, like the Trusted Platform Module (TPM) [40], has long been proposed as the solution for bootstrapping trust, enabling the detection of changes to system state that might indicate compromise, and establishing cryptographic identities. Unfortunately, TPMs have not been widely deployed in IaaS cloud environments due to a variety of challenges. First, the TPM and related standards for its use are complex and difficult to implement. Second, since the TPM is a cryptographic co-processor and *not* an accelerator, it can introduce substantial performance bottlenecks (e.g., 500+ms to generate a single digital signature). Lastly, the TPM is a physical device by design and most IaaS services rely upon virtualization, which purposefully divorces cloud nodes from the hardware on which they run. At best, the limitation to physical platforms means that only the cloud provider would have access to the trusted hardware, not the tenants [17, 20, 31]. The Xen hypervisor includes a virtualized TPM implementation that links its security to a physical TPM [2, 10], but protocols to make use of the vTPM in an IaaS environment do not exist.

To address these challenges we identify the following desirable features of an IaaS trusted computing system:

- **Secure Bootstrapping** – the system should enable the tenant to securely install an initial root secret into each cloud node. This is typically the node's long term cryptographic identity and the tenant chains other secrets to it to enable secure services.

- **System Integrity Monitoring** – the system should allow the tenant to monitor cloud nodes as they oper-

ate and react to integrity deviations within one second.

- **Secure Layering (Virtualization Support)** – the system should support tenant controlled bootstrapping and integrity monitoring in a VM using a TPM in the provider's infrastructure. This must be done in collaboration with the provider in least privilege manner.

- **Compatibility** – the system should allow the tenant to leverage hardware-rooted cryptographic keys in software to secure services they already use (e.g., disk encryption or configuration management).

- **Scalability** – the system should scale to support bootstrapping and monitoring of thousands of IaaS resources as they are elastically instantiated and terminated.

Prior cloud trusted computing solutions address a subset of these features, but none achieve all. Excalibur [31] supports bootstrapping at scale, but does not allow for system integrity monitoring or offer full support for tenant trusted computing inside a VM (i.e., layering). Manferdelli et al. created a system that supports secure layering and bootstrapping, but does not support system integrity monitoring, is incompatible with existing cryptographic services, and has not demonstrated cloud scale operation [25]. Finally, the Cloud Verifier [34] enables system integrity measurement and cloud scalability but does not fully address secure layering or enable secure bootstrapping.

In this paper, we introduce `keylime`; an end-to-end IaaS trusted cloud key management service that supports all the above desired features. The key insight of our work is to utilize trusted computing to bootstrap identity in the cloud and provide integrity measurement to support revocation, but then allow high-level services that leverage these identities to operate independently. Thus, we provide a clean and easy to use interface that can integrate with existing security technologies (see Figure 1).

We introduce a novel bootstrap key derivation protocol that combines both tenant intent and integrity measurement to install secrets into cloud nodes. We then leverage the Cloud Verifier [34] pattern of Schiffman et al. to enable periodic attestation that automatically links to identity revocation. `keylime` supports the above with secure layering in both bare-metal and virtualized IaaS resources in a manner that minimizes trust in the cloud provider. We demonstrate the compatibility of `keylime` by securely enabling cloud provisioning with `cloud-init`[1], encrypted communication with IPsec, configuration management with Puppet[2], secret management with Vault[3], and storage with LUKS/dm-crypt encrypted disks. Unlike existing solutions [39, 25], these services don't need to be trusted computing aware, they just need to use an identity key and respond to key revocations.

Finally, we show that `keylime` can scale to handle thousands of simultaneous nodes and perform integrity checks on nodes at rates up to 2,500 integrity reports (quotes) verified per second. We present and evaluate multiple options for deploying our integrity measurement verifier both in the cloud, in a low-cost cloud appliance based on a Raspberry Pi, and on-premises. We show that the overhead of securely

[1]http://launchpad.net/cloud-init
[2]http://puppetlabs.com/
[3]http://hashicorp.com/blog/vault.html

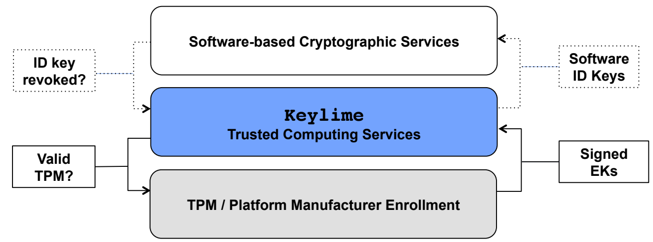

Figure 1: Interface between trusted hardware and existing software-based security services via the `keylime` trusted computing service layer.

provisioning a key using `keylime` takes less than two seconds. Finally, we find that our system can detect integrity measurement violations in as little as 110ms.

2. BACKGROUND

Trusted Computing The TPM provides the means for creating trusted systems that are amenable to system integrity monitoring. The TPM, as specified by the Trusted Computing Group (TCG)[4], is a cryptographic co-processor that provides key generation, protected storage, and cryptographic operations. The protected storage includes a set of Platform Configuration Registers (PCRs) where the TPM stores hashes. The TPM uses these registers to store *measurements* of integrity-relevant components in the system.

To store a new measurement in a PCR, the `extend` operation concatenates the existing PCR value with the new measurement, securely hashes[5] that value, and stores the resulting hash in the register. This hash chain allows a verifier to confirm that a set of measurements reported by the system has not been altered. This report of measurements is called an *attestation*, and relies on the `quote` operation, which accepts a random nonce and a set of PCRs. These PCRs can include measurements of the BIOS, firmware, boot loader, hypervisor, OS, and applications, depending on the configuration of the system. The TPM reads the PCR values, and then signs the nonce and PCRs with a key that is only accessible by the TPM. The key the TPM uses to sign quotes is called an attestation identity key (AIK). We denote a quote using $Quote_{AIK}(nonce, PCR_i : d_i ...)$ for a quote using AIK from the TPM with the associated $nonce$ and one or more optional PCR numbers PCR_i and corresponding data d_i that will be hashed and placed in PCR_i.

The TPM contains a key hierarchy for securely storing cryptographic keys. The root of this hierarchy is the Storage Root Key (SRK) which the owner generates during TPM initialization. The SRK in turn protects the TPM AIK(s) when they are stored outside of the TPM's nonvolatile storage (NVRAM). Each TPM also contains a permanent credential called the Endorsement Key (EK). The TPM manufacturer generates and signs the EK. The EK uniquely identifies each TPM and certifies that it is a valid TPM hardware device. The private EK never leaves the TPM, is never erased, and can only be used for encryption and decryption during AIK initialization to limit its exposure.

[4]http://trustedcomputinggroup.org
[5]TPM specification version 1.2 uses SHA-1 for measurements. TPM specification version 2.0 adds SHA-256 to address cryptographic weaknesses in SHA-1.

Integrity Measurement To measure a system component, the underlying component must be *trusted computing-aware*. The BIOS in systems with a TPM supports measurement of firmware and boot loaders. TPM-aware boot loaders can measure hypervisors and operating systems [22, 19, 29]. To measure applications, the operating system must support measurement of applications that are launched, such as the Linux Integrity Measurement Architecture [30, 21]. One limitation of approaches like IMA is the inability to monitor the *run-time state* of the applications. Nexus aims to address this limitation with a new OS that makes trusted computing a first-class citizen, and supports introspection to validate run-time state [37]. Several proposals exist for providing *run-time integrity monitoring* including LKIM [24] and DynIMA [8]. These systems ensure that a *running system* is in a known state, allowing a verifier to validate not only that what was loaded was known, but that it has not been tampered with while it was running.

In addition to operating system validation, others have leveraged trusted computing and integrity measurement to support higher-level services, such as protected access to data when the client is offline [23], or to enforce access policies on data [26]. Others have proposed mechanisms to protect the server from malicious clients, e.g., in online gaming [1], or applications from a malicious operating system [6, 7, 15]. However, these proposals do not account for the challenges of migrating applications to a cloud environment, and often assume existing infrastructure to support trusted computing key management.

IaaS Cloud Services In the IaaS cloud service model, users request an individual compute resource to execute their application. For example, users can provision physical hardware, virtual machines, or containers. In this paper, we refer to any of these tenant-provisioned IaaS resources as *cloud nodes*. Users provision nodes either by uploading a whole image to the provider or by configuring a pared-down base image that the provider makes available. Users often begin by customizing a provider-supplied image, then create their own images (using a tool like `Packer`[6]) to decrease the amount of time it takes for a node to become ready.

`cloud-init` is a standard cross-provider (e.g., Amazon EC2, Microsoft Azure...) mechanism that allows cloud tenants to specify bootstrapping data. It accepts a YAML-formatted description of what bootstrapping actions should be taken and supports plugins to take those actions. Examples of such actions include: adding users, adding package repositories, or running arbitrary scripts. Users of cloud computing resources at scale typically spawn new cloud instances using an application programming interface and pass along enough bootstrapping information to allow the instance to communicate with a configuration management platform (such as Chef[7] or Puppet, etc.) for further instance-specific configuration. These bootstrapping instructions are not encrypted, meaning that a provider could intercept secrets passed via the bootstrapping instructions. In our research, we found that organizations will either (a) send an *unprotected* pre-shared key for Puppet in their `cloud-init` bootstrapping actions, or (b) rely on some weaker method of proving identity such as going off the certificate's common name (hostname).

[6]https://www.packer.io/

[7]https://www.chef.io/chef/

3. DESIGN

To address the limitations of current approaches, we consider the union of trusted computing and IaaS to provide a hardware root-of-trust that tenants leverage to establish trust in the cloud provider's infrastructure and in their own systems running on that infrastructure. This section considers the threats that `keylime` addresses, and how to leverage existing trusted computing constructs in a virtualized environment while limiting complexity and overhead.

3.1 Threat Model

Our goal is to minimize trust in the cloud provider and carefully account for all concessions we must make to enable trusted computing services. We assume the cloud provider is semitrusted, i.e., they are organizationally trustworthy but *are still* susceptible to compromise or malicious insiders. We assume the cloud provider has processes, technical controls, and policy in place to limit the impact of such compromise from spreading across their entire infrastructure. Thus, in the semitrusted model, we assume that some fraction of the cloud provider's resources may be under the control of the adversary (e.g., a rogue system administrator may control a subset of racks in an IaaS region).

Specifically, we assume that the adversary can monitor or manipulate compromised portions of the cloud network or storage arbitrarily. We assume that the adversary may *not* physically tamper with any host's (e.g., hypervisor or bare metal node) CPU, bus, memory, or TPM[8]. In virtualized environments, the security of our system relies upon keeping cryptographic keys in VM memory. Therefore, we assume that the provider does not purposefully deploy a hypervisor with the explicit capability to spy on tenant VM memory (e.g., Ether [9]). We assume that TPM and system manufacturers have created the appropriate endorsement credentials and have some mechanism to test their validity (i.e., signed certificates)

Finally, we assume that the attacker's goal is to obtain persistent access to a tenant system in order to steal, disrupt, or deny the tenant's data and services. To accomplish persistence the attacker must modify the code loading or running process. We assume that such modifications would be detected by load-time integrity measurement of the hypervisor or kernel [19], runtime integrity measurement of the kernel [24], and integrity measurement of applications [30].

3.2 Architecture

To introduce the architecture of `keylime` we first describe a simplified architecture for managing trusted computing services for a single organization, or cloud tenant, without virtualization. We then expand this simplified architecture into the full `keylime` architecture, providing extensions that allow layering of provider and tenant trusted computing services and supporting multiple varieties of IaaS execution isolation (i.e., bare metal, virtual machines, or containers). Figure 3 depicts the full system architecture with layering.

The first step in bootstrapping the architecture is to create a tenant-specific registrar. The registrar stores and certifies the public AIKs of the TPMs in the tenant's infrastructure. In the simplified architecture, the tenant registrar can be hosted outside the cloud in the tenant's own infrastructure

[8]This is similar to the threat model assumed by the TPM, where physical protections are not a strict requirement to be compliant with the specification.

Table 1: Keys used by keylime and their purpose.

Key	Type	Purpose
EK	RSA 2048	Permanent TPM credential that identifies the TPM hardware.
SRK	RSA 2048	TPM key that protects TPM created private keys when they are stored outside the TPM.
AIK	RSA 2048	TPM key used to sign quotes.
K_e	AES-256	Enrollment key created by the registrar and used to activate the AIK.
K_b	AES-256	Bootstrap key the tenant creates. `keylime` securely delivers to the node.
U, V	256bit random	Trivial secret shares of K_b, derived with random 256bit V: $U = K_b \oplus V$.
NK	RSA 2048	Non-TPM software key used to protect secret shares U, V in transit.

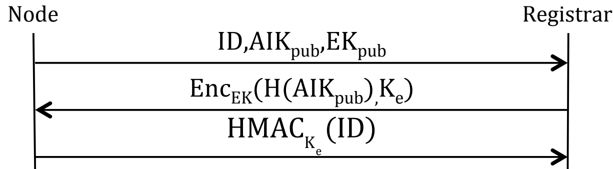

Figure 2: Physical node registration protocol.

or could be hosted on a physical system in the cloud. The registrar is only a trust root and does not store any tenant secrets. The tenant can decide to trust the registrar only after it attests its system integrity. Since the registrar is a simple a component with static code, verifying its integrity is straight forward.

To create a registrar, we can leverage existing standards for the creation and validation of AIKs by creating a TCG Privacy CA [38]. To avoid the complexity of managing a complex PKI and because there's no need for privacy within a single tenant's resources, we created a registrar that simply stores valid TPM AIK public keys indexed by node UUID. Clients request public AIKs from the registrar through a server-authenticated TLS channel.

To validate the AIKs in the registrar, we developed a TPM-compatible enrollment protocol (see Figure 2). The node begins by sending its ID and standard TPM credentials (EK_{pub}, AIK_{pub}) to the registrar. The registrar then checks the validity of the TPM EK with the TPM manufacturer. Importantly, the generation and installation of the EK by the TPM manufacturer roots the trust upon which the rest of our system relies. If the EK is valid, the registrar creates an ephemeral symmetric key K_e and encrypts it along with a hash of the public AIK, denoted $H(AIK_{pub})$, with the TPM EK_{pub}. The node uses the `ActivateIdentity` TPM command to decrypt K_e. The TPM will only decrypt K_e if has EK_{priv} and if it has AIK_{priv} corresponding to $H(AIK_{pub})$. The nodes uses an HMAC to prove that it can decrypt the ephemeral key K_e. The registrar then marks that AIK as being valid so that it can be used to validate quotes.

The core component of `keylime` is an out of band cloud verifier (CV) similar to the one described by Schiffman et al. [34]. Each cloud organization will have at least one CV that is responsible for verifying the system state of the organization's IaaS resources. The tenant can host the CV in the IaaS cloud or on-premises at their own site (we give options for tenant registrar and CV deployment in Section 3.2.1). The CV relies upon the tenant registrar for validating that the AIKs used to sign TPM quotes are valid, or more specifically, that the AIKs are recognized by the tenant as being associated with machines (physical or virtual) owned by the tenant. The registrar, CV, and cloud node service are the only components in `keylime` that manage and use keys and public key infrastructures associated with the TPM.

The CV participates in a three party key derivation protocol (we describe in detail in Section 3.2.2) where the CV and tenant cooperate to derive a key, K_b, at the cloud node to support initial storage decryption. The tenant uses K_b to protect tenant secrets and trust relationships. The tenant can use this key to unlock either its disk image or to unlock tenant-specific configuration provided by `cloud-init`.

This protocol is akin to the method by which a user can use the TPM to decrypt his or her disk in a laptop. To allow the decryption key to be used to boot the laptop, the user must enter a password (demonstrating the user's intent) and TPM PCRs must match a set of whitelisted integrity measurements (demonstrating the validity of the system that will receive the decryption key). In an IaaS cloud environment, there is neither a trusted console where a user can enter a password nor is there a way to pre-seed the TPM with the storage key or measurement whitelist. Our protocol uses secret sharing to solve these problems by relying externally upon the CV for integrity measurement and by having the tenant directly interact with the cloud node to demonstrate intent to derive a key. The protocol then extends beyond bootstrapping to enable continuous system integrity monitoring. The CV periodically polls each cloud node's integrity state to determine if any runtime policies have been violated. The frequency with which the CV requests and verifies each node's integrity state will define the latency between an integrity violation and detection.

To cleanly link trust and integrity measurement rooted in the TPM to higher-level services, we create a parallel software-only PKI and a simple service to manage it. The goal is to remove the need to make each service trusted computing-aware, e.g. integrating Trusted Network Connect into StrongSwan[9]. We refer to this parallel software-only service as the software CA. To bootstrap this service, we use the key derivation bootstrap protocol to create a cloud node to host the service. Since the bootstrap key derivation protocol ensures that the node can only derive a key if the tenant authorizes it and if the node's integrity state is approved, we can encrypt the private key for the software CA and pass it to the node upon provisioning. Once established, we can then start other cloud nodes and securely pass them keys signed by this CA. The linkage to the hardware root of trust, the secure bootstrapping of relevant keys, and user intent to create new resources are again ensured using the

[9]https://wiki.strongswan.org/projects/strongswan/wiki/TrustedNetworkConnect

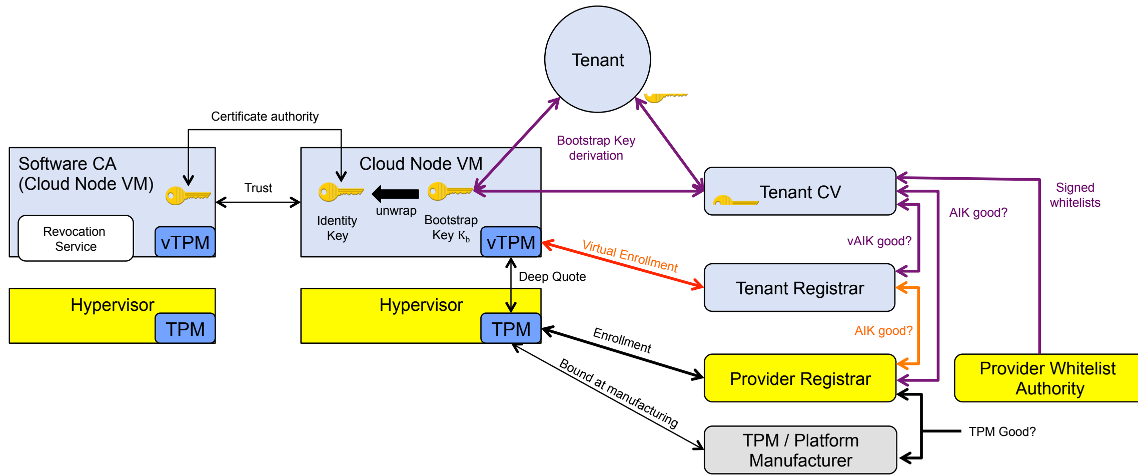

Figure 3: Layered `keylime` trusted computing architecture.

bootstrap key derivation protocol. Once established, standard tools and services like IPsec or Puppet can now directly use the software CA identity credentials that each node now possesses.

To complete the linkage between the trusted computing services and the software identity keys, we need a mechanism to revoke keys in the software PKI when integrity violations occur in the trusted computing layer. The CV is responsible for notifying the software CA of these violations. The CV includes metadata about the nature of the integrity violation, which allows the software CA to have a response policy. The software CA supports standardized methods for certificate revocation like signed revocation lists or by hosting an OCSP responder. To support push notifications of failures, the software CA can also publish signed notifications to a message bus. This way services that directly support revocation actions can subscribe to notifications (e.g., to trigger a re-key in a secret manager like Vault).

3.2.1 Layering Trust

We next expand this architecture to work across the layers of virtualization common in today's IaaS environments. Our goal is to create the architecture described previously that cleanly links common security services to a trusted computing layer in a *cloud tenant's* environment. Thus, in a VM hosting environment like Amazon EC2 or OpenStack, we aim to create trusted computing enabled software CAs and tenant nodes inside of virtual machine instances. Note that, in a bare-metal provisioning environment like IBM Softlayer[10], HaaS [13], or OpenStack Ironic[11], we can directly utilize the simplified architecture where there is no trust layering.

We observe that IaaS-based virtual machines or physical hosts all provide a common abstraction of isolated execution. Each form of isolated execution in turn needs a root of trust on which to build trusted computing services. Due to the performance and resource limitations of typical TPMs (e.g., taking 500 or more milliseconds to generate a quote, and only supporting a fixed number of PCRs), direct multiplexed use of the physical TPM will not scale to the numbers of virtual machines that can be hosted on a single modern system. As described by Berger et al. [2] and as implemented in Xen [10], we utilize a virtualized implementation of the TPM. Each VM has its own software TPM (called a vTPM) whose trust is in turn rooted in the hardware TPM of the hosting system. The vTPM is isolated from the guest that use it, by running in a separate Xen domain.

The vTPM interface is the same as a hardware TPM. The only exception to this, is that the client can request a *deep-quote*[12] that will get a quote from the hardware TPM in addition to getting a quote from the vTPM. These quotes are linked together by including a hash of the vTPM quote and nonce in the hardware TPM quote's nonce. Deep quotes suffer from the slow performance of hardware TPMs, but as we'll show in later this section, we can limit the use of deep quotes while still maintaining reasonable performance and scale and maintaining security guarantees.

To assure a chain of trust that is rooted in hardware, we need the IaaS provider to replicate some of the trusted computing service infrastructure in their environment and allow the tenant trusted computing services to query it. Specifically, the provider must establish a registrar for their infrastructure, must publish an up-to-date signed list of the integrity measurements of their infrastructure (hosted by a whitelist authority service), and may even have their own CV. The tenant CV will interact with the whitelist authority service and the provider's registrar to verify deep quotes collected by the infrastructure.

Despite the fact that most major IaaS providers run closed-source hypervisors and would provide opaque integrity measurements [16], we find there is still value in verifying the integrity of the provider's services. By providing a known-good list of integrity measurements, the provider is committing to a version of the hypervisor that will be deployed widely across their infrastructure. This prevents a targeted attack where the a single hypervisor is replaced with a malicious version designed to spy on the tenant (e.g., the provider is coerced by a government to monitor a specific cloud ten-

[10]http://www.softlayer.com
[11]https://wiki.openstack.org/wiki/Ironic

[12]We use similar notation for quotes as we do for deep quotes, $DeepQuote_{AIK,vAIK}(nonce, PCR_i : d_i, vPCR_j : d_j)$, except that PCRs may be from both physical and virtual sets of PCRS. We use virtual PCR #16 to bind data.

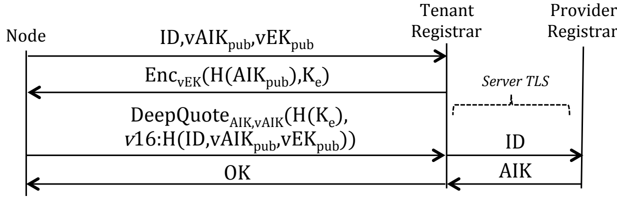

Figure 4: Virtual node registration protocol.

ant). Thus, an attacker must subvert both the code loading process on all the hypervisors and the publishing and signing process for known-good measurements. In our semitrusted threat model, we assume the provider has controls and monitoring which limit the ability of a rogue individual to accomplish this.

As in Section 3.2, we begin with the establishment of a tenant registrar and cloud verifier. There are multiple options for hosting these services securely: 1) in a bare metal IaaS instance with TPM, 2) on-tenant-premises in tenant-owned hardware, 3) in a small trusted hardware appliance deployed to the IaaS cloud provider, and 4) in an IaaS virtual machine. The first three of these options rely upon the architecture and protocols we've already discussed. The last option requires the tenant to establish an on-tenant-premises CV and use that to bootstrap the tenant registrar and CV. This on-tenant-premises CV identifies and checks the integrity of the tenant's virtualized registrar and CV, who then in turn are responsible for the rest of the tenant's virtualized infrastructure.

The primary motivations for a tenant choosing between these options are the detection latency for integrity violations, scale of IaaS instances in their environment, bandwidth between the tenant and the IaaS cloud, and cost. Option 1 provides maximum performance but at higher cost. Option 2 will by limited by bandwidth and requires more costs to maintain resources outside of the cloud. Option 3 is a good trade-off between cost and performance for a small cloud tenant with only tens of nodes or who can tolerate a longer detection latency. Finally, Option 4 provides compatibility with current cloud operations, good performance and scalability, and low cost at the expense of increased complexity. In Section 5, we examine the performance trade-offs of these options including a low-cost registrar and CV appliance (Option 3) we implemented on a Raspberry Pi.

Once we have created the tenant registrar and CV, we can begin securely bootstrapping nodes into the environment. As before, the first node to establish is a virtualized software CA and we do this by creating a private signing key offline and protecting it with a key that will be derived by the bootstrap key derivation protocol. The following process will be the same for all tenant cloud nodes. When a node boots, it will get a vTPM from the IaaS provider.

The process of enrolling a vTPM into the tenant registrar needs to securely associate the vTPM credentials, e.g., $(vEK, vAIK)$, with a physical TPM in the provider's infrastructure (see Figure 4). The tenant registrar cannot directly verify the authenticity of the vEK because it is virtual and has no manufacturer. To address this, we use a deep quote to bind the virtual TPM credentials to a physical TPM AIK.

The vTPM enrollment protocol begins like the physical TPM enrollment protocol by sending $ID, (EK_{pub}, AIK_{pub})$ to the tenant registrar. The tenant registrar then returns $Enc_{vEK}(H(AIK_{pub}), K_e)$ without additional checks. The virtual node then decrypts K_e using `ActivateIdentity` function of its vTPM. The node next requests a deep quote using a hash of K_e as the nonce to both demonstrate the freshness of the quote and knowledge of K_e to the tenant registrar. It also uses virtual PCR #16 to bind the vTPM credentials and ID to the deep quote. Upon receiving the deep quote, the tenant registrar asks the provider registrar if the AIK from the deep quote is valid. The tenant registrar also requests the latest valid integrity measurement whitelists from the provider. Now the tenant registrar can check the validity of the deep quote's signature, ensure that the nonce is $H(K_e)$, confirm the binding data in PCR #16 matches what was provided in the previous step, and check the physical PCRs values in the deep quote against the provider's whitelist. Only if the deep quote is valid will the tenant registrar mark the $vAIK$ as being valid.

When considering the cost of performing a deep quote, the provider must carefully consider the additional latency of the physical TPM. Deep quotes provide a link between the vTPM and the physical TPM of the machine, and new enrollments should always include deep quotes. When considering if deep quotes should be used as part of periodic attestation, we must understand what trusted computing infrastructure the provider has deployed. If the provider is doing load time integrity only (e.g., secure boot), then deep quotes will only reflect the one-time binding at boot between the vTPM and the physical TPM and the security of the vTPM infrastructure. If the provider has runtime integrity checking of their infrastructure, there is value in the tenant performing periodic attestation using deep quotes. In the optimal deployment scenario, the provider can deploy `keylime` and provide tenants with access to the integrity state of the hypervisors that host tenant nodes. To limit the impact of slow hardware TPM operations, the provider can utilize techniques like batch attestation where multiple deep quote requests from different vTPMs can be combined into a single hardware TPM operation [28, 31].

3.2.2 Key Derivation Protocol

We now introduce the details of our bootstrap key derivation protocol (Figure 5). The goal of this protocol is for the cloud tenant to obtain key agreement with a cloud node they have provisioned in an IaaS system. The protocol relies upon the CV to provide integrity measurement of the cloud node during the protocol. The tenant also directly interacts with the cloud node to demonstrate their intent to spawn that resource and allow it to decrypt sensitive contents. However, the tenant does not directly perform integrity measurement. This separation of duties is beneficial because the attestation protocols may operate in parallel and it simplifies deployment by centralizing all integrity measurement, white lists, and policy in the CV.

To begin the process, the tenant generates a *fresh* random symmetric encryption key K_b. The cloud tenant uses AES-GCM to encrypt the sensitive data to pass to the node d with K_b, denoted $Enc_{K_b}(d)$. The tenant then performs trivial secret sharing to split K_b into two parts U, which the tenant will retain and pass directly to the cloud node and V, which the tenant will share with the CV to provide to the node upon successful verification of the node's integrity state. To obtain these shares the tenant generates a secure random

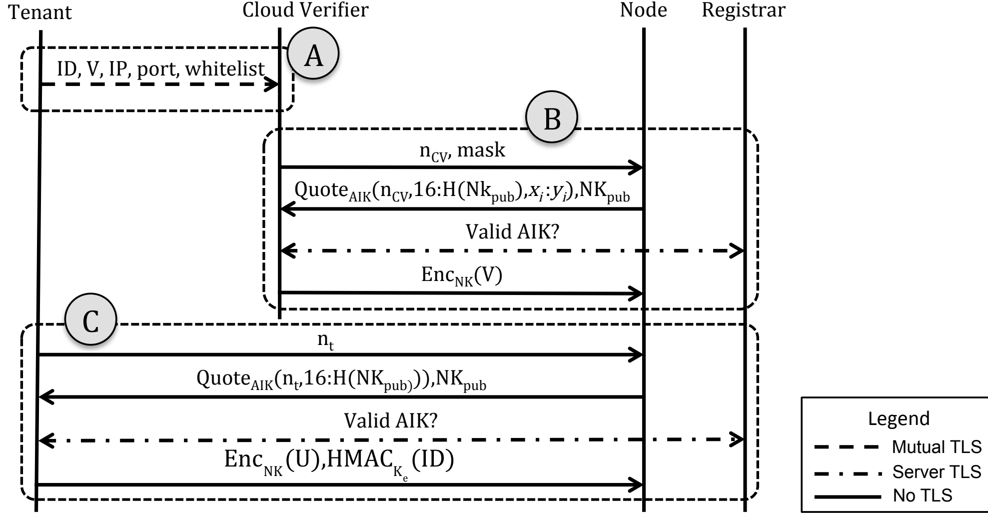

Figure 5: Three Party Bootstrap Key Derivation Protocol.

value V the same length as K_b and computes $U = K_b \oplus V$.

In the next phase of the protocol, the tenant requests the IaaS provider to instantiate a new resource (i.e., a new virtual machine). The tenant sends $Enc_{K_b}(d)$ to the provider as part of the resource creation. The data d may be configuration metadata like a `cloud-init` script[13]. Upon creation, the provider returns a unique identifier for the node *uuid* and an IP address at which the tenant can reach the node.

After obtaining the node *uuid* and IP address, the tenant notifies the CV of their intent to boot a cloud node (see area A in Figure 5). The tenant connects to the CV over a secure channel, such as mutually authenticated TLS, and provides v, *uuid*, node IP, and a TPM policy. The TPM policy specifies a white list of acceptable PCR values to expect from the TPM of the cloud node. At this point the CV and tenant can begin the attestation protocol in parallel.

The attestation protocol of our scheme is shared between the interactions of the CV and the cloud node (B) and that of the tenant and the cloud node (C) with only minor differences between them (Figure 5). The protocol consists of two message volleys the first for the initiator (either CV or tenant) to request a TPM quote and the second for the initiator to provide a share of K_b to the cloud node upon successful validation of the quote. Since we use this protocol to bootstrap keys into the system, there are no existing software keys with which we create a secure channel. Thus, this protocol must securely transmit a share of K_b over an untrusted network. We accomplish this by creating an ephemeral asymmetric key pair on the node, denoted NK, outside of the TPM[14]. As in Section 3.2.1, we use PCR #16's value in a TPM quote to bind NK to the identity of the TPM thereby authenticating NK. The initiator can then encrypt its share of K_b using NK_{pub} and securely return it to the cloud node.

The differences in the attestation protocol between CV and tenant arise in how each validates TPM quotes. Because we wish to centralize the adjudication of integrity measurements to the CV, the TPM quote that the tenant requests *only* verifies the identity of the cloud node's TPM and doesn't include any PCR hashes. Since the tenant generates a fresh K_b for each cloud node, we are not concerned with leaking U to a node with invalid integrity state. Furthermore, because V is only one share of K_b, the CV cannot be subverted to decrypt resources without user intent.

We now describe the attestation protocol in detail. The initiator first sends a fresh nonce (n_t for the tenant as in B from Figure 5 and n_{CV} for the cloud verifier as in C from Figure 5) to the cloud node along with a mask indicating which PCRs the cloud node should include in its quote. The CV sets the mask based on TPM policy exchanged earlier and the tenant creates an empty mask. We extend a hash of NK_{pub} into a freshly reset PCR #16. The initiator requests a quote from the TPM with the given PCR mask. The node then returns $Quote_{AIK}(n, 16 : H(NK_{pub}), x_i, : y_i), NK_{pub}$ to the initiator. Additional PCR numbers x_i and values y_i are only included in the quote returned to the cloud verifier based on the TPM policy it requested. During the protocol to provide U, the tenant also supplies $HMAC_{K_b}(ID)$ to the node. This provides the node with a quick check to determine if K_b is correct.

The initiator then confirms that the AIK is valid according to the tenant registrar over server authenticated TLS. If the initiator is the CV, then it will also check the other PCRs to ensure they are valid according to the tenant-specified whitelist. If the node is virtual, then the quote to the CV will also include a deep quote of the underlying hardware TPM. The CV will in turn validate it as described in the previous section. Upon successful verification, the initiator can then return their share of K_b. Thus, the tenant sends $Enc_{NK}(U)$ and the cloud verifier sends $Enc_{NK}(V)$ to the node. The cloud node can now recover K_b and proceed with

[13]Because K_b may not be re-used in our protocol, the cost of re-encrypting large disk images for each node may be prohibitive. We advocate for encrypting small sensitive data packets like a `cloud-init` script, and then establish local storage encryption with ephemeral keys.

[14]NK could also be generated and reside inside the TPM. However, since it is ephemeral, is only used for transport security and it is authenticated by the TPM using the quote, we found the added complexity of also storing it in the TPM unneeded.

the boot/startup process.

The cloud node does not retain K_b or V after decryption of d. To support node reboot or migration, the cloud node stores U in the TPM NVRAM to avoid needing the tenant to interact again. After rebooting, the node must again request verification by the CV to obtain V and re-derive K_b. If migration is allowed, the provider must take care to also securely migrate vTPM state to avoid losing U.

4. IMPLEMENTATION

We implemented `keylime` in approximately 5,000 lines of Python in four components: registrar, node, CV, and tenant. We use the IBM Software Trusted Platform module library [18] to directly interact with the TPM rather than going through a Trusted Software Stack (TSS) like Trousers. The registrar presents a REST-based web service for enrolling node AIKs. It also supports a query interface for checking the keys for a given node UUID. The registrar use HMAC-SHA384 to check the node's knowledge of K_e during registration.

The node component runs on the IaaS machine, VM, or container and is responsible for responding to requests for quotes and for accepting shares of the bootstrap key K_b. It provides an unencrypted REST-based web service for these two functions.

To support vTPM operations, we created a service the IaaS provider runs to manage hardware TPM activation and vTPM creation/association. This service runs in a designated Xen domain and has privileges to interact with the Xen `vtpmmgr` domain [11]. We then implemented a utility for the deep quote operation. Since the Xen vTPM implementation does not directly return the PCR values from the virtual TPM (i.e., the shallow quote) during a deep quote, we chose to first execute a shallow quote, hash its contents with the provided nonce, and place them in the nonce field of the deep quote. This operation cryptographically binds them together. This operation is not vulnerable to man-in-the-middle attack since there is no other interface to directly manipulate the nonce of a deep quote [36]. We then return both the shallow and deep quotes and require the verifier checks both signatures and sets of PCR values.

The cloud verifier hosts a TLS-protected REST-based web service for control. Tenants add and remove nodes to be verified and also query their current integrity state. Upon being notified of a new node, the CV enqueues metadata about the node onto the `quote_request` queue where a configurable pool of worker processes will then request a deep quote from the node. Upon successful verification of the quote, the CV will use an HTTP POST to send V to the node. The CV uses PKCS#1 OAEP and with RSA 2048 keys to protect shares of K_b in transit.

The tenant generates a random 256-bit AES key and encrypts and authenticates the bootstrap data using AES with Galois Counter Mode [27]. The tenant uses trivial XOR-based secret sharing to split K_b into V and U. The tenant executes a simplified version of the same protocol that the CV uses. The tenant checks with the registrar to determine if the quote signing AIK is valid and owned by the tenant.

Upon receiving U and V, the node can then combine them to derive K_b. To limit the impact of rogue CVs or tenants connecting to the node's unauthenticated REST interface, the node stores all received U and V values and iteratively tries each combination to find the correct K_b. Once the node has correctly derived K_b, it mounts a small in-memory file system using `tmpfs` and writes the key there for other applications to access.

4.1 Integration

While the key derivation protocol of `keylime` is generic and can be used to decrypt arbitrary data, we believe the most natural cloud use-case for it is to decrypt a small IaaS node-specific package of data. To enable this use-case we have integrated `keylime` with the `cloud-init` package, the combination we call `trusted-cloud-init`. As described in Section 2, `cloud-init` is widely adopted mechanism to deploy machine-specific data to IaaS resources. To integrate `keylime` and `cloud-init`, we patched `cloud-init` to support AES-GCM decryption of the `user-data` (where `cloud-init` stores tenant scripts and data). We modified the upstart system in Ubuntu Linux to start the `keylime` node service before `cloud-init`. We then configure `cloud-init` to find the key that `keylime` creates in the `tmpfs` mounted file system. After successful decryption, `cloud-init` deletes the key and scrubs it from memory.

To support applications that need node identities that do not manage their own PKIs, we implemented a simple software CA. The tenant provisions the software CA by creating the CA private key offline and delivering it to a new node using `trusted-cloud-init`. We also deliver certificates to the software CA that allow it and the tenant to mutually authenticate each other via `trusted-cloud-init`. To demonstrate the clean separation between the trusted computing layer and the software key management layer, we use the ZMQ Curve secure channel implementation [14]. This system uses an elliptic curve cryptography scheme dissimilar from the cryptographic algorithms, keys, and other techniques the TPM uses.

To enroll a new node, the tenant first generates a node identity key pair using the software CA client. The software CA supports a plugin architecture that allows the tenant to specify what type of key pairs to create (e.g., X.509 RSA 2048). The tenant then connects securely to the software CA over ZMQ and gets the node's identity certificate signed. The tenant can now provision a new node with this identity using `trusted-cloud-init`. The software CA also supports receiving notifications from the CV if a node later fails integrity measurement.

To support transparent integration with an IaaS platform, we patched OpenStack Nova and libvirt to support the creation of companion vTPM Xen domains for each user created instance. We link the OpenStack UUID to the `keylime` provider registrar. We then implemented a wrapper for the OpenStack Nova command line interface that enables `trusted-cloud-init`. Specifically, our wrapper intercepts calls to `nova boot` and automatically encrypts the provided `user-data` before passing it to OpenStack. It then calls the `keylime` tenant, which begins the bootstrapping protocol. This allows OpenStack users to transparently use `keylime` and `cloud-init` without needing to fully trust the OpenStack provider not to tamper or steal the sensitive contents of their `user-data`.

4.2 Demonstration Applications

We next describe how `keylime` can securely bootstrap and handle revocation for existing non-trusted computing-aware applications and services common to IaaS cloud de-

ployments.

IPsec To enable secure network connectivity similar to TNC [39], we implemented `trusted-cloud-init` scripts to automatically encrypt all network traffic between a tenant's IaaS resources. The scripts use the OpenStack API for IP address information and then build configurations for the Linux IPsec stack and `raccoon`[15]. This configuration is also easily extensible to a TLS-based VPN like OpenVPN[16].

Puppet To enable secure system configuration management, we integrated `keylime` with Puppet We do so by generating the signed RSA keys that Puppet uses to communicate with the Puppet master using the Software CA process described previously. These steps bypass the need to either use the insecure `autosign` option in the Puppet master to blindly accept new nodes or to have an operator manually approve/deny certificate signing requests from new nodes. To support continuous attestation and integrity measurement, we implemented a plug-in for the CV that notifies the tenant's Puppet master when a node fails its integrity measurements. The master can then revoke that node's access to check-in and receive the latest configuration data.

Vault While tools like Puppet are often used to provision secrets and keys, tenant operators can instead use a dedicated secret management system that supports the full lifecycle of cryptographic keys directly. To demonstrate this, we have integrated `keylime` with Vault, a cloud-compatible secret manager. Like Puppet, we use the Software CA to provision RSA certificates for each node and configure Vault to use them. We also implemented a revocation plugin for the CV that notifies Vault to both revoke access to a node that fails integrity measurement and to re-generate and redistribute any keys to which that node had access.

LUKS Finally, to demonstrate our ability to provision secrets instead of cryptographic identities, we implemented a `trusted-clout-init` script that provides the key to unlock an encrypted volume on boot.

5. EVALUATION

In this section we evaluate the overhead and scalability of `keylime` in a variety of scenarios. We ran our experiments on a private OpenStack cluster, a Xen host, and a Raspberry Pi. In OpenStack, we used standard instance flavors where the m1.small has 1 vCPU, 2GB RAM, and a 20GB disk, and the m1.large has 4 vCPUs, 8GB RAM, an 80GB disk. We used Ubuntu Linux 14.10 as the guest OS in OpenStack instances. The Xen host had one Xeon E5-2640 CPU with 6 cores at 2.5Ghz, 10Gbit NIC, 64 GB RAM, a WinBond TPM, and ran Xen 4.5 on Ubuntu Linux 15.04. The Raspberry Pi 2 had one ARMv7 with 4-cores at 900Mhz, 1GB RAM, 100Mbit NIC, and ran Raspbian 7. We ran each of the following experiments for 1-2 minutes and present averages of the performance we observed.

5.1 TPM Operations

We first establish a baseline for the performance of TPM operations with the IBM client library, our Python wrapper code, the Xen virtual TPM, and the physical TPM. We benchmarked both TPM quote creation and verification on the Xen host (Table 2). We collected the physical TPM measurements on the same system with a standard (non-Xen) kernel. We collected both vTPM quote and deep quote measurements from a domain running on Xen. As expected, operations that require interaction with the physical TPM are slow. Verification times, even for deep quotes that include two RSA signature verifications, are comparatively quick.

Table 2: Average TPM Operation Latency (ms).

	TPM	vTPM	Deep quote
Create Quote	725	68.5	1390
Check Quote	4.64	4.64	5.33

5.2 Key Derivation Protocol

We next investigate the CV latency of different phases of our protocol. In Figures 6 and 7, we show the averaged results of hundreds of trials of the CV with 100 vTPM equipped VMs. Each operation includes a full REST interaction along with the relevant TPM and cryptographic operations. We also benchmarked the latency of the protocol phases emulating zero latency from the TPM (Null TPM). This demonstrates the minimum latency of our CV software architecture including the time required to verify quotes. The results from the Null TPM trials indicate that our network protocol and other processing impose very little additional overhead, even on the relatively modestly powered Raspberry Pi. The bare metal system had a slightly larger network RTT to the nodes it was verifying causing it to have a higher latency than the less powerful m1.large.

In Figure 7, we see that latency for the quote retrieval process is primarily affected by slow TPM operations and is comparable to prior work [31]. The bootstrapping latency is the sum of the latencies for retrieving a quote and providing V. We find the bootstrapping latency for bare metal and virtual machines to be approximately 793ms and 1555ms respectively. Virtual nodes doing runtime integrity measurement after bootstrapping benefit from much lower latency for vTPM operations. Thus, for a virtual machine with a vTPM, `keylime` can detect integrity violations in as little as 110ms. The detection latency for a system with a physical TPM (781ms for our Xen host) is limited by the speed of the physical TPM at generating quotes.

5.3 Scalability of Cloud Verifier

Next we establish the maximum rate at which the CV can get and check quotes for sustained integrity measurement. This will define the trade-off between the number of nodes a single CV can handle and the latency between when an integrity violation occurs and the CV detects it. Since the CV quote checking process is a simple round robin check of each node, it is easy to parallelize across multiple CVs further enhancing scalability. We emulate an arbitrarily large population of real cloud nodes using a fixed number test cloud nodes. These test cloud nodes emulate a zero latency TPM by returning a pre-constructed quote. This way the test nodes appear like a larger population where the CV will never have to block for a lengthy TPM operation to complete. We found that around 500 zero latency nodes were sufficient to achieve the maximum quote checking rate.

We show the average number of quotes verified per second for each of our CV deployment options in Figure 8. Because of our process-based worker pool architecture, the primary factor affecting CV scalability is the number of cores and

[15]http://ipsec-tools.sourceforge.net/
[16]https://openvpn.net/

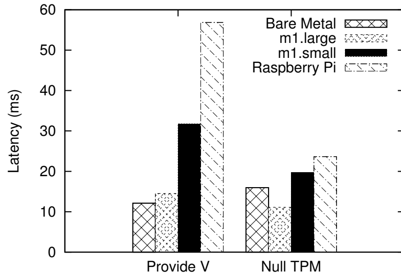

Figure 6: Latency of keylime bootstrapping protocol.

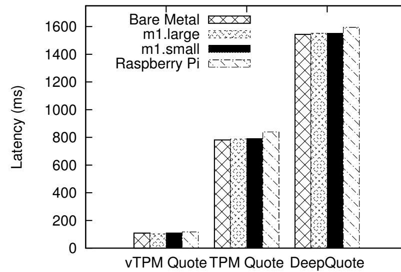

Figure 7: Latency of TPM operations in bootstrapping protocol.

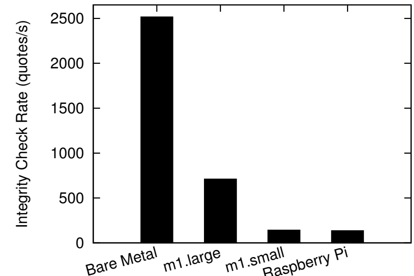

Figure 8: Maximum CV quote checking rate of keylime.

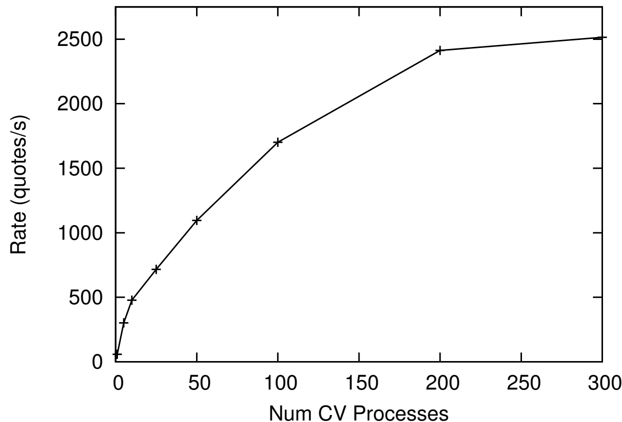

Figure 9: Scaling the CV on bare metal

RAM available. These options provide a variety of choices for deploying the CV. For small cloud tenants a low cost VM or inexpensive hardware appliance can easily verify hundreds of virtual machines with moderate detection latency (5-10s). For larger customers, a well-resourced VM or dedicated hardware can scale to thousands with similar latency. For high security environments, all options can provide subsecond detection and response time. In future work, we plan to implement a priority scheme that allows the tenant to set the rate of verifications for different classes of nodes.

We next show how our CV architecture can scale by adding more cores and parallelism. We use the bare metal CV and show the average rate of quotes retrieved and checked per second for 500 test nodes in Figure 9. We see linear speedup until we exhaust the parallelism of the host CPU and the concurrent delay of waiting for many cloud nodes. This performance represents a modest performance improvement over Schiffman et al.'s CV which was able to process approximately 2,000 quotes per second on unspecified hardware [34] and a substantial improvement over the Excalibur monitor's ability to check approximately 633 quotes per second [31].

5.4 On-Premises Cloud Verifier

Finally, we investigate the performance of the CV when hosted at the tenant site away from the cloud. We show the results of our bare metal system's quote verification rate and the bandwidth used for a variety of network delays we inserted using the `comcast`[17] tool in Table 3. These results show that it is possible to run the CV on-premises at the tenant site at the cost of a reduction in quote checking rate (and therefore detection latency) and several Mbit/s of bandwidth. As such, we recommend the highest performance and lowest cost option is to run the CV in the cloud alongside the nodes it will verify.

Table 3: On-Premises bare metal CV verifying 250 Cloud Nodes using 50 CV processes.

Network RTT (ms)	**Rate** (quotes/s)	**Bandwidth** (Kbits/s)
4ms (native)	937	3085
25ms	613	2017
50ms	398	1310
75 ms	282	928.3
100 ms	208	684.7
150 ms	141	464.2

6. RELATED WORK

Many of the challenges that exist in traditional enterprise networks exist in cloud computing environments as well. However, there are new challenges and threats including shared tenancy and additional reliance on the cloud provider to provide a secure foundation [4, 16]. To address some of the challenges, many have proposed trusted computing.

The existing specifications for trusted computing rely on trusted hardware, and assume a single *owner* of the system. With the advent of cloud computing, this assumption is no longer valid. While both the standards community [41] and prior work [2] is beginning the process of supporting virtualization, no end-to-end solution exists. For example, the cTPM system [5] assumes a trustworthy cloud provider and requires modifications to trusted computing standards. Another proposal for higher-level validation of services provides a cryptographically signed audit trail that the hypervisor provides to auditors [12]. The audit trail captures the execution of the applications within the virtual machine. This proposal does not provide a trusted foundation for the audit trail, and assumes a benign hypervisor. Bleikertz, et al., propose to use trusted computing to provide cryptographic services for cloud tenants[3]. Their Cryptography-as-a-Service (CaaS) system relies on trusted computing, but does not address bootstrapping and requires hypervisor modifications

[17] https://github.com/tylertreat/Comcast

that cloud providers are unlikely to support.

To address the issues of scalability, several proposals exist to monitor a cloud infrastructure, and allow for validation of the virtual machines controlled by the tenants of the cloud [34, 32, 33, 35]. The cloud verifier pattern proposed by Schiffman, et al., allows a single verifier to validate trust in the cloud infrastructure, and in turn the cloud verifier "vouches" for the integrity of the cloud nodes. This removes the need for tenants to validate the integrity of the hypervisor hosts prior to instantiating cloud nodes on them and avoids the need for nodes to mutually attest each other before communicating. The tenant simply provides their integrity verification criteria to the verifier, and the verifier ensures that the tenant's integrity criteria are satisfied as part of scheduling resources. We utilize the cloud verifier pattern in our work, with some important differences. First we extend it to support secure system bootstrapping for both bare metal and virtualized IaaS environments. Second, we do not host any tenant-owned parts of the integrity measurement infrastructure in the provider's control as they do. This means that our solution is substantially less invasive to the cloud provider's infrastructure (e.g., they required nearly 5,000 lines of code to be added to OpenStack) and is less prone to compromise. For example, `keylime` relies upon the vTPM integrity measurements inside tenant nodes rather than enabling the cloud provider to have explicit virtual machine introspection (i.e., secret stealing) capabilities.

Excalibur works to address the scalability problems of trusted computing by leveraging ciphertext policy attribute-based encryption (CPABE) [31]. This encryption scheme allows data to be encrypted using keys that represent attributes of the hypervisor hosts in the IaaS environment (e.g., software version, country, zone). Using Excalibur, clients can encrypt sensitive data, and be assured that a hypervisor will only be given access to the data if the policy (the specified set of attributes) is satisfied. Excalibur only addresses trusted bootstrapping for the underlying cloud platform. Therefore, a compromised tenant node would be neither detected nor prevented. The Excalibur monitor is a provider-owned (but attested) component that holds the encryption keys that allow a node to boot on a particular hypervisor. `keylime` uses secret sharing to avoid having bootstrap key stored (and therefore vulnerable) in any cloud systems except for in the cloud node for which they are intended.

The CloudProxy Tao system provides building blocks to establish trusted services in a layered cloud environment [25]. The Tao environment relies on the TPM to establish identity and load-time integrity of the nodes and software in the system. Their system does not support system integrity monitoring as they assume that all interactions will *only* be with other trusted programs running in the Tao environment. Tao relies upon mutual attestation for all communicating nodes, but is unable to use TPM-based keys because they are not fast enough to support mutual attestation. Using the out-of-band CV, we avoid mutual attestation while maintaining rapid detection of integrity violations. The Key Server in Tao holds all the secret keys to the system, must interact with hosts to load new applications, and must be fully trusted. The Key Server does not offer compatible deployment options for IaaS environments, especially for small tenants who cannot afford secure facilities or hardware security modules. Furthermore, CloudProxy Tao does not detail the secure bootstrapping of their Key Server or Privacy CA component for TPM initialization. `keylime` explicitly describes bootstrapping of all relevant components and enables multiple realistic secure deployment options for CV and registrar hosting.

7. CONCLUSION

In this paper, we have shown that `keylime` provides a fully integrated solution to bootstrap and maintain hardware-rooted trust in elastically provisioned IaaS clouds. We have demonstrated that we can bootstrap hardware-rooted cryptographic identities into both physical and virtual cloud nodes, and leverage those identities in higher-level security services, without requiring each service to become trusted computing-aware. `keylime` uses a novel key derivation protocol that incorporates a tenant's *intent* to provision new cloud resources with integrity measurement. Finally, we have demonstrated and evaluated several deployment scenarios for our system's critical component, the cloud verifier. `keylime` can securely derive a key in less than two seconds during the provisioning and bootstrapping process, and requires as little as 110ms to respond to an integrity violation. Furthermore, we have shown that `keylime` can scale to support thousands of IaaS nodes while maintaining quick response to integrity violations.

8. REFERENCES

[1] S. Balfe and A. Mohammed. Final fantasy – securing on-line gaming with trusted computing. In B. Xiao, L. Yang, J. Ma, C. Muller-Schloer, and Y. Hua, editors, *Autonomic and Trusted Computing*, volume 4610 of *Lecture Notes in Computer Science*, pages 123–134. Springer Berlin Heidelberg, 2007.

[2] S. Berger, R. Cáceres, K. A. Goldman, R. Perez, R. Sailer, and L. van Doorn. vtpm: Virtualizing the trusted platform module. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, Berkeley, CA, USA, 2006. USENIX Association.

[3] S. Bleikertz, S. Bugiel, H. Ideler, S. Nürnberger, and A.-R. Sadeghi. Client-controlled cryptography-as-a-service in the cloud. In M. Jacobson, M. Locasto, P. Mohassel, and R. Safavi-Naini, editors, *Applied Cryptography and Network Security*, volume 7954 of *Lecture Notes in Computer Science*, pages 19–36. Springer Berlin Heidelberg, 2013.

[4] S. Bouchenak, G. Chockler, H. Chockler, G. Gheorghe, N. Santos, and A. Shraer. Verifying cloud services: Present and future. *SIGOPS Oper. Syst. Rev.*, 47(2):6–19, July 2013.

[5] C. Chen, H. Raj, S. Saroiu, and A. Wolman. ctpm: A cloud tpm for cross-device trusted applications. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, Seattle, WA, Apr. 2014. USENIX Association.

[6] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. *SIGPLAN Not.*, 43(3):2–13, Mar. 2008.

[7] J. Criswell, N. Dautenhahn, and V. Adve. Virtual ghost: Protecting applications from hostile operating systems. *SIGARCH Comput. Archit. News*, 42(1):81–96, Feb. 2014.

[8] L. Davi, A.-R. Sadeghi, and M. Winandy. Dynamic integrity measurement and attestation: towards defense against return-oriented programming attacks. In *Proceedings of the 2009 ACM workshop on Scalable trusted computing*, pages 49–54. ACM, 2009.

[9] A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: Malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, pages 51–62, New York, NY, USA, 2008. ACM.

[10] M. Fioravante and D. D. Graaf. vTPM. http://xenbits.xen.org/docs/unstable/misc/vtpm.txt, November 2012.

[11] D. D. Graaf and Q. Xu. vTPM manager. http://xenbits.xen.org/docs/unstable/misc/vtpmmgr.txt.

[12] A. Haeberlen, P. Aditya, R. Rodrigues, and P. Druschel. Accountable virtual machines. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–16, Berkeley, CA, USA, 2010. USENIX Association.

[13] J. Hennessey, C. Hill, I. Denhardt, V. Venugopal, G. Silvis, O. Krieger, and P. Desnoyers. Hardware as a service - enabling dynamic, user-level bare metal provisioning of pools of data center resources. In *2014 IEEE High Performance Extreme Computing Conference*, Waltham, MA, USA, Sept. 2014.

[14] P. Hintjens. Curvezmq authentication and encryption protocol. http://rfc.zeromq.org/spec:26, 2013.

[15] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel. Inktag: Secure applications on an untrusted operating system. *SIGPLAN Not.*, 48(4):265–278, Mar. 2013.

[16] W. Huang, A. Ganjali, B. H. Kim, S. Oh, and D. Lie. The state of public infrastructure-as-a-service cloud security. *ACM Comput. Surv.*, 47(4):68:1–68:31, June 2015.

[17] IBM. Ibm and intel bring new security features to the cloud. http://www.softlayer.com/press/ibm-and-intel-bring-new-security-features-cloud, September 2004.

[18] IBM. Software trusted platform module. http://sourceforge.net/projects/ibmswtpm/, April 2014.

[19] Intel. Intel Trusted Boot (tboot). https://software.intel.com/en-us/articles/intel-trusted-execution-technology.

[20] Intel. Cloud integrity technology. http://www.intel.com/p/en_US/support/highlights/sftwr-prod/cit, 2015.

[21] T. Jaeger, R. Sailer, and U. Shankar. Prima: Policy-reduced integrity measurement architecture. In *Proceedings of the Eleventh ACM Symposium on Access Control Models and Technologies*, SACMAT '06, pages 19–28, New York, NY, USA, 2006. ACM.

[22] B. Kauer. Oslo: Improving the security of trusted computing. In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, SS'07, pages 16:1–16:9, Berkeley, CA, USA, 2007. USENIX Association.

[23] R. Kotla, T. Rodeheffer, I. Roy, P. Stuedi, and B. Wester. Pasture: Secure offline data access using commodity trusted hardware. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 321–334, Hollywood, CA, 2012. USENIX.

[24] P. A. Loscocco, P. W. Wilson, J. A. Pendergrass, and C. D. McDonell. Linux kernel integrity measurement using contextual inspection. In *Proceedings of the 2007 ACM Workshop on Scalable Trusted Computing*, STC '07, pages 21–29, New York, NY, USA, 2007. ACM.

[25] J. Manferdelli, T. Roeder, and F. Schneider. The cloudproxy tao for trusted computing. Technical Report UCB/EECS-2013-135, EECS Department, University of California, Berkeley, Jul 2013.

[26] P. Maniatis, D. Akhawe, K. Fall, E. Shi, S. McCamant, and D. Song. Do you know where your data are?: Secure data capsules for deployable data protection. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, pages 22–22, Berkeley, CA, USA, 2011. USENIX Association.

[27] D. A. McGrew and J. Viega. The galois/counter mode of operation (gcm). *NIST*, 2005.

[28] T. Moyer, K. Butler, J. Schiffman, P. McDaniel, and T. Jaeger. Scalable web content attestation. *IEEE Transactions on Computers*, Mar 2011.

[29] S. Munetoh. GRUB TCG Patch to Support Trusted Boot. http://trousers.sourceforge.net/grub.html.

[30] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and implementation of a tcg-based integrity measurement architecture. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, SSYM'04, pages 16–16, Berkeley, CA, USA, 2004. USENIX Association.

[31] N. Santos, R. Rodrigues, K. P. Gummadi, and S. Saroiu. Policy-sealed data: A new abstraction for building trusted cloud services. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 175–188, Bellevue, WA, 2012. USENIX.

[32] J. Schiffman, T. Moyer, C. Shal, T. Jaeger, and P. McDaniel. Justifying integrity using a virtual machine verifier. In *Computer Security Applications Conference, 2009. ACSAC '09. Annual*, pages 83–92, Dec 2009.

[33] J. Schiffman, T. Moyer, H. Vijayakumar, T. Jaeger, and P. McDaniel. Seeding clouds with trust anchors. In *Proceedings of the 2010 ACM Workshop on Cloud Computing Security Workshop*, CCSW '10, pages 43–46, New York, NY, USA, 2010. ACM.

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

[35] J. Schiffman, H. Vijayakumar, and T. Jaeger. Verifying system integrity by proxy. In *Proceedings of the 5th International Conference on Trust and Trustworthy Computing*, TRUST'12, pages 179–200, Berlin, Heidelberg, 2012. Springer-Verlag.

[36] A. Segall. Using the tpm: Machine authentication and attestation. http://opensecuritytraining.info/IntroToTrustedComputing_files/Day2-1-auth-and-att.pdf, Oct 2012.

[37] 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*, SOSP '11, pages 249–264, New York, NY, USA, 2011. ACM.

[38] Trusted Computing Group. TCG Infrastructure Working Group A CMC Profile for AIK Certificate Enrollment. http://www.trustedcomputinggroup.org/files/resource_files/738DF0BB-1A4B-B294-D0AF6AF9CC023163/IWG_CMC_Profile_Cert_Enrollment_v1_r7.pdf.

[39] Trusted Computing Group. Trusted Network Communications. http://www.trustedcomputinggroup.org/developers/trusted_network_communications.

[40] Trusted Computing Group. Trusted Platform Module. http://www.trustedcomputinggroup.org/developers/trusted_platform_module.

[41] Trusted Computing Group. Virtualized Platform. http://www.trustedcomputinggroup.org/developers/virtualized_platform.