Policy-Sealed Data: A New Abstraction for Building Trusted Cloud Services

Nuno Santos, Rodrigo Rodrigues[†]*, Krishna P. Gummadi, Stefan Saroiu*[‡]
MPI-SWS, [†]*CITI/Universidade Nova de Lisboa,* [‡]*Microsoft Research*

Abstract

Accidental or intentional mismanagement of cloud software by administrators poses a serious threat to the integrity and confidentiality of customer data hosted by cloud services. Trusted computing provides an important foundation for designing cloud services that are more resilient to these threats. However, current trusted computing technology is ill-suited to the cloud as it exposes too many internal details of the cloud infrastructure, hinders fault tolerance and load-balancing flexibility, and performs poorly. We present Excalibur, a system that addresses these limitations by enabling the design of trusted cloud services. Excalibur provides a new trusted computing abstraction, called *policy-sealed data*, that lets data be *sealed* (i.e., encrypted to a customer-defined policy) and then *unsealed* (i.e., decrypted) only by nodes whose configurations match the policy. To provide this abstraction, Excalibur uses attribute-based encryption, which reduces the overhead of key management and improves the performance of the distributed protocols employed. To demonstrate that Excalibur is practical, we incorporated it in the Eucalyptus open-source cloud platform. Policy-sealed data can provide greater confidence to Eucalyptus customers that their data is not being mismanaged.

1 Introduction

Managing cloud computing services is complex and error-prone. Cloud providers therefore delegate this task to skilled cloud administrators who manage the cloud infrastructure software. However, it is difficult to assure that their actions are error-free. In particular, an accidental or, in some cases, intentional action from a cloud administrator could leak, corrupt, or lose customer data. The threat of potential violations to the integrity and confidentiality of customer data is often cited as a key barrier to the adoption of cloud services [2,15]. Furthermore, publicized incidents involving the loss of confidentiality or integrity of customer data [1, 4, 7, 23, 25] and the growing amount of security-sensitive data outsourced to the cloud [3,6] only heightens these concerns.

Recently, several proposals [22,39,45,53] have advocated leveraging trusted computing technology to make cloud services more resilient to integrity and confidentiality concerns. This technology relies on a secure coprocessor – typically a Trusted Platform Module (TPM) chip [17] – deployed on every node in the cloud. Each TPM chip would store a strong identity (unique key) and a fingerprint (hash) of the software stack that booted on the cloud node. TPMs could then restrict the upload of customer data to cloud nodes whose identities or fingerprints are considered *trusted*. This capability offers a building block in the design of trusted cloud services by securing data confidentiality and integrity against insiders, or confining the data location to a desired geographical or jurisdictional boundary.

Despite their benefits, current trusted computing abstractions are ill-suited to the requirements of cloud services for three main reasons. First, TPM abstractions were designed to protect data and secrets on a standalone machine; they are thus cumbersome to use in a multi-node datacenter environment where data migrates across multiple nodes with potentially different configurations. Second, TPM abstractions over-expose the cloud infrastructure by revealing the identity and software fingerprint of individual cloud nodes; external agents could use this information to exploit vulnerabilities in the cloud infrastructure or gain business advantage [40]. Third, the current implementation of TPM abstractions is inefficient and can introduce scalability bottlenecks to cloud services.

This paper presents Excalibur, a system that provides cloud service designers with new trusted computing abstractions that overcome these barriers. These abstractions provide another critical building block for constructing services that offer better guarantees regarding data integrity, confidentiality, or location. Excalibur's design includes two main innovations crucial to overcoming the concerns posed by using TPMs in the cloud.

First, Excalibur provides a new trusted computing abstraction, called *policy-sealed data*, that allows customer data to be encrypted according to a customer-chosen policy and guarantees that only the cloud nodes whose configuration satisfies that policy can decrypt and retrieve the data. We devised this abstraction to address the first two limitations of current TPM abstractions; the abstraction permits multiple nodes with or without identical configurations to flexibly access data as long as they satisfy the customer policies. Moreover, since it allows policies to be specified using human-readable attributes, policy-sealed data hides the low-level identities and software fingerprints of nodes.

Second, Excalibur implements the policy-sealed data abstraction in a way that overcomes the inefficiency hur-

dles of current TPMs and scales to the demand of cloud services. To do this, we designed a centralized *monitor* that checks the integrity of cloud nodes and acts as a single point-of-contact for customers to bootstrap trust in the cloud infrastructure. To prevent the potential scalability challenges associated with a centralized monitor, we designed a set of distributed protocols to efficiently implement the new abstractions. Our protocols use the Ciphertext Policy Attribute-Based Encryption (CPABE) encryption scheme [11], which drastically reduces the communication needs between the monitor and production nodes by requiring each node contact the monitor only once during a boot cycle, a relatively infrequent operation. We validated the correctness of Excalibur's cryptographic protocols using a protocol verifier [12].

To demonstrate the practicality of Excalibur, we built a proof-of-concept compute service akin to EC2. Based on the Eucalyptus open source cloud management platform [36], our service leveraged Excalibur to give users better guarantees regarding the type of hypervisor or the location where their VM instances run. Our experience shows that Excalibur's primitive is simple and versatile: our changes required minimal modifications to the Eucalyptus codebase.

Our evaluation suggests that Excalibur scales well. Due to CPABE, the monitor's load scales independent of the workload. In addition, according to our simulations, one server acting as a monitor was sufficient to manage a large cluster; for example, a server took $\sim$15 seconds to check the node configurations of a cluster with 10K nodes that all rebooted simultaneously. Finally, offering trusted computing guarantees to the EC2-like service added modest overhead during VM management operations only.

2 Trusted Computing Concepts

The success of a cloud provider hinges on its customers being willing to entrust the provider with their data [2, 15]. A key factor in strengthening customers' trust is providing strong assurances about the integrity of the cloud infrastructure. TPMs can play a fundamental role in providing these assurances.

The integrity of the cloud infrastructure depends on the security of its hardware and software components. For hardware security, cloud providers already rely on surveillance devices and physical access control that severely restrict physical access to cloud nodes, even by cloud provider staff [19]. In certain cases, by deploying cloud nodes in sealed containers, they ensure that physical access is fully disallowed [19]. For software security, providers could take advantage of techniques that reduce the size of the TCB [53], narrow the management interfaces [34], and verify the TCB code [24]. These techniques help designers build secure software platforms (e.g., secure hypervisors) to host customers' data and computations.

However, current cloud architectures provide scant assurances that the data that customers ship to the cloud is being handled by integrity-protected nodes running secure software platforms. Insecure software platforms (e.g., ones that have been tampered with or that run unpatched software versions) put at risk cloud service integrity and thus customer data. Trusted computing technology addresses this problem by providing customers with integrity guarantees of the cloud nodes themselves.

Trusted computing technology provides the hardware support needed to bootstrap trust in a computer [38]. To do so, it offers system designers four main abstractions. First, *strong identities* let the computer be uniquely identified without having to trust the OS or the software running on the computer. Second, *trusted boot* produces a unique fingerprint of the software platform running on the computer; the fingerprint consists of hashes of software platform components (e.g., BIOS, firmware controlling the computer's devices, bootloader, OS) computed at boot time. Third, this fingerprint can be securely reported to a remote party using a *remote attestation* protocol; this protocol lets the remote party authenticate both the computer and the software platform so it can assess whether the computer is trustworthy, e.g., if it is a trusted platform that is designed to protect the confidentiality and integrity of data [20, 32]. Fourth, *sealed storage* allows the system to protect persistent secrets (e.g., encryption keys) from an attacker with the ability to reboot the machine and install a malicious OS that can inspect the disk; the secrets are encrypted so that they can be decrypted only by the same computer running the trusted software platform specified upon encryption.

An important instance of trusted computing hardware is the Trusted Platform Module (TPM) [17], a secure co-processor widely deployed on desktops, laptops and increasingly on servers. To offer a strong identity, the TPM uses an Attestation Identity Key (AIK). To track the hash values that constitute a fingerprint, the TPM uses special registers called Platform Configuration Registers (PCRs). Whenever a reboot occurs, the PCRs are reset and updated with new hash values. To perform remote attestation, the TPM can issue a *quote*, which includes the PCR values signed by the TPM with an AIK. For sealed storage, the TPM offers two primitives, called *seal* and *unseal*, to encrypt and decrypt secrets, respectively. Seal encrypts the input data and binds it to the current set of PCR values. Unseal validates the identity and fingerprint of the software platform before decrypting sealed data.

3 Threat Model

Our premise is that the attacker seeks to compromise customer data by extracting it from integrity-protected cloud nodes. An attack is successful if either the data is accessible on a machine running an insecure software platform or is moved outside the provider's premises.

The attacker is assumed to be an agent with privileged access to the cloud nodes' management interface. Such an agent is typically a cloud provider's employee who manages cloud software and behaves inappropriately due either to negligence (e.g., misconfiguring the nodes where a computation should run) or to malice (e.g., desire to steal customer data). The management interface is accessible only from a *remote site*. Therefore, we assume the attacker cannot launch physical attacks. In fact, software and hardware management roles are usually differentiated and assigned to different teams.

The management interface grants the attacker privileges to the software platform running on the node (e.g., access to the root account) and to a dedicated hardware component for power cycling the nodes. These privileges empower him to access customer data on the nodes: he can reboot any node, access its local disk after rebooting, install arbitrary software on the node, and eavesdrop the network. However, whenever cloud nodes boot a secure software platform whose TCB we assume to be correct, the attacker can no longer exploit vulnerabilities through the software platform's interface.

Multiple trusted parties perform all other management tasks in the cloud provider's infrastructure. These tasks include, e.g., procuring and deploying the hardware, securing the premises, developing the software platforms, managing the provider's private keys, endorsing whether a software platform is secure, certifying the software and hardware, etc. Trusted parties can be employees of the cloud provider or external trusted organizations. Due to the nature of their roles, however, trusted parties *do not* have access to the cloud nodes' management interface.

We assume that the TPMs are correct, and we do not consider side-channel attacks.

4 Policy-sealed Data

This section makes the case for our new trusted computing abstraction, called *policy-sealed* data. We first discuss the limitations of existing TPM abstractions in the context of the design of a strawman trusted cloud service. We then describe how policy-sealed data addresses these limitations.

4.1 Strawman Design of a Trusted Cloud Service

Our strawman trusted cloud service offers features similar to Amazon's EC2 but aims to provide better protection against the inspection or corruption of customer VMs by a cloud administrator.

The first step in designing the strawman is to protect the state of customer VMs running on cloud nodes. To do this, we use recent proposals from research and industry that offer such guarantees but *on a single node only*. For example, CloudVisor [53] retrofits Xen so that the hypervisor guarantees the integrity and confidentiality of data and software running in guest VMs even in the presence of a malicious system administrator. Customers can leverage the TPM's remote attestation capability to verify that a cloud node is running CloudVisor before uploading data to it.

However, this verification step checks these guarantees only for the cloud node on which the data is first uploaded. Once in the cloud, the customer's data and VMs often migrate from one node to another, or are suspended to disk and resumed at a later time. To offer end-to-end protection, the checks must be repeated upon such events.

Thus, to accommodate VM migration, the strawman design of a trusted EC2 must perform remote attestation each time a customer's VM migrates to verify that: (1) the destination node's identity is signed by the cloud provider, and (2) the fingerprint matches that of CloudVisor. To protect the VM upon suspension to disk, the VM state must be encrypted using sealed storage before suspension occurs.

4.2 Limitations of TPM Abstractions

The strawman design highlights some shortcomings of current TPM abstractions stemming from a fundamental principle upon which TPMs were built: they were designed to offer guarantees about one single computer. In particular, TPMs suffer from three major problems when they are used to build trusted cloud services.

First, the sealed storage abstraction was not designed for a distributed and dynamic environment like the datacenters where cloud services operate. It precludes the application developer from encrypting and storing sensitive data in an untrusted medium (e.g., a local hard drive, or the Amazon S3 service) and retrieving it from a different node or from the same node running a software configuration that differs from that in place when the data was encrypted. However, developers might be interested in suspending the VM to disk and resuming it later on a different node (e.g., if, in the interim, the original node was shut down to save power) or on the same node running a different configuration (e.g., if, in the interim, the hypervisor was upgraded to a more recent version).

Second, today's TPMs are not built for high performance. TPMs can execute only one command at a time, and many TPM commands, such as remote attestation,

Attribute	Value	Description
service	"EC2"	service name
version	"1"	version of the service
vmm	"Xen", "CloudVisor"	virtual machine monitor
type	"small", "large"	resources of a VM
country	"US", "DE"	country of deployment
zone	"Z1", "Z2", "Z3", "Z4"	availability zone

Table 1: Example of service attributes. In this case, EC2 supports two types of VM instances, two types of VMMs, and four availability zones (datacenters) in the US and Germany.

Node	Configuration
N	*service* : "EC2" ; *version* : "1" ; *type* : "small" ; *country* : "DE" ; *zone* : "Z2" ; *vmm* : "CloudVisor"

Table 2: Example of a node configuration. This configuration contains the values for the attributes that characterize the hardware and software of a specific node N.

Policy	Policy Specification
P_1	*service* = "EC2" and *vmm* = "CloudVisor" and *version* $\geq$ "1" and *instance* = "large"
P_2	*service* = "EC2" and *vmm* = "CloudVisor" and (*zone* = "Z1" or *zone* = "Z3")
P_3	*service* = "EC2" and *vmm* = "CloudVisor" and *country* = "DE"

Table 3: Examples of policies. P_1 expresses version and VM instance type requirements, P_2 specifies a zone preference for different sites, and P_3 expresses a regional preference.

take approximately one second to complete. This inefficiency hampers the scalability of cloud services that use the TPM and can even open avenues for denial of service attacks if the TPM abstractions were invoked by customer-accessible operations.

Finally, the cloud infrastructure may be overexposed. By revealing TPM node identities and allowing customers to remotely attest the nodes, any outsider could learn, for instance: (1) the number of cloud nodes that constitute the infrastructure of the cloud provider, and (2) the distribution of different platforms they run. This information could be used by external attackers to trace vulnerabilities in the infrastructure, or by competitors to learn business secrets. Handing over such information is often unacceptable to cloud providers.

Recent proposals for TPMs in the cloud do not completely address these TPM limitations. Systems like Nexus [50] or CloudVisor [53] use TPMs to allow customers to remotely attest only a single cloud node and therefore do not address the preceding issues. Essentially, these systems address the complementary problem of securing the platform running on a single node. Our previous workshop paper [45] took preliminary steps to address some of these issues, but its solution did not handle situations where sensitive data needed to be secured persistently, which is unrealistic to assume on real-world cloud services; our prior solution also suffered from scalability limitations.

4.3 The Policy-sealed Data Abstraction

To overcome these limitations, we propose the new *policy-sealed data* abstraction. This abstraction allows customer data to be bound to cloud nodes whose configuration is specified by a customer-defined policy. Policy-sealed data offers two primitives for securing customer data: *seal* and *unseal*. Seal can be invoked anywhere – either on the customer's computer or on the cloud nodes. It takes as input the customer's data and a *policy* and outputs ciphertext. The reverse operation, unseal, can be invoked only on the cloud nodes that need to decrypt the data. Unseal takes as input the sealed data and decrypts it *if and only if* the node's configuration satisfies the policy specified upon seal; otherwise, decryption fails.

With our abstraction, each cloud node has a configuration, which is a set of human-readable *attributes*. Attributes express features that refer to the node's software (e.g., "vmm", "version") or hardware (e.g., "location"). A policy expresses a logical condition over the attributes supported by the provider (e.g., "vmm=Xen and location=US"). Table 1 shows an example of the attributes of a hypothetical deployment of a service akin to EC2. Table 2 illustrates the configuration of a particular node, and Table 3 lists example policies over node configurations in that deployment.

Our primitive can replace the existing remote attestation and sealed storage calls for securing customer data on the cloud. In particular, to protect data upon upload or migration, the customer needs only to seal the data to a policy: if the destination cannot unseal the data, then its configuration does not match the policy; therefore, the node is not trusted from the perspective of the customer who originally specified the policy.

5 Excalibur Design

This section presents Excalibur, a system that provides policy-sealed data support for building trusted cloud services.

5.1 Design Goals & Assumptions

Our central goal is to design and implement a system that offers the policy-sealed data primitive by making use of commodity TPMs. Furthermore, the system design must overcome the preceding limitations of the interface offered by current TPMs.

We focus on the design of the primitive used by the cloud platforms running on individual nodes. Therefore, we are not concerned with securing these platforms themselves. In particular, our goal is not to prevent the management interface exposed to cloud administrators from leaking or corrupting sensitive data (e.g., direct memory inspection of VM memory). Similarly, we require that the individual cloud platforms pro-

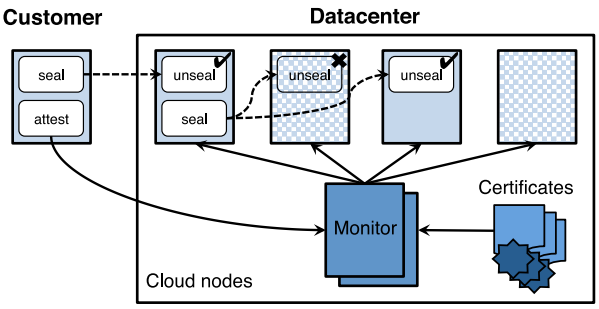

Figure 1: Excalibur deployment. The dashed lines show the flow of policy-sealed data, and the solid lines represent interactions between clients and the monitor. The monitor checks the configuration of cloud nodes. After a one-time monitor attestation step, clients can seal data. Data can be unsealed only on nodes that satisfy the policy (unshaded boxes).

tect certain key material used to seal and unseal data, and that the system interface does not allow the fingerprint stored in the TPM to be changed so that it becomes inconsistent with the current system state. To address these complementary goals, applications must make use of a series of existing systems and hardening techniques [20, 24, 33, 53].

5.2 System Overview

The design of Excalibur is based on a centralized component, called a *monitor*. The monitor is a dedicated service running on a single cloud node (or, as we will explain, on a small set of nodes for fault tolerance and scalability). It coordinates the enforcement of policy-sealed data on the entire cloud infrastructure by mapping TPM identities and fingerprints of the cloud nodes to policy-sealed data attributes. Only the monitor can trigger TPM primitives on the cloud nodes, minimizing the negative performance impact of TPM operations and preventing the exposure of infrastructure details.

Figure 1 illustrates a deployment of Excalibur, highlighting the separation between the two main system components: the *client* and the *monitor*. The client consists of a library that allows the implementation of a trusted cloud service to use the policy-sealed data primitives. This library can be used on both the customer side (e.g., before uploading data) and by the software platforms running on the cloud nodes (e.g., before migrating data between nodes). The customer-side client does not expose the unseal primitive since the notion of a configuration applies to cloud nodes only.

Whenever a cloud node reboots, the monitor runs a special remote attestation protocol to obtain the fingerprint and identity of the node and translates these to a node configuration by consulting an internal database. The node configuration — which expresses physical characteristics, like hardware or location, and software features as a set of attributes — is then encoded as credentials that are sent to the node. These credentials are required by cloud nodes to unseal policy-sealed data and are destroyed whenever the nodes reboot.

The monitor exposes a narrow management interface that lets the cloud administrator configure the mappings between attributes and identities (i.e., fingerprints). This is necessary for routing system maintenance as new software platforms and cloud nodes are deployed on the infrastructure. The management interface also allows multiple clones of the monitor to be securely spawned in order to scale up the system. To assure customers that it is properly maintained, the monitor accepts only mappings that are vouched for by special *certificates*; customers can directly attest the monitor in order to check its authenticity and integrity.

Though our high-level design is simple, we still need to overcome two main challenges: 1) to cryptographically enforce policies in a scalable, fault tolerant and efficient way, and 2) to assure customers that the monitor operates correctly despite the fact that it is managed by untrusted cloud administrators. To address these challenges, we: 1) use CPABE cryptography to enforce policies, and 2) devise certificates and a scalable monitor attestation mechanism to ensure that the monitor is trustworthy. We next explain these design choices in more detail.

5.3 Cryptographic Enforcement of Policies

The main challenge in implementing the seal and unseal primitives is avoiding scalability bottlenecks. A possible design is for the monitor itself to evaluate the policies: upon sealing, the client encrypts the data with a symmetric key and sends this key and the policy to the monitor; the monitor then encrypts this key and the policy with a secret key and returns the outcome to the client. To unseal, the encrypted key is sent to the monitor, which internally recovers the original symmetric key and policy, evaluates the policy, and releases the symmetric key if the node satisfies the policy. Although this solution implements the necessary functionality, it involves the monitor in every seal and unseal operation and thereby introduces a scalability bottleneck.

An alternative design is to evaluate the policies on the client side using public-key encryption. Each cloud node receives from the monitor a set of private keys that match its configuration; in this scheme, each key corresponds to an attribute-value pair of the configuration. Sealing is done by encrypting the data with the corresponding public keys according to the attributes defined in the policies. This solution avoids the bottlenecks of the first approach because all cryptographic operations take place on the client side, without involving the monitor. Its main shortcoming is complicated key manage-

ment due to the number of key-pairs that nodes must handle to reflect all possible attribute combinations usable by policies.

The solution we chose uses a cryptographic scheme called Ciphertext Policy Attribute-Based Encryption (CPABE) [11]. This scheme first generates a pair of keys: a public *encryption key* and a secret *master key*. Unlike traditional public key schemes, the encryption key allows a piece of data to be encrypted and bound to a *policy*. A policy is a logical expression that uses conjunction and disjunction operations over a set of terms. Each term tests a condition over an attribute, which can be a string or a number; both types support the equality operation, but the numeric type also supports inequalities (e.g., $a = x$ *or* $b > y$). CPABE can then create an arbitrary number of *decryption keys* from the same master key, each of which can embed a set of attributes specified at creation time. The encrypted data can be decrypted only by a decryption key whose attributes satisfy the policy (e.g., keys embedding the attribute $a = x$ can decrypt a piece of data encrypted with the preceding example policy).

Excalibur uses CPABE to encode the runtime configurations of the cloud nodes into decryption keys. At setup time, the monitor generates a CPABE encryption and master key pair and secures the master key. Whenever it checks the identity and software fingerprint of a cloud node, the monitor sends the appropriate credentials to the node, which include a CPABE decryption key embedding the attributes that correspond to the configuration of the node; the decryption key is created from the master key and forwarded to all the nodes featuring the same configuration. Sealing is done by encrypting the data using the encryption key and a policy, and unsealing is done by decrypting the sealed data using the decryption key. Policies are expressed in the CPABE policy language used to specify the examples in Table 3 as well as more elaborate policies.

The security of the system then depends on the security of the CPABE keys. The monitor protects the master key by: 1) ensuring that it cannot be released through the monitor's management interface, and 2) encrypting it before storing it on disk, as described in Section 6.3. Additionally, cloud platforms must protect decryption keys. A software platform must prevent leakage or corruption of key material through its management interface (e.g., by direct memory inspection of VM memory); it must hold the key in volatile memory so that key material is destroyed upon reboot. Moreover, the software platform must force a reboot after changing TCB components that get measured during a trusted boot (e.g., subsequent to upgrading the hypervisor). These properties ensure that the CPABE decryption keys of cloud nodes remain consistent with their TPM fingerprints and therefore reflect current node configurations.

The benefits of using CPABE are twofold. First, it lets the system scale independently of the workload since the seal and unseal primitives do not interact with the monitor (and run entirely on the client side). Second, it permits the creation of expressive policies directly supported by the CPABE policy specification language while only requiring two keys – the CPABE encryption and decryption keys – to be sent to the nodes.

The cost using CPABE is a performance hit when compared to traditional cryptographic schemes. Section 6 explains how this impact can be minimized. A second cost of using CPABE is key revocation, which is typically difficult in identity- and attribute-based cryptosystems. Since Excalibur assumes that the TCB of nodes' software platforms is secure, any TCB vulnerability accessible through the administrator's interface will invalidate the guarantees provided by our system. To handle revocation of decryption keys, our current design requires that all sealed data whose original policy satisfies the attributes of the compromised keys be resealed. This operation can be done efficiently by reencrypting only a symmetric key, not the data itself.

5.4 Trusting the Monitor

Since the monitor is managed by the cloud administrator, mismanagement threats that affect any cloud node could also affect the monitor. Thus, another challenge is to ensure that the monitor operates correctly and to efficiently convey this guarantee to customers.

To meet this challenge, we must first prevent the monitor from accepting flawed attribute mappings. For example, a mapping would be flawed if the attribute "location=DE" were mapped to the identity of a node located in the US, or if the attribute "vmm=Xen" were mapped to the fingerprint of CloudVisor. To prevent this, the monitor only accepts attribute mappings that are vouched for by a *certificate*. A certificate is issued by one or multiple *certifiers*, which validate the correctness of mappings. For example, a certifier checks the location of nodes and the fingerprints of software platforms. This role could be played by the provider itself, or by external trusted parties akin to Certification Authorities.

Since anyone can issue certificates, the monitor must let customers know the certifier's identity so they can judge the certifier's trustworthiness and thereby be confident that the attribute mappings are correct. Furthermore, even if the certifier were judged trustworthy, the system must nevertheless provide additional guarantees about the authenticity and integrity of the monitor: only in this case can the customer be sure that the certificate-based protections and the security proto-

cols implemented by the monitor are correct. To provide these guarantees, customers must directly attest the monitor when first using the system.

5.5 Monitor Scalability and Fault Tolerance

To improve scalability and make Excalibur resilient to faults, we enable several monitor replicas (*clones*) to be spawned, and we optimize the monitor attestation protocol.

Monitor clones can be elastically launched and terminated by the administrator, using the protocol described in Section 6.7. The cloud provider can then use standard load balancers to evenly distribute client attestation requests from clients among clones. Each clone can serve requests without communicating with other clones.

To eliminate critical bottlenecks within a clone, we introduce two optimizations. The first improves the throughput of clone attestations triggered by customers. Due to TPM inefficiencies, the maximum throughput of a monitor clone using a standard attestation protocol is close to one attestation per second, clearly insufficient even when spawning a reasonable number of clones. We therefore enhance the attestation protocol with a technique based on Merkle trees that can batch a large number of attestation requests into a single TPM quote (see Section 6).

A second optimization improves the throughput of decryption key requests issued by the cloud nodes. The algorithm for decryption key generation is also inefficient, which could significantly slow down servicing keys to the cloud nodes if a new key were to be generated per request. Since many machines in the datacenter share the same configuration (e.g., machines that belong to the same cluster), the monitor clone can instead securely cache the decryption keys and send them to all the nodes with the same profile.

6 Detailed Design

This section presents the design of Excalibur in more detail. We first introduce certificates, which constitute the root-of-trust of the system. We then describe the interfaces offered by Excalibur for building cloud services and managing the system. Finally, we present the security protocols that enforce policy-sealed data.

Notation. For CPABE keys, $K^{\mathbb{M}}$, $K^{\mathbb{E}}$ and $K^{\mathbb{D}}$ denote master, encryption, and decryption keys, respectively. For asymmetric cryptography, K and K^P denote private and public keys, respectively. For symmetric keys, we drop the superscript. Notation $\langle x \rangle_K$ indicates data x encrypted with key K, and $\{y\}_K$ indicates data y signed with key K. We represent nonces as n. Session keys and nonces are randomly generated. Notation D, P, E, and M denote data, policy, envelope, and manifest; these terms are clarified in Section 6.2.

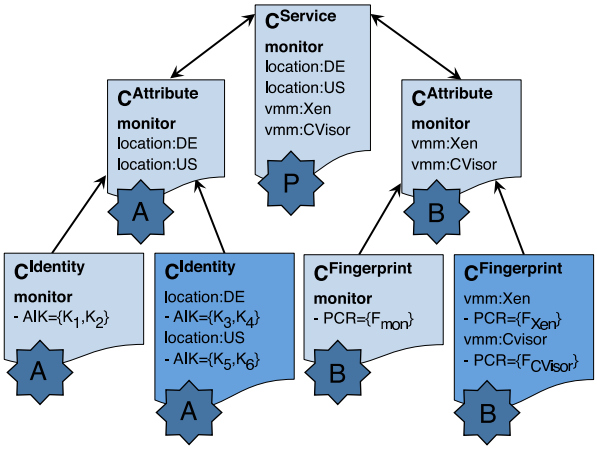

Figure 2: Example certificate tree. The certificates in light colored boxes form the *manifest* that validates the monitor's authenticity and integrity.

6.1 Certificate Specification

Excalibur uses certificates to validate mappings between attributes specific to a trusted cloud service and identities, i.e., fingerprints of cloud nodes. Certificates are used both by the monitor, to check the configuration of cloud nodes and attest new monitor clones, and by the customer-side client, to attest the monitor. Our certificate specification supports multiple certifiers since a single certifier may not have the expertise to assess all the attributes of the cloud service, or simply to increase customer trust. Therefore, certificates form a hierarchical tree. Figure 2 shows how a provider P can use the certificates that correspond to the internal nodes in the tree to delegate the certification of different attributes to two certifiers, A and B. Additionally, each leaf in the certificate tree vouches for a mapping between the attributes that appear in node configurations and low-level measurements, namely software fingerprints (PCRs) or hardware identities (AIK keys).

Due to space limitations, we defer a discussion of the details regarding the certification procedure, certificate expiration, certificate revocation, and certificate management to a separate technical report [46].

6.2 System Interfaces

Excalibur's interface has two parts: a *service interface*, which supports the implementation of cloud services, and a *management interface*, which lets cloud administrators maintain the system.

The service interface exported by the client library supports three operations, summarized in Table 4. Before the data can be sealed on the customer-side, *attest-monitor* must be invoked to check the monitor's authenticity and integrity. It returns the encryption key $K^{\mathbb{E}}$

attest-monitor$(mon\text{-}addr)$	$\rightarrow (K^{\mathbb{E}}, M)$ *or* FAIL
seal$(K^{\mathbb{E}}, P, D)$	$\rightarrow E = \langle P, D \rangle K, \langle K \rangle K^{\mathbb{E}}$
unseal$(K^{\mathbb{E}}, K^{\mathbb{D}}, E)$	$\rightarrow (D, P)$ *or* FAIL

Table 4: Excalibur service interface.

needed for sealing and a *manifest* M, which contains the certificates needed to validate the monitor's identity and fingerprint (see Figure 2). The manifest is passed to the customer, who learns from it which attributes can be used in policies and identifies the provider and certifier identities needed to decide whether the service is trustworthy. Since the client saves the manifest and encryption key for sealing, this operation needs to be performed only when the cloud service is first used.

The core primitives are *seal* and *unseal*. Seal can be invoked by both cloud nodes and customers; it takes as arguments the encryption key $K^{\mathbb{E}}$, a policy P, and the data D and produces an envelope E. This envelope is passed to unseal, which returns the decrypted data D or fails if its caller does not satisfy the policy. In addition to the decryption key $K^{\mathbb{D}}$, unseal receives as an argument the encryption key $K^{\mathbb{E}}$, which is required by CPABE decryption; the cloud node that invokes unseal must obtain this key from the monitor. Unseal also returns the original policy P so that a cloud node can re-seal the data with the customer's policy. The CPABE policy language is used to express policies.

The management interface lets the cloud administrator remotely maintain the monitor using a console. Its main operations permit the administrator to initialize the system, manage certificates, and spawn monitor clones.

6.3 System Initialization

Before the system can be used, the monitor must be initialized by binding a unique CPABE key pair to the service. To do this, the cloud administrator loads the certificates that validate the service attributes into the monitor and instructs the monitor to generate the key pair. If these certificates form a consistent certificate tree, the monitor creates unique encryption and master keys and binds them to the tree's root certificate (see Figure 2). To permit for system maintenance, the administrator can remove or add certificates as long as they form a valid certificate tree.

The monitor maintains its persistent state in a *certificate* store and a *key* store. Both stores keep their contents in XML files on a local disk. The certificate store contains the certificates loaded into the monitor. The key store contains all the CPABE keys. To secure the key material, the key store is sealed using the TPM seal primitive, which ensures that the key store can be accessed only under a trusted monitor configuration in case the monitor reboots.

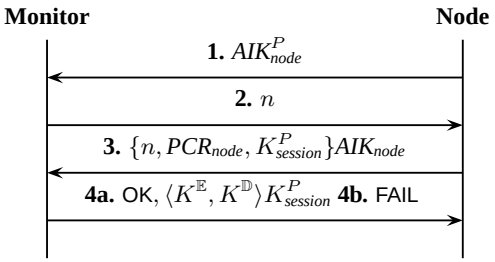

Figure 3: Node attestation protocol.

6.4 Node Attestation Protocol

Once the setup is complete, the monitor delivers to each cloud node a credential that reflects the boot time configuration of that node, which will allow the node to unseal and re-seal data. The goal of the node attestation protocol is to deliver these credentials securely. Recall that, under our assumptions, when a cloud node reboots, the credentials kept by the node in volatile memory are lost. Therefore, this protocol must be executed each time a cloud node reboots so it can obtain a fresh credential.

The monitor first obtains a quote from the node that is signed by the node's AIK and contains the current PCRs. Then, the monitor looks in the certificate database for certificates that match the node's PCRs and AIK. If any are found, the monitor obtains the node configuration by combining all the attributes of the matching certificates into a list like that shown in Table 2. Next, the monitor sends the credentials to the node; these include the encryption and decryption keys embedding these attributes. Since generating a new decryption key is expensive, the monitor caches these keys in the key store so they can be resent to nodes with the same configuration.

Figure 3 shows the precise messages exchanged between the monitor and the customer-side client. The protocol is based on a standard remote attestation in which a nonce n is sent to the node (message 2), and the node replies with a quote (message 3); the nonce is used to check the freshness of the attestation request. Message 3 includes a session key $K^{P}_{session}$ that is used in message 4 to securely send credentials $K^{\mathbb{E}}$ and $K^{\mathbb{D}}$ to the node. Since the session key is ephemeral, an adversary could not perform a TOCTOU attack by rebooting the machine after finishing attestation (message 3) but before receiving the decryption key (message 4).

Note that the node does not need to authenticate the monitor to preserve the security of policy-sealed data. In the worst case, a node may receive a compromised decryption key from an attacker. However, given that customers seal their data with the encryption key obtained from the legitimate monitor, unseal would fail in such a scenario, and this attack would fail to compromise customer data.

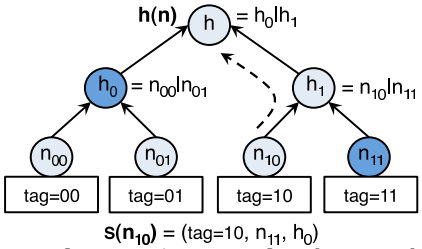

Figure 4: Batch attestation example. The tree is built from 4 nonces. A summary for nonce n_{10} comprises its tag and the hashes in the path to the root.

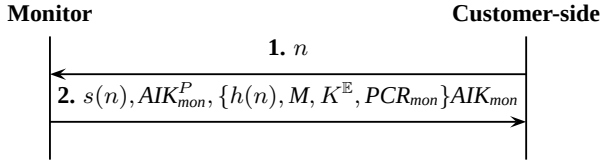

Figure 5: Monitor attestation protocol.

6.5 Monitor Attestation Protocol

The monitor attestation protocol is triggered by the *attest-monitor* operation, which lets customers detect if the monitor is legitimate by checking its authenticity and integrity. In addition, this protocol obtains: 1) the encryption key, which is used for sealing data, and 2) the set of certificates that form the manifest, which let the customer check the identity of certifiers and learn the attributes that are available. The monitor is legitimate if its identity and fingerprint are validated by the manifest.

The main challenge in designing this protocol is scalability. If every customer-side client were to run a standard remote attestation, then the throughput of the monitor would be extremely low due to TPM inefficiency.

To overcome this scalability problem, we batch multiple attestation requests into a single quote operation using a Merkle tree, as shown in Figure 4. The Merkle tree lets the monitor quote a batch of N nonces n_i expressed as an aggregate hash $h(n_{i=0}^{N})$ and send an evidence – summary $s(n_i)$ – to each customer-side client that its nonce n_i is included in the aggregate hash in a network-efficient manner (i.e., instead of sending all N nonces, it sends just a summary of size $O(log(N))$).

The detailed monitor attestation protocol is shown in Figure 5. In the first message, the customer-side client sends nonce n for freshness and then uses the information returned in message 2 to validate the monitor in two steps. First, it checks in the manifest M for the certificates with attribute "monitor"; it uses them to authenticate the monitor key AIK_{mon}^{P} and to validate the fingerprint of the monitor's software platform PCR_{mon} (see Figure 2). Second, to validate the freshness of the received messages, it compares nonce n and the summary $s(n)$ against the aggregate hash $h(n)$ produced by batch attestation. If all tests pass, the monitor is trustworthy, and the encryption key $K^{\mathbb{S}}$ is authentic. The customer can then seal data securely.

6.6 Seal and Unseal Protocols

The use of CPABE lets seal and unseal execute without contacting the monitor. In implementing these primitives, we take into account two aspects of CPABE related to performance and functionality. First, since CPABE is significantly more inefficient than symmetric encryption, seal encrypts the data with a randomly generated symmetric key and uses CPABE to encrypt the symmetric key. Second, given that CPABE decryption does not return the original policy (which unseal must return to let cloud nodes re-seal the data), we include in the envelope the original policy and a digest for integrity protection (see Table 4).

6.7 Clone Attestation Protocol

To scale the monitor elastically, the cloud administrator can create multiple monitor clones. To do so, an existing monitor instance must share the CPABE master key with the new clone so the latter can generate and distribute decryption keys to the cloud nodes. However, this can be done only if the new clone can be trusted to secure the key and to comply with the specification of Excalibur protocols.

To enforce this condition, the existing monitor instance and the clone candidate run a clone attestation protocol analogous to that shown in Figure 3, but with two differences. First, after message 3, the monitor assesses if the candidate is trustworthy by checking whether its AIK and PCR values map to the "monitor" attribute contained in the manifest; if not, cloning is aborted. Second, if the test passes, the monitor authorizes cloning and sends the master key, the encryption key, and a digest to the candidate. The digest identifies the head of the certificate tree associated with the keys. The new clone refrains from using the keys until the administrator uploads the corresponding certificates to it.

7 Implementation

We implemented Excalibur in about 22,000 lines of C. This included the monitor, a client-side library providing the service interface, a client-side daemon for securing the CPABE decryption key on the cloud nodes, a management console, and a certificate toolkit for issuing certificates. The console communicates with the monitor over SSL, and all other protocols used UDP messages. We used the OpenSSL crypto library [37] and the CPABE toolkit [8] for all cryptographic operations, and we used the Trousers software stack and its related tools [51] to interact with TPMs.

We extended a cloud service so it could use Excalibur to help us understand the effort needed to adapt services

```
1324  sock.send("receive\n")
1325  sock.recv(80)
1326
1327  pipe = subprocess.Popen("/xen-/bin/seal",
1328     stdin=subprocess.PIPE,
1329     stdout=sock.fileno())
1330  fd_pipe = pipe.stdin.fileno()
1331
1332  XendCheckpoint.save(fd_pipe, dominfo, True,
1333     live, dst)
1334  os.close(fd_pipe)
1335  sock.close()
```

Figure 6: Hook to intercept migration (from file *XendDomain.py*.) We redirect the state of the VM through a process that seals the data before it proceeds to the destination on socket *sock* (lines 1327-1330).

for Excalibur and to estimate the performance impact of Excalibur on cloud services.

The example cloud service we adapted is an elastic VM service where customer VMs can be deployed in compute clusters in multiple locations, similar to Amazon's EC2 service. Our extension used Excalibur to better assure customers that their VMs would not be accidentally or intentionally moved outside of a cluster in a certain area (e.g., the EU).

Our base platform was Eucalyptus [36], an open source system that provides an elastic VM service with an EC2-compatible interface. Eucalyptus supports various VMMs; we used Xen [9] because it is open source.

Our implementation modified Xen to invoke seal and unseal when the customer's VM was created on a new node, migrated from one node to another, or suspended on one node and resumed on another. An attempt to migrate the VM to a node outside the specified locations would fail because the node would lack the credentials to unseal the policy-sealed VM.

Implementing these changes was straightforward. Integration with Excalibur required modifications to Xen, in particular to a Xen daemon called *xend*, which manages guest VMs on the machine and communicates with the hypervisor through the OS kernel of Domain 0. In particular, the VM operations *create*, *save*, *restore*, and *migrate* sealed or unsealed the VM memory footprint whenever the VM was unloaded from or loaded to physical memory, respectively. To streamline this implementation, we took advantage of the fact that *xend* always transfers VM state between memory and the disk or the network in a uniform manner using file descriptors. Therefore, we located the relevant file descriptors and redirected their operations through an OS process that sealed or unsealed according to the transfer direction. Figure 6 shows a snippet of *xend* that illustrates this technique applied to migration. Overall, our code changes were minimal: we added/modified 52 lines of Python code to *xend*.

The other two changes we made included: (1) hardening the software interfaces to prevent the system administrator from invoking any VM operations other than the four noted above, and (2) using a TPM-aware bootloader [5] to measure software integrity and to extend a TPM register with the Xen configuration fingerprint.

8 Evaluation

This section evaluates the correctness of Excalibur protocols using an automated tool. We also assess the performance of Excalibur and our example service.

8.1 Protocol Verification

We verified the correctness of our protocols using an automated theorem prover. We used a state-of-the-art tool, ProVerif [12], which supports the specification of security protocols for distributed systems in concurrent process calculus (pi-calculus).

To use the tool, we specified all protocols used by our system, which included all cryptographic operations (including CPABE operations), a simplified model of the TPM identity and fingerprint, the format of all certificate types in the system, the monitor protocols, and seal and unseal operations. In total, the specification contained approximately 250 lines of code in pi-calculus.

ProVerif proved the semantics of policy-sealed data in the presence of an attacker with unrestricted network access. The attacker could listen to messages, shuffle them, decompose them, and inject new messages into the network; this model covers, for example, eavesdropping, replay, and man-in-the-middle attacks. ProVerif proved that whenever a customer sealed data, the resulting envelope could be unsealed only by a node whose configuration matched the policy. We provide the specification and proof online [35].

8.2 Performance Evaluation

To evaluate Excalibur's performance, we first evaluated the monitor's scalability by measuring its performance overhead as well as its throughput for its three main activities: generating CPABE decryption keys, delivering these keys to nodes, and serving monitor attestation requests. We then measured the performance overhead of seal and unseal on the client side.

8.2.1 Setup and Methodology

We used two different experimental setups. The first used a two-node testbed; one node acted as a monitor, and the other acted as a regular cloud node making requests to the monitor. The second setup was used to evaluate the monitor throughput for attesting cloud nodes and serving customer attestation requests. For attesting cloud nodes, we simulated 1,000 nodes by using one machine acting as the monitor and five machines acting as cloud nodes, all running parallel instances of the node attestation protocol. For monitor attestations, we used a single machine acting as customers running

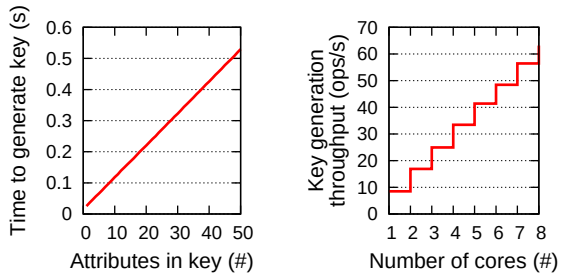

Figure 7: Performance of decryption key generation. Time to generate key as we vary the number of attributes (left), and throughput for 10 attributes as we vary the number of cores (right).

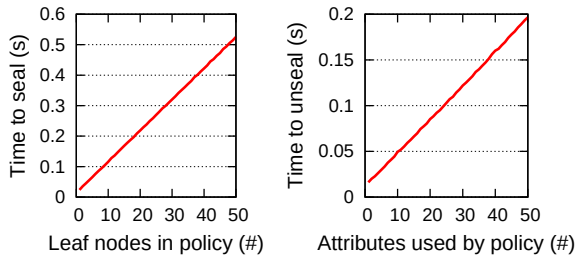

Figure 8: Performance overhead of sealing and unsealing data as a function of the complexity of the policy, with input data of constant size (1K bytes).

parallel instances of the monitor attestation protocol. This number of nodes was sufficient to exhaust monitor resources and ensure that there were no bottlenecks in the client nodes.

Both setups used Intel Xeon machines, each one equipped with 2.83GHz 8-core CPUs, 1.6GB of RAM, and TPM version 1.2 manufactured by Winbond. All machines ran Linux 2.6.29 and were connected to a 10Gbps network. We repeated each experiment ten times and reported median results; the standard deviation was negligible.

8.2.2 Decryption Key Generation

The overhead of generating a CPABE decryption key depends on the number of attributes embedded in the key. We measured the time to generate a decryption key stemming from the same master key, in which we varied the number of attributes from one to 50. This range seemed reasonable to characterize a node configuration.

Figure 7 shows the results, which confirm two relevant findings of the original authors of CPABE. First, the overhead of generating keys grows linearly with the number of attributes present in the key. Second, generating CPABE keys is expensive, e.g., a key with ten attributes took 0.12 seconds to create, which corresponds to a maximum rate of 8.33 keys/sec on a single core.

Although CPABE key generation is inherently inefficient, we consider that its performance is acceptable when throughput pressure on the monitor is relatively low because large groups of machines are likely to have the same configuration. The latency to generate a key is experienced only by the first node that reboots with a configuration new to the monitor. Since the key is cached, it is reused in future identical requests without additional costs.

8.2.3 Node Attestation

The latency of the node attestation protocol took 0.82 seconds. The bulk of the attestation cost (96%) was due to the node's performing a TPM quote operation necessary for remote attestation. This result is not surprising since such operations are known to be inefficient [31].

Most of the work required by this protocol is carried out by cloud nodes. Therefore, the attestation latency should not represent a bottleneck to the coordinator. To confirm this, we evaluated the monitor's throughput when running multiple parallel instances of this protocol. Results showed that the monitor could deliver up to 632.91 keys per second, which is efficient and would allow a single monitor machine to scale to serve a large number of nodes.

8.2.4 Monitor Attestation

We measured the performance of the monitor attestation protocol. This protocol had a latency of 1.21 seconds and a throughput of approx. 4800 reqs/sec on a single node. The quote operation performed by the monitor's local TPM accounted for the bulk of the latency (0.82 seconds), and the remaining time was due to cryptographic operations and network latency. The high peak throughput we observed was enabled by batch attestation. When we disabled batching, the throughput dropped sharply to 0.82 reqs/sec. Thus, this technique is crucial to the scalability of the monitor and delivered a throughput speedup of over 5000x.

8.2.5 Sealing and Unsealing

The performance overhead of the seal and unseal operations performed by Excalibur clients was dominated by the two cryptographic primitives: CPABE and symmetric cryptography (which uses AES with a 256-bit key size). We study their effects in turn.

To understand the overall performance overhead of CPABE, we set the input data to a small, constant size. Figure 8 shows the performance overhead of sealing and unsealing 1KB of data as a function of policy complexity. On the left is the cost of a seal operation as a function of the number of tests contained in the policy. For instance, policy *A=x and (B=y or B=z)* contains three comparisons. Our findings show that the sealing cost grows linearly with the number of attributes. The cost of sealing for a policy with 10 attributes was about 128 milliseconds.

On the right, Figure 8 shows the cost of an unseal operation. Unlike encryption, CPABE decryption depends

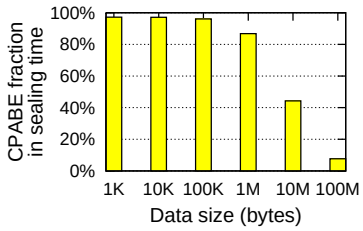

Figure 9: CPABE fraction in the performance overhead of sealing, varying the size of the input data.

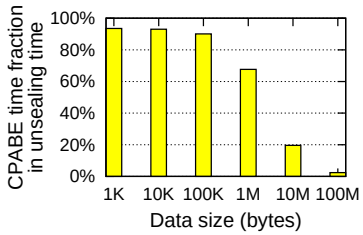

Figure 10: CPABE fraction in the performance overhead of unsealing, varying the size of the input data.

on the number of attributes in the decryption key that are used to satisfy the policy. For example, consider a decryption key with attributes *A:x* and *B:y*, and policies P_1 : *A=x*, and P_2 : *A=x and B=y*. Policy P_1 uses one attribute, whereas P_2 uses two. As before, the performance overhead of unseal grows linearly with the size of the policy. The time required to unseal a policy with 10 attributes was 51 milliseconds.

To study the relative effect of CPABE on the overall performance of Excalibur primitives, we varied the size of the input data. Figures 9 and 10 show the fraction of overhead due to CPABE, and Table 5 lists the absolute operation times. Our findings show that CPABE accounts for the most significant fraction of performance overhead. Sealing 1 MB of data with a policy containing 10 leaf nodes took 134 milliseconds, and 87% of the total cost of sealing was due to CPABE encryption. For unsealing, the fraction of CPABE was slightly lower than for sealing, but it was still very significant. Unsealing 1 MB of data with a policy satisfying 10 attributes of the private key took 68 milliseconds, where 68% of the latency was due to CPABE.

In summary, our evaluation of Excalibur showed these results: the costs of generating decryption keys and the node attestation protocol were reasonable when taking into account how infrequently they are required; the monitor scaled well with the number of cloud customers that used the service for the first time and with the number of cloud nodes that were attested upon reboot; the monitor could be further scaled up using cloning, and the latency of seal and unseal was reasonable and dominated by the cost of symmetric key encryption for large data items.

Data (bytes)	Latency (ms)	
	Sealing	Unsealing
1K	120	50
10K	120	49
100K	121	51
1M	134	68
10M	264	243
100M	1522	1765

Table 5: Performance overhead of sealing and unsealing data, varying the size of the input data.

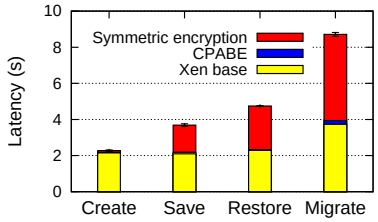

Figure 11: Latency of VM operations in Xen. Encrypting the VM state accounts for the largest fraction of the overhead, while the execution time of CPABE is relatively small. Encryption runs AES with 256-bit key size.

8.3 Cloud Compute Service

We now evaluate the performance overhead that the changes to Xen incur on its VM management operations, namely *create*, *save*, *restore* and *migrate*. We measured the time to complete each operation using an example VM for 10 trials. The example VM ran a Debian Lenny distribution, with Linux-xen 2.6.26, used a 4GB disk image, and its memory footprint was 128MB.

Figure 11 shows the results of our experiments. The performance impact is noticeable, especially for the *save*, *restore*, and *migrate* operations, where the completion time roughly doubled. The overhead, however, came from encrypting the VM's entire memory footprint; using Excalibur to secure or recover the encryption key added a small delay. Unlike the other operations, *create* experienced a small overhead increase of only 4%. This is because the system only decrypted the kernel image, which occupied 4.6MB, instead of the larger VM footprint as it did for the other operations.

As the results show, seal and unseal introduced noticeable overhead to the VM operations due to the symmetric encryption of the VM image. However, given that these operations occur infrequently, and considering the additional benefits to data security, we argue that these results reflect an acceptable trade-off between security and performance.

9 Related Work

Over the past several years, there has been considerable work on trusted computing [38]. Most of this work targets single computers with the goal of enforcing application runtime protection [16,20,26,30,31], virtualizing

trusted computing hardware [10], and devising remote attestation solutions based on both software [18,48] and hardware [13, 21, 42–44, 49]. Other work, focusing on distributed environments, provides integrity protection on shared testbeds [14] or distributed mandatory access control [29]. More recently, trusted computing primitives have been adapted to mobile scenarios to provide increased assurances about the authenticity of data generated by sensor-equipped smartphones [27]. Our work concentrates on the specific challenges of cloud computing environments, which fall outside the scope of these prior efforts.

Excalibur shares some ideas with property-based attestation [42], whose goal is to make hash-based software fingerprints more meaningful to humans. Like Excalibur, property-based attestation maps low-level fingerprints to high level attributes (properties) and relies on a monitor (controller) to perform this mapping. However, this prior work offers an abstract model without an associated system. Moreover, Excalibur extends this work by proposing new trusted computing primitives.

Nexus [50], a new operation system for trustworthy computing, introduces active attestation, which allows attesting a program's application-specific runtime properties and supports access control policies per application. Both Nexus policies and policy-sealed data can bind data based on attributes. However, the two systems target complementary problems: Nexus policies focus on nodes running Nexus and restrict the applications that can access the data; Excalibur policies focus on multi-node settings and restrict the cloud nodes that can access the data, supporting multiple software platforms. Thus, Nexus would be a good candidate to use as an attribute in an Excalibur policy.

The work by Schiffman et al. [47] aims to improve the transparency of IaaS cloud services by providing customers with integrity proofs of their VMs and underlying VMMs. Like Excalibur, a central component, called cloud verifier (CV), mediates attestations of nodes and uses high-level properties (attributes) for reasoning about node configurations. However, the scope of this work is narrower than ours: while the CV provides only integrity proofs, Excalibur builds on these proofs to enforce policy-sealed data, which is a general, data-centric abstraction for protecting customer data in the cloud. In addition, the CV administrator is assumed to be trustworthy, representing a weaker threat model; in our view, this assumption does not address an important class of problems that occur in cloud services today. Finally, their system does not address the shortcomings of sealed storage TPM primitives, which could raise concerns of data management inflexibility and isolation crippling if these primitives need to be used by cloud services to secure persistent data.

Multiple software systems have been proposed to increase the security of sensitive data. At the OS layer, hypervisors and OSes can protect the confidentiality and integrity of data using isolation [24, 30, 39, 53] or information flow control [52] techniques. At the middleware layer, frameworks that build Web services to offer their users strict control over their data hosted at the provider site [22] enable controlled sharing of sensitive data using differential privacy [41] or provide general-purpose encapsulation mechanisms for data [28]. These proposals are complementary to our work: despite their potential to increase security and control over data in the cloud, these proposals lack a scalable mechanism for bootstrapping trust in the multi-node cloud environment. By combining these platforms with Excalibur, cloud providers could build new trusted cloud services.

10 Conclusion

This paper presented Excalibur, a system that implements policy-sealed data. This new abstraction addresses the limitations of trusted computing when used in the cloud and enables the design of trusted cloud services. Excalibur leverages TPMs, a novel architecture with a set of associated protocols, and CPABE to offer developers two new primitives, seal and unseal, for constructing cloud services with stronger protection over how data is managed. We demonstrated the simplicity and flexibility of policy-sealed data by using Excalibur to build an elastic VM cloud computing service based on Xen and Eucalyptus, which accesses customer's data only on customer-approved platform configurations.

Acknowledgements: We would like to thank Peter Drushel, Pedro Fonseca, Aniket Kate, Jay Lorch, Massimiliano Marcon, Bryan Parno, Himanshu Raj, and Alec Wolman for their valuable comments and conversations that improved our work. We are also grateful to the anonymous reviewers and Mihai Christodorescu, our shepherd, for their feedback.

References

[1] Blippy Users Credit Card Numbers Exposed in Google Search Results. `http://mashable.com/2010/04/23/blippy-credit-card-numbers`.

[2] Cloudcamp: Five key concerns raised about cloud computing. `http://www.itnews.com.au/News/223980,cloudcamp-five-key-concerns-raised-about-cloud-computing.aspx`.

[3] Federal Government's Cloud Plans: A $20 Billion Shift. `http://www.cio.com/article/671013/Federal_Government_s_Cloud_Plans_A_20_Billion_Shift`.

[4] T-mobile: All your sidekick data has been lost forever. `http://mashable.com/2009/10/10/t-mobile-sidekick-data`.

[5] Trusted GRUB. `http://trousers.sourceforge.net/grub.html`.

[6] Verizon to Put Medical Records in the Cloud. http://www.networkcomputing.com/cloud-computing/229501444.

[7] Insecurity of Privileged Users: Global Survey of IT Practitioners. Technical report, Ponem Institute and HP, 2011. http://h30507.www3.hp.com/hpblogs/attachments/hpblogs/666/62/1/HP%20Privileged%20User%20Study%20FINAL%20December%202011.pdf.

[8] Advanced Crypto Software Collection. http://acsc.cs.utexas.edu.

[9] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP*, 2003.

[10] S. Berger, R. Cáceres, K. A. Goldman, R. Perez, R. Sailer, and L. van Doorn. vTPM: virtualizing the trusted platform module. In *USENIX Security Symposium*, 2006.

[11] J. Bethencourt, A. Sahai, and B. Waters. Ciphertext-policy attribute-based encryption. In *Symposium on Security and Privacy*, 2007.

[12] B. Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *CSFW*, 2001.

[13] E. Brickell, J. Camenisch, and L. Chen. Direct Anonymous Attestation. In *CCS*, 2004.

[14] C. Cutler, M. Hibler, E. Eide, and R. Ricci. Trusted disk loading in the Emulab network testbed. In *WCSET*, 2010.

[15] ENISA. Cloud Computing - SME Survey, 2009. http://www.enisa.europa.eu/act/rm/files/deliverables/cloud-computing-sme-survey/.

[16] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A Virtual Machine-Based Platform for Trusted Computing. In *SOSP*, 2003.

[17] T. C. Group. TPM Main Specification Level 2 Version 1.2, Revision 130, 2006.

[18] V. Haldar, D. Chandra, and M. Franz. Semantic Remote Attestation - A Virtual Machine directed approach to Trusted Computing. In *VM*, 2004.

[19] J. Hamilton. An Architecture for Modular Data Centers. In *CIDR*, 2007.

[20] H. Härtig, M. Hohmuth, N. Feske, C. Helmuth, A. Lackorzynski, F. Mehnert, and M. Peter. The Nizza secure-system architecture. *CollaborateCom*, 2005.

[21] T. Jaeger, R. Sailer, and U. Shankar. PRIMA: policy-reduced integrity measurement architecture. In *SACMAT*, 2006.

[22] J. Kannan, P. Maniatis, and B.-G. Chun. Secure data preservers for web services. In *WebApps*, 2011.

[23] M. Keeney. Insider Threat Study: Computer System Sabotage in Critical Infrastructure Sectors. Technical report, U.S. Secret Service and CMU, 2005. http://www.secretservice.gov/ntac/its_report_050516.pdf.

[24] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *SOSP*, 2009.

[25] E. Kowalski. Insider Threat Study: Illicit Cyber Activity in the Information Technology and Telecommunications Sector. Technical report, U.S. Secret Service and CMU, 2008. http://www.secretservice.gov/ntac/final_it_sector_2008_0109.pdf.

[26] D. Lie, C. A. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. C. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software. In *ASPLOS*, 2000.

[27] H. Liu, S. Saroiu, A. Wolman, and H. Raj. Software abstractions for trusted sensors. In *MobiSys*, 2012.

[28] 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 *HotOS*, 2011.

[29] J. M. McCune, T. Jaeger, S. Berger, R. Caceres, and R. Sailer. Shamon: A System for Distributed Mandatory Access Control. In *ACSAC*, 2006.

[30] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. D. Gligor, and A. Perrig. TrustVisor: Efficient TCB Reduction and Attestation. In *IEEE Symposium on Security and Privacy*, 2010.

[31] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An Execution Infrastructure for TCB Minimization. In *EuroSys*, 2008.

[32] Microsoft. BitLocker Drive Encryption. http://www.microsoft.com/whdc/system/platform/hwsecurity/default.mspx.

[33] A. G. Miklas, S. Saroiu, A. Wolman, and A. D. Brown. Bunker: a privacy-oriented platform for network tracing. In *NSDI*, 2009.

[34] D. G. Murray, G. Milos, and S. Hand. Improving Xen security through disaggregation. In *VEE*, 2008.

[35] N. Santos. ProVerif scripts for the Excalibur protocols, 2011. http://www.mpi-sws.org/~nsantos/excalibur/xcproverif.zip.

[36] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. Eucalyptus: A Technical Report on an Elastic Utility Computing Architecture Linking Your Programs to Useful Systems. Technical Report 2008-10, UCSB.

[37] OpenSSL. http://www.openssl.org.

[38] B. Parno, J. M. McCune, and A. Perrig. Bootstrapping trust in commodity computers. In *IEEE Symposium on Security and Privacy*, 2010.

[39] H. Raj, D. Robinson, T. B. Tariq, P. England, S. Saroiu, and A. Wolman. Credo: Trusted Computing for Guest VMs with a Commodity Hypervisor. Technical Report MSR-TR-2011-130, Microsoft Research, 2011.

[40] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, You, Get Off of My Cloud! Exploring Information Leakage in Third-Party Compute Clouds. In *CCS*, 2009.

[41] I. Roy, S. T. Setty, A. Kilzer, V. Shmatikov, and E. Witchel. Airavat: Security and privacy for mapreduce. In *NSDI*, 2010.

[42] A.-R. Sadeghi and C. Stüble. Property-based attestation for computing platforms: caring about properties, not mechanisms. In *NSPW*, 2004.

[43] R. Sailer, T. Jaeger, X. Zhang, and L. van Doorn. Attestation-based policy enforcement for remote access. In *CCS*, 2004.

[44] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and implementation of a tcg-based integrity measurement architecture. In *USENIX Security Symposium*, 2004.

[45] N. Santos, K. P. Gummadi, and R. Rodrigues. Towards trusted cloud computing. In *HotCloud*, 2009.

[46] N. Santos, R. Rodrigues, K. Gummadi, and S. Saroiu. Excalibur: Building Trustworthy Cloud Services. Technical Report MPI-SWS-2011-004, MPI-SWS, 2011.

[47] J. Schiffman, T. Moyer, H. Vijayakumar, T. Jaeger, and P. McDaniel. Seeding clouds with trust anchors. In *WCCS*, 2010.

[48] A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla. Swatt: Software-based attestation for embedded devices. *IEEE Symposium on Security and Privacy*, 2004.

[49] E. Shi, A. Perrig, and L. V. Doorn. Bind: A fine-grained attestation service for secure distributed systems. In *IEEE Symposium on Security and Privacy*, 2005.

[50] 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 *SOSP*, 2011.

[51] TrouSerS. http://trousers.sourceforge.net.

[52] S. Vandebogart, P. Efstathopoulos, E. Kohler, M. Krohn, C. Frey, D. Ziegler, F. Kaashoek, R. Morris, and D. Mazières. Labels and event processes in the asbestos operating system. *ACM Trans. Comput. Syst.*, 2007.

[53] F. Zhang, J. Chen, H. Chen, and B. Zang. Cloudvisor: Retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *SOSP*, 2011.