Excalibur: Building Trustworthy Cloud Services

Nuno Santos — *MPI-SWS*
Rodrigo Rodrigues — *MPI-SWS*
Krishna P. Gummadi — *MPI-SWS*
Stefan Saroiu — *Microsoft Research*

Technical Report: MPI-SWS-2011-004

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 for 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. Excalibur enables the design of trustworthy cloud services by providing a new trusted computing abstraction called *policy-sealed data*. This abstraction enables data to be *sealed* (i.e., encrypted to a customer-defined policy) such that it can only be *unsealed* (i.e., decrypted) 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 that are 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 data is processed exclusively by nodes that meet their preferences.

1 Introduction

Managing cloud computing services is a complex and error-prone task. Cloud providers delegate this task to skilled cloud administrators who are responsible for managing the software of the cloud infrastructure. However, it is difficult to assure that their actions are error free. In particular, an accidental or, in some cases, an intentional action from a cloud administrator may lead to leaking, corrupting or losing customer data. In fact, the threat of potential violations to the integrity and confidentiality of customer data is often cited as one of the main barriers to the adoption of cloud services [2, 14]. Furthermore, high-profile incidents that involved the loss of confidentiality or integrity of customer data [1, 4] and the growing amount of security-sensitive data that is outsourced to the cloud [3, 6] only heightens such concerns.

Recently, several proposals [21, 39, 45] have advocated leveraging trusted computing technology to help build cloud services that are more resilient to the violation of the integrity and confidentiality of customer data. This technology relies on specialized hardware – typically a Trusted Platform Module (TPM) chip [16] – that could be deployed on every node that is part of the cloud infrastructure. Each TPM chip stores on its die a strong identity (unique key) and a fingerprint (hash) of the software stack that booted on the cloud node. TPMs can be used to restrict the upload of customer data to cloud nodes whose identity or fingerprint are considered *trusted*. This capability offers a building block in the design of trustworthy cloud services that provides customers better guarantees for protecting the confidentiality and integrity of their data against insiders, or confining their data location within desired geographical or jurisdictional boundaries.

Despite their benefits, the current trusted computing abstractions are ill-suited for the requirements of cloud services, and this increases the burden of adoption of such technology in the cloud. This is because TPM abstractions were designed for protecting data and secrets on a standalone machine, and, consequently, they are cumbersome to use in a multi-node datacenter environment, in which data can be migrated across multiple nodes. In addition, TPM abstractions force the cloud infrastructure to be over-exposed, since they reveal the identity and software fingerprint of individual cloud nodes, which external agents could use to exploit vulnerabilities in the cloud infrastructure or gain business advantage [34]. Furthermore, the current TPM implementation of these abstractions is inefficient, which can introduce scalability bottlenecks to cloud services.

This paper presents Excalibur, a system that provides the designers of cloud services with new trusted computing abstractions that overcome these barriers. These abstractions can be used as a key building block for the construction of services that offer better guarantees regarding the integrity, confidentiality, or location of customer data. The design of Excalibur includes two main innovations, which are crucial to overcoming the concerns with using TPMs in the cloud.

First, Excalibur provides a new trusted computing abstraction called *policy-sealed data*, that allows customer data to be encrypted 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 in a way that addresses the limitations of current TPM abstractions. Since it allows policies to be specified using human readable attributes, policy-sealed data hides the low-level identities and software fingerprints of nodes. Moreover, it allows for data to be flexibly accessed by multiple nodes with or without identical configurations as long as they satisfy the customer policies.

Second, Excalibur implements the policy-sealed data abstraction in a way that overcomes the inefficiency hurdles of current TPMs and scales to the demand of cloud services. For this, we design 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 [10] which drastically reduces the communication needs between the monitor and the production nodes. With our protocols, each node needs to contact the monitor only once during a boot cycle, a relatively infrequent operation. We also demonstrate the correctness of Excalibur's cryptographic protocols by using a protocol verifier [11].

To demonstrate the practicality of Excalibur, we built a proof-of-concept compute service akin to EC2. Our new service is based on the Eucalyptus open source cloud management platform [31] and leverages 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: we made minimal modifications to the Eucalyptus codebase to integrate Excalibur into the service.

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

2 Brief primer on trusted computing and TPMs

Trusted computing technology provides the hardware support to bootstrap trust in a computer [33]. In particular, this technology can enable a remote user to check whether a computer is trustworthy by verifying what software platform has been booted on the machine. This feature is crucial for checking the integrity of systems that have been designed to protect the confidentiality and integrity of data [19, 28].

Trusted computing requires the presence of special hardware on a computer, and provides the system designers with four main abstractions: strong identities, trusted boot, remote attestation, and sealed storage. *Strong identities* allow the computer to be uniquely identified without having to trust the OS or the software running on the computer. *Trusted boot* produces a unique *fingerprint* of the software platform running on the computer which consists of hashes of the software platform components (e.g., BIOS, firmware controlling the computer's devices, bootloader, OS) computed at boot time. This fingerprint can be securely reported to a remote party using a *remote attestation* protocol, which allows the remote party to authenticate both the computer and the software platform so that it can assess whether the computer is trustworthy. Finally, *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. For this, the secrets are encrypted in such a way that they can only be decrypted by the same computer running the trusted software platform specified upon encryption.

The most common special hardware used to implement these trusted computing abstractions is a chip called the Trusted Platform Module (TPM) [16]. A TPM is a secure co-processor, which is widely deployed on desktops, laptops and increasingly on servers. A TPM offers a strong identity which can be provided by an Attestation Identity Key (AIK). To keep track of the hash values that constitute a fingerprint, the TPM uses special registers called Platform Configuration Registers (PCRs). Whenever a reboot takes place, the PCRs are reset and updated with new hash values. To assist 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. This allows the unseal primitive to validate the identity and fingerprint of the software platform before decrypting sealed data.

3 The case for policy-sealed data

This section makes the case for a new trusted computing abstraction called *policy-sealed data* designed to serve the needs of cloud computing. Before describing our abstraction, we start by discussing the limitations of existing TPM abstractions in the context of the design of a trustworthy cloud service.

3.1 Strawman design of a trustworthy cloud service

When applied to the cloud, trusted computing is a crucial block for building trustworthy cloud services — cloud services that offer guarantees to customers. For example, a trustworthy cloud service akin to Amazon's EC2 could provide better protections against inspection or corruption of customers' VMs by a cloud administrator.

The research community and industry are already designing systems that offer such guarantees but *on a single node only*. For example, a recent research project [45], called CloudVisor, retrofits Xen in such a way that the hypervisor guarantees the integrity and confidentiality of the data and software running in guest VMs. A customer can leverage the TPM's remote attestation abstraction to verify that a cloud node is running CloudVisor before uploading data to the cloud.

However, such a verification step only checks these guarantees for the cloud node where the data is uploaded first. Once in the cloud, the customer's data and VMs are often migrated from one node to another, or suspended to disk and resumed at a later point in time. To offer end-to-end protection, these checks must be performed throughout these multiple stages.

To accommodate VM migration, a strawman design of a trustworthy EC2 would perform remote attestation each time a customer's VM is migrated 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 suspending the VM onto disk. This design confines VMs to cloud nodes running CloudVisor, thereby keeping the VMs safe from the actions of cloud administrators.

3.2 Limitations of TPM abstractions

Current TPM abstractions are not well suited for building trustworthy cloud services similar to the strawman design described above. Their limitations are fundamental – TPMs were designed to offer guarantees about one single computer. In particular, TPMs suffer from three major problems when used for building trustworthy cloud services.

First, the sealed storage abstraction is not designed for a distributed and dynamic environment like the datacenters where cloud services operate. In particular, it precludes the application developer from encrypting and storing sensitive data in an untrusted medium (e.g., a local hard drive, or Amazon S3 service) and retrieving it on a different node or the same node running a different trustworthy configuration, without knowing the future configuration at encryption time. In the previous example, one might be interested in suspending the VM to disk and resume it at a later point on a different node (e.g., if in the meanwhile the original node had been shut down for saving power) or on the same node running a different configuration (e.g., if in the meanwhile the hypervisor had been upgraded to a more recent version).

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

Finally, the cloud infrastructure may be severely 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.

3.3 The policy-sealed data abstraction

To overcome these limitations, we propose a new abstraction aimed at building trustworthy cloud services called *policy-sealed data*. 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 used 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 can refer to the software (e.g., "vmm", "version") or to the hardware (e.g., "location") of a node. A policy expresses a logical condition over the attributes supported by the provider (e.g., "vmm=Xen and location=US"). Table 1 gives an exam-

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 attributes of a service. Suppose 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. 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* = “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 zone preference for different sites, and P_3 expresses a regional preference.

ple 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 only needs to seal the data to a policy: if the destination cannot unseal the data then its configuration does not match the policy and therefore the node is not trustworthy to the needs of the customer who originally specified the policy.

Policy-sealed data addresses the limitations of standard TPM abstractions. First, data can be moved flexibly among the nodes that satisfy the same customer policy. Attributes can express a large variety of software and hardware configurations independently of the cloud service model, and policies let customers express their preferences using rich attribute expressions. Second, we will show how policy-sealed data can be implemented in an efficient manner bypassing the TPMs' performance limitations. Finally, our abstraction prevents infrastructure overexposure because it hides the nodes' individual TPM-based identities and fingerprints.

4 Excalibur design

This section presents Excalibur, a system that enables the design of trustworthy cloud services by providing policy-sealed data support. The intuition behind its design is simple: it is based on a centralized component called a *monitor*, which maps the abstractions used by policy-sealed data to the TPM-based identities and fingerprints. This design aspect is important because only the monitor will trigger TPM primitives on the cloud nodes, thus minimizing their negative performance impact. Though this fundamental design choice is simple, we still need to overcome two significant 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.

Next, we outline our assumptions. Then, we present an overview of Excalibur and show how we address the two design challenges above.

4.1 Assumptions and threat model

We consider that the cloud provider's employees responsible for administering the cloud infrastructure may behave erroneously. Such actions may result in leakage or corruption of customer data or computations. The causes for misbehavior may range from negligence (e.g., not applying security patches) to malice (e.g., theft of customer data). This motivates our adoption of an adversarial model, which contemplates the worst case scenario of an administrator actively trying to access customer data located on the cloud nodes. However, we differentiate physical from remote attacks.

Regarding physical attacks, which could compromise the correctness of TPMs, we assume that these are outside the reach of the attacker. The rationale is that providers already secure their premises. In particular, hardware in the cloud is put under sharp surveillance and restricted access control policies. In certain situations, physical accessis even completely disallowed [18].

Our focus is on attacks that can be performed remotely, which are far easier to perpetrate. In fact, cloud nodes are normally managed remotely through a management interface, which comprises the interface exposed by the software platform running on the node (e.g., the ability to login), and a dedicated interface for tasks like power cycling. The management interface gives an administrator great power to access customer data on the cloud nodes: e.g., it can reboot any node, access its local disk after rebooting, and install arbitrary software platforms on the node. We thus assume that the

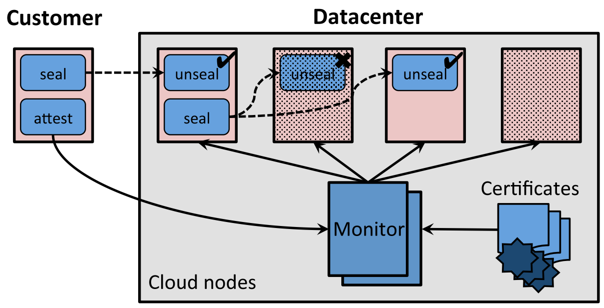


Figure 1: Excalibur deployment: The dashed lines show the flow of policy-sealed data, and the filled lines represent interactions between clients and the monitor. The monitor checks the configurations of every cloud node. The customer starts by attesting the monitor to check its integrity, and seals the data. The data can only be unsealed on nodes that satisfy the policy (represented by unshaded boxes).

attacker has remote access to the management interface of every cloud node and can also eavesdrop the network.

The software platforms that implement a trustworthy cloud service must satisfy three properties. First, the management interface exposed to cloud administrators must prevent operations that could lead to leakage or corruption of sensitive data on the node (e.g., direct memory inspection of VM memory). Second, a software platform should be able to protect sensitive data (e.g., key material) in volatile memory so that they are destroyed upon reboot; we rely on this feature for protecting some credentials that the monitor sends to the nodes. Third, the software platform must be crafted to force a reboot whenever the software platform changes (e.g., system upgrades); this operation ensures that the fingerprint stored in the TPM is consistent with the current state of the system. These properties can be enforced by making use of existing systems and hardening techniques [19, 22, 29, 45], and therefore this does not constitute the focus of this paper.

We assume that cloud providers themselves are not malicious, since it is in their best interest to leverage trusted computing to improve the security of their services. In particular, cloud providers will deploy a TPM chip on every cloud node.

4.2 System overview

Figure 1 illustrates the deployment of Excalibur in the cloud. Excalibur comprises two components: a *client* and a *monitor*. The client consists of a library that allows the implementation of a trustworthy cloud service to use the policy-sealed data primitives. This library can be used both at 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). Note that the customer-side client does not expose the unseal primitive; since the notion of a configuration only applies to cloud nodes, it only makes sense to invoke unseal on these nodes. With policy-sealed data, a trustworthy cloud service can then confine customer data to the nodes deemed trustworthy by the customers.

The monitor is a dedicated service running on either a single or a small set of cloud nodes. It forms the core of Excalibur since it coordinates the enforcement of policy-sealed data on the entire cloud infrastructure. It maps TPM identities and fingerprints of the cloud nodes to policy-sealed data attributes. 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 of the node as a set of attributes, is then encoded as credentials that are then sent to the node. These credentials are required by cloud nodes for unsealing policy-sealed data, and are destroyed whenever the nodes reboot.

The monitor exposes a narrow management interface that allows the cloud administrator to configure the mappings between attributes and identities / fingerprints. This is necessary for the regular maintenance of the system 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 the monitor is properly maintained, the monitor only accepts mappings that are vouched for by special *certificates*, and customers can directly attest the monitor in order to check its authenticity and integrity. Provided that customers trust the issuers of certificates, then the service is trustworthy.

Next we describe how we address the two main design challenges: scalable and efficient enforcement of policies, and customer trust bootstrap via the monitor.

4.3 Cryptographic enforcement of policies

The monitor needs to coordinate the cryptographic enforcement of the seal and unseal primitives. In particular, the data encrypted upon seal should only be decrypted upon unseal by the nodes whose configuration satisfy the policy used to seal the data. The main implementation challenge is avoiding scalability bottlenecks.

One possibility, which corresponds to the choice we had made in a preliminary design for this system, 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; then the monitor encrypts this key and the policy with a secret key, and sends the outcome back to the client. Upon unsealing, 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, which introduces a scalability bottleneck.

A second design is to evaluate the policies at the client side using public-key encryption. Each cloud node receives from the monitor a set of private keys that match its configuration, in which 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 at the client side, which alleviates the pressure on the monitor. The main shortcoming is that, in practice, it complicates key management due to the number of key-pairs that nodes need to handle in order to reflect all possible attribute combinations that can be used in policies.

4.3.1 The need for Attribute-Based Encryption

To address these problems, we use a cryptographic scheme called Ciphertext Policy Attribute-Based Encryption (CPABE) [10], which works as follows. It starts with the generation of 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 using 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 inequality operators (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 only be decrypted 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 example policy above).

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 keeps the master key securely. Whenever the monitor 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 and unsealing are done by encrypting the data using the encryption key and a policy, and decrypting the sealed data using the decryption key, respectively. The data can only be unsealed by nodes that satisfy the specified policy. The monitor protects the master key by 1) ensuring that the master key cannot be released through the monitor's management interface, and 2) encrypting the master key before storing it on disk as described in Section 5.3.

The benefits of this solution are twofold. First, it allows the system to scale independently of the workload since the seal and unseal primitives do not interact with the monitor, but run entirely at the client side. Second, it allows for 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 price to pay for using CPABE is a performance hit when compared to traditional cryptographic schemes. In Section 5 we explain how this impact is minimized.

4.4 Trusting the monitor

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

To achieve this, we first need to 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 have the responsibility of validating the correctness of mapping. For example a certifier would check the location of the nodes, and the fingerprints of the 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 inform the customers about the identity of the certifier so that they can judge whether the certifier is trustworthy, and thereby be confident that the attribute mappings are correct. Nevertheless, even if the certifier is trustworthy, the system must provide further 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 protocols implemented by the monitor are correct. To provide this guarantee, customers can directly attest the monitor. To perform this attestation, customers must obtain 1) the identity and fingerprint of the monitor using remote attestation, and

2) a certificate that validates that the identity belongs to a cloud node owned by the provider and that the fingerprint is of a trustworthy monitor implementation. If both conditions hold, it it safe to trust the monitor.

However, this solution raises a scalability issue, since the maximum throughput of a monitor clone equipped with a single TPM would be in the order of one attestation per second. This is clearly insufficient to cope with the demand of a cloud service, even if we spawn a reasonable number of monitor clones. To make this solution scalable, we employ a technique based on Merkle trees for batching a large number of attestation requests into a single TPM quote which we detail in Section 5.

5 Detailed design

This section describes how Excalibur enables building trustworthy cloud services by providing policy-sealed data support. We first introduce the 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 5.2.

5.1 Certificate specification

Excalibur uses certificates to validate the mappings between the attributes specific to a trustworthy cloud service and the identities / 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 aims to support multiple certifiers, since a single certifier may not have the expertise to assess all the attributes of the cloud service, or simply because the provider might be willing to hire multiple certifiers in order to reinforce customers' trust. Provided that customers trust the certifiers that issued the certificates, they can be sure that the mappings are correct, and thus policy-sealed data is properly enforced.

To meet the needs of our security protocols and support multiple certifiers, we allow individual attributes to be independently vouched for by different certifiers (e.g., just the VMM model or the location). As a result, certificates form a hierarchical tree like in the example

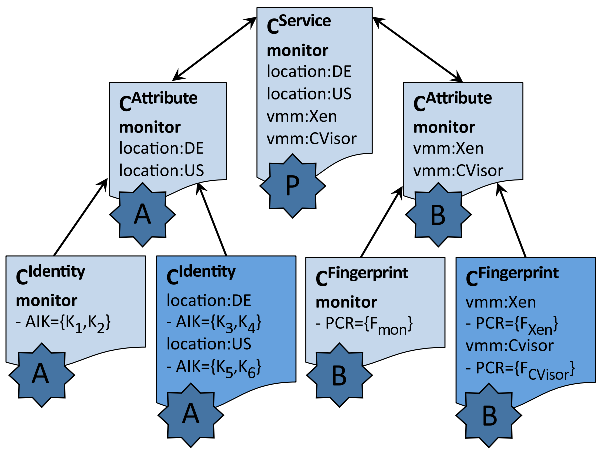


Figure 2: Example certificate tree. This covers an infrastructure that comprises two clusters with three nodes each, and three software platforms. Provider P certifies the service, certifier A the location of nodes, and certifier B their software fingerprints. The certificates in light colored boxes form the *manifest*; it validates the monitor's authenticity and integrity.

shown in Figure 2. This 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 different certifiers, A and B. Additionally, each leaf in the certificate tree vouches for a mapping between the attributes that appear in node configurations and the low-level measurements, namely software fingerprints (PCRs) or hardware identities (AIK keys).

5.2 System interfaces

The interface of Excalibur can be divided into two parts: a *service interface* to support the implementation of cloud services, and a *management interface* to allow cloud administrators to maintain the system.

The service interface exported by the client library consists of the three operations summarized in Table 4. Before the data can be sealed at the customer-side, the *attest-monitor* must be invoked to check the authenticity and integrity of the monitor. It returns the encryption key $K^{\mathbb{E}}$ needed for sealing and a *manifest* M which consists of a set of certificates that are necessary and sufficient for validating the identity and fingerprint of the monitor (see Figure 2). The manifest is passed to the customer, who can learn which attributes can be used in policies, and identify the provider and certifiers so that the customer can decide whether the service is trustworthy. Since the local client saves the manifest and encryption key for sealing, this operation only needs to be done the first time the cloud service is used.

The core primitives are *seal* and *unseal*. Seal can be invoked by both cloud nodes and customers, and it takes the encryption key $K^{\mathbb{E}}$, a policy P, and the data D,

attest-monitor(*mon-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.

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 these keys 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 language for expressing policies is the CPABE policy language.

The management interface allows the cloud administrator to remotely maintain the monitor using a console. Its main operations allow for initializing the system, managing certificates, and spawning monitor clones.

5.3 System initialization

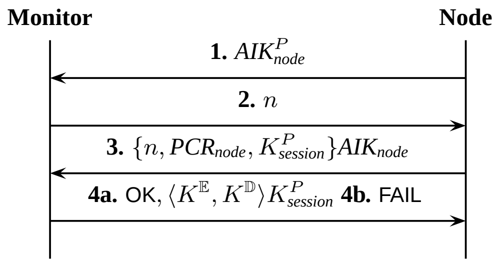
For a trustworthy cloud service to take advantage of policy-sealed data, the monitor must be initialized by binding a unique CPABE key pair to the service. To do this, the cloud administrator loads the certificates that validate the attributes of the service into the monitor, and instructs the monitor 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 root certificate of the tree (see Figure 2). To allow for maintenance of the system, 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 database*, and a *key database*. The certificate database contains the certificates loaded into the monitor. The key database contains all the CPABE keys. To secure the key material, the key database is sealed using the TPM seal primitive, which ensures that the key database can only be accessed under a trustworthy monitor configuration in case the monitor reboots.

Once the setup is complete, the monitor can check the configurations of cloud nodes, as explained below.

5.4 Node attestation protocol

The main task of the monitor is to deliver credentials to each cloud node reflecting the node's boot time configuration, which 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 will be lost. Therefore this protocol must be executed each time a cloud node reboots, so it can obtain a fresh credential.



Monitor
Node
1. AIK^{P}_{node}
2. n
3. $\{n, PCR_{node}, K^{P}_{session}\}AIK_{node}$
4a. OK, $\langle K^{\mathbb{E}}, K^{\mathbb{D}}\rangle K^{P}_{session}$ 4b. FAIL

Figure 3: Node attestation protocol.

To deliver the credentials to a node, the monitor starts by obtaining a quote from the node signed by the node's AIK and containing 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 joining all the attributes of the matching certificates onto a list similar to the one shown in Table 2. Next, the monitor sends the node the credentials, which include the encryption key, and the decryption key embedding these attributes. Since generating a new decryption key is expensive, the monitor caches these keys in the key database, and shares the same decryption key among the nodes with similar 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. We introduce a first message, sent by the node after it boots up, to request credentials. Message 3 includes a session key $K^{P}_{session}$, which will be used in message 4 for securely sending credentials $K^{\mathbb{E}}$ and $K^{\mathbb{D}}$ to the node.

A noteworthy aspect of this protocol is 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, the data can only be unsealed by a decryption key generated from the master key owned by the legitimate monitor; this property is guaranteed by CPABE. As a result, the attacker cannot compromise the customer's data. Next, we show how the customer obtains the encryption key from the legitimate monitor.

5.5 Monitor attestation protocol

The monitor attestation protocol is triggered by the *attest-monitor* operation and allows customers to detect

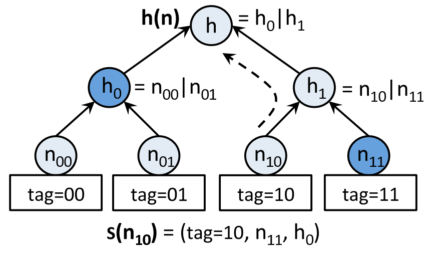
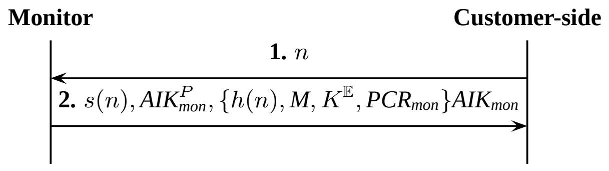


Figure 4: Batch attestation example: The tree is built from 4 nonces. A summary for nonce n_{10} comprises its tag and the hashes in the path to the root.



Monitor
Customer-side
1. n
2. $s(n), AIK^{P}_{mon}, \{h(n), M, K^{\mathbb{E}}, PCR_{mon}\}AIK_{mon}$

Figure 5: Monitor attestation protocol.

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 lets the customer check the identity of certifiers and learn about the attributes available in the service. The monitor is legitimate if its identity and fingerprint are validated by the manifest.

The main challenge in designing this protocol is scalability. To validate the monitor, the customer-side client must obtain a fresh quote from the monitor's TPM. The quote contains the monitor's PCRs and a nonce sent by the customer-side client for freshness; both elements are signed by the monitor's AIK. However, since this protocol depends on a fresh nonce signed in the TPM, the attestation throughput would be very low, which creates a scalability bottleneck as multiple customers run the protocol concurrently.

To overcome this problem, we need to batch multiple attestation requests into a single quote operation. More precisely, we want to allow the monitor to quote a batch of N nonces n_i expressed as an aggregate hash $h(n^{N}_{i=0})$, and yet send evidence $s(n_i)$ to each customer-side client that its nonce n_i is included in the aggregate hash.

To make the protocol network efficient when sending the evidence back to customers, the produced evidence must be significantly smaller than the accumulated batch of nonces. For this purpose, we use a Merkle tree as illustrated in the example of Figure 4. While the monitor TPM is busy processing a batch of requests, the monitor accumulates the nonces sent by multiple customers. As soon as the TPM becomes free, the monitor builds a Merkle tree where the leaves hold the nonces of the batch, each inner node is a hash of the concatenated children, and the head is the aggregate hash $h(n)$. The monitor quotes the TPM with the aggregate hash. When quote finishes, the monitor creates a summary $s(n)$ of size $O(log(N))$ for each nonce, and sends each summary to the respective customer-side client. To validate the freshness of the quote, a client takes its nonce n and the received summary $s(n)$, recomputes the aggregate hash, and compares the result with the received $h(n)$. This allows the monitor to efficiently accumulate many attestation requests into a single quote, which increases throughput dramatically.

We now explain the precise monitor attestation protocol 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", and uses them to authenticate the monitor key AIK^{P}_{mon}, 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 uses nonce n and the summary $s(n)$ to compare 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 safely.

5.6 Seal and unseal protocols

Thanks to the use of CPABE, invoking seal and unseal to respectively encrypt and decrypt data can be done without having to further contact the monitor (see Table 4).

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 allow cloud nodes to re-seal the data, we include the original policy in the envelope along with a digest for integrity protection.

5.7 Clone attestation protocol

To elastically scale the monitor, the cloud administrator can create multiple monitor clones. In creating a new clone, an existing monitor instance must share the CPABE master key with the new clone so that the latter can generate and distribute decryption keys to the cloud nodes. However, this can only be done 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 the one shown in Figure 3, but with two differences. First, after message 3, the monitor checks if the candidate is trustworthy by checking if the candidate's 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.

6 Implementation

We implemented Excalibur in about 22,000 lines of C code. This includes the implementation of 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 use UDP messages. We use the OpenSSL crypto library [32] and the CPABE toolkit [7] for all our cryptographic operations, and the Trousers software stack and its related tools [43] to interact with TPMs.

6.1 Example cloud compute service

We extended an existing cloud service to use Excalibur in order to examine two issues. First, we wanted to understand the effort it would take to adapt existing cloud services to use Excalibur. Second, we wanted to use our implementation to estimate the performance impact of integrating Excalibur into cloud services.

The example cloud service we decided to adapt is an elastic VM service where customer VMs can be deployed in compute clusters in multiple locations, similar to the EC2 service from Amazon. Our extension uses Excalibur to enable a customer to gain better assurance that its VMs will not be accidentally or intentionally moved outside of the provider's compute cluster in a certain area (e.g., the EU).

To build such a service, we started from Eucalyptus [31], which is an open source system that provides an elastic VM service with an interface compatible to that of Amazon EC2. It consists of several subsystems for controlling the allocation of virtual machines. Each node runs a virtual machine monitor (VMM) for hosting the VMs, which Eucalyptus coordinates centrally. Eucalyptus supports various VMMs, but we used Xen [8] because it is open source.

To implement the confinement of VM placement we modified Xen to invoke seal and unseal when the customer's VM is 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

```
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 (*seal.sh*), which seals the data before the data proceeds to the destination on socket *sock* (lines 1327-1330).

node outside the specified locations would fail, as the node would lack the credentials to unseal the policy-sealed VM.

Implementing these changes was relatively straightforward. The integration with Excalibur required modifications to Xen, in particular to a Xen daemon called *xend*, which is responsible for the management of the 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* need to seal or unseal the VM memory footprint whenever the VM is 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 redirect the operations on these file descriptors through an OS process that seals or unseals according to the direction of transfer. Figure 6 shows a snippet of *xend* which illustrates this technique applied to migration. The modifications took place in the Python code in *xend*. Overall, the changes to *xend* were minimal: we added/modified 52 lines of python code.

The other two changes we needed to perform consisted of (i) hardening the software interfaces to prevent the system administrator from invoking any operations other than the four VM management operations listed above, and (ii) using a TPM-aware bootloader [5] to measure the integrity of the software and to extend a TPM register with the Xen configuration fingerprint.

Overall, adapting Eucalyptus to use Excalibur primitives required little programming effort.

7 Evaluation

In this section we evaluate the correctness of Excalibur protocols using an automated tool, and the performance of both Excalibur and our example cloud service.

7.1 Protocol verification

We verified the correctness of Excalibur protocols using an automated theorem prover. We used a state-of-the-art tool named ProVerif [11], which supports the specification of security protocols for distributed systems in concurrent process calculus (pi-calculus).

To use the tool, we specified all the 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 contains approximately 250 lines of code in pi-calculus.

ProVerif was able to prove the semantics of policy-sealed data in the presence of an attacker with unrestricted access to the network. The attacker can 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 seals data, the resulting envelope can only be unsealed by a node whose configuration matches the policy.

For more details about the specification and the proof, we refer the interested reader to [30].

7.2 Excalibur performance

The performance evaluation of Excalibur has two parts. First, we evaluate the scalability of the monitor by measuring the performance overhead and throughput of its three main activities: generating CPABE decryption keys, delivering these keys to nodes, and serving monitor attestation requests. We then measure the performance overhead of seal and unseal operations at the client side.

7.2.1 Experimental setup and methodology

To evaluate Excalibur's performance, we use two different experimental setups. The first setup uses a two-node testbed where one of the nodes acts as a monitor and the other acts as a regular cloud node making requests to the monitor. The second setup is used to evaluate the monitor throughput for attesting cloud nodes and serving attestation requests by customers. For attesting cloud nodes, we simulate 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 use a single machine acting as customers running parallel instances of the monitor attestation protocol. This number of nodes is sufficient to exhaust the resources of the monitor, and to ensure that there were no bottlenecks in the client nodes.

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

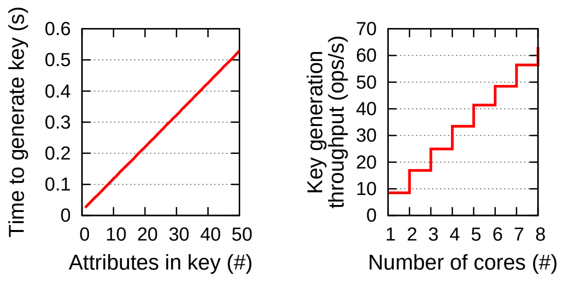


Figure 7: Performance of decryption key generation: time as we vary the number of attributes (left), and throughput for 10 attributes as we vary the number of cores (right).

7.2.2 Decryption key generation

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

Figure 7 shows, on the left, the performance overhead for generating a decryption key as a function of the number of attributes it contains. On the right, we show the peak throughput of key generation with 10 attributes on a single processor using up to eight cores. The results 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 takes 0.12 seconds to create, which would correspond 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 in this context. The reason is that the 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 will only be experienced by the first node which reboots with a configuration that is new to the monitor. Since the key is cached, it is reused in future identical requests without additional costs.

7.2.3 Node attestation

We measured the latency of the node attestation protocol, and found that it takes 0.82 seconds. We also observe that the bulk of the attestation cost (96%) is due to the node performing a TPM quote operation necessary

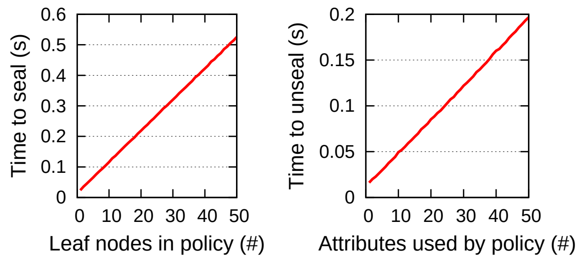


Figure 8: Performance overhead of sealing and unsealing data as a function of the complexity of the policy, with input data of constant size (1K bytes).

for remote attestation. This result is not surprising as TPM quote operations are known to be inefficient [27].

Since most of the work required by this protocol is carried out by the cloud nodes, this should not represent a bottleneck to the coordinator. To confirm this, we evaluated the throughput of the monitor when conducting multiple parallel instances of this protocol, and the results show that the monitor can deliver up to 632.91 keys per second, which is efficient and allows a single monitor machine to scale to serve a large number of nodes.

7.2.4 Monitor attestation

We measured the performance of the monitor attestation protocol. This protocols has a latency of 1.21 seconds and a throughput of approx. 4800 reqs/sec on a single node. The quote operation performed by the local TPM of the monitor is responsible for the bulk of the latency (0.82 seconds), and the remaining time is due to cryptographic operations and network latency. The value observed for the peak throughput is high, and is a consequence of using batch attestation. If we disable batching, the throughput drops sharply to 0.82 reqs/sec. Thus, this technique is crucial to the scalability of the monitor, delivering a throughput speedup of over 5000x.

7.2.5 Sealing and unsealing

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

To understand the effect of CPABE in the overall performance overhead, we set the input data to be of a small, constant size. Figure 8 shows the performance overhead of sealing and unsealing 1KB of data as a function of the complexity of the policy.

On the left, Figure 8 shows 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 cost

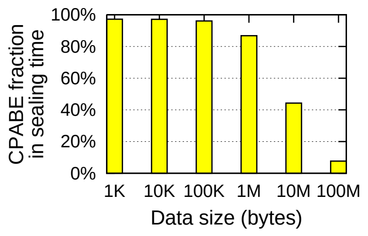


Figure 9: CPABE fraction in the performance overhead of sealing, varying the size of the input data.

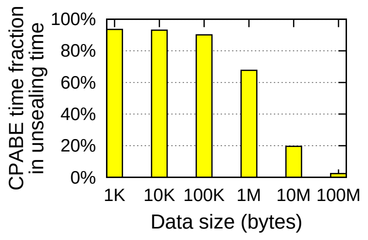


Figure 10: CPABE fraction in the performance overhead of unsealing, varying the size of the input data.

of sealing grows linearly with the number of attributes. The cost of sealing for a policy with 10 attributes is about 128 milliseconds.

On the right, Figure 8 shows the cost of an unseal operation. Unlike encryption, CPABE decryption depends 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 would use one attribute, whereas P_2 would use 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 is 51 milliseconds.

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

7.2.6 Summary

Our results show that the latencies of decryption key generation and the node attestation protocol are reason-

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.

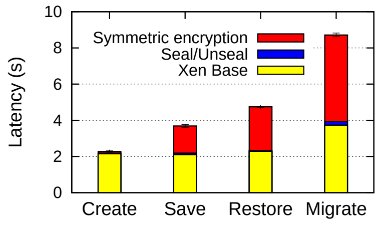


Figure 11: Latency of VM operations in Xen: Encrypting the VM state accounts for the largest fraction of the overhead, while the execution time of seal/unseal is relatively small. Encryption runs AES with 256 bit key size.

able, particularly since the monitor computes the former once per configuration, and executes the latter only once for each reboot of a node.

Our evaluation shows that the monitor scales well. A generic single-core machine running as a monitor node can check the configurations of up to 633 nodes per second. The monitor can withstand a peak throughput of 4.8K attestation requests from external customers per second on a single node, which is high. Since the protocol and the key generation are easily parallelizable, increasing the number of monitor nodes (or cores) can scale up Excalibur even further.

Finally, we show that the latency of the seal and unseal operations is reasonable. In particular, it is on the order of tens or about one hundred miliseconds for small data items. For large data the latency increases but it is dominated by the cost of symmetric encryption.

7.3 Cloud compute service

We evaluate the performance overhead that the changes to Xen incur on its VM management operations, namely *create*, *save*, *restore* and *migrate*. We measure the time to complete each operation using an example VM for 10 trials. The example VM runs a Debian Lenny distribution, with Linux-xen 2.6.26; it uses a disk image with 4GB, and its memory footprint is 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 doubles. The overhead, however, comes from encrypting the entire memory footprint of the VM; the usage of Excalibur to secure or recover the encryption key adds a small delay. Unlike the other operations, *create* experiences a small overhead – only a 4% increase. This is because the system only decrypts the kernel image which occupies 4.6MB, instead of the larger VM footprint, as in the remaining operations.

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

8 Related work

Over the past several years, there has been a lot of work on trusted computing [33]. Most of this work targets single computers with the goal of enforcing application runtime protection [15, 19, 23, 26, 27], virtualizing trusted computing hardware [9], and devising remote attestation solutions based on both software [17, 41] and hardware [12, 20, 36–38, 42]. Other work, focusing on distributed environments, provides integrity protection on shared testbeds [13], or distributed mandatory access control [25]. 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 [36], whose goal is to make the hash-based software fingerprints more meaningful to humans. Just like in Excalibur, it maps low-level fingerprints to high level attributes (properties), and relies on a monitor (controller) for this mapping. However, this work offers mostly an abstract model without an associated system. Moreover, Excalibur goes beyond this work by proposing new abstractions geared toward cloud services.

The work by Schiffman et al. [40] aims to improve the transparency of IaaS cloud services by providing customers with integrity proofs of their VMs and underlying VMMs. Similarly to 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 only provides 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 administrator of the CV is assumed to be trustworthy, representing a weaker threat model, which, in our view, does not address an important class of problems that occur in cloud services today. Finally, as opposed to Excalibur, this system does not address the shortcomings of the sealed storage TPM primitives, which could raise concerns of data management inflexibility and isolation crippling if they 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, we find hypervisors and OSes that can protect the confidentiality and integrity of data using isolation [22,26,45] or information flow control [44] techniques. At the middleware layer, we find frameworks, e.g., to build web services that can offer their users strict control over their data hosted at the provider site [21], enable controlled sharing of sensitive data using differential privacy [35], or provide general purpose encapsulation mechanisms for data [24]. These proposals are complementary to our work, since, even though these software platforms have a lot of potential for increasing the security and control of data on the cloud, they lack an adequate mechanism for bootstrapping trust. By combining these platforms with Excalibur, cloud providers have the opportunity to build new trustworthy cloud services.

In our previous workhop paper [39], we proposed a trusted cloud architecture to protect the confidentiality and integrity of customer VMs. Excalibur, on the one hand, embodies the high-level vision proposed in this work, but, on the other hand, differs in many design directions, like, for instance, in the fact that Excalibur uses CPABE to avoid involving the monitor in the operations that encrypt and decrypt customer data. Furthermore, unlike our previous workshop paper, we present here a complete architecture and set of associated protocols, and a real system implementation.

9 Conclusion

In this paper we present Excalibur, a system that implements policy-sealed data: a new abstraction that address the limitations of trusted computing on the cloud, and enables the design of trustworthy cloud services. Excalibur leverages TPMs, a novel architecture with a set of associated protocols, and CPABE to provide the developers of cloud services two new primitives, seal and unseal, which enable the construction of cloud services that offer stronger protection over how data is managed. We demonstrate the simplicity and flexibility of policy-sealed data by using Excalibur to build an elastic VM cloud computing service based on Xen and Eucalyptus that accesses customer's data only on platform configurations approved by the customer.

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

[8] 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.

[9] 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.

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

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

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

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

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

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

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

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

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

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

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

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

[22] 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.

[23] 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.

[24] 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.

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

[26] 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.

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

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

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

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

[31] 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.

[32] OpenSSL. `http://www.openssl.org`.

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

[34] 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.

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

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

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

[38] 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.

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

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

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

[42] 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.

[43] TrouSerS. `http://trousers.sourceforge.net`.

[44] 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.

[45] 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.