CloudProxy: The Tao of Trusted Cloud Computing

John L. Manferdelli,
Intel Science and Technology Center for Secure Computing,
University of California, Berkeley,

Tom Roeder, Google Inc.,

Fred B. Schneider, Cornell University

January 14, 2013

Abstract

The *CloudProxy framework*, or *Tao*, protects confidentiality and integrity of remotely-hosted programs and data. The framework supports policy enforcement, including access control for data controlled by each program. CloudProxy also provides protected key management and storage services without requiring changes to existing cloud data center operating procedures. Program data need never be transmitted or stored in unencrypted form, and no cryptographic keys are ever visible to insiders or co-tenants in a shared facility. FileProxy instantiates the CloudProxy framework and illustrates how the Tao is used to build a large scale, distributed application running on many computers. FileProxy consists of (possibly many instances of) two cooperating programs, *FileClient* and *FileServer*, running on different computers in potentially different cloud data centers. A supporting infrastructure component, called *KeyNegoServer*, is employed for key management.

1 Overview

Trusted computing research has focused mostly on system primitives and services [10, 24]. A foundational element of trusted computing, the integrity-protected boot of a software stack, appeared as early as 1989 [11]. Sealed storage to protect secrets for software and attestation for key management were added along with authenticated boot to provide remotely verifiable code identity and strong program isolation [29].

We combine these elements to obtain flexible trusted computing services that ensure the integrity and confidentiality of processing, data, and keys in a cloud data center operated by a third party. The protection is enforced in the presence of provisioning or operations errors by cloud data center *operators* as well as attacks by data center *insiders*[1].

An *activity* is a set of related processing possibly spanning many computers and data centers. Each activity[2] uses several component programs, which we call *activity elements*. The activity is designed, programmed, distributed and run on behalf of an *activity owner*.

[1] Insiders refers to operators, technicians and any other person who is at the data center or controls data center operations.

[2] An activity is thus a distributed application or service.

The activity owner requires that the following *security properties* hold:

- *Computational Integrity*: Activity elements are the ones provided by the activity owner. They cannot be modified prior to execution.
- *Isolation*: While activity elements execute, no computation, including intermediate results, can be observed by unauthorized parties, including data center *tenants*[3] and data center insiders
- *Policy enforcement:* Requests and directives received by activity elements during execution are performed by an activity element if and only if they comply with *activity owner policy*. This policy consists of authenticated assertions provided by the activity owner, or its delegates, before or after activity elements have been deployed.
- *Data Integrity*: Data identified by the activity owner that is used or produced by activity elements can only be written or modified by *bona fide* activity elements to the extent authorized by the activity owner policy.
- *Data Confidentiality*: No data identified as confidential by the activity owner can be disclosed or used by any person or program unless the disclosure and use complies with activity owner policy. This includes data stored on disks and data transmitted between activity elements.
- *Attestation*: An activity owner can use untrusted communications channels to verify statements from activity elements including policy statements and assertions demonstrating that the security properties are being enforced by the activity elements.

These security properties are enforced using the *CloudProxy Tao*[4]. The Tao provides primitives, including cryptographic primitives to measure and authenticate activity elements and generate, validate and distribute keys, which are revealed only to validated activity elements. These primitives are used to establish and maintain the security properties. The Tao does this without requiring any special provisioning at the data center; the Tao simply requires distribution, by possibly insecure means, of activity elements.

To illustrate the Tao, we built a prototype system called FileProxy. FileProxy consists of two activity elements: FileClient and FileServer. FileServer controls access to files stored on an assigned storage area in some accessible secondary memory (e.g., a disk or storage service). The files are encrypted and integrity protected using cryptographic keys protected by the Tao. FileServer also maintains a separate encrypted and integrity protected file with metadata containing authorization information for files stored by activity elements. Authorization information describes what programs and users can read, write, modify or create files. FileServer responds to connection requests received over an untrusted TCP connection from client activity elements (possibly including other FileServer activity elements). It authenticates client activity elements using the Tao and, symmetrically, the client activity elements authenticate the FileServer activity element using the Tao. Once authenticated, FileServer and the client activity elements negotiate an encrypted, integrity protected communications channel over which file services can be requested and satisfied. Clients can retrieve, upload or modify files under the control of activity owner policy. FileServer determines whether it will fulfill file service requests based on authorization evidence supplied by the client activity element. FileClient, as the name suggests, is a client of FileServer. Using files obtained

[3] A *tenant* is a program that is provided by a data center customer and runs in a data center.

[4] Tao means "the way" in Chinese.

from FileServer, FileClient performs application-specific processing and stores results locally or uploads them to FileServer.

FileProxy, having two components running on different machines, which must establish a trusted, authenticated relationship and cooperate to achieve an overall system goal, illustrates the functionality required to build a large heterogeneous distributed system at scale using the Tao. This functionality is general and can be used to share, store and process encrypted data flexibly under policy control among many activity elements.

A useful paradigm in computer science is recursive instantiation. The Tao was designed to support recursively layered stacks: A host system supporting the Tao bears enough capability to allow its hosted system to provide the Tao for systems the hosted system itself hosts. In the FileProxy prototype, there are three recursive layers: *Trusted Hardware* provides the Tao for an operating system called the *Trusted OS*. The Trusted OS provides the Tao for the two possible activity elements, FileClient and FileServer.

This paper is organized as follows: Section 2 describes the Tao. For the Tao to apply in our target cloud setting, there are prerequisites related to physical security and requirements related to the Trusted Hardware; Section 3 describes these prerequisites and requirements and discusses the protection model afforded by the Tao in this setting. In Section 4, we describe the FileProxy design and implementation. Section 5 describes the performance of FileProxy. Section 6 discusses related work, and finally, Section 7 discusses further research to extend and generalize FileProxy.

2 CloudProxy Tao

We describe the CloudProxy Tao for a host system and its hosted systems. First, we define some standard notation.

2.1 Standard Notation

We designate a hosted program by hostedProgram. If we wish to emphasize the hosted program's identity, we enclose its common name in parenthesis, for example, hostedProgram(FileClient).

We designate a generic Host for the Tao as host. Again, if we wish to specify a particular host, we add its name in parenthesis: host(TrustedOS).

Security Principals are entities that can be named as subjects in a security assertion[5] Users (identified by name and represented by public/private key pairs) as well as programs (identified by measurements and represented by public/private key pairs) are examples of Security Principals. In an authorization assertion, subjects are the entities to whom rights or privileges are conferred and objects are the resources affected by the designated right or privilege. For example, in the security assertion described informally by "John may read /Users/John/MyDiary", John is a subject, the file resource named by /Users/John/MyDiary is the object and the right is "may read."

Symmetric keys are denoted with a upper case K. Often, the key bears a subscript designating the security principal that controls the key and a superscript identifying its purpose. Thus, $K^{\mathsf{sealKey}}_{\mathsf{host(TrustedOS)}}$ represents a symmetric key controlled by the host(TrustedOS) used to seal objects.

[5] A security assertion is an unambiguous, enforceable statement, in a formal language, describing the state of a system related to security. Authentication assertions, authorization assertions and directives naming actions that can or should be carried out are all examples of security assertions.

Encryption and decryption operations employing a symmetric key, K, are denoted by $E(K, \cdot)$ and $D(K, \cdot)$, respectively.

A public/private key pair is designated PK, pK where PK is the public key and pK is the private key. As with symmetric keys, we employ superscripts and subscripts to identify the security principal represented by the key pair and its purpose. Thus the private portion of the activity policy key for the FileProxy activity for the activity owner John is represented as $pK^{\mathsf{activityPolicyKey(FileProxy)}}_{\mathsf{activityOwner(John)}}$. Encryption and decryption operations employing a public/private key pair PK/pK are denoted by $E(PK, \cdot)$ and $D(pK, \cdot)$ respectively.

A message, M, signed by the private key pK is denoted by $\langle M \rangle_{pK}$. A signed claim or *certificate* recites the designated assertion (commonly a public key, its validity period and properties in the case of a certificate) in a canonical form. This is cryptographically hashed and padded and the resulting object is encrypted using pK to form a *signature*. To verify a signature, a verifier also cryptographically hashes the original assertion in canonical form and then decrypts the signature using PK; if the padding in the decrypted signature is correct and the hash in the decrypted signature matches the one the verifier computes, the signature is valid. A certificate naming the public key $PK^{\mathsf{attestKey}}_{\mathsf{host}}$ and an identifier called $\mathsf{hostIdentifier}$ is written as

$$\langle \mathsf{Certificate}(PK^{\mathsf{attestKey}}_{\mathsf{host}}, \mathsf{hostIdentifier}) \rangle_{pK^{\mathsf{activityPolicyKey}}_{\mathsf{activityOwner}}}.$$

2.2 Tao Overview

The Tao consists of three interrelated framework elements.

Tao Environment. A set of primitives implemented by a *host system* and used by and for the benefit of a *hosted system*. The Tao Environment ensures that the memory (including processor registers, state, cache, etc.) assigned to a correctly loaded hosted system cannot be read or modified by any other hosted program (to enforce Isolation). It provides a compact, unforgeable representation of a hosted system (this is called the hosted system's *measurement*, which is used to authenticate a hosted system in enforcing Computational Integrity, Policy Enforcement, Data Integrity, Data Confidentiality and Attestation). It safeguards secrets, generated or stored by a hosted system, which will only be revealed a hosted system with the correct measurement (called *sealed storage*) and it cryptographically signs messages on behalf of the hosted system, citing the hosted system's measurement, demonstrating the provenance and integrity of hosted system assertions (called *attestation*); these last two primitives work with the others to enforce all the security properties.

Tao Policy Enforcement. Each activity element contains the public portion of a public/private key pair provided by an activity owner called the *activity policy key*. An activity element uses the activity policy key to validate the origin and authenticity of security assertions expressed in a formal language (using XML syntax) and called *claims*, made by the activity owner or its delegates. Activity elements rely on such claims and will carry out activities specified by them. The activity policy is the union of all such authentic claims.

Tao Key Management. An activity element can use primitives provided by the Tao Environment to generate, protect, and certify keys. Protecting sensitive keys is provided by sealed storage. Certifying a key refers to signing a claim, naming the public portion of a key, supported by claims chaining up to the activity policy key. This signed claim can be verified by activity elements.

2.3 Tao Environment

Each hosted program in a CloudProxy has an associated *measurement*, denoted $\mu(\mathsf{hostedProgram})$. For example, the measurement for hosted instance of FileServer is denoted by $\mu(\mathsf{hostedProgram(FileServer)})$ This measurement is typically a cryptographic hash[6] of the executable code, initialized data and any configuration information affecting program execution[7]. The measurement serves is an unforgeable representation of the hosted program. Because of differences due to hardware, measurements used in FileProxy prototype vary in minor ways depending on stack layer.

When the host Tao Environment[8] loads a hosted program, hostedProgram, the Tao Environment first computes $\mu(\mathsf{hostedProgram})$. The host Tao Environment securely retains the association between the hostedProgram and $\mu(\mathsf{hostedProgram})$ while the hostedProgram is executing. Although a host may execute many hosted programs, from the moment a hostedProgram is loaded throughout execution, the host ensures that no program or person (except the host itself) can read or modify hostedProgram or its data. In this way, the host implements computational isolation.

The host Tao Environment employs two keys that persist across activations of the host. The host *sealing key* is a symmetric key denoted $K_{\mathsf{host}}^{\mathsf{sealKey}}$. The *host attestation key*, is a public/private key pair $PK_{\mathsf{host}}^{\mathsf{attestKey}}/pK_{\mathsf{host}}^{\mathsf{attestKey}}$. $K_{\mathsf{host}}^{\mathsf{sealKey}}$ and $pK_{\mathsf{host}}^{\mathsf{attestKey}}$ are known only to the host.

The Tao Environment has a certificate, signed by the activity owner, which names the host system and the attest public key, $PK_{\mathsf{host}}^{\mathsf{attestKey}}$, together with other information like validity period and revocation information. This certificate, called the *host attest certificate* is denoted,

$$\langle \mathsf{Certificate}(PK_{\mathsf{host}}^{\mathsf{attestKey}}, \mathsf{hostIdentifier}) \rangle_{pK_{\mathsf{activityOwner}}^{\mathsf{activityPolicyKey}}}.$$

If the hosted system is a program, hostIdentifier is $\mu(\mathsf{hostProgram})$.

The host Tao Environment provides the following services, called the *Tao Primitives*, for its hosted systems.

Seal(dataToSeal) seals a caller supplied data item, dataToSeal. To seal dataToSeal, for hostedProgram, the host encrypts a nonce, $\mu(\mathsf{hostedProgram})$ and dataToSeal using its sealing key and returns the resulting encrypted blob, sealedData. Thus, we have

$$\mathsf{sealedData} := E(K_{\mathsf{host}}^{\mathsf{sealKey}}, \mathsf{nonce}||\mu(\mathsf{hostedProgram})||\mathsf{dataToSeal}).$$

[6] A cryptographic hash, H, is a function that maps variable length strings into a fixed length bit string. Popular cryptographic hashes are SHA-1, which has a 160-bit output length and SHA256, which has a 256-bit output length. Cryptographic hashes have several important properties that we rely on. The first is called *pre-image resistance* and it means that given y, where $y = H(x)$, it is computationally infeasible to find x; by contrast, given x, computing $y = H(x)$ is efficient. A related property is *collision resistance*, which requires that given x, y such that $y = H(x)$, it is computationally infeasible to find $x', \neq x$ such that $H(x) = H(x')$. A slightly stronger property of cryptographic hashes is: It is computationally infeasible to find $x \neq x'$ such that $H(x) = H(x')$. For us, a critical application of these properties is that if we compute a cryptographic hash of an item (like a program) and change even a single a bit, the hash will change. Further, for a given hashed program it is impossible to find another program with the same hash.

[7] Configuration information may include, for example, command line flags and arguments in the shell level invocation of an application program.

[8] The host contains an implementation of the Tao Environment but usually contains other functions used by the hosted system such as I/O and scheduling.

Unseal(sealedData) unseals the caller supplied sealed data item sealedData. To unseal sealedData, the host decrypts sealedData using the sealing key. It compares the decrypted μ(hostedProgram) with the measurement of the hosted system requesting the unseal and if the measurements match, it returns the decrypted dataToSeal, otherwise it returns an error.

Attest(dataToAttest) attests the caller supplied data item denoted dataToAttest. To attest to dataToAttest, the host system computes a composite cryptographic hash of dataToAttest and μ(hostedProgram) and signs it using $pK_{\mathsf{host}}^{\mathsf{attestKey}}$. An attestation is a claim reciting dataToAttest and μ(hostedProgram) along with the computed signature. Thus an attestation has a format similar to a certificate; it is denoted $\langle \mathsf{Attest}(\mathsf{dataToAttest}, \mu(\mathsf{hostedProgram})) \rangle_{pK_{\mathsf{host}}^{\mathsf{attestKey}}}$

GetHostedMeasurement returns μ(hostedProgram), the hosted system measurement of the calling hostedProgram.

GetAttestCertificate returns the host attest certificate of the calling program. A party receiving an attestation uses this to determine whether it trusts this host's attestation key.

GetEntropy(*numBits*) returns a cryptographically secure random number of size *numBits* bits.

StartHostedProgram(newHostedProgram) causes the host system to start a new hosted program, newHostedProgram. Before newHostedProgram, is started, it is measured, resulting in μ(newHostedProgram)). In accordance with the CloudProxy Tao, the host Tao Environment securely retains the association between newHostedProgram and μ(newHostedProgram) during execution and ensures that from the moment newHostedProgram is loaded throughout execution, no program or person (except the host itself) can read or modify newHostedProgram or its data. StartHostedProgram returns an opaque handle[9] that identifies the newHostedProgram's execution context in the host.

Nonces are included in the results produced by some of the primitives to prevent replay attacks.

2.4 Tao Policy Enforcement

The CloudProxy Tao provides a uniform mechanism that allows hosted systems, acting as activity elements, to enforce policies. Before distributing activity elements, the activity owner generates a public/private key pair called the activityPolicyKey. For simplicity in notation, and due to the central role of the activity owner, we employ a shorthand notation and write pK_0 for $pK_{\mathsf{activityOwner}}^{\mathsf{activityPolicyKey}}$ and write PK_0 for $PK_{\mathsf{activityOwner}}^{\mathsf{activityPolicyKey}}$. pK_0 is kept secret by the activity owner. It serves as a root key that is used to sign claims (or delegate the right to sign such claims) to be followed by activity elements. It also signs root certificates trusted by activity elements.

The activity owner also embeds PK_0 (say, as initialized data in the program executable) in all activity elements. Evidence supporting a claim presented to the hosted program must originate with a claim signed by pK_0. Since PK_0 is part of the hosted system, it cannot be changed without changing μ(hostedProgram). Sealing, unsealing and attesting by the host system on behalf of a hosted system is tied to this measurement. This unbreakable connection enables us to tie PK_0 and hence policy enforcement to the hosted system measurement.

In schematic form, if a hosted system receives a chain of claims of the form $\langle PK_{\mathsf{B}}^{\mathsf{A}}$ says X $\rangle_{pK_{\mathsf{B}}^{\mathsf{A}}}$, $\langle PK_0$ says $PK_{\mathsf{B}}^{\mathsf{A}}$ maysay X $\rangle_{pK_0}$, it applies the public portion of the activity policy key to the

[9] Like a PID.

second claim and verifies that the second claim was signed by pK_0. It then uses $PK_{\mathsf{B}}^{\mathsf{A}}$ to verify that the first claim was signed by $pK_{\mathsf{B}}^{\mathsf{A}}$. If these two signatures correctly verify, the hosted system knows that X complies with activity owner policy and that the directive expressed by X was *authorized*. The hosted system then performs X. The body of the claim, X, consisting of predicates, is expressed in a formal language.

2.5 Tao Key Management and the Tao Initialization

CloudProxy uses cryptography to provide confidentiality, integrity and authentication. As a result, generating, deploying, certifying and replacing cryptographic keys, collectively called *key management*, is foundational.

All key management is rooted in the ability of a hosted system to prove to the activity owner, using cryptographic statements provided by a host Tao Environment, that a public key generated by the hosted system could only have come from the hosted system while it was isolated and that the hosted system has a mechanism to ensure it and only it has access to the corresponding private key.

Public keys provisioned in this way are used in two important ways. First, they enable a hosted system to provide a Tao Environment to its hosted systems since a key pair so provisioned can serve as an attestation key for the original hosted system. Second, when a hosted system serves as an activity element, such a key can be used to authenticate the hosted program[10]

The provisioning procedure involves an infrastructure component called *KeyNegoServer* operated by the activity owner. When a hosted system first runs on a host Tao Environment, it conducts a protocol, called *Tao Initialization*, with KeyNegoServer over an untrusted communication link.

During the Tao Initialization, the hosted program, hostedProgram, after it is isolated and measured, generates a new public/private key pair using the GetEntropy primitive. We call this key the provisionedKey. Using the seal operation provided by the Tao Environment, hostedProgram seals $pK_{\mathsf{hostedProgram}}^{\mathsf{provisionedKey}}$ and saves the resulting encrypted blob for use across a restart in the future. Then hostedProgram, using the attest primitive provided by the Tao Environment, requests an attestation naming $PK_{\mathsf{hostedProgram}}^{\mathsf{provisionedKey}}$, and obtains the attestation

$$\langle \mathsf{Attest}(PK_{\mathsf{hostedProgram}}^{\mathsf{provisionedKey}}, \mu(\mathsf{hostedProgram})) \rangle_{pK_{\mathsf{host}}^{\mathsf{attestKey}}}.$$

This attestation is transmitted to KeyNegoServer along with the host attest certificate, which was obtained from the host Tao Environment using GetAttestCertificate. KeyNegoServer verifies the attestation signature and compares $\mu(\mathsf{hostedProgram})$ in the attestation with measurements belonging to trusted activity elements. If the measurement matches one of these, and the supplied host attest certificate is valid, KeyNegoServer uses pK_0 to sign a certificate, naming $PK_{\mathsf{hostedProgram}}^{\mathsf{provisionedKey}}$ and $\mu(\mathsf{hostedProgram})$. The resulting certificate is

$$\langle \mathsf{Certificate}(PK_{\mathsf{hostedProgram}}^{\mathsf{provisionedKey}}, \mu(\mathsf{hostedProgram})) \rangle_{pK_0}.$$

This certificate is returned to the requesting hosted system.

Symbolically, this message exchange is:

[10] Authenticating a program means verifying its measurement or identity. This may be done indirectly, as in this case, where *only* the hosted program with a specified measurement can obtain a private key. Proof of possession of this key verifies the program identity.

1. $\mathsf{hostedProgram} \rightarrow \mathsf{KeyNegoServer}$: $\langle \mathsf{Attest}(PK^{\mathsf{provisionedKey}}_{\mathsf{hostedProgram}}, \mu(\mathsf{hostedProgram}))\rangle_{pK^{\mathsf{attestKey}}_{\mathsf{host}}}$, $\langle \mathsf{Certificate}(PK^{\mathsf{attestKey}}_{\mathsf{host}}, \mathsf{hostIdentifier})\rangle_{pK_0}$.

2. $\mathsf{KeyNegoServer} \rightarrow \mathsf{hostedProgram}$: $\langle \mathsf{Certificate}(PK^{\mathsf{provisionedKey}}_{\mathsf{hostedProgram}}, \mu(\mathsf{hostedProgram}))\rangle_{pK_0}$.

Note, at the conclusion of Tao Initialization, the hostedProgram has a signed certificate naming hostedProgram's public key and its measurement. The corresponding private key is only known to hostedProgram running in the Tao Environment provided by its host.

When the provisionedKey is used as an attestation key by a hosted system that is acting as a Tao Environment, the public key is denoted as $PK^{\mathsf{attestKey}}_{\mathsf{hostedProgram}}$, the certificate returned by KeyNegoServer is the hostedProgram's host attest certificate:

$$\langle \mathsf{Certificate}(PK^{\mathsf{attestKey}}_{\mathsf{hostedProgram}}, \mu(\mathsf{hostedProgram}))\rangle_{pK_0}.$$

When the provisionedKey is used by hostedProgram to authenticate itself, the key is referred to as the *programKey* and the certificate becomes the hostedProgram's *program certificate*:

$$\langle \mathsf{Certificate}(PK^{\mathsf{programKey}}_{\mathsf{hostedProgram}}, \mu(\mathsf{hostedProgram}))\rangle_{pK_0}.$$

The security properties of a hosted program depend only on the hosted program and its host. Maintaining the security properties of a hosted program does not depend on other hosted programs of the same host. Indeed, a hosted program is expressly protected from sibling hosted programs by the host.

A properly written hosted program, using keys that are only available to it, can encrypt and integrity protect all input and output as well as authenticate itself. It can also verify any claims it receives over a possibly insecure network using Tao Policy Enforcement. This enables the Tao to provide key management mechanism in which no secret key needed by the hostedProgram is ever handled by or made available to other programs or people in a data center.

The Tao is complete.

3 Hardware Requirements and Protection Model

As we pointed out in the overview, CloudProxy security is rooted in specialized hardware capabilities of Trusted Hardware.

Trusted Hardware requirements mandate that the CPUs and related motherboard components provide measured boot, memory isolation[11], and the ability to restrict what memory locations bus masters with Direct Memory Access (or DMA) can access to ensure the memory isolation cannot be subverted by bus masters. Measured boot means the booted software, including any security critical boot parameters, which is called the *measured launch environment* or *MLE*, is measured in the sense of the Tao primitives by the Trusted Hardware. Trusted Hardware also must provide sealing, unsealing and attestation for the MLE citing its measurement.

Existing server boards with Intel CPUs supporting SMX, TXT and VT-d and TPM 1.2 fulfill the Trusted Hardware requirements and are already widely deployed; consult [14] for these

[11] For example, Extended Page Tables.

capabilities. AMD CPUs supporting SVD with IOMMUs and TPM 1.2 also meet this need. The TPM, the only motherboard part which is not on every server board, costs under a dollar.

In addition to the foregoing hardware component requirements, we impose access restrictions on computers in a cloud data center. We require computers be in locked enclosures preventing physical access during the period they execute Tao application activity elements and for a few minutes thereafter. Ideally, compliance with these restrictions can be audited. The requirement for a period of repose after system shutdown is to avoid cold boot attacks [13]. Many Infrastructure as a Service (IaaS) providers already have facilities like this for some of their customers and employ good physical auditing (cameras in the data center, two-person access rules, etc.).

Trusted Hardware requirements only affect computers running CloudProxy systems. They do not impose any requirements on peripherals, such as disks, or network infrastructure or computers that do not run CloudProxy systems.

The protection model specifies what security promise CloudProxy offers in the presence of what attacks by what adversaries. Provided the hardware requirements (including access restrictions on computers running CloudProxy systems) are in place, CloudProxy ensures that the security properties set forth in the overview hold.

We assume activity elements, as supplied, have no exploitable flaws.

The attack model for CloudProxy specifies the access and capability of an adversary seeking to compromise our protection model.

Our adversaries include data center insiders, network attackers and other tenants. However, we assume the entity operating the data center is not an adversary and is motivated, whether by contract, the desire to protect its brand or third party compliance procedures, to ensure that physical access restrictions are maintained. Fortunately, these restrictions impose modest costs (sometimes no incremental cost) and are easily audited.

Except for Trusted Hardware executing CloudProxy systems, our adversaries have physical access to all data center hardware and infrastructure. Disks can be removed and examined offline at any time. Data center infrastructure software may be flawed or contain maliciously inserted components that intentionally target the activity. Tenants in the data center may be adversarial. In our attack model, network traffic can be examined and changed by an adversary.

We do not rule out, nor defend against, denial of service attacks (insiders can turn off power or break network connections or simply refuse to schedule activity elements from running). Denial of service attacks, however, could be mitigated by replication of activity elements in several data centers.

4 FileProxy

FileProxy has two activity elements, FileServer and FileClient. In deployments operating at scale, there can be many copies of FileServer and FileClient running simultaneously. For simplicity, we describe a full use case using a single instance of each.

4.1 FileProxy Activity Structure

A deployment involves three physically-separate computers, connected to a shared network.

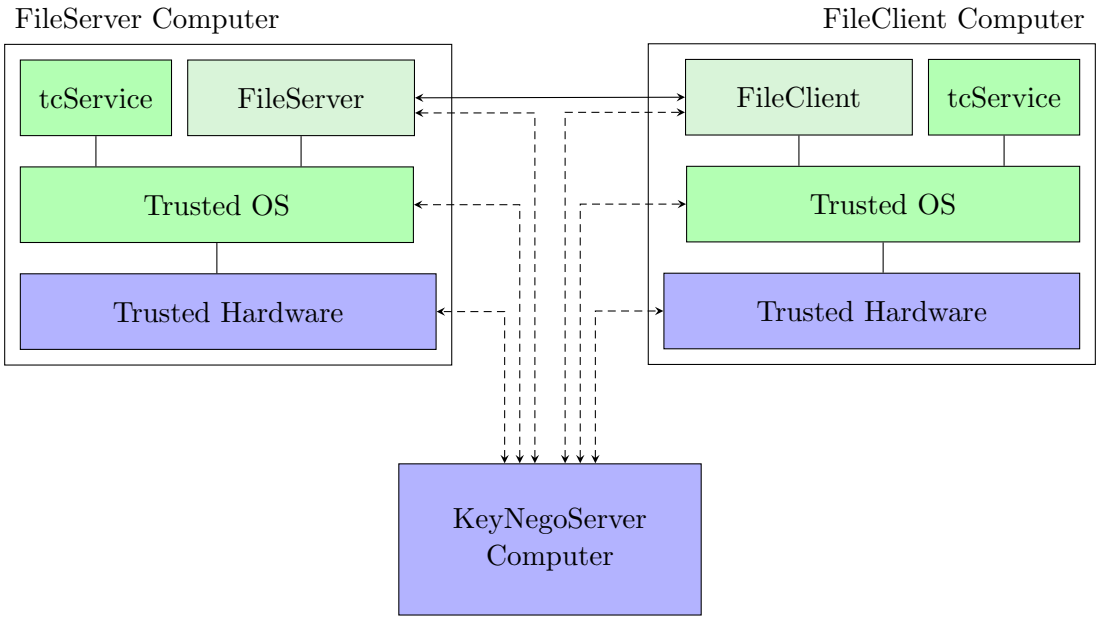

Figure 1: Layered FileProxy System Components

The *FileServer Computer* runs the FileServer activity element on the Trusted OS on Trusted Hardware.

The *FileClient Computer* runs FileClient activity element on the Trusted OS on Trusted Hardware.

The *KeyNegoServer Computer* runs a conventional operating system, which is configured to run a single web service, KeyNegoServer.

4.2 Layered Tao Environments for FileProxy

A critical design element in FileClient and FileServer is their reliance on the Tao to provide the security properties. The FileProxy design makes use of the ability of the CloudProxy Tao to support a recursively layered composed collection of systems. FileProxy has two stacks, the FileClient stack and the FileServer stack, each consisting of three Tao layers:

- The Trusted Hardware, as host, provides the Tao for a Trusted OS as a hosted program.
- The Trusted OS, as host, provides the Tao for the FileClient activity element and the FileServer activity element.
- FileClient and FileServer employ the Tao for key management, to protect keys, for policy enforcement and to authenticate themselves in communications protocols.

The Trusted OS is a modified Linux operating system with several kernel changes to Linux and a custom initialization procedure. FileServer and FileClient run as Linux processes. Figure 1 depicts the hosting relationships and communications channels used in the FileProxy activity. It also depicts the KeyNegoServer component that is used in Tao Initialization. Solid lines in the figure represent communication paths that are active during the operation of the service; dashed

communication paths are only active during key establishment during setup. And paths between computers represent TCP/IP connections.

The mechanisms implementing the Tao vary a little from layer to layer. When the host is a software program, the Tao primitives are implemented in all layers using the same API and essentially the same implantation. For example, except for primitives implementing isolation, the Tao primitives are implemented using the same core library in FileClient, FileServer and the Trusted OS. In the case of the Trusted OS, this library executes in a trusted user mode service program called *tcService*. This user mode process communicates with hosted applications through a special device driver.

Table 1 lays out the major Tao implementation features that vary from layer to layer. For the Trusted Hardware, Tao Initialization actually happens offline as explained later.

Layer provided by	**Measurement provided by**	**Isolation provided by**	**Host primitives initialization (seal, unseal, attest)**	**Tao Initialization**
Trusted Hardware	TPM internal	Single Hosted Program	TPM using TPM device driver TPM library	Out of band signature
Trusted OS	TPM cryptographic hash of boot image	Single Hosted Program	tcService using standard library	KeyNegoServer protocol
FileClient FileServer	Library cryptographic hash of executable	N/A	standard library	KeyNegoServer protocol

Table 1: FileProxy Tao layers

4.3 Initializing the Policy Key and KeyNegoserver Database

At the heart of the *CloudProxy Tao* is the role of the PK_0 in policy enforcement and key management.

The activity owner must embed PK_0 in every data center distributed activity application element. While, in principle, each stack layer may employ a different policy key, in FileProxy, the same key is used in every layer. At the Trusted OS layer of fileProxy, PK_0 is placed as initialized data in a device driver, which is statically linked in the Trusted OS image. For both FileServer and FileClient, PK_0 is placed as initialized data in the program image of each program.

After embedding the policy public key, the activity owner measures the resulting Trusted OS, FileServer and FileClient in the same way as they would be measured in the data center. For FileServer and FileClient, this simply involves calculating a SHA-256 based hash of the executable files. In the case of the Trusted OS, this involves calculating the two 160 bit SHA1 based hashes that the Trusted Hardware will compute and save in PCR 17 and PCR 18 of the TPM as well as the composite hash representing $\mu(\mathsf{TrustedOS})$, which is computed from these values. All these measurements, as well as the programs they are associated are recorded for further use.

The activity owner can now deploy the Trusted OS, FileServer and FileClient to whatever data center it chooses or simply place them on an insecure server for subsequent download.

The private portion of the policy key is used to sign certificates. For example, at each CloudProxy computer, the AIK (which is the Attest Key used by the Trusted Hardware as a host for the Trusted OS) must be signed so it can be trusted by activity elements. To accomplish this, the TPM generates evidence, ultimately signed by a hardware manufacturer's private key[12], demonstrating the trustworthiness of the AIK. This information is provided to an off-line program, in a secure facility operated by the activity owner, which has a database of Trusted Hardware public keys as well as a list of any hardware whose trusted status was revoked. The off-line program, using this database evaluates the TPM-produced evidence and if satisfied, signs a certificate with pK_0.

Certificates for the Trusted OS Attest key and the FileClient and FileServer program keys are also signed by pK_0 in the on-line Tao Initialization procedure using KeyNegoServer. Since KeyNegoServer needs to use the private portion of the policy key, KeyNegoServer must be securely operated throughout its lifetime protecting this critical private key. There are many possible measures that might be taken to provide this protection: The key may be generated in a secure Hardware Security Module ("HSM"), so the private key is never in the clear. The KeyNegoServer installation should be physically secure and the activity owner may wish to employ multi-person access control on critical operations involving the KeyNegoServer. Finally, the activity owner may wish to build KeyNegoServer with the same CloudProxy Tao employed to protect activity application elements. However this is done, KeyNegoServer and pK_0 must be absolutely secure.

In evaluating evidence provided by the Trusted OS, FileClient and FileServer, KeyNegoServer uses the program measurements made by the activity owner prior to deploying those images and stored in the activity owner's KeyNegoServer database. Again this may be supplemented with revocation information about no longer trusted components.

The KeyNegoServer protocol, which was described in Tao Initialization, is implemented using two XML encoded messages; these XML message formats appear in the appendix.

4.4 FileProxy Operation

After Tao Initialization, the operational phase of FileProxy begins.

When the FileServer activity element starts after Tao Initialization, it retrieves the files containing its sealed symmetric and private keys and asks the Trusted OS to unseal them.

Next, the FileServer activity element opens a TCP request channel at a well-known port awaiting connection attempts from instances of FileClient. When one arrives, FileServer and FileClient negotiate a channel. This process is called *channel establishment*. Channel establishment results in mutual authentication of each participant (i.e.- FileClient and FileServer, as well as additional security principals, like users, FileClient operates on behalf of) and the establishment of shared encryption and integrity keys for the channel using their program public keys. This protocol, which is similar to that employed by TLS with client authentication, is described below.

Since the private keys corresponding to the public keys in the FileClient and FileProxy program certificates can only be known by Tao isolated and measured copies of FileClient and FileServer after unsealing, the authentication achieved by channel establishment, provides communicating

[12]Namely, the hardware manufacturer who built the Trusted Hardware.

FileServer and FileClient activity elements with cryptographic proof of the isolation regime and code identity (measurement) of its channel partner. This code identity represents the primary security principal on whose behalf the channel was established. As part of channel establishment, additional security principals (such as users, on whose behalf FileClient operates) are also authenticated by one side providing proof of possession of a private key corresponding to a public key, which represents this principal: the channel "speaks for" these principals.

Once the channel is established, FileClient may request file services from FileServer including upload files, retrieve files, create files (or directories) and delete files (or directories).

FileServer separately maintains metadata associated with each file it manages. The metadata consists of a URI (a universal resource identifier, such as "//www.activity.com/files/MyDiary.txt") to refer to the file, a cryptographic hash of the file and the public keys of the file owners. Owners have full rights (read, write, modify, delete) to a file and may grant those rights to another party or delegate the right to grant those rights to another party.

The cryptographic hash of a file can be used to search for the file based on exact content match. FileServer has an assigned storage area in some accessible secondary memory (i.e.- a disk or storage service) with a hierarchical name space like a Unix file system. FileServer may actually replicate its data in many such storage areas but we omit this in our description for simplicity. This secondary memory is neither confidential nor uncorruptible so FileServer uses the keys it generated and protects, using the Tao Primitives, to encrypt and integrity protect its files.

FileProxy metadata is stored in a single encrypted, integrity protected file. The secondary memory employed in FileProxy is simply a Linux file system on a mounted (but untrusted) disk.

When requesting file access, FileClient transmits evidence, supporting the request to FileServer over the secure channel they established. Evidence is represented by signed claims. We discuss claims in detail later. To make an access decision, FileServer simply verifies a chain of evidence, presented by FileClient, starting at a claim signed by a file owner and terminating in a grant naming one or more of the principals the channel speaks for. If access is allowed, FileServer decrypts the file (with the keys it maintains for file decryption and integrity control) and sends the file over the (encrypted, integrity controlled) channel to FileClient.

FileClient is equipped with the same Tao Primitives as FileServer; it processes file data, potentially storing protected intermediate results locally, or sending results to FileServer for storage and protection.

4.5 Channel Establishment between FileClient and FileServer

When FileClient contacts FileServer to initiate a session, they must negotiate a secure channel. This protocol consists of XML encoded messages. There are four client messages and three server messages. The first three client messages and the first two server messages are essentially the same as the TLS handshake specialized to a case that requires client authentication and only supports RSA signatures (i.e., it does not also support DSA signatures).

1. FileClient's first message to FileServer contains a client generated nonce and supported cipher suites (typically only one). Symbolically,
   FileClient → FileServer: ClientNonce || Cipher-Suites.

2. FileServer replies with a server generated nonce, the selected cipher suite and the FileServer program key certificate. Symbolically,
   FileServer $\rightarrow$ FileClient: $\mathsf{ServerNonce} \,||\, \mathsf{Cipher\text{-}Suite} \,||$
   $\langle \mathsf{Certificate}(PK^{\mathsf{programKey}}_{\mathsf{FileServer}}, \mu(\mathsf{FileServer}))\rangle_{pK_0}$.

3. FileClient's second message to FileServer contains a randomly generated premaster secret encrypted with the verified $PK^{\mathsf{programKey}}_{\mathsf{FileServer}}$, the server nonce decrypted with $pK^{\mathsf{programKey}}_{\mathsf{FileClient}}$, which serves to authenticate FileClient to FileServer, and the FileClient program key certificate. After FileClient's second message, all communications are encrypted and integrity-protected with keys derived, using a key derivation function (KDF), from the pre-master secret and the client and server nonces just as in TLS. By agreeing to a key, FileServer is implicitly authenticated to FileClient. Now both are confident of the code identity of their channel partner. Symbolically,
   FileClient $\rightarrow$ FileServer: $E(PK^{\mathsf{programKey}}_{\mathsf{FileServer}}, \mathsf{preMasterSecret}) \,||$
   $D(pK^{\mathsf{programKey}}_{\mathsf{FileClient}}, \mathsf{ServerNonce}) \,||\, \langle \mathsf{Certificate}(PK^{\mathsf{programKey}}_{\mathsf{FileClient}}, \mu(\mathsf{FileClient}))\rangle_{pK_0}$.

4. FileClient sends its third message immediately after its second. This message consists of the hash of the first three messages, denoted by $\mathsf{Hash}_{1:3}$, signed by FileClient's program key. This serves to bind the previous messages, preventing splicing attacks as well as authenticating FileClient to FileServer. Symbolically,
   FileClient $\rightarrow$ FileServer: $\langle \mathsf{Hash}_{1:3}\rangle_{PK^{\mathsf{programKey}}_{\mathsf{FileClient}}}$.

5. FileServer's second message contains a new random challenge (C) used to authenticate any additional security principals and a hash of all previous messages, denoted $\mathsf{Hash}_{1:4}$. Symbolically,
   FileServer $\rightarrow$ FileClient: C, $\mathsf{Hash}_{1:4}$.

6. FileClient's final message contains the certificates of the additional principals to be authenticated along with the first challenge encrypted with the first additional security principal's private key, the challenge plus 1 ($C + 1$) encrypted with the second security principal's private key and so on. Symbolically,
   FileClient $\rightarrow$ FileServer: $\mathsf{principalCertificate}_1 \,||\, \ldots \,||\, \mathsf{principalCertificate}_n \,||$
   $D(pK^{\mathsf{authenticationKey}}_{\mathsf{principal}_1}, C) \,||\, \ldots \,||\, D(pK^{\mathsf{authenticationKey}}_{\mathsf{principal}_n}, C + n - 1)$.

7. If the encrypted challenges are verified, FileServer's final message concludes the protocol indicating successful completion. Symbolically,
   FileServer $\rightarrow$ FileClient: $\mathsf{Success}$.

4.6 FileProxy Request/Response Protocol

After channel establishment, FileClient transmits XML encoded requests to FileServer. File service requests contain the file name requested, length (if this is a file write), access requested and signed claims, chaining up to a file owner, supporting the access request.

FileServer responds with an accept or reject indication, possibly an error code, the file name and file length. In the case of a read request, the file follows if the request was accepted. In the case of a write request, FileClient transmits the file it wishes to upload after receiving the success indication. The request types are: getResource, sendResource, createResource (accompanied by owner certificate), deleteResource, addOwner and deleteOwner.

4.7 Authentication and authorization

The most commonly used security assertions are related to authentication and authorization. In FileProxy, all claims, including authentication and authorization assertions, are DSig[13] signed and formatted. These assertions may incorporate many named properties, for example, authentication assertions may name security principal types (policy principal, machine principal, user principal, program principal). Although security principals have names for convenience, validation of a security principal depends only on its public key.

To fix attention, here is a set of claims, in cartoon form, that might be used by a requesting principal, Fred, represented by $PK_{\mathsf{Fred}}^{\mathsf{authenticationKey}}$[14], to read the file with universal name //www.activity.com/files/fileX.txt, which is owned by John, represented by $PK_{\mathsf{John}}^{\mathsf{authenticationKey}}$:

1. $\langle PK_{\mathsf{John}}^{\mathsf{authenticationKey}}$ speaksfor John$\rangle_{pK_0}$.
2. $\langle PK_{\mathsf{Fred}}^{\mathsf{authenticationKey}}$ speaksfor Fred$\rangle_{pK_0}$.
3. $\langle PK_{\mathsf{John}}^{\mathsf{authenticationKey}}$ says $PK_{\mathsf{Fred}}^{\mathsf{authenticationKey}}$ mayread //www.activity.com/files/fileX.txt$\rangle_{pK_{\mathsf{John}}^{\mathsf{authenticationKey}}}$.

In our cartoon example, $PK_{\mathsf{Fred}}^{\mathsf{authenticationKey}}$ would have been one of the principals authenticated in channel establishment and $PK_{\mathsf{John}}^{\mathsf{authenticationKey}}$ would be an owner key for the file //www.activity.com/files/fileX.txt in the file metadata.

The authorization language employed in the prototype is a variant of the says/speaksfor formalism of Lampson and his colleagues [18]. Our authorization language, though simple, is perfectly adequate for the FileProxy activity.

All delegation is explicit and is effected by a signed XML statement using the *maysay* predicate. For example, if $PK_{\mathsf{Owner}}^{\mathsf{authenticationKey}}$ wishes to let $PK_{\mathsf{John}}^{\mathsf{authenticationKey}}$ delegate read permission and $PK_{\mathsf{John}}^{\mathsf{authenticationKey}}$ then grants rights to $PK_{\mathsf{Fred}}^{\mathsf{authenticationKey}}$, the following assertions are used:

1. $\langle PK_{\mathsf{Owner}}^{\mathsf{authenticationKey}}$ says $PK_{\mathsf{John}}^{\mathsf{authenticationKey}}$ maysay * mayread //www.activity.com/files/fileX.txt$\rangle_{pK_{\mathsf{Owner}}^{\mathsf{authenticationKey}}}$.
2. $\langle PK_{\mathsf{John}}^{\mathsf{authenticationKey}}$ says $PK_{\mathsf{Fred}}^{\mathsf{authenticationKey}}$ mayread //www.activity.com/files/fileX.txt$\rangle_{PK_{\mathsf{John}}^{\mathsf{authenticationKey}}}$.

Constrained delegation based on time, location or other factors may be incorporated in the policy language.

Our authorization system is neither an ACL system nor a capability system. We refer to it as a claims based system. It is not an ACL system since there is no centralized access control list anywhere. It is not really a capability system since the mere possession of a claim is insufficient to grant access (or delegate).

FileClient must present evidence in order of evaluation. FileServer simply verifies the evidence, it does not construct a proof itself, so evaluation time is bounded. After evaluation, FileServer currently discards the evidence after evaluation.

[13] Dsig refers to the XML Signature Syntax and Processing (Second Edition), which can be found at http://www.w3.org/TR/xmldsig-core/.

[14] $PK_{\mathsf{Fred}}^{\mathsf{authenticationKey}}$ speaksfor Fred.

4.8 The Trusted Hardware for FileProxy

Our FileProxy prototype runs on a Dell Intel Core i7-2600 based 3.4GHz, 8M, Optiplex 990 workstation with a TPM 1.2 and 16GB of memory. Establishing the Tao Environment isolation and measuring the Trusted OS is accomplished using Dynamic Root of Trust Measurement Boot (or DRTM) [15]. An open source package, TBOOT [8], performs the DRTM boot of the Trusted OS and works as follows.

Grub loads the TBOOT image, the Trusted OS, a special filesystem (the *initramfs* file system) and a special program called sinit.bin into memory. These first three images form the MLE. TBOOT performs a measured launch of the Trusted OS using a special processor instruction[15]. This instruction does a soft reset of all the CPUs in a multiprocessor system, thus setting the CPUs into a known safe state, computes a cryptographic hash of sinit.bin and loads this 160-bit hash into a register in the TPM, called PCR17 and computes a 160-bit cryptographic hash of the MLE, which is loaded into PCR18 of the TPM. It executes sinit.bin, a signed module, which initializes the CPUs and chipsets ensuring that only trusted System Management Mode handlers can run. A composite hash of PCR17 and PCR18 forms the measurement of our Trusted OS, $\mu(\mathsf{TrustedOS})$. This measurement is referenced for TPM seal, unseal and attest operations. These PCR's may not be changed without a reboot. TBOOT sets up an environment required to perform multi-processor initialization and shutdown after Linux exits. Next, the CPU is prepared so that Linux can be called in 64 bit mode with an expected memory map. Finally, TBOOT calls Linux main. From this point forward, Linux proceeds in the usual way.

The net effect of DTRM boot is that the entire Linux image together with initramfs is measured and Linux is started in a state that enables it to perform any required additional initialization securely. Initramfs contains all security critical configuration information as well as any libraries or support programs required by the Trusted OS, FileServer or FileClient. It also contains the FileServer or FileClient executable files.

PK_0 is directly compiled into the (measured) Trusted OS image. The Trusted OS has a statically linked interface to the TPM to access the Tao Primitives supplied by the Trusted Hardware, as the host for the Trusted OS.

To complete the description of the Trusted Hardware as a Tao Environment, we describe the CloudProxy Initialization for the Trusted Hardware in more detail. Readers who are willing to believe that $PK^{\mathsf{sealKey}}_{\mathsf{TrustedHardware}}$, $pK^{\mathsf{attestKey}}_{\mathsf{TrustedHardware}}$, and the *Trusted Hardware attest certificate*,

$$\langle \mathsf{Certificate}(PK^{\mathsf{attestKey}}_{\mathsf{TrustedHardware}}, \mathsf{TrustedHardwareIdentifier})\rangle_{pK_0}$$

have been generated and stored prior to booting the Trusted OS can safely skip the remainder of this paragraph. For the remainder of this paragraph, the reader is assumed to understand TPM 1.2 operation and terminology (see [29]). Prior to first boot of the Trusted OS. A preparatory Linux OS and TPM initialization application is booted on the Trusted Hardware. The TPM initialization application takes ownership of the TPM, causing the TPM to generate a Storage Root Key used for sealing. The TPM initialization application causes the TPM to generate a 2048-bit RSA public/private key pair, called the Attestation Identity Key (AIK), which will become, and we now designate as, the $PK^{\mathsf{attestKey}}_{\mathsf{TrustedHardware}}$. Next, the TPM initialization application causes the TPM to generate evidence (using the ActivateIdentity interface) that is intended for a "privacy CA." The privacy CA (in a manner analogous to the KeyNegoServer) uses this evidence

[15] GETSEC[SENTER] on Intel processors.

to decide whether it trusts the AIK and if it does, it signs a certificate for the AIK. We could have added support in KeyNegoServer to accept this evidence directly and sign the Trusted Hardware attest certificate. Instead, for simplicity, we take this evidence and employ an offline verification process; if it succeeds, we cause pK_0 to sign $PK^{\mathsf{attestKey}}_{\mathsf{TrustedHardware}}$ along with some information about the Trusted Hardware thus generating the Trusted Hardware attest certificate. At the end of this out-of-band process, $PK^{\mathsf{sealKey}}_{\mathsf{TrustedHardware}}$ and $pK^{\mathsf{attestKey}}_{\mathsf{TrustedHardware}}$ reside in, and are never disclosed outside, the TPM[16]. They can only be used for TPM seal, unseal and attest operations. Further, the Trusted Hardware attest certificate has been stored for later retrieval by the Trusted OS.

Since the Trusted OS is the only program booted by the DRTM boot, it is isolated. The TPM holds the Trusted OS measurement, $\mu(\mathsf{TrustedOS})$, which is the MLE. After the DRTM boot, the Trusted OS can request seal, unseal and attest operations from the TPM, which cites this measurement. The TPM can also generate cryptographically secure random numbers, which can be requested by the Trusted OS while it is running[17]. This discharges all the obligations required of the Trusted Hardware to provide the Tao Environment for the Trusted OS. The Tao for this layer is complete.

4.9 The Trusted OS

The Trusted OS is a specially-configured Linux system running a restricted set of services. As explained in the previous section, the Trusted OS executable and data, as well as its initramfs file system, containing all configuration information, required libraries and support programs and a copy of tcService, FileClient or FileServer, is measured and isolated at boot. The TPM contains the measurement of this MLE, which completely characterizes all security critical components of the TrustedOS.

The swap device is encrypted and integrity protected by a key generated randomly at boot using dm-crypt [9], which is also statically linked into the Trusted OS, so no cryptographically unprotected information is used by or exported by the Trusted OS due to paging or swapping. After boot, the Trusted OS treats any mounted device as untrusted (e.g.- disallows SUID programs from these devices). All security critical data is encrypted and integrity protected before being stored by the TrustedOS on these storage devices.

Normally, Linux uses initramfs for a short period of time, discarding it after mounting an external system disk. By contrast, we retain initramfs as our permanent system disk.

The startup script employed by Trusted OS on the FileClient computer will only start the FileClient application and the startup script employed by Trusted OS on the FileServer computer will only start the FileServer application.

To support the Tao Primitives, we employ two device drivers, the TPM driver and a custom driver called *tcioDD*; these are statically linked into the Trusted OS image.

All trusted services (seal, unseal, attest, measure, etc.) are implemented on behalf of the Trusted OS by a user mode program, *tcService.exe*, which is supplied as part of the initramfs filesystem

[16]Actually, the sealing key is a 2048 bit public private key pair on most TPMs. This is unfortunate because it means the TPM unseal operation is very slow, more than a thousand times slower than it would be if it were a symmetric key. For this reason, we only use the TPM to seal a symmetric key and the Trusted OS uses this key to seal other quantities. The seal and unseal operation in the TPM is thus used only once per boot of the Trusted OS. This is merely a performance improvement and doesn't affect the conceptual description at this level.

[17]The need for randomness from the TPM is obviated in "Ivy Bridge" Intel processors which have special random number generation instructions.

and started at boot by the system startup script. Since tcService.exe, as well as the dynamic link libraries it needs[18], are part of the MLE, they cannot be modified without changing the measured identity of the Trusted OS.

tcService has several roles. It opens the TPM device driver exclusively[19]. After startup, tcService opens tcioDD and posts a read, waiting for service requests from hosted software coming through tcioDD. This interposition by the kernel is required as tcService must be isolated and protected. It also must know the PID and code identity of any application on whose behalf it provides a service. As a kernel component, tcioDD is authoritative about this.

tcService carries out the Tao Initialization for the Trusted OS, generating $K^{\text{sealKey}}_{\text{TrustedOS}}$ and $PK^{\text{attestKey}}_{\text{TrustedOS}}$, $pK^{\text{attestKey}}_{\text{TrustedOS}}$ and interacts with KeyNegoServer to obtain the Trusted OS attest certificate.

tcService incorporates a software library[20] implementing the Tao Primitives using $K^{\text{sealKey}}_{\text{TrustedOS}}$ and $pK^{\text{attestKey}}_{\text{TrustedOS}}$. These services are accessed using an interface called *TaoEnvironment* for the Trusted OS, on behalf of its hosted programs. tcService also has an interface, called *TaoHostServices*, that provides an interface to host services. TaoHostServices provides an interface to the TPM driver to get entropy, seal, unseal and attest operations for the Trusted OS as a hosted system on the Trusted Hardware.

The TaoEnvironment and TaoHostServices interfaces, are also linked into FileServer and FileClient. Once FileServer and FileClient obtain their unsealed symmetric and private keys, the TaoEnvironment interface and software library implements encryption, sealing, unsealing and private key operations related to the program key. TaoHostServices, using the same API, serves as the interface to Tao Primitives provided by the Trusted OS. TaoHostServices maintains a read/write channel to tcioDD on behalf the application. This channel is used to communicate Tao Primitive requests and responses, in formatted packets, to the TrustedOS. tcioDD can sometimes satisfy a service request directly, for example, PK_0 is embedded in tcioDD and can be provided directly. However, tcioDD usually forwards requests to tcService. tcService carries out requests and writes the results back to tcioDD for return to the application.

As an illustration, when an application wishes to start another application, it calls the appropriate TaoHostServices interface (StartMeasuredProgram) with the executable file name. TaoHostServices packages the request in a buffer and writes the command packet request to tcioDD; it then posts a read, which will return status and information from tcioDD after the request is processed. After validating the request and noting the caller's PID, tcioDD sends the request to tcService. tcService locks the executable file and computes a cryptographic hash of its contents. It then forks, changes ownership of the forked child process to the requestor's UID (all of them), records the PID of the new child process and the corresponding cryptographic hash, closes any sensitive resources that are inherited by the child process and exec's the application file in the child process. tcService then returns the PID of the child process by writing to tcioDD, which in turn returns it to the requesting process. tcService maintains the association between the PID (or opaque handle) of an activity element and the hosted program's measurement thus allowing tcioDD to ensure results are returned to the correct hosted program. The interaction

[18]These libraries can be determined by calling the Linux command *ldd* on tcService.exe.

[19]In most Linux systems, a software library from IBM called TSS is used to interact with the TPM. TSS is an excellent system but it is too large for our purpose and offers services our system does not need (including multiprogram access). We implement all required TPM service interaction with a much smaller library written for this purpose and linked into tcService.

[20]This library includes implementations of AES, HMACSHA-256, RSA, SHA-1 and SHA-2.

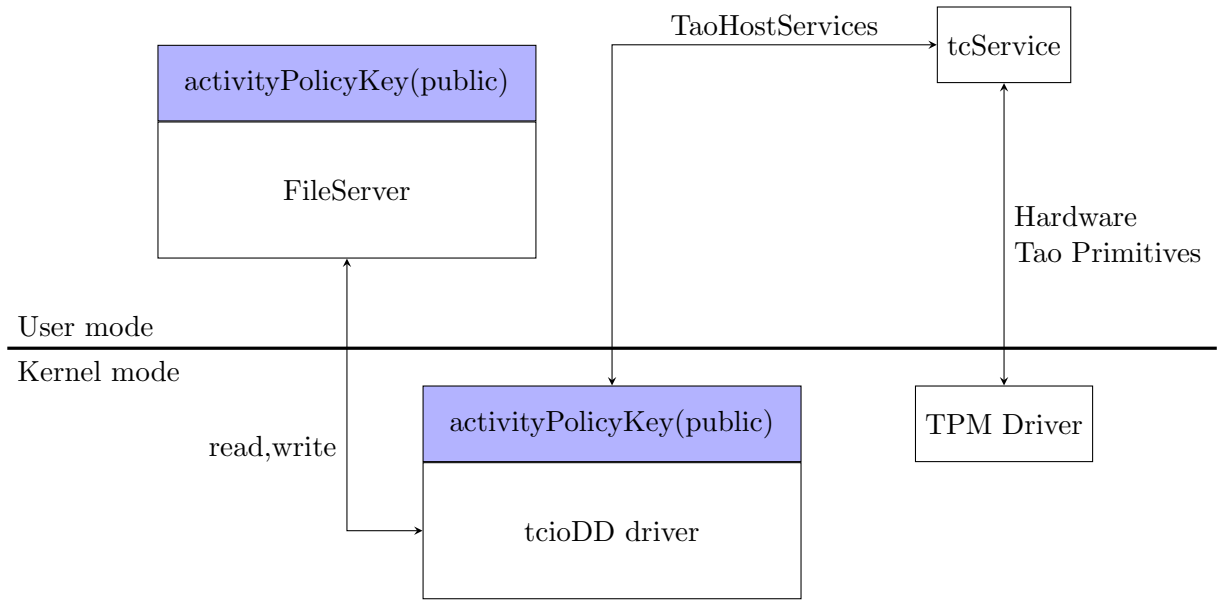

Figure 2: FileServer, TrustedOS, and tcService interactions

between FileServer and tcService is depicted in Figure 2.

To seal, unseal, or attest, an application calls the appropriate TaoHostServices interface with the needed arguments (the blob to seal or unseal, or the blob to be attested). TaoHostServices packages these requests with the parameters, writes the command packet to tcioDD and posts a read for the request. tcioDD validates the request, notes the requesting application's PID in the command packet and forwards the command packet to tcService. tcService looks up the cryptographic hash of the requesting application using the requestor's PID in the table it maintains[21]. In the case of seal, tcService encrypts the hash (using its custodial Trusted OS keys). In the case of attest, tcService signs a composite hash of the requesting application and the data to be attested, using $pK^{\mathsf{attestKey}}_{\mathsf{TrustedOS}}$.

When the Trusted OS starts for the first time on the Trusted Hardware, tcService performs the Tao Initialization generating pair of 128 bit symmetric keys that it uses for encryption and integrity protection of its seal and unseal operation ($K^{\mathsf{sealKey}}_{\mathsf{TrustedOS}}$) and a 1024-bit RSA public/private key pair, which becomes the $PK^{\mathsf{attestKey}}_{\mathsf{TrustedOS}}$. It uses the Trusted Hardware Tao Primitives to seal $K^{\mathsf{sealKey}}_{\mathsf{TrustedOS}}$ and $PK^{\mathsf{attestKey}}_{\mathsf{TrustedOS}}$ and stores the sealed blobs in the standard place on an untrusted storage device it can access after subsequent reboot. Finally, tcService contacts KeyNegoServer resulting in the TrustedOS attest certificate:

$$\langle \mathsf{Certificate}(PK^{\mathsf{attestKey}}_{\mathsf{TrustedOS}}, \mathsf{validityPeriod}, \mathsf{revocationPolicy}, \mu(\mathsf{TrustedOS}))\rangle_{pK_0}.$$

The Trusted OS attest certificate is then stored it in the standard place on an untrusted storage device it can access after subsequent reboots. This completes the Tao Initialization for the Trusted OS.

If this is not the first time the Trusted OS was booted on this Trusted Hardware, immediately after boot, the tcService retrieves, from the standard place on an untrusted storage device it can

[21]The table was updated with the PID and measurement of each application when it is started via the previously described start application interface.

access, the previously sealed blobs containing $K^{\mathsf{sealKey}}_{\mathsf{TrustedOS}}$ and $pK^{\mathsf{attestKey}}_{\mathsf{TrustedOS}}$ and an unencrypted copy of the Trusted OS attest certificate. It then uses TaoHostServices to interact with the TPM to unseal the two keys, which can then be used to provide seal, unseal and attest for the Trusted OS, a host system, and for its hosted systems.

The Trusted OS measures and starts one of two possible hosted applications, namely, FileClient or FileServer.

Because of this startup procedure, FileClient, in the case of the FileClient computer, and FileServer, in the case of the FileServer computer, are isolated. Using the TPM seeded deterministic pseudo-random number generator, the Trusted OS can supply cryptographically secure random numbers to its hosted applications. This discharges two of the obligations required for the Trusted OS to support the Tao Environment.

tcService implements the Tao Primitives and performs Tao Initialization, thus discharging the final obligations for the Trusted OS to support the Tao Environment.

The Tao for this layer is complete.

4.10 FileServer and FileClient

Having described the CloudProxy Primitives, protocols and initialization, there is surprisingly little to add about FileServer or FileClient themselves.

FileClient and FileServer use the TaoEnvironment interface to seal and unseal the keys employed to encrypt and decrypt files and metadata[22] as well as generated, certify, seal and unseal and use their program key.

When FileClient wants a file, it constructs supporting evidence (starting at an owner grant), transmits it over the secure channel to FileServer, retrieves the result and processes.

Obviously, FileClient functionality is quite general and any cloud processing that requires preserving the confidentiality and integrity of computation and data fits comfortably into this model.

5 Performance

The overhead introduced by FileProxy over the processing mandated by the cloud service itself (i.e., whatever computations FileClient performs) consists of:

1. A slight (nearly negligible) delay for DRTM boot of the Trusted OS.
2. The time required to seal and unseal keys.
3. The Tao Initialization, including key generation and KeyNegoServer interaction. This happens once when a FileServer or FileClient runs for the very first time on a computer. This is the only phase when the TPM attestation Primitive is used.

[22]For large systems, several different keys are used to encrypt files and metadata. In this case, FileClient and FileServer use TaoEnvironment sealed keys to encrypt and decrypt these keys. FileClient also uses its sealing keys (or keys protected by those sealing keys) to protect private keys for users used in the "proof of possession" protocol in channel establishment.

4. Channel Establishment, which happens once per FileClient connection.
5. The processing time required to authenticate claims and validate access rights when requests for file services are made by FileClient.
6. The delay caused by symmetric encryption and decryption of transmissions over the channel between FileServer and FileClient. Again, this is nearly negligible.

We examine each of these in turn and conclude with a macrobenchmark.

5.1 DRTM Boot

DRTM boot adds between 10 and 20 milliseconds to boot time for Linux (depending on TPM and code size). This is similar to what is reported in [20].

5.2 Retrieving keys

The Trusted OS unseal operation, using the TPM, is used once at boot. This single unseal operation takes almost a second, seal is three to four times faster. Because the TPM seal and unseal operations are so slow, we only use the TPM to seal and unseal a single symmetric encryption/integrity protection key. This symmetric key is used for further sealing and unsealing for the Trusted OS. For example, the private portion of the Trusted OS Attest Key is sealed and unsealed with this symmetric key. Not only is symmetric encryption orders of magnitude faster than RSA operations on the same processor, but the symmetric encryption and decryption runs on the native CPU (rather than the TPM) which is much faster than the TPM. Using the artifice of only sealing the symmetric key with the TPM improved the performance of retrieving keys by several orders of magnitude.

Sealing and unsealing for applications (performed by the Trusted OS using its keys) runs at native CPU speed. The ring transition costs incurred in reading the encrypted blobs overwhelm any encryption overhead. As a result, this delay for the first seal or unseal is comparable to a read or write of a file block, subsequent seal and unseal operations are dominated by the transition costs from the Trusted OS to tcioDD and tcioDD to tcService.

5.3 Tao Initialization

The performance is dominated by normal TCP/IP latency, which can easily be 100 ms for each KeyNegoServer message sent via a standard broadband provider. By contrast, KeyNegoServer running inside a data center can have latency delays that are shorter by a factor of 25. Non-network related overhead is confined to a few public key operations and a single private key operation, which fits into the noise of network performance variation even with our relatively slow public key implementation performance.

5.4 Channel Establishment

This is comparable in time to an unoptimized TLS channel negotiation together with the additional time required to authenticate any additional user principals. The customary TLS handshake uses four messages while ours has six. As with Tao Initialization processing, Channel

Establishment performance is completely dominated by network latency delays. The actual single thread incremental processing time is less than 20ms, again dominated by private key operations.

5.5 Authentication and authorization

Authentication and authorization processing time is very dependent on the length and complexity of the chain of claims presented by FileClient to demonstrate access rights. For our simple use cases, this involves three to four RSA signature checks on average.

5.6 Encryption operations

File and channel encryption performance is similar to that generally reported for symmetric key cryptographic operations. Our default AES implementation is almost identical to that employed in common encryption packages. Our AES NI implementation single threaded performance is about three times faster. We currently use AES-128-CBC for bulk encryption. Using the default AES algorithm implementation, this runs at about 800MB/sec.

Our public key operations employ RSA and are about four times slower than an optimized implementation in standard packages like OpenSSL. Single threaded RSA-1024 encryption, for example, runs at about 3000 operations per second. Introducing a few standard optimizations would make our public key performance comparable to OpenSSL's. However, we would likely move to Elliptic Curve based public key operations in any proposed production use (see [21]). In the same spirit, bulk encryption in production would likely employ AES-128-GCM, which we wrote but did not make part of the default FileProxy implementation.

5.7 Macro-benchmark

To quantify overall performance, we run two activities. The first activity consists of a pair of Linux applications, like FileClient and FileProxy but with no Tao Primitives. The first program requests files from the second program, which retrieves them locally and returns the file over an unprotected TCP channel. The second activity is the FileProxy activity with all protection mechanisms in place.

6 Related Work

The secure execution of client code on remote servers has long been a goal, and an extensive literature relates to this problem. The solution described in this paper goes beyond the problem of running code on a single machine; it solves the problem of building a trusted distributed system. Two areas of work are related.

- *Trusted Computing* is concerned with way to gain assurance in a given software stack and how to bootstrap this trust to perform operations with trusted hosts.
- Distributed Computing focuses on algorithms and protocols for computations involving servers, with or without assumptions about trust and the kind of failures to be tolerated.

6.1 Trusted Computing

A *Trusted Computing* environment is one in which trust that users have for their computers is justified: hardware and software cannot be subverted, and their properties can be verified by users, given trust in some weaker assumptions. Many of the ideas in Trusted Computing have long been known. For example, trusted boot was first described by Gasser, Goldstein, Kaufman, and Lampson in 1989 [11]. Parno, McCune, and Perrig [23] survey the problems in this area and many of the solutions. Recent work (e.g., [7, **?**, **?**]), focuses on implementing a trusted host in the cloud using virtual machines (VMs).

Self-Service Cloud (SSC) Computing [7] implements a cloud computing service over the Xen [5] hypervisor; security-critical functionality is separated into: (i) a system-level domain that can perform high-level tasks for the machine as a whole, and (ii) a per-VM administrative domain that can be used to perform tasks specific to the VM (or collection of VMs). The system-level domain cannot read the contents of VMs or interfere with their operation, and operations performed on the VM level cannot interfere with other VMs. SSC provisions client VM keys in the per-VM domain during domain establishment. Domain separation for virtual machines could be applied to a version of CloudProxy that has been extended to operate over virtual machines (see Section 7 for details of how to implement this).

CARMA [30] implements trusted hardware, even in the face of tampering with values transferred along hardware buses or memory. The implementation of Trusted Computing in CARMA requires a CPU and a verification device hence reduces the total amount of trust required in the system, since many other systems additionally require trust in other hardware devices. The techniques introduced in CARMA are orthogonal to the Tao; any method of establishing trust in software can be used to establish trust in software that implements the Tao, which seems preferable, because methods that require less trust are more likely to be accepted in production use.

Flicker [20] also allows clients to reduce the amount of software they must trust. With Flicker, clients may switch briefly into a trusted computing mode to execute (extremely) small pieces of code, sometimes as little as a couple of hundred lines, which is well within the province of formal verification (though it still depends on the correctness of all the underlying hardware mechanisms and the trustworthiness of the root of trust). Flicker could be an efficient way to realize the Tao's key negotiation server or otherwise difficult-to-verify systems. A server receiving the request would switch to the key negotiation code only for key negotiation and would otherwise not be subject either to attack or compromise. And clients could still verify that responses came from KeyNegoServer using the authentication techniques of CloudProxy.

6.2 Distributed Computing

Trusted Computing implementors have assumed that hardware executes instructions as directed. However, Secure Function Evaluation (SFE) or the more general Secure Multi-Party Computation (MPC) [12] does not even require this assumption: MPC allows a collection of hosts to compute jointly on the output of a function without any individual host ever learning more than its own input. Early work in MPC established the existence of such protocols but did not provide practical instantiations.

More recent work in MPC has focused on two classes of problems: achieving efficiency under specialized assumptions and implementing MPC under general assumptions but for specialized problems.

Salus [17] provides practical secure function evaluation under the assumption of two non-colluding cloud providers. This assumption can be discharged with relatively good assurance in the modern cloud computing landscape; CloudProxy provides a complementary approach: a client could run two different versions of the server-side software in the cloud, and all sides in a computation could gain trust in a cloud server through code identity verification.

SecPAL [6] is a distributed authorization language that allows relying parties to perform complex reasoning (based on Datalog) involving statements about rights of a requesting party. Similarly, NAL [26] extends the says and speaksfor logic [18] for use in an OS with a TPM-based root of trust. Authorization in CloudProxy employs a simpler language than NAL, but more complex languages like SecPAL and NAL are valuable for certain applications.

Proof of storage [3] provides a way for a remote server to verifiably store data for a client and allow the client to make probabilistic checks that the data is being stored correctly. Proof of data storage comes in two forms. Each provides a different level of assurance in the correctness of the data stored: Proof of Data Possession [3, 2] provides the weaker guarantee, and Proof of Retrievability [16, 27] provides a stronger guarantee. Designing a Proof of Retrievability scheme that supports dynamic update operations efficiently is an open problem.

A proof of storage scheme could be used in conjunction with CloudProxy to attest to freshness of data provided by a trusted host. The current FileProxy implementation guarantees that data read was correctly written by a FileProxy component that was securely booted—it does not guarantee that data is the most current, since an adversary could replace a disk with an old copy, and FileProxy would not be able to tell the difference. Proof of storage supplies the missing freshness guarantee and would be a valuable addition to FileProxy.

7 Discussion

FileClient and FileServer implement a simple client-server, but more general distributed systems also can be built using the same infrastructure. Systems built using CloudProxy are distinguished by the approach to key establishment and the extra level of program authentication provided by the Trusted Computing platform.

FileServer assumes it is the only application running in the only operating system on the physical computer during execution. The same is true of FileClient. While some services might warrant this degree of isolation, cloud data centers providing Infrastructure as a Service time-multiplex computers among tenants. This time-multiplexing is customarily done using a hypervisor. Our Tao is easily adapted to hypervisor hosting; another CloudProxy Tao layer is simply deployed.

7.1 Hypervisors

A hypervisor that securely hosts adversarial partitions (i.e., virtual machines provided by different tenants) must provide strong isolation between guests and between guests and the hypervisor.

Ideally, these security properties are verified by careful design, testing and formal methods. Thus an ideal hypervisor would be small and change rarely. This sort of hypervisor (a "Type 1" hypervisor) does not permit loadable modules. It exports functionality, like I/O, to guest partitions with help from supporting hardware.

Early hypervisors [22], enjoyed surprisingly good performance and security [4]. This success was partly due to device virtualization (e.g., I/O processors on IBM mainframes). The ability to keep

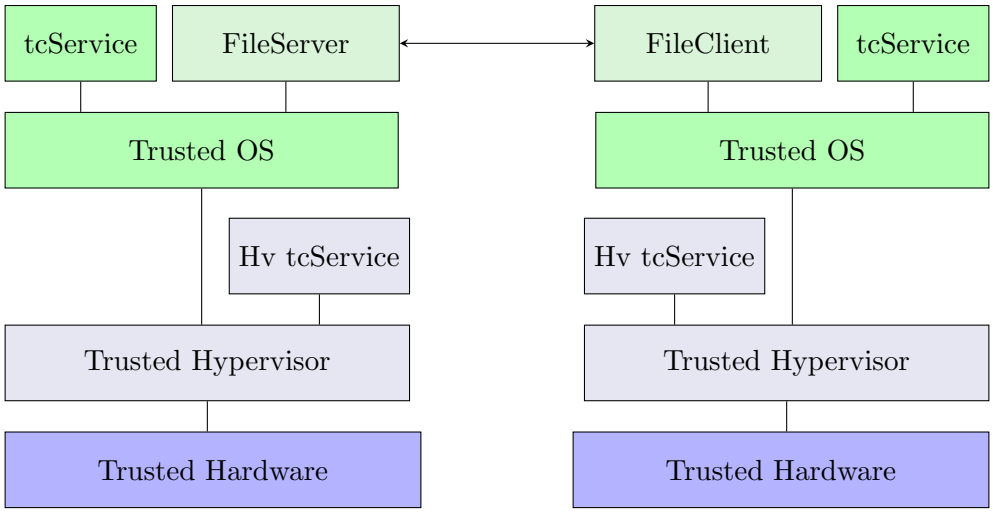

Figure 3: Hypervisor-hosted FileProxy components

an ever growing set of device drivers out of the TCB is critical, and modern processors provide device virtualization and isolation using IOMMUs [1].

Xen [5] is not a type 1 hypervisor hence it has a huge TCB. In contrast, Nova [28], is small, provides excellent performance, and can be relied on to provide these strong isolation properties.

CloudProxy generally, and FileProxy, in particular, can be rapidly adapted to a Nova-like hypervisor, as follows

1. Use TBOOT to perform a DRTM Boot of the hypervisor.
2. Add an authenticated boot operation to the hypervisor so that it can perform a DRTM-like boot for its guests.
3. Provide a special partition designated at start time by the hypervisor, to provide Tao Primitives for the hypervisor on behalf of guest partitions. This partition consists almost exclusively of the tcService program we have already developed.
4. Ensure that the hypervisor zeroizes all guest memory when it reassigned or removed from exclusive guest control.

Figure 3 extends Figure 1 to such a hypervisor-based version; it shows the hosting relationships and communication between FileServer and FileClient under this architecture. In hypervisor-based use, in accordance with the CloudProxy Tao, Trusted Hardware boots and measures a small, well understood *Trusted Hypervisor*, which hosts the Trusted OS. The Trusted OS, in turn, hosts an application activity element (as depicted below). Since tcService requires little OS support, the tcService partition can be implemented as an encapsulated partition with a library OS, thereby vastly reducing the TCB size.

Additional work is required for the Tao to provide secure resource scheduling (by providing another simple, Management Partition). Data centers have typically already developed scheduling software for their services; such software can also be adapted with little change.

Hypervisor-based isolation brings a few additional challenges, notably, preventing side channels between guests. There are simple, known countermeasures that prevent side channels in

cryptographic processing, and side channels for non-cryptographic processing are generally difficult to exploit [25]. User supplied VM's can be designed to avoid side channel leakage for especially sensitive applications. Additionally, side channel exposure can be reduced or eliminated if a sensitive application is assigned to a set of cores sharing critical resources like caches. Sophisticated attacks on VMs (see [31]) can be made substantially more difficult by denying accurate timing interval information[23] between different VM invocations[24].

7.2 Further improvements

Improvements can be made to the current design and implementation of CloudProxy and its FileProxy instantiation. First, KeyNegoServer should be hardened against crashes and other faults. Second, the cryptography should be extended to support revocation and cryptographic agility. Third, the system should allow a richer distributed authorization language for specifying claims. Fourth and finally, our user process should be moved to the Linux kernel for efficiency.

CloudProxy was designed to enable highly scalable operations through redundancy. Key management in the current implementation, though adequate for medium scale operation (without change), should be supplemented to provide support for systems with many cooperating elements. Resilience to KeyNegoServer failures can be achieved by executing Byzantine-fault tolerant protocols for the (relatively simple) operations of the KeyNegoServer.

The current CloudProxy prototype does not implement revocation or other forms of cryptographic agility. For example, the public key system is fixed as RSA with 1024-bit keys. Larger keys are required for actual use, and an Elliptic Curve based system should be used. We used a standard AES implementation but have written an alternative AES NI [14] based implementation that has much better performance and side channel attack resiliency.

Similarly, the cryptographic protocols used by servers and clients in the system should admit updates without recompilation. None of these modifications alters the fundamental design or implementation of CloudProxy or the FileProxy application.

Authorization and authentication operations can be made considerably faster (but more complicated) by caching previously verified claims. Further, the authorization system could be expanded to protect other operations and abstractions. The easiest way to implement this would be to adopt an existing distributed authorization language like SecPAL [6] or NAL [26].

Linux offered an attractive platform for deploying all of the required OS changes (initial trusted file system, application measurement, encrypted swap). However, with modest effort, these features could be integrated directly into the Trusted OS. Integration would bring benefits. It eliminates the need for tcioDD, saving two context switches per Tao Primitive request. Similarly, an executable is currently being read twice in the course of execution: first to hash the executable and later to load it. Full OS support would eliminate the duplication. Although libraries not included in initramfs can be statically linked into an application, a native OS implementation with a more efficient mechanism for incorporating dynamic libraries in code identity would reduce memory consumption and CPU time.

[23] For example, access to the rdtsc instruction on Intel processors.
[24] In production, cross VM rdtsc timing is seldom required.

7.3 Platforms

It is difficult to ensure security without a simple programming model. The model in CloudProxy is simple and would be accessible to developers. To demonstrate the simplicity of the model, one of the authors taught a single undergraduate class session at UC, Berkeley describing the programming model and virtually every student was able to understand how to build a secure cloud service with these basic components.

Our focus on cloud center servers enabled us to achieve our goal by relying only on modest changes to Linux and the hosted applications. However, CloudProxy functionality on traditional client platforms outside a data center (potentially even on phones) is possible. These clients could support trusted interaction with cloud services for medical record use, banking, policy separated environments (home/work, classified/unclassified, etc.), and a long list of other activities.

Data is ultimately used on client systems, so client systems should have strong confidentiality and integrity protection. Many, if not most, of the publicized data center security failures have been caused not by bad data center software but by bad client-hosted processing executed on behalf of operators responsible for maintaining the data center [19].

Furthermore, many recently-developed client systems are based on System-on-a-Chip (SoC) hardware. So, the process of adopting the required hardware changes to implement FileClient (or similar applications) can be easier than for current OEM systems. Adapting FileClient to a client system, requires, among other things, better authenticated user I/O[25]. Clients also need additional support for user-installed applications. General adoption of Trusted Computing systems like FileProxy will depend on the usability as well as the security of clients.

References

[1] D. Abramson, J. Jackson, S. Muthrasanallur, G. Neiger, G. Regnier, R. Sankaran, I. Schoinas, R. Uhlig, B. Vembu, and J. Wiegert. Intel virtualization technology for directed i/o. *Intel Technology Journal*, 10:179–192, August 2006.

[2] G. Ateniese, R. Bruns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song. Provable data possession at untrusted stores. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, pages 598–609. ACM, 2007.

[3] G. Ateniese, S. Kamara, and J. Katz. Proofs of storage from homomorphic identification protocols. In *Proceedings of the 15th International Conference on the Theory and Application of Cryptology and Information Security: Advances in Cryptology*, pages 319–333, 2009.

[4] C. R. Attanasio, P. W. Markstein, and R. J. Phillips. Penetrating an operating system: a study of VM/370 integrity. *IBM Systems Journal*, 15(1):102–116, March 1976.

[5] 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 *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, pages 164–177. ACM, 2003.

[25] Graphics cards and USB systems need to be changed so that users cannot be fooled into thinking display output comes from or user input goes to a different program than it expected. Client systems also require additional effort to ensure that GPUs cannot break isolation boundaries as can happen in current client systems.

[6] M. Becker, C. Fournet, and A. D. Gordon. Design and semantics of a decentralized authorization language. *Journal of Computer Security*, 18(4):619–665, 2010.

[7] S. Butt, H. A. Lagar-Cavilla, A. Srivastava, and V. Ganapathy. Self-service cloud computing. In *Proceedings of the 2012 ACM Conference on Compute and Communications Security*, pages 253–264. ACM, 2012.

[8] J. Cihula. Tboot. http://sourceforge.net/projects/tboot.

[9] dm-crypt: Linux kernel device-mapper crypto. http://code.google.com/p/cryptsetup/wiki/DMCrypt.

[10] P. England, B. W. Lampson, and J. Manferdelli. A trusted open platform. *IEEE Computer*, 36(7):55–62, 2003.

[11] M. Gasser, A. Goldstein, C. Kaufman, and B. Lampson. The digital distributed system security architecture. In *Proceedings of the 1989 National Security Conference*, pages 305–319. NIST, 1989.

[12] O. Goldreich, S. Micali, and A. Widgerson. How to play any mental game or a completeness theorem for protocols with honest majority. In *Proceedings of the 19th Annual Symposium on the Theory of Computing*, pages 218–229. ACM, 1987.

[13] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest we remember: cold book attacks on encryption keys. In *Proceedings of the 17th USENIX Security Symposium*, pages 45–60. USENIX Association, 2008.

[14] Intel Corporation. Intel architecture manual. Available at http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.

[15] Intel Corporation. Measured launched environment developer's guide, March 2011. http://www.intel.com/content/www/us/en/software-developers/intel-txt-software-development-guide.html.

[16] A. Juels and B. S. K. Jr. PORs: Proofs of retrievability for large files. In *Proceedings of the 2007 ACM Conference on Computer and Communications Security*, pages 584–597. ACM, 2007.

[17] S. Kamara, P. Mohassel, and B. Riva. Salus: A system for server-aided secure function evaluation. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, pages 797–808. ACM, 2012.

[18] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems*, 10(4), November 1992.

[19] McAfee Labs. Protecting your critical assets. lessons from "operation aurora", 2010. Available at http://www.wired.com/images_blogs/threatlevel/2010/03/operationaurora.

[20] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: an execution infrastructure for TCB minimization. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, pages 315–328. ACM, 2008.

[21] National Security Agency. NSA Suite B Cryptography. Available at http://www.nsa.gov/ia/programs/suiteb_cryptography.

[22] R. P. Parmalee, T. I. Peterson, C. C. Tillman, and D. J. Hatfield. Virtual storage and virtual machine concepts. *IBM Systems Journal*, 11(2):99–130, June 1972.

[23] B. Parno, J. M. McCune, and A. Perrig. Bootstrapping trust in commodity computers. In *Proceedings of the 2010 IEEE Sympoisum on Security and Privacy*, pages 414–429. IEEE, 2010.

[24] M. Peinado, Y. Chen, P. England, and J. Manferdelli. NGSB: A trusted open system. In *Information Security and Privacy: 9th Australasian Conference*, pages 86–97, 2004.

[25] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, pages 199–212. ACM, 2009.

[26] F. B. Schneider, K. Walsh, and E. G. Sirer. Nexus authorization logic (NAL): Design rationale and applications. *ACM Transactions on Information and System Security*, 14(1), May 2011.

[27] H. Shacham and B. Waters. Compact proofs of retrievability. In *Proceedings of Asiacrypt 2008*, pages 90–107. Springer-Verlag, 2008.

[28] U. Steinberg and B. Kauer. NOVA: a microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th European Conference on Computer Systems*, pages 209–222. ACM, 2010.

[29] Trusted Computing Group. Trusted platform module, version 1.2, 2011. http://www.trustedcomputinggroup.org/resources/tpm_main_specification.

[30] A. Vasudevan, J. M. McCune, J. Newsome, A. Perrig, and L. van Doorn. CARMA: A hardware tamper-resistant isolated execution environment on commodity x86 platforms. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS)*. ACM.

[31] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-VM side channels and their use to extract private keys. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, pages 305–316. ACM, 2012.