The CloudProxy Tao for Trusted Computing

John Manferdelli
Tom Roeder
Fred Schneider

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2013-135
http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-135.html

July 19, 2013

Acknowledgement

Intel Science and Technology Center for Secure Computing, UC, Berkeley

The CloudProxy Tao for Trusted Computing

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

Tom Roeder
Google, Inc.
Mountain View, CA

Fred B. Schneider*
Department of Computer Science
Cornell University
Ithaca, NY

Abstract

Applications running in a cloud data center face several challenges, including secure deployment, insider attacks, and cryptographic key management. Previous research shows how to securely start and run individual programs using the Trusted Platform Module or other secure co-processors, but none of that work solves the end-to-end problem of deploying and gaining assurance in distributed applications running on third-party hardware. And key management in existing systems often requires trust in potentially untrustworthy certificate authorities.

CloudProxy is a new framework that provides secure deployment of applications to the cloud, defends against insider attacks, and provides protocols for automatic key management. Data managed by CloudProxy is never stored or transmitted in unencrypted form, and cryptographic keys are provisioned in a way that defends against malicious operators or other data-center insiders. Protocols are provided for remote or local clients to authenticate the executable and execution environment of a server and for a server to authenticate the executable and execution environment of its clients. Three prototype applications were implemented to evaluate the utility of CloudProxy: FileProxy, a file service; AuthProxy, an authentication service for remote third parties; and BidProxy, an auction service. Performance measurements show that CloudProxy is a practical way to support secure, distributed applications.

1 Introduction

The *CloudProxy Tao* (henceforth, "the Tao") is a recipe for creating secure, distributed, cloud-based services by combining ingredients that are already available in many cloud data centers. The Tao is realized as an interface that can be implemented at any layer of a system. *CloudProxy* implements multiple layers of the Tao and provides means for

- protecting the confidentiality and integrity of information stored or transmitted by some *hosted program*,
- establishing that the code executed as a hosted program in a cloud is the expected code and is being run in the expected environment, and
- authenticating requests to the hosted program to check that they come from a client executing some expected program in an expected environment, either remotely or locally in the cloud.

CloudProxy is the first implemented, fully fleshed-out system providing these properties along with key management and an appropriate trust model for all principals. To demonstrate the versatility of CloudProxy, we used it to implement three prototypes: *FileProxy* is a simple file service; *AuthProxy* is a stateless authentication service; and *BidProxy* is a distributed auction service.

The Tao combines hardware-based memory isolation along with unforgeable *measurement-based security principals* [18, 10, 35], as supported by Trusted Platform Modules (TPMs) [42] and other secure co-processors. With

*Schneider is supported in part by AFOSR grants F9550-06-0019 and FA9550-11-1-0137, National Science Foundation grants 0430161, 0964409, and CCF-0424422 (TRUST), ONR grants N00014-01-1-0968 and N00014-09-1-0652, and grants from Microsoft.

- *Computational Integrity*: Activity elements have been started from unchanged code in well-known configurations.
- *Isolation*: Intermediate results from activity elements cannot be observed by unauthorized data center operators, tenants, employees, or intruders.
- *Policy Enforcement*: Requests received by activity elements are performed if and only if they comply with activity owner policy.
- *Data Integrity*: Data identified by the activity owner as requiring integrity checks is written or modified only by trusted programs and only to the extent authorized by activity owner policy.
- *Data Confidentiality*: Neither data at rest nor data in transit that is identified as confidential by the activity owner will be disclosed unless that disclosure complies with activity owner policy.
- *Attestation*: An activity owner can authenticate statements from activity elements by using untrusted communications channels.

Figure 1: CloudProxy Security Properties

measurement-based security principals, a cryptographic key can be associated with some measurement value. Only principals having that associated measurement value are permitted to access and use the key. The measurement value typically combines the hash of a requesting program's executable with any environmental information that affects program execution—for example, boot parameters and information identifying the host execution environment. Consequently, the capability to generate digital signatures or to decrypt data is available only to unmodified programs being executed in unmodified environments.

Specialized hardware is just one way to implement measurement-based security principals, and embodiments of the Tao are not limited to the lowest level of a system's software stack. Moreover, the Tao can be deployed recursively, because a *host system* supporting the Tao necessarily has means to enable its hosted programs to instantiate the Tao for programs that they host.[1] For example, CloudProxy is a stack of three levels, each instantiating the Tao: the *Trusted Hardware* (TrHW) provides the Tao for an operating system called *Trusted OS* (TrOS); and TrOS provides the Tao for programs running as *activity elements* which, together, comprise an *activity*. An activity is an instance of a distributed computation executing on behalf of some *activity owner*. And an *activity owner policy* specifies authenticated claims that must accompany a request in order for that request to be deemed authorized.

Cryptography is a key ingredient for the Tao. It protects confidentiality and integrity of data stored on secondary storage or sent over to clients of a hosted application. It is used to authenticate activity elements to their clients. And it is used in *claims-based authorization* [8, 37] for controlling access to activity functionality. So, the Tao includes means to provision cryptographic keys, providing a small, independently-deployed component (hence, easily trusted) along with protocols that defend against malicious actions by data-center operators or employees.

The remainder of this paper is organized as follows: Section 2 describes hardware assumptions related to physical security and our threat model. Section 3 describes the Tao. In section 4, we describe the design and implementation of CloudProxy; section 5 describes our example applications. Section 6 describes performance of these applications. Section 7 discusses related work. And section 8 discusses on-going work to extend and generalize CloudProxy.

2 Goals and Assumptions

CloudProxy enforces the security properties listed in Figure 1. The enforcement of these properties depends critically on several assumptions.

CloudProxy assumes that each activity element, if executed in a benign environment, will behave as expected for

[1]This is conceptually similar to the recursive microkernel/virtual machine work of Ford et al. [17] and the nested virtualization work of Ben-Yehuda et al. [9].

any input to any of its interfaces. We thus have proceeded expecting that methods developed to achieve activity element correctness will be independent of any machinery CloudProxy provides for defending against attacks connected to cloud data center deployments.

CloudProxy defenses address a threat model corresponding to possible exploitation of data center vulnerabilities:

- The adversary may control networks in the data center and networks that connect the data center to clients.
- The adversary may control machines outside of the data center that are not executing CloudProxy programs.
- The adversary may have physical access to all data center hardware and infrastructure, except hardware executing CloudProxy activity elements or programs supporting their execution.
- Data center insiders and tenants may collude with the adversary.

So, for example, disks are easily removed and replaced in server machines, so our threat model admits an adversary that might, at any time, remove a disk from a powered-down machine, examine and modify its contents, and then re-install the updated disk. The threat model also includes data centers managed using flawed software that harbors maliciously-inserted components for attacking specific activity elements.

The threat model restricts physical access to processors during execution of CloudProxy programs. One way to discharge this assumption in practice would be to enclose racks of processors in cages. Ideally, each cage would be under video surveillance and would remain locked while its computers are running and for a few minutes after shutdown (to help prevent cold boot attacks [22]). Providers of Infrastructure as a Service (IaaS) often employ facilities like this for customers.

CloudProxy software runs on Trusted Hardware, comprising CPUs and related motherboard components that support measured boot, memory isolation (including the capability to restrict access by bus masters with Direct Memory Access), and provide for sealing, unsealing, and attestation by measurement-based security principals.[2] Measured boot capability enables the launch environment—including booted software and any security-critical boot parameters—to be measured by the trusted hardware. The resulting *measured launch environment* (MLE) thereafter can serve as a measurement-based security principal. Peripherals, such as disks, network infrastructure, or computers that do not run CloudProxy systems need not be restricted.[3] Note, however, that our prototype currently does not defend against attacks that roll back the state of disk to an earlier valid state.[4]

Notably absent from the list of security properties in Figure 1 is an availability guarantee. The usual defenses against maliciously-excessive loads are not precluded by the CloudProxy protocols. Cloud data center insiders also have additional opportunities for compromising availability of an activity. A malicious insider could turn off power, sever network connections, or simply refuse to schedule activity elements. The only known defense against such attacks is replication of activity elements on systems that cannot all fall under the control of a single adversary. CloudProxy is compatible with protocols to implement such replication.

3 The Tao Environment

A *Tao environment* is implemented by a *host* and used by and for the benefit of a hosted program *hp*. It offers

- memory (e.g., processor registers and cache) that, once assigned to a correctly loaded hosted program, cannot be read or modified by any other program,
- a secure association between *hp* and an unforgeable measurement value $\mu(hp)$, where $\mu(hp)$ is a representation of a correctly configured instance of *hp*, and
- means to ensure confidentiality and integrity for secrets generated and handled by the measurement-based security principal $\mu(hp)$ associated with *hp*.

[2]Existing server boards that incorporate Intel CPUs supporting SMX, TXT, VT-d, and TPM 1.2 fulfill these requirements [23] and are already widely deployed in cloud data centers; AMD CPUs supporting AMD-V with IOMMUs and TPM 1.2 also satisfy the requirements [2]. The TPM is the only component not currently included in every server board, and its cost is under a dollar.

[3]A potential exception is the swap device; see section 4.

[4]For techniques to mitigate the attack, see section 8.

In a Tao environment, two cryptographic keys[5] persist across activations of whatever system *host* is serving as the host. *Sealing key* K_{host}^{seal} is a symmetric key; *host attestation key* $PK_{host}^{\mathsf{attest}}/pK_{host}^{\mathsf{attest}}$ is a public/private key pair. K_{host}^{seal} and $pK_{host}^{\mathsf{attest}}$ are known only to *host*.

Prior to installing hosted programs to serve as activity elements on a host, the activity owner generates a public/private key pair PK_0/pK_0. The activity owner keeps private key pK_0 secret. Certificates signed by pK_0 can thus be trusted by activity elements. Private key pK_0 often serves as a root key for chains of certificates that convey claims (or delegate the right to sign claims) for authorizing actions at an activity element.

The following primitives are provided by any implementation of a Tao environment:

Hosted Program Initiation. $StartHostedProgram(hp)$ causes a host to load and start executing a new hosted program *hp*. The invocation returns an opaque handle that identifies the execution context for *hp* on that host.

Measurement and Attestation. A hosted program *hp* may invoke $GetHostedMeasurement()$ to learn measurement value $\mu(hp)$. Invocation of $Attest(data)$ by *hp* generates and returns a certificate[6] $\langle \mathbf{attest}(data, \mu(hp)) \rangle_{pK_{host}^{\mathsf{attest}}}$ that binds *data* to *hp* on *host*. And invocation of $GetAttestCertificate()$ returns a *host attest certificate* $\langle \mathbf{cert}(PK_{host}^{\mathsf{attest}}, hostIdentifier) \rangle_{pK_0}$ that binds host attestation public key $PK_{host}^{\mathsf{attest}}$ to *hostIdentifier*, which typically is a measurement value for the program that implements host *host*.

Sealing and Unsealing. Invocation of $Seal(data)$ by a hosted program *hp* executing in a Tao environment that has sealing key K_{host}^{seal} produces a *sealed blob*[7] $E(K_{host}^{\mathsf{seal}}, nonce \mid\mid \mu(hp) \mid\mid data)$. Invocation of $Unseal(sb)$ by a hosted program hp' on some host $host'$ extracts and returns *data* from *sb* only if *sb* is a sealed blob generated by hp' executing $Seal(\cdot)$ on that same host. The host implements this by using its sealing key to decrypt *sb* and then checking whether the measurement value in *sb* equals $\mu(hp')$—a check that succeeds only if the host system and hosted program for the decryption of *sb* are the same as for the $Seal(data)$ that generated sealed blob *sb*.[8]

Entropy. $GetEntropy(n)$ returns a cryptographically-strong random number of size *n* bits.

3.1 Cryptographic Keys for Use by Activities

A hosted program *hp* that digitally signs messages using some private key pK_{hp}^{auth} it generates must be able to prove that (i) *hp* is the only program that has access to the private key; and (ii) the public/private key pair was generated by *hp*. The first requirement is satisfied if *hp* can prove that it is executing in a Tao environment, because guarantees about memory isolation then protect a key stored in memory, and $Seal(\cdot)$ can protect a key stored on disk or elsewhere outside of the hosted program.

Discharging the second requirement is not difficult if the activity owner is willing to employ a separate KeyServer component that (i) can be reached over a communications channel,[9] (ii) knows pK_0 for the activity, and (iii) knows $\mu(hp)$ for the activity's hosted program. To protect pK_0 and $\mu(hp)$, the KeyServer installation should be physically secure and employ multi-person access control when performing critical operations. Moreover, since KeyServer can be the only component that needs to know pK_0, it might use an attached Hardware Security Module (HSM) to generate pK_0—the key would then never be available outside of KeyServer. However, implementation of $GetAttestCertificate()$ requires communication with KeyServer if that is where pK_0 is stored.

Figure 2 gives a protocol for *hp* executing in a Tao environment on *host* to generate a new public/private key pair $PK_{hp}^{\mathsf{auth}}/pK_{hp}^{\mathsf{auth}}$ and obtain from KeyServer a certificate

$$\langle \mathbf{cert}(PK_{hp}^{\mathsf{auth}}, \mu(hp)) \rangle_{pK_0}$$

attesting that PK_{hp}^{auth} comes from a hosted program *hp* executing in a Tao environment. The host attest certificate retrieved in step 3 is trusted by the activity owner, hence also by activity elements, because it is signed by pK_0. It

[5]In this paper, a symmetric key is denoted with a upper case K, often decorated with a subscript that designates the security principal controlling the key and with a superscript identifying the purpose of the key. A public/private key pair is denoted PK/pK, where PK is the public key and pK is the corresponding private key. Each may be decorated with a subscript or superscript as for symmetric keys.

[6]Notation $\langle M \rangle_{pK}$ denotes a message M digitally signed by private key pK.

[7]Encryption and decryption operations employing a key K are denoted $E(K,\cdot)$ and $D(K,\cdot)$, respectively; initialization vectors are left implicit. And, we write $v \mid\mid w$ to denote serialization of v and w into a structure that can later be (uniquely) parsed.

[8]Note that some trusted hardware, like TPMs, provide NVRAM with similar semantics to Seal/Unseal and better performance, and this could be used to implement Tao Seal/Unseal operations in some cases.

[9]In CloudProxy, physical communications channels are not assumed to be secure, and we employ protocols to provide secure communications over these untrusted channels. KeyServer communications use such channels.

1. *hp*: Generate $PK_{hp}^{\mathsf{auth}}/pK_{hp}^{\mathsf{auth}}$ using entropy from an invocation of $\mathit{GetEntropy}(\cdot)$.
2. *hp*: Invoke $\mathit{Attest}(PK_{hp}^{\mathsf{auth}})$ to obtain certificate

$$\langle \mathbf{attest}(PK_{hp}^{\mathsf{auth}}, \mu(hp))\rangle_{pK_{host}^{\mathsf{attest}}}.$$

3. *hp*: Invoke $\mathit{GetAttestCertificate}()$ to obtain host attest certificate for Tao environment executing on *host*

$$\langle \mathbf{cert}(PK_{host}^{\mathsf{attest}}, \mathit{hostIdentifier})\rangle_{pK_0}.$$

4. $hp \rightarrow \mathit{KeyServer}$: $\langle \mathbf{attest}(PK_{hp}^{\mathsf{auth}}, \mu(hp))\rangle_{pK_{host}^{\mathsf{attest}}} \,||$
   $\langle \mathbf{cert}(PK_{host}^{\mathsf{attest}}, \mathit{hostIdentifier})\rangle_{pK_0}$
5. *KeyServer*: Verify that the certificate comes from a trusted host; in the current implementation, all host certificates are signed by the activity policy key so this step can be implemented by verifying the signature on the certificate. Compare $\mu(hp)$ in **attest** certificate against a list of measurements of known, trusted programs. If $\mu(hp)$ is in the list, then:

   $\mathit{KeyServer} \rightarrow hp$: $\langle \mathbf{cert}(PK_{hp}^{\mathsf{auth}}, \mu(hp))\rangle_{pK_0}$

Figure 2: Protocol for Provisioning PK_{hp}^{auth} / pK_{hp}^{auth}

asserts that $PK_{host}^{\mathsf{attest}}$ is a host attestation key for a Tao environment. So, the activity owner trusts that the certificate retrieved in step 2 was produced by a Tao environment; that certificate asserts that an $\mathit{Attest}(PK_{hp}^{\mathsf{auth}})$ primitive was invoked by a hosted program with measurement value $\mu(hp)$—that is, the certificate was produced by hosted program *hp*.

It might seem like an adversary could performing a man-in-the-middle attack in step 4 to replace the attestation with one of its own choosing. However, note that the certificate for $PK_{host}^{\mathsf{attest}}$ is bound to the measurement of the host, as certified by KeyServer itself. So, KeyServer can check that this message came from a program with the provided measurement. And *hp* can verify the signature in step 5 with its embedded copy of PK_0 to check that this message came from KeyServer.

It might also seem like freshness of the key generated in step 1 is not guaranteed, since there is no nonce in this protocol. However, recall that *hp* is guaranteed to be a known, trusted program, hence KeyServer can be certain that it generated a fresh key for this interaction.

3.2 Authorization

In the Tao, authorization for operations that hosted programs provide is enforced by requiring client requests to be accompanied by sets of *claims*. Each claim is a digitally signed certificate. A client request is deemed authorized only if the accompanying set of claims authorizes the request by satisfying the hosted program's activity owner policy. Typically, such a policy will be rooted in a public/private key pair *pK*/*PK*, where *pK* is controlled by the activity owner. Claims are ignored unless they are signed by *pK* or by some other key to which delegations have been made—directly or indirectly—in claims signed by *pK*.

A hosted program *hp* can independently determine whether a set of claims satisfies such an activity owner policy provided that activity owner's public key *PK* is available to verify signatures on the certificates. A particularly attractive solution is to incorporate *PK* into the executable for *hp*. This construction forges an unbreakable binding between *PK* and *hp*, since changes to *PK* now cause changes to $\mu(hp)$. Sealing, unsealing, and attestation all depend on the value of $\mu(hp)$, so the construction puts changes to the authorization policy on par with other changes to a hosted program.

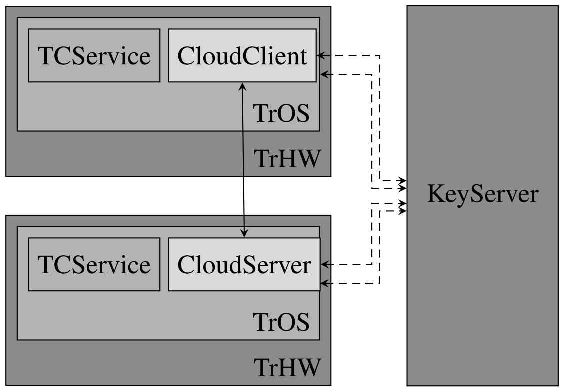

Figure 3: Layered CloudProxy Components

Layer	TrHW	TrOS	applications
Measurement	TPM	TPM	software
Isolation	program	program	N/A
Primitives	device driver	TCService	software
Initialization	out of band	KeyServer	KeyServer

Table 1: CloudProxy Tao layers

4 CloudProxy

CloudProxy is a layered system, where each layer implements an instance of the Tao by using functionality provided by lower layer(s).

- TrHW, as host, provides the Tao for TrOS as a hosted program.
- TrOS, as host, provides the Tao for activity elements.

Figure 3 depicts the hosting relationships and communications channels used for two notional activity elements—*CloudServer* and *CloudClient*—along with KeyServer. CloudServer and CloudClient represent server and client roles, respectively, that can be filled by activity elements in CloudProxy applications. Note that CloudClient may execute in a data center but is not required to execute there. Solid lines in the figure represent communication paths that are active during operation of the activity; dashed lines represent communication paths required for one-time key provisioning at initialization. Each hosted program is contained in its host system, and darker-colored boxes represent more-trusted systems.

The mechanisms implementing the Tao in CloudProxy differ across layers. Table 1 summarizes. TrHW was described in section 2. When the host is software, Tao primitives are implemented using essentially the same code. For example, CloudClient, CloudServer and TrOS all use the same library for their implementations of the Tao primitives. And all certificates are DSig [45] signed and formatted.

TrOS is a modified Linux operating system. We performed two types of modifications to stock Linux: removing and hardening attack surfaces, and adding a new kernel/user component to implement the Tao.

We hardened a stock Linux image in four ways. First, we reduced the size and complexity of the Linux system by removing all unnecessary applications. Second, we disabled dynamic module loading in the kernel to prevent changes to a measured kernel after trusted boot. Third, all data and programs run by TrOS are supplied in an in-memory file system, `initramfs`, which forms part of the measurement of TrOS. Fourth and finally, we changed the Linux init script to retain `initramfs` after boot and continue to use it as the system disk throughout execution.

Since only `initramfs` file contents are authenticated, all other filesystems mounted by TrOS after boot are mounted as untrusted; that means TrOS will not execute any programs, especially `suid` programs, from these untrusted file systems. Mounted filesystems are used only for (untrusted) storage; integrity and confidentiality protection of storage is provided by cryptographic means using keys that only TrOS knows.

If swapping or demand paging is enabled, then the swap device should be encrypted with `dm-crypt` [15] using keys that are generated at boot time. Since `dm-crypt` provides confidentiality but not integrity protection, the swap device must be physically located inside the physical protection barrier that protects processors during program execution.

A user mode program called TCService is included as part of `initramfs` and is launched by TrOS after boot. TCService implements the Tao primitives for the TrOS. Since TCService and the dynamic link libraries it needs are retrieved from `initramfs` and are part of the MLE, they cannot be modified without changing $\mu(\text{TrOS})$ and causing the lower-level Tao environment (the TPM in TrHW) to block the modified TCService from accessing its sealed storage or making the same attestations. Because hosted programs (processes, under Linux) must be reliably measured and authenticated by TCService, a device driver called `tcioDD` is statically linked into the TrOS as a trusted interposition channel between the activity elements requesting Tao host services from TrOS and TCService. Figure 4 illustrates the interaction between CloudServer, TCService, and `tcioDD`.

TrHW and TrOS in CloudProxy use basically the same approach to enforcing the security properties given in Figure 1. Computational Integrity is enforced by performing measurement of hosted programs before they execute. Isolation comes from running only a single hosted program within a Tao environment. Policy Enforcement uses claims, as described in section 3.2. Data Integrity and Data Confidentiality are achieved cryptographically using keys that are sealed to the hosted program. Attestation is achieved by sending and verifying program measurements during secure channel establishment.

In CloudProxy, all software layers that implement the Tao do so using a common library called the CloudProxy Library. This library contains all cryptographic[10] support required by the Tao and CloudProxy programs, the implementation of all functions required for initialization, storage and retrieval of sealed blobs, sealing, unsealing, attestation and verification of attestation, implementation of all Tao protocols including secure channel establishment, certificate, claim verification and the access control guard used to make authorization determinations. The CloudProxy Library has a standard interface used by hosted programs to obtain services provided by a host as well as standard interfaces to functions implementing host services performed by a host.

4.1 CloudProxy Initialization

In CloudProxy, the same key PK_0 is used in every layer.[11] At the TrOS layer of CloudProxy, PK_0 is contained in initialized data in `tcioDD`. For both CloudServer and CloudClient, PK_0 is stored as initialized data in the program image of each program.

After embedding PK_0, the activity owner measures the resulting TrOS, CloudServer, and CloudClient in exactly the same way as they will be measured in the data center. For CloudServer and CloudClient, this simply involves calculating a SHA-256 based hash of the executable files. In the case of TrOS, this involves calculating two 160 bit SHA1 based hashes that TrHW will compute and save in PCR 17 and PCR 18 of the TPM as well as the composite hash representing $\mu(TrOS)$, which is computed from these values. All these measurements and their associated programs are recorded for further use.

Trusted Hardware Initialization. CloudProxy initialization of the hardware layer differs from the rest of CloudProxy initialization. TrHW depends on a preparatory Linux OS and TPM initialization application booted on TrHW. The TPM initialization application takes ownership of the TPM, causing the TPM to generate a Storage Root Key which will later be used for sealing and which we denote by $PK_{TrHW}^{\mathsf{seal}}$. The TPM initialization application also causes the TPM to generate a 2048-bit RSA public/private key pair, called the Attestation Identity Key (AIK); we denote the public portion of this key as $PK_{TrHW}^{\mathsf{attest}}$; $pK_{TrHW}^{\mathsf{attest}}$ is stored in the TPM.

Next, the TPM initialization application causes the TPM to generate evidence that is intended for a privacy CA to demonstrate the trustworthy nature of the AIK and its generation. This evidence is saved for later use. The AIK must be signed so it can be trusted by activity elements. This is done in a secure facility operated by the activity owner by an offline program which verifies the saved evidence. The use of a secure facility allows the activity owner to certify only TPMs of its choice. If verification succeeds, it uses pK_0 to sign $PK_{TrHW}^{\mathsf{attest}}$ along with some information about TrHW, thus generating TrHW attest certificate

$$\langle \mathbf{cert}(PK_{TrHW}^{\mathsf{attest}}, TrHWIdentifier)\rangle_{pK_0}. \tag{1}$$

At the end of this out-of-band process, $PK_{TrHW}^{\mathsf{seal}}$ and $pK_{TrHW}^{\mathsf{attest}}$ reside in, and are never disclosed outside of, the TPM. And the TrHW attest certificate is copied to a storage location from which it can be accessed by TrOS after boot.

[10] Random number generation, RSA encryption, signing, key generation, verification, AES, HMAC, cryptographic hash functions (SHA1 and SHA-256) and symmetric key generation.

[11] Other instantiations of the Tao could use different keys at each layer, but CloudProxy employs the same key for simplicity in key distribution.

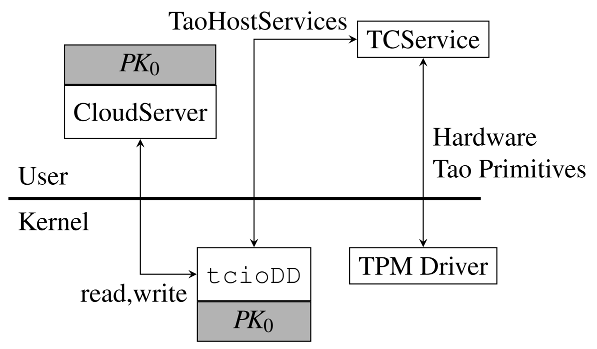

Figure 4: CloudServer, TrOS, and TCService interactions

TrOS Initialization. Tao Environment isolation and measurement for TrOS is accomplished using Dynamic Root of Trust Measurement (DRTM) [24] Boot. `tboot` [14] performs the DRTM boot of TrOS by loading the `tboot` image, TrOS, a special filesystem (`initramfs`), and a special program called `sinit.bin` as the MLE. `tboot` performs a safe boot of the hardware and puts measurements of the MLE in read-only registers PCR17 and PCR18 in the TPM; a composite hash of these registers serves as $\mu(\text{TrOS})$.

The net effect of DTRM 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 TrOS, CloudServer, and CloudClient. It also contains CloudServer or CloudClient executables.

TCService initializes TrOS, generating K^{seal}_{TrOS} and key pair $PK^{\text{attest}}_{TrOS}$ / $pK^{\text{attest}}_{TrOS}$, then interacts with KeyServer to obtain the TrOS attest certificate. The TrOS attest certificate is stored in a standard location on an untrusted storage device that TrOS can access after subsequent reboots. TCService uses `tcioDD` to manage all application initialization and handles all Tao Primitive requests for `tcioDD`. Kernel support for these requests is necessary because our implementation uses PIDs as handles, and the kernel (hence `tcioDD`) is authoritative about process identity.

The TrOS unseal operation using the TPM is invoked once, at boot. This unseal operation takes almost a second; seal is three to four times faster. Because TPM seal and unseal operations are so slow, we only use the TPM to seal and unseal K^{seal}_{TrOS}. This key is used for further sealing and unsealing for TrOS. For example, $pK^{\text{attest}}_{TrOS}$ is sealed and unsealed with K^{seal}_{TrOS}. Not only is symmetric encryption orders of magnitude faster than RSA operations on the same processor, but symmetric encryption and decryption run on the native CPU, which is much faster than the TPM. Replacing TPM operations with a sealed symmetric key improves the performance of retrieving keys by several orders of magnitude.

CloudServer and CloudClient Initialization. Each CloudProxy application is started as a child process of TCService; the application sends a message to TCService through `tcioDD` requesting execution as a measured program. Upon receiving this message, TCService performs the measurement (and can perform any other checks it might need to verify the provenance or correctness of the application) and starts it as a child process. `tcioDD` inserts the requesting program's PID in the message and sends the message to TCService. TCService computes the cryptographic hash of the program (which must statically link all required libraries except those in `initramfs`), then forks, changes the UID of the forked child to that of the requesting program (which it can determine from the PID in the request) and executes the executable program in the child. TCService inserts PID, UID, and $\mu(hp)$ in a table it maintains and returns the PID of the child process through `tcioDD` to the requesting program. Subsequent requests from the newly-started, measured child will contain its PID; TCService can look up its measurement in its measured program table. This mechanism allows TrOS to implement the Tao primitives. And this allows TCService to ensure that only measured applications are running. See Figure 4 for a depiction of the interaction between TCService and components of CloudProxy.

Certificates for the CloudServer and CloudClient auth keys are signed by pK_0 in the online Tao Initialization protocol described in section 3.1.

1. CloudClient $\rightarrow$ CloudServer: TLS v1.2 channel establishment.

   All later messages (including the following steps) are exchanged on this channel with confidentiality and integrity protection. After the channel is established, CloudClient and CloudServer each check that the certificate used by its communication partner attests to a code identity that matches the identity of a known trusted component.

2. CloudServer $\rightarrow$ CloudClient: C

   where C is a fresh random nonce

3. CloudClient $\rightarrow$ CloudServer: $principalCertificate_1 \; || \; \ldots \; || \; principalCertificate_n \; || \; \langle C \rangle_{pK^{\mathsf{auth}}_{principal_1}} \; || \; \ldots \; || \; \langle C+n-1 \rangle_{pK^{\mathsf{auth}}_{principal_n}}$.

   where each $pK^{\mathsf{auth}}_{principal_i}$ is a key representing a security principal $principal_i$ to authorize. If all signatures pass verification, then this message concludes the challenge-response protocol.

Figure 5: CloudProxy Channel Establishment

4.2 CloudProxy Operation

After Tao Initialization for each Tao layer, the operational phase of CloudProxy begins. When the CloudServer activity element starts, it retrieves files containing its certificate, sealed symmetric and private keys and it unseals the keys using Tao Primitives provided by TrOS.

Next, CloudServer opens a TCP connection at a well-known port and awaits connection attempts from CloudClient instances. For each request, CloudServer and CloudClient negotiate a channel. This process is called *channel establishment*; it results in mutual authentication of each participant (i.e., CloudClient, and CloudServer, and additional security principals, like users) and shared encryption and integrity keys for the channel.

CloudProxy Channels. The CloudProxy channel establishment protocol is a specialization of TLS v1.2 [34] along with an extra challenge-response protocol to authenticate other principals on the channel. We employ TLS v1.2 with client authentication, and we only accept RSA signatures. Our implementation additionally accepts only certificates bound to TPM-measured and policy-authorized code. The specialized form of TLS employed by CloudProxy allows our implementation to collapse some TLS messages and operations into single steps for efficiency. Figure 5 presents the protocol.

Once the channel is established, CloudClient may request operations from CloudServer. When requesting a CloudProxy operation, CloudClient transmits claims to CloudServer by communicating over the secure channel they share. To make an access decision, CloudServer verifies these claims. CloudClient processes data from the operation, potentially protecting and storing intermediate results locally or sending results to CloudServer.

5 Applications

5.1 FileProxy

The goal of FileProxy is to protect confidentiality and integrity of files sent to a server by a client. The service also enforces access control restrictions on client reads and updates to files and meta-data. These properties are not provided directly by the security properties of CloudProxy; FileProxy must also (i) protect the binding between file names and files, and (ii) handle file-specific claims. Users of FileProxy get assurance in its security properties by examining the code, since the CloudProxy Tao allows users to be certain about the identity of code running on remote servers.

FileProxy consists of two components: *FileServer* implements the CloudServer role and exposes `read`, `write`, `create`, `delete`, `addowner`, and `removeowner` operations for files it stores. *FileClient* implements the CloudClient role and requests file operations on behalf of a user.

FileServer maintains metadata for each file it manages; all FileProxy metadata is stored in a single encrypted file with integrity protection. The metadata consists of a universal resource identifier (URI) to identify the file, a cryptographic hash of the file, and the public keys of the file owners. Storing the cryptographic hash of a file prevents an adversary with access to the disk from changing access control requirements for a file by changing the disk to change the name of the file in the filesystem by modifying the disk directly. A principal that creates a file or directory is implicitly granted the right to perform any operation on this file or directory. Note that `create` operations can only be performed by principals authorized by the activity owner policy for FileProxy.

Disks assigned to FileProxy can be read and written by adversaries, so FileServer encrypts its files by using the Tao Primitives along with keys it generates; the encryption scheme provides confidentiality and integrity protection. However, an adversary might remove, rewrite, and replace disks, so data retrieved from disk might not be the most recent copy of this data. Section 8 discusses ways to mitigate this attack.

Requests to FileServer contain the file name requested, the length (for file write operations), the access requested, and signed claims that chain up to a file owner and support the access request. If the request is supported by the provided claims, and file access is needed, then FileServer decrypts the file (using its keys for file decryption and integrity checks) and sends the file over its secure channel to FileClient.

FileServer responds to each request with an accept or reject indication, an error code, the file name, and file length. In the case of a read request, the file itself is transmitted after a read request. FileClient transmits a file after receiving the success indication for a write request.

Authorization. The authorization language employed in FileProxy is a simplification of the says/speaksfor formalism first proposed by Lampson et al. [27]. Security principals are associated with public/private key pairs, where the public key serves as a name for the principal. A principal requests an operation by signing a certificate that gives parameters of the request. Here is an example for a request by $PK^{\mathsf{auth}}_{Alice}$ to perform a `read` on a file $\mathscr{F}$:

$$\langle PK^{\mathsf{auth}}_{Alice}\ \textbf{says}\ \texttt{read}\ \mathscr{F}\rangle_{pK^{\mathsf{auth}}_{Alice}}. \tag{2}$$

To authorize a security principal (named by a public key) to perform an operation on a file, the file owner signs a claim granting that privilege. For example, suppose PK^{auth}_{Bob} is the public key for the owner of a file $\mathscr{F}$. Then claim

$$\langle PK^{\mathsf{auth}}_{Bob}\ \textbf{says}\ PK^{\mathsf{auth}}_{Alice}\ \textbf{may}\ \texttt{read}\ \mathscr{F}\rangle_{pK^{\mathsf{auth}}_{Bob}} \tag{3}$$

grants $PK^{\mathsf{auth}}_{Alice}$ the authority needed for `read` $\mathscr{F}$. Claim (3) would have to accompany request (2) for that `read` to be authorized.

Authorization is delegated in FileProxy by using claims involving the **maysay** connective. The claim

$$\langle PK^{\mathsf{auth}}_{Bob}\ \textbf{says}\ PK^{\mathsf{auth}}_{Alice}\ \textbf{maysay}\ *\ \textbf{may}\ \texttt{read}\ \mathscr{F}\rangle_{pK^{\mathsf{auth}}_{Bob}} \tag{4}$$

delegates to security principal $PK^{\mathsf{auth}}_{Alice}$ the authority to grant to other principals the authority to `read` file $\mathscr{F}$. A combination of (4) and

$$\langle PK^{\mathsf{auth}}_{Alice}\ \textbf{says}\ PK^{\mathsf{auth}}_{Charlie}\ \textbf{may}\ \texttt{read}\ \mathscr{F}\rangle_{pK^{\mathsf{auth}}_{Alice}} \tag{5}$$

authorizes $PK^{\mathsf{auth}}_{Charlie}$ to `read` $\mathscr{F}$ if PK^{auth}_{Bob} is the owner of $\mathscr{F}$. FileProxy accepts chains of delegations if the chain is rooted with a claim that is signed by the file's owner. Claim (4) followed by claim (5) is an example of such a chain.

Finally, bindings between public keys and other kinds of security principals are supported by claims written in terms of the **speaksfor** connective. Such claims must be signed by the FileProxy activity owner's key pK_0: for example, $\langle PK^{\mathsf{auth}}_{Alice}\ \textbf{speaksfor}\ \text{Alice}\rangle_{pK_0}$. In effect, these claims create for FileProxy a local space of names for security principals it serves.

5.2 AuthProxy

AuthProxy provides a service that translates CloudProxy claims into third-party credentials. For example, AuthProxy could transform a claim into a Kerberos [33] or an X.509-based [44] credential. The nature of the translation depends on statements in the claim. Our implementation currently only supports conversion to an RSA signature under a fixed key. As for FileProxy, clients get assurance in the correctness of the translation by examining the code, since CloudProxy provides evidence of code identity.

AuthProxy consists of two components: AuthServer (like FileServer) for responding to requests from clients, and AuthClient (like FileClient) to make requests. AuthServer implements the CloudServer role, and AuthClient implements the CloudClient role. In this case, however, AuthServer is stateless and produces and signs credentials for properly authorized client requests.

AuthServer exposes a single operation: GetCredential, which must be accompanied by claims that authorize this client to get a credential of the requested type. AuthServer thus converts authorization statements from our policy language into authorization tokens of other forms to be consumed by third-party services.

5.3 BidProxy

BidProxy is a distributed auction system. It manages offers from an authorized seller and determines the winner of an auction among authorized buyers. The seller sets the duration of the auction. BidProxy provides an RSA signature over the winner's identity so the winner can complete the transaction with the seller.

BidProxy has one security property in addition to the properties provided by CloudProxy: no information about any bidder or bid is disclosed except for revealing the identity of the winner and value of the winning bid to the seller. Note that, as for FileProxy, the validity of the auction procedure can be determined by examination of the source code, since CloudProxy attests to the identity of running code.

BidProxy has three components.

- BidClient: BidClient is a CloudClient application used to submit bids. The buyer provides BidClient with its public/private key pair, and BidClient submits authenticated requests signed by the buyer.
- SellerClient: SellerClient works like BidClient but for the seller instead of the buyer. It reads the bids from BidServer and computes the winner.
- BidServer: This CloudServer component collects bids, noting the bid price, time of receipt, and public key of the registered BidClient.

6 Performance Evaluation

We implemented CloudProxy in C++ from scratch, including all symmetric and asymmetric cryptographic primitives. The full implementation consists of about 65k LOC for all applications, along with KeyServer, the Tao host implementation in Linux, a device driver to implement the lower-level Tao and TPM access, and all test code. For DRTM boot, we use `tboot` version 1.7.0 and the Intel SINIT module `3rd_gen_i5_i7_SINIT_51.BIN`.

All data points reported are the median of at least 10 executions. Error bars represent the median absolute deviation (Δ) over the points, which is computed as the median of the absolute value of the difference between each data point and the median data point; symbolically, for a set X, we write $\Delta(X) = \text{median}_{x \in X}(|x - \text{median}(X)|)$. We use Δ because it is a more robust variance measure than standard deviation in the face of outliers.

We executed our tests on two Lenovo T430s laptops each with Intel Core i5-3320M 2.60 GHz processors; all tests ran single-threaded on the processors. These processors have lower performance than processors used by normal data center computers, so these results should be seen as providing a lower bound on the performance. We used Linux kernel version 3.7.5 with a custom small `initramfs` for our tests. The network card for each laptop was an Intel PRO/1000 with a crossover connection between the two laptops.

Our implementation uses 128-bit AES-CBC for bulk and channel encryption with HMAC-SHA256 for integrity. Our tests currently do not use `dm-crypt` to encrypt swap on the client and server, since all our code for tests runs in memory on `initramfs` and does not swap.

6.1 Microbenchmarks

We performed microbenchmark experiments over FileProxy components, and we see similar results for AuthProxy and BidProxy.

DRTM Boot. To measure the time added by the DRTM boot of Linux, we measured wall-clock time between the moment an entry was selected in `grub` to the moment when the initial screen appeared during the boot process. This encompasses execution time of the Linux kernel, the `tboot` code, and the Intel `SINIT` component started by `tboot`.

In our measurements, a normal boot took a median of 8.91 s, with Δ 0.06 s, and enabling `tboot` led to an median boot time of 15.79 s with Δ 0.14 s. So, DRTM boot with `tboot` and `SINIT` adds about 7 s to the boot time.

This extra cost occurs only at boot time and has no additional effect on execution time of binaries running on the system.

Retrieving keys. Sealing and unsealing for applications (performed by TrOS using K_{TrOS}^{seal}) runs at native CPU speed. For example, in a simple test that creates a resource on the server, client key seal time had a median of 7 μs with Δ 0 μs, while client key unseal time had a median of 4 μs with Δ 0 μs.

Initialization. Initialization for TCService, FileServer, and FileClient is dominated by normal TCP/IP latency, which can easily be 100 ms for each KeyServer message sent using a standard broadband provider. By contrast, KeyServer running inside a data center can have latency delays that are shorter by a factor of 25 or more. 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. In our tests, with KeyServer running on localhost, median Tao environment initialization for the first startup of a program was 465 μs with Δ 52 μs.

Channel Establishment. The performance of FileProxy on channel establishment depends on our unoptimized cryptographic code, so it is slower than TLS. In our tests, we had a median connection establishment time of 144.777 ms with Δ 0.253 ms.

Authentication and authorization. Authentication and authorization processing time depends on the length and complexity of the chain of claims presented by FileClient to demonstrate access rights. The simple policy used in our tests contains only a single claim, and we measured a median access check time of 674 μs with Δ 4 μs. The cost of the access check is dominated by RSA signature verification, and it rises linearly in the number of signed assertions.

Encryption operations. File and channel encryption performance is similar to generally reported performance for symmetric key cryptographic operations.

Server decryption for a set of bulk file transfers between client and server using AES-NI operated at a median of 1004 Mbps with Δ 2.04 Mbps for 1 GB files, and encryption operated at a median of 916.877 Mbps with Δ 2.84 Mbps for the same files. These microbenchmarks show that encryption and decryption performance using AES-NI supports high network throughput, as can be seen in the macrobenchmark. The encryption benchmark includes the time needed to write data to `initramfs`, and decryption includes the time to read data from `initramfs`.

6.2 Macrobenchmarks

6.2.1 FileProxy Macrobenchmark

To quantify overall performance for FileProxy, we ran a sequence of tests that show the incremental cost of adding levels of protection to a simple application that transfers files between a client and a server. Encryption and integrity protection for files is a common task and can be implemented using existing libraries, like OpenSSL [29]. To determine the costs associated with FileProxy, we wrote a program, called `ossl`, that uses OpenSSL 1.0.1 [29] and demonstrates the costs of adding levels of protection to file transfer and storage, starting with completely unprotected communication and storage and ending with the same cryptographic protections as FileProxy. We compare this final version to FileProxy to investigate the overhead of our additional algorithms.

The `ossl` server performs one of four possible tasks:

1. (None) The server listens on a port for TCP connection from a client. It takes the bytes received on this connection and writes them to standard output; this output is piped to a file.

2. (TLS) The server listens on a port for a TLS v1.2 connection with client authentication. It then writes to standard output the bytes received on this connection; as in None, the output is piped to a file. This shows the additional cost of protecting the communication channel.

3. (Enc) The server does the same as in TLS and additionally encrypts the bytes using 128-bit AES-CBC and writes the encrypted bytes to standard output. This shows the additional cost of storage encryption.

4. (Full) The server does the same as in Enc and additionally performs an HMAC-SHA256 computation over the encrypted bytes. The server outputs the HMAC value at the end of the encrypted bytes. This shows the additional cost of adding storage integrity protection.

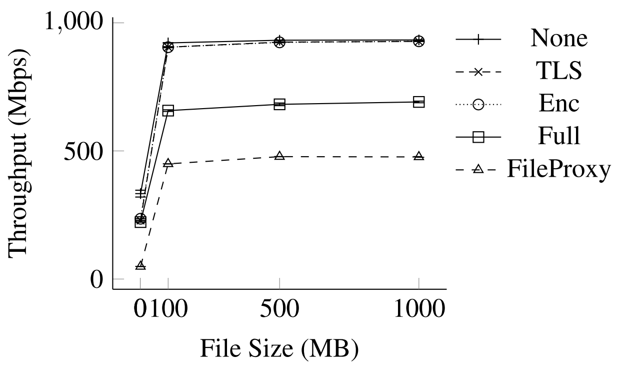

Figure 6: Throughput of implementations of file transfer

Compression is disabled on the TLS connection, and both client and server use self-signed 1024-bit RSA certificates for authentication; client and server each use a certificate file containing their peer's certificate as a trust database. The entire `ossl` application is 357 lines of code, not counting comments or blank lines.

Note that our FileProxy application not only transmits files over a protected channel and encrypts them on the server with integrity protection but also uses TPM-protected keys and requires both client and server to check program identity as authenticated by DRTM boot and TPM-based attestation on the protected channel.

Figure 6 shows the results for a range of file sizes. None, TLS, and Enc saturate the 1 Gbps link between the two laptops. This shows that OpenSSL provides high-performance network and cryptographic primitives. Adding HMAC-SHA256 lowers the performance to about 700 Mbps. And FileProxy reaches about 470 Mbps—less than the equivalent `ossl` implementation.

FileProxy only achieves about 2/3 the speed of the comparable protocol in `ossl`. However, most of the additional cost in FileProxy is caused by a difference in cryptographic implementation. The `ossl` application uses the highly tuned OpenSSL library, while FileProxy uses our research prototype implementation of channel and cryptographic primitives. We believe that FileProxy would have the same performance as `ossl` Full if we used OpenSSL, since the additional costs in FileProxy are one-time setup costs either for the programs or the channel and can be easily amortized over the transmission of a 1 GB file. We did not use OpenSSL because one of the goals of this project was to investigate the costs of building a full application from scratch. See section 8 for discussion of this goal and the results.

6.2.2 AuthProxy Macrobenchmark

We implemented the AuthProxy protocol and executed tests in which AuthClient connects to AuthServer and requests a new credential. In our tests, the median time for a client to contact a server and get a fresh credential is 707.883 ms with Δ 66.601 ms.

6.2.3 BidProxy Macrobenchmark

For BidProxy, we implemented the BidClient, BidServer, and SellerClient and ran tests in which varying numbers of BidClients connect to a BidServer and bid in an auction. SellerClient actions are performed offline later and are not measured in these benchmarks, though they are similarly efficient. The cost of a single bid submission is median 212 ms with Δ 3.74 ms. As expected, a linear increase in the number of clients leads to a linear increase in the time needed to complete bidding.

7 Related Work

The Trusted Computing Agenda. The CloudProxy Tao derives from ideas originally discussed [3] for trusted boot as well as ideas for implementing secure distributed systems [18]. However, it adds a novel key management infrastructure that allows for stateless application distribution and does not require cumbersome and vulnerable separate state and key distribution components.

The trusted boot concept has since been expanded into the so-called *trusted computing* agenda, premised on the availability of a TPM or other secure co-processor that controls measurement-based security principals' access and use of cryptographic keys. Parno, McCune, and Perrig [32] survey problems and solutions in the area. More recent work [12, 46, 36] begins to explore the use of trusted computing in cloud data centers.

Self-Service Cloud (SSC) Computing [12] implements a cloud computing service over the Xen [7] hypervisor. Security-critical functionality is separated into (i) a system-level domain that performs high-level tasks for the machine as a whole and (ii) a per-VM administrative domain that performs tasks specific to the VM (or collection of VMs). The system-level domain cannot access the state of the VMs or interfere with their operation, and operations performed in the VM cannot interfere with the state or execution of other VMs. SSC provisions client VM keys in the per-VM domain during domain establishment. This domain separation for virtual machines could be applied to a version of CloudProxy that has been extended to operate over virtual machines (see section 8). However, SSC does not provide code identity assurance like the Tao.

Several systems provide trusted hardware that could be used to implement the lowest level of Tao Primitives. These systems offer flexibility to Tao adopters, since they provide other ways to satisfy the Trusted Hardware requirement of CloudProxy. AEGIS [41] is a secure single-chip processor that allows third parties to perform remote attestation of programs and protects the integrity of code executing on the processor. Similarly, CARMA [43] implements trusted hardware, even in the face of tampering with values transferred along hardware buses or memory. And DataSafe [13] is an architecture with hardware-supported protection of data. It protects data with TPM-based encryption and provides a hardware implementation that prevents data leakage.

Flicker [28] provides a different view of trusted computing. With Flicker, clients switch briefly into a trusted computing mode to execute code, sometimes as little as a couple of hundred lines, which is well within the province of formal verification (though correct functioning still depends on correctness of all the underlying hardware mechanisms and the trustworthiness of the root of trust). Flicker could be an efficient way to realize activity elements or KeyServer in CloudProxy.

Other work implements trusted computing on mobile devices. For example, On-board Credentials (ObCs) [16] is an open architecture for secure credentials; the ObCs implementation executes on a mobile phone using M-shield [39]. ObCs employ a method of provisioning keys that has similarities to our KeyServer protocol: a remote server certifies keys to be sent to a device. However, ObCs focuses on single problem domain rather than providing a general framework like the Tao.

Multi-Party Computation. An alternative to having the processors and their software in the TCB is to use a protocol that allows servers to perform joint computations with cryptographic security guarantees. Secure Function Evaluation (SFE) or the more general Multi-Party Computation (MPC) [21] allows a collection of hosts to compute jointly the output of a function without any individual host ever learning more than its own input and the resulting output. 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.

For example, Salus [26] 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.

Fully-homomorphic encryption [19, 11, 20] provides another way to realize secure computation on remote servers. Here, arbitrary computations are performed on encrypted data. Such functionality would completely obviate the need for CloudProxy or any of the other remote-execution schemes discussed above. Unfortunately, the cost of fully homomorphic encryption at present is prohibitive.

8 Next Steps

Past research has focused on circumscribed use of Tao-like primitives. For example, McCune et al. [28] focus on portions of applications encapsulated in non-interruptible functions (like a virtual HSM). This can require extensive application redesign, and does not protect most of the computation. Most other trusted computing research has focused on a single layer of the software stack (like the hypervisor), and not the full system.

In contrast, the recursive design of the CloudProxy Tao can be applied to all layers and provides a simple programming model as well as a clear attack model (data center insiders and tenants). Our work differs from previous systems in being designed for secure distributed computing, including automatic key management (using KeyServer). Alternative key distribution mechanisms use pairwise agreement between activity elements, which imposes an impractical burden on each element.

The CloudProxy Tao does not appear to have limitations that would prevent a full-scale, production deployment. Our CloudProxy prototype, however, does. We are thus exploring extensions to overcoming the limitations we know about and we are engaged in analysis to convince ourselves there are no others. Some of the problems being addressed and solutions we are pursuing are outlined below.

CloudProxy in a Virtual Machine. The CloudProxy prototype assumes it is the only application running in the only operating system on a physical computer. Yet cloud data centers providing IaaS use hypervisors to time-multiplex computers among tenants. Incorporating a hypervisor into CloudProxy looks to be a straightforward task—the hypervisor serves as another Tao layer in the recursive stack. Compromised performance and security are the primary concerns.

Early hypervisors [30] enjoyed surprisingly good performance and security [6]. This success was partly due to device virtualization (e.g., I/O processors on IBM mainframes). The ability to keep an ever growing set of device drivers out of the TCB remains critical, and modern processors provide device virtualization and isolation using IOMMUs [1]. So, hypervisors intended for securely hosting virtual machines provided by different tenants generally are able to provide strong isolation between guests and between guests and the hypervisor.

Ideally, a hypervisor would be small (so it can be carefully analyzed) and it would change rarely (so any analysis can long-lived). It would not not permit loadable modules, and it likely would export complicated functionality, like I/O, to guest partitions, with help from supporting hardware. These requirements—and the large TCB—rule out Xen [7]. But the Nova [40] hypervisor is small, provides excellent performance, and implements strong isolation properties.

We are thus now engaged in re-hosting CloudProxy on a Nova-like hypervisor. This requires making a few changes to that hypervisor. First, the hypervisor must be booted with DRTM using the same code and methods used to boot TrOS. And the hypervisor itself must perform a measured boot of its guests; this involves code similar to that in TCService. Second, the hypervisor must provide a special partition that implements Tao Primitives for its guest partitions; this also uses substantially the same code as in TCService. Third and finally, the hypervisor must provide hypercall interfaces for Tao Services and ensure the same isolation properties that TrOS does. This involves zeroing guest memory on reassignment and equivalent support for swapping or paging.

In the resulting system, TrHW would boot and measure a small, well-understood *Trusted Hypervisor*, which hosts TrOS. As before, TrOS runs an activity element. Since TCService requires little OS support, the TCService partition can be implemented as an encapsulated partition with a library OS, thereby vastly reducing TCB size.

Costs for Cryptography. The CloudProxy prototype employs a cryptography library that we built from scratch for this project. This library is not yet as efficient as OpenSSL, and it provides fewer operations. One direction for additional work would be to integrate CloudProxy with OpenSSL for higher speed cryptography. Another would be to continue optimizing our crypto library to match the performance of OpenSSL.

Despite the lower performance of our cryptographic library, we believe that CloudProxy demonstrates the potential for efficient and practical Trusted Computing in the cloud; even with a prototype library, FileProxy was still able to match 2/3 the performance of the industry standard library.

Defending Against Storage Replay. CloudProxy currently does not defend against storage replay attacks. These attacks could be mitigated using proofs of storage [5, 4, 25, 38], which provide a way for a remote server to verifiably store data for a client and allows the client to make probabilistic checks that the data is being stored correctly. A proof of storage scheme could be used in conjunction with CloudProxy to attest to freshness of data provided by a trusted host. Another alternative would be to employ a system like Memoir [31], which protects state from rollback using techniques based on TPM NVRAM.

System Upgrade. Measurement-based security principals can become problematic when software needs to be upgraded. The new version of a hosted program (which does not have the same cryptographic hash as earlier versions) is no longer able to access data encrypted using keys available to its predecessor. We are currently experimenting with four mechanisms to perform such upgrades. This shows that the problem is far from intractable.

- File encryption keys might be shared among servers by treating these keys as access-controlled objects (just like files). So, a new version of FileServer could request encryption keys for existing encrypted files and reseal those encryption keys with its new sealing key.
- KeyServer could be extended to manage file encryption keys and distribute them during the key distribution protocol.
- For all layers except the one authenticated directly by hardware, the measurement value for a hosted program might be implemented as the hash of some meta-data naming a public key rather than the hash of an executable. Then this public key would sign a measurement value to bind the key to each new hosted program.
- The old version of the program could unseal and then encrypt (with the public key of the new version) all of the keys that need to be transferred.

9 Contributions and design choices

The principal contribution of this work is a complete system that provides the security properties set forth in Figure 1 in a data center operated by a third party. Especially noteworthy is the ability of CloudProxy to protect not only from co-tenants but from insiders, including system administrators. This protection is not now afforded by any system in a comparable setting. Additionally, CloudProxy allows programmers to develop activities in a familiar programming environment (e.g., a Linux application). Prior Trusted Computing solutions work in more limited, unfamiliar and poorly supported environments and sometimes require significant application or system refactoring. These limitations may help explain the limited adoption of Trusted Computing technology. CloudProxy demonstrated a complete and flexible off-the-shelf solutiona adaptable to a wide range of applications.

The demands of the Cloud Computing model motivate some of our design choices. We have made an effort to explain the implementation details critical for ensuring the cloud protection model; for example, the encryption of swap, isolation of applications, and verification of initial program data in the Trusted OS. Without this, the Cloud protection model is not achieved and the system becomes closer to existing Trusted Computing solutions.

We also emphasized building the entire system from scratch. This is important because it facilitates verification and prevents vulnerabilities associated with large support libraries which conceal assumptions or configuration dependencies and greatly increase the effort a developer must expend to verify the secure use of the large library components. Also, large libraries cause image bloat. CloudProxy systems favor completely independent and frugal images to improve deployability in data centers as well as security. As an example, OpenSSL, which is a well-written cryptographic package, has been criticized by N. Heninger and her colleagues who expressed a belief "that not even the developers of OpenSSL understand it." One person, who reviewed an earlier draft of this paper, suggested using jTPM, a large library with TPM support. Developers of this library warn "Please keep in mind that IAIK XKMS is experimental software. The use of IAIK XKMS in a real production environment is discouraged. No guarantees for data produced by IAIK XKMS can be made. Use the software at your own risk!" This forthright statement is understandable for a large library. We went to great lengths to ensure that no unintended consequences arose from the use of such libraries.

We have significantly improved the performance of the crypto implementation since these benchmarks were recorded but even as reported, the benchmarks demonstrate that the entire infrastructure imposes essentially negligible overhead on application performance.[12]

There are important contributions that are not present in other systems (real or proposed) including, for example, [10] and [18].

vTPM [10] describes virtualizing TPM support for OSes hosted by a hypervisor. While a physical TPM is one (of several) critical hardware mechanisms that can ground the Tao, it is not the only one and the Tao focuses on a much simpler, and we believe, more flexible set of primitives for programmers *at every level of a software stack*. vTPM [10] does not propose a workable key management solution for distributed applications. Simply doing pairwise attestation requires programmers (in this limited case, hypervisor users) to record and maintain a comprehensive list of all trusted programs, which is not scalable. By contrast, KeyServer frees individual components of this responsibility, which makes update and deployment much simpler. KeyServer also can perform key rollover as applications change.

[12]The benchmarks reported were designed to provide upper bounds on performance degradation. The Tao only affects performance related to initialization and shutdown, secure channel establishment, and I/O (and even then only slightly); the Tao has no performance impact on other application computation.

While cryptographic hash as code identity is needed for the system software booted by the hardware, we pointed out that higher levels of the Tao can use more flexible notions of code identity that simplify update. Strict adherence to the TPM model (including PCRs) for measurement, sealing and attestation prevents this. In addition, most TPM functionality (including legacy support) is not needed by the Tao and need never be learned (or misunderstood) by CloudProxy developers. Thus the programming model for [10] which provides less capability than the Tao is more complicated than the Tao. Finally, [10] is silent on anchoring policy management and policy enforcement. A simple policy anchor is a critical feature and is particularly important for cloud distribution where simplicity is paramount. It also obviates the need for trust infrastructure outside the control of the activity owner; this has been a critical problem for browsers in light of the Diginotar and Comodo attacks. With CloudProxy, distributing the application image is all that is required for fail-safe operation.

By contrast, [18] focuses on the distributed application problem, including elements common to our system like an integrity-protected, encrypted channel between distributed program elements with authenticated principals that the channel speaks for. However, the protection model in [18] excludes protecting the operating system ("protecting the operating system is outside the scope" of the paper). Thus the ability demonstrated by CloudProxy to remotely verify measurement, key management and isolation lies outside the scope of this important paper. The paper suggests DNS as the underlying authentication model for the endpoint (probably the only reasonable one available at the time) but this does not address the stated protection goal of CloudProxy to avoid dependence on external infrastructure or data center insiders. As a practical matter, not only do insiders represent a threat because of intentional misconfiguration, but in a cloud setting, unintentional misconfiguration, unsafe image management and unprotected data storage are equally important threat vectors for distributed programs hosted in clouds. Still, this paper is a direct inspiration for the current work.

10 Conclusions

CloudProxy has several novel aspects. First, we believe that CloudProxy is the first system to provide distributed trusted computing based on secure hardware elements like TPMs. Second, we provide a key management solution for trusted components that requires no manual key generation or intervention by data center personnel. Third, we provide new recursive design for distributed trusted computing. Fourth and finally, we provide a complete implementation that can be used to build applications.

The source code for CloudProxy will be available under an open source licence.

Acknowledgements. We are grateful to Eric Grosse for his encouragement and support for this project. We are also grateful to the Defense Science Board study committee on "Cyber Security and Reliability in a Digital Cloud" for providing Manferdelli and Schneider with an impetus to pursue the problems CloudProxy solves.

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] AMD Virtualization. `http://sites.amd.com/us/business/it-solutions/virtualization/Pages/virtualization.aspx`.

[3] W. A. Arbaugh, D. J. Farber, and J. M. Smith. A secure and reliable bootstrap architecture. In *1997 IEEE Symposium on Security and Privacy*, pages 65–71. IEEE, 1997.

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

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

[6] 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.

[7] 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 *Nineteenth ACM Symposium on Operating Systems Principles*, pages 164–177. ACM, 2003.

[8] 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.

[9] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour. The turtles project: Design and implementation of nested virtualization. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2010)*, pages 423–436. USENIX Association, 2010.

[10] S. Berger, R. Cáceres, K. A. Goldman, R. Perez, R. Sailer, and L. van Doorn. vTPM: Virtualizing the Trusted Platform Module. In *15th USENIX Security Symposium*, pages 305–320. USENIX Association, 2006.

[11] Z. Brakerski and V. Vaikuntanathan. Efficient fully homomorphic encryption from (standard) lwe. In *IEEE 52nd Annual Symposium on Foundations of Computer Science (FOCS 2011)*, pages 97–106. IEEE, 2011.

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

[13] Y.-Y. Chen, P. A. Jamkhedkar, and R. B. Lee. A software-hardware architecture for self-protecting data. In *2012 ACM conference on Computer and communications security (CCS '12)*, pages 14–27. ACM, 2012.

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

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

[16] J.-E. Ekberg, N. Asokan, K. Kostiainen, and A. Rantala. On-board credentials with open provisioning. Technical Report NRC-TR-2008-007, Nokia Research Center, 2008.

[17] B. Ford, M. Hibler, J. Lepreau, P. Tullmann, G. Back, and S. Clawson. Microkernels meet recursive virtual machines. In K. Petersen and W. Zwaenepoel, editors, *Second USENIX Symposium on Operating Systems Design and Implementation (OSDI '96)*, pages 137–151, 1996.

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

[19] C. Gentry. Fully homomorphic encryption using ideal lattices. In M. Mitzenmacher, editor, *41st Annual ACM Symposium on Theory of Computing (STOC 2009)*, pages 169–178. ACM, 2009.

[20] C. Gentry, S. Halevi, and N. P. Smart. Homomorphic evaluation of the aes circuit. In *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference*, pages 850–867. Springer, 2012.

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

[22] 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 boot attacks on encryption keys. In *17th USENIX Security Symposium*, pages 45–60. USENIX Association, 2008.

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

[24] Intel Corporation. Intel Trusted Execution Technology, Measured Launched Environment Developers Guide, March 2011. `http://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/intel-trusted-execution-technology-software-dev-guide.pdf`.

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

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

[27] 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.

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

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

[30] 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.

[31] B. Parno, J. R. Lorch, J. R. Douceur, J. Mickens, and J. M. McCune. Memoir: Practical state continuity for protected modules. In *32nd IEEE Symposium on Security and Privacy*, pages 379–394. IEEE, 2011.

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

[33] RFC 4120: The Kerberos Network Authentication Service. `https://tools.ietf.org/html/rfc4120`.

[34] RFC 5246: The Transport Layer Security (TLS) Protocol Version 1.2. `https://tools.ietf.org/html/rfc5246`.

[35] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and implementation of a tcg-based integrity measurement architecture. In *13th USENIX Security Symposium*, pages 223–238. USENIX Association, 2004.

[36] N. Santos, R. Rodrigues, K. P. Gummadi, and S. Saroiu. Policy-sealed data: A new abstraction for building trusted cloud services. In *21st USENIX conference on Security symposium*, Security'12, pages 10–10, Berkeley, CA, USA, 2012. USENIX Association.

[37] 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.

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

[39] J. Srage and J. Azema. M-Shield mobile security technology. TI white paper. `http://focus.ti.com/pdfs/wtbu/ti_mshield_whitepaper.pdf`, 2005.

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

[41] G. E. Suh, D. E. Clarke, B. Gassend, M. van Dijk, and S. Devadas. AEGIS: Architecture for tamper-evident and tamper-resistant processing. In *17th Annual International Conference on Supercomputing (ICS 2003)*, pages 160–171. ACM, 2003.

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

[43] 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 *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*. ACM, 2012.

[44] Recommendation X.509. `http://www.itu.int/rec/T-REC-X.509/en`.

[45] XML Signature Syntax and Processing (Second Edition). `http://www.w3.org/TR/xmldsig-core`.

[46] F. Zhang, J. Chen, H. Chen, and B. Zang. CloudVisor: Retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 203–216, New York, NY, USA, 2011. ACM.