cTPM: A Cloud TPM for Cross-Device Trusted Applications

Chen Chen, *Carnegie Mellon University;* Himanshu Raj, Stefan Saroiu, and Alec Wolman, *Microsoft Research*

https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/chen

This paper is included in the Proceedings of the
11th USENIX Symposium on Networked Systems
Design and Implementation (NSDI ’14).

April 2–4, 2014 • Seattle, WA, USA

ISBN 978-1-931971-09-6

cTPM: A Cloud TPM for Cross-Device Trusted Applications

Chen Chen[†], Himanshu Raj, Stefan Saroiu, and Alec Wolman
Microsoft Research and [†]CMU

Abstract:

Current Trusted Platform Modules (TPMs) are ill-suited for cross-device scenarios in trusted mobile applications because they hinder the seamless sharing of data across multiple devices. This paper presents cTPM, an extension of the TPM's design that adds an additional root key to the TPM and shares that root key with the cloud. As a result, the cloud can create and share TPM-protected keys and data across multiple devices owned by one user. Further, the additional key lets the cTPM allocate cloud-backed remote storage so that each TPM can benefit from a trusted real-time clock and high-performance, non-volatile storage.

This paper shows that cTPM is practical, versatile, and easily applicable to trusted mobile applications. Our simple change to the TPM specification is viable because its fundamental concepts – a primary root key and off-chip, NV storage – are already found in the current specification, TPM 2.0. By avoiding a clean-slate redesign, we sidestep the difficult challenge of re-verifying the security properties of a new TPM design. We demonstrate cTPM's versatility with two case studies: extending Pasture with additional functionality, and re-implementing TrInc without the need for extra hardware.

1 Introduction

People are increasingly relying on more than one mobile device. Recent news reports estimate that: the average US consumer owns 1.57 mobile devices [8]; Singapore has 7.8 million mobile devices, which translates to 150% mobile penetration [36]; and the average Australian will own five mobile devices by 2040 [37]. Given this trend, mobile platforms are recognizing the need for "cross-device" functionality that automatically synchronizes photos, videos, apps, data, and even games across all devices owned by a single user.

Simultaneously, laptops, smartphones, and tablets are increasingly incorporating trusted computing hardware. For example, Google's Chromebooks use TPMs to prevent firmware rollbacks and to store and attest user's data encryption keys [11]. Windows 8 (on tablets and phones) offers BitLocker full-disk encryption [21] and virtual smart cards [23] using TPMs. Recent research leverages TPMs to build new trusted mobile services [30, 32, 9, 17, 14], new trusted cloud services [31], and new operating systems [33].

Unfortunately, these two trends may be at odds: trusted hardware, such as the trusted platform module (TPM), does not provide good support for cross-device functionality. By design, TPMs offer a hardware root-of-trust bound to a single, standalone device. TPMs come equipped with encryption keys whose private parts never leave the TPM hardware chip, reducing the possibility those keys may be compromised. The tension between single-device TPM guarantees and the need for cross-device sharing makes it difficult for trusted applications to cope with multi-device scenarios. For example, Pasture [14], a TPM-based secure offline data access system that can be used for movie rentals, limits all its guarantees to one device due to TPM limitations. Similarly, Windows TPM-based virtual smart cards are single-device only – users have to provision and renew their credentials separately on each device they own.

Support for cross-device sharing requires altering the TPM design, which raises the following question: Can a small-scale TPM design change overcome these limitations? While a clean-slate TPM re-design could provide a variety of additional security properties, there are two pragmatic reasons why a smaller change is preferable. First, TPMs have undergone a decade of API and implementation revisions to reduce the likelihood of vulnerabilities. A clean-slate re-design would demand considerable time and effort to provide a mature codebase. Second, TPM manufacturers would more willingly adopt smaller and simpler changes.

This paper proposes a single, simple design change to the TPM, called cTPM, that overcomes the limitations noted above by equipping the TPM with one additional root key that is shared with the cloud. This key lets trusted applications overcome their cross-device limitations with the cloud's assistance. It ensures that the cloud can control only a portion of TPM resources: those encrypted with the shared key. The cloud remains restricted from accessing the TPM resources protected by all other device-local, TPM root keys. We verified the security of the communication protocol between the TPM and the cloud using a protocol verifier [3].

The new key also lets cTPM allocate non-volatile (NV) storage in the cloud. The cTPM's remote storage enables the cloud to provide a trusted, synchronized and highly accurate source of time by periodically recording the time to remote storage. Today's TPMs lack a trusted source of time (i.e., a trusted real-time clock). Although the TPM provides an internal trusted timer, this timer

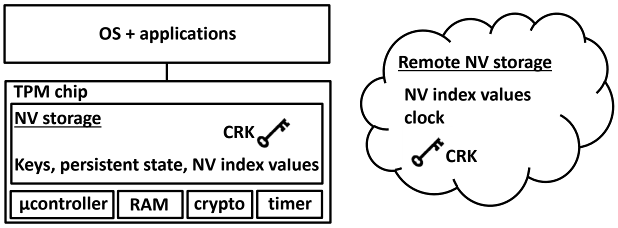

Figure 1. **Diagram of cTPM architecture.**

alone is insufficient to build a trusted real-time clock. Finally, the cTPM's remote cloud storage offers TPM applications large amounts of NV storage and lets them perform frequent NV writes. In contrast, TPM chips cannot offer such resources because they suffer from serious resource and performance limitations. These limitations drastically reduce the use cases for TPMs in both mobile and server scenarios, and have led researchers investigate alternatives to TPMs such as trusted devices whose storage offers increased performance [16]. Figure 1 shows a diagram of cTPM's architecture.

We demonstrate the benefits of cTPM by presenting two case studies. First, the cloud's ability to manage a portion of the TPM's state provides Pasture [14] with additional functionality. With cTPM, Pasture can extend its guarantees across all devices owned by a single user and can support server-side revocation, an operation not offered by the original Pasture protocol. Further, cTPM's trusted clock enables Pasture to grant data access at a specific time in the future (e.g., *make this movie available on Friday at midnight*). Second, the high-performance nature of cTPM's remote storage improves the performance of applications that require frequent writes. We re-implement TrInc [16] without the need of extra hardware (TrInc requires a smartcard).

2 Background

TPM Primer. At manufacturing time, TPM chips are provisioned with a couple of public/private key-pairs for cryptography (i.e., digital signatures and asymmetric encryption). The TPM design guarantees that the private keys of these *root* key-pairs never leave the TPM, thereby reducing the possibility of compromise. TPMs can also generate public/private key-pairs with private keys stored in the TPM's NV storage. However, TPMs have limited NV storage and thus cannot store many such key-pairs.

TPMs are equipped with a set of platform configuration registers (PCRs) guaranteed to be reset upon a computer reboot. PCRs are primarily used to store fingerprints of a portion of the software booting on a computer (e.g., the BIOS, firmware, and OS bootloader); Chromebooks [11] and BitLocker [21] use PCRs in this way.

TPMs can perform cryptographic algorithms for encrypting, authenticating, and attesting data. Implementing functionality beyond that offered by TPMs in a trustworthy manner can be done using secure execution mode, a form of hardware protection offered by x86 CPUs. Intel's secure execution architecture, called Trusted Execution Technology (TXT), offers a runtime environment strongly isolated from other software running on the computer.

The TPM spec does not provide minimum performance requirements, and, as a result, today's commodity TPMs are slow and inefficient [19, 14]. TPM vendors have little incentive to use faster but more expensive internal parts when building their TPM chips. This performance handicap has limited the use of TPMs to scenarios that do not require fast or frequent operations. However, no technological constraints prevents a hardware vendor from building a high-performance TPM.

Describing the full functionality of a TPM is beyond the scope of this paper. Ryan [29] and Challener et al. [5] provide good overviews of how TPMs work, although the TPM specs [39] remain the authoritative source for a full description of TPM functionality.

TPM 2.0. The Trusted Computing Group (TCG) is currently defining the specification for TPM version 2.0 [38], the next version of the TPM. TPM 2.0 offers several improvements, including a more complete set of cryptographic algorithms, i.e., SHA-2 and elliptic curve cryptography (ECC) in addition to SHA-1 and RSA offered by TPM 1.2. TPM 2.0 also provides more PCRs and supports more flexible authorization policies that control access to TPM-protected data. Finally, TPM 2.0 provides a reference implementation, while TPM 1.2 provides only an open-source implementation developed by a third party [10]. A complete list of differences between the two versions is provided by the TCG [38].

In TPM 2.0, three entities can control the TPM's resources: the platform manufacturer, the owner, and the privacy administrator. The TPM 2.0 spec *control domain* refers to the specific resources that each entity controls. The platform firmware control domain overseen by the platform manufacturer updates the TPM firmware as needed. The owner control domain protects keys and data on behalf of users and applications. The privacy administrator control domain safeguards privacy-sensitive TPM data. This role can be played by anyone; for example, in an enterprise the IT department acts as the privacy administrator for all its machines' TPMs.

Each TPM 2.0 control domain has a primary seed, which is a large, random value permanently stored in the TPM. Primary seeds are used to generate symmetric/asymmetric keys and proofs for each control domain.

3 Motivation

This section first describes how the additional TPM functionality can be implemented at present and why this approach is problematic. We then discuss specific feature limitations of existing TPMs for cross-device sharing, trusted clock, and NV storage. Finally, we describe how cTPM addresses each limitation.

3.1 Secure Execution Mode Limitations

Extending the TPM functionality can be done at present by leveraging its extensibility mechanism, which is a *secure execution mode* integrated into the system CPU. Both Intel's TXT and AMD's SEM are extensibility mechanisms for the TPM – they enable the development of trusted computing features not easily achieved solely through the built-in TPM commands. Unfortunately, major stumbling blocks prevent a secure execution mode from providing needed features. The first stumbling block is performance; to use the secure execution mode, CPU interrupts must be disabled, and, in a multiprocessor system, only one CPU core can be enabled. Entering this mode requires the OS to save its state and suspend execution, operations that are relatively heavyweight. Second, none of today's smartphones and tablets includes a CPU that supports TXT or SEM; these features are provided only on laptops and desktops sold to enterprises. Third, even if they did, using secure execution mode is remarkably difficult. It requires support from the motherboard chipset, BIOS , and, in the case of Intel's TXT, an additional chipset-specific authenticated code module. Also, once in secure execution mode, the code only has access to a "barebones" machine without any I/O, OS, or library support. Building such support without relying on interrupts may be challenging. We know of no production software that uses secure execution mode[1].

Section 6 will describe how cTPM solves this problem in a simpler way that does not require the use of secure execution mode. Despite cTPM's benefits, however, changing the TPM design raises a legitimate concern: Does the verification cost of introducing TPM changes outweigh the additional benefits of the new design? Although we cannot provide a reliable estimate of these costs, we deliberately kept our design changes minimal. The cTPM design affects only the TPM commands that access NV data (to indicate whether operations are local or remote) and adds three TPM commands that synchronize data between the device TPM and the cloud TPM. The vast majority of the TPM logic remains the same.

3.2 Limitation 1: Cross-Device Data Sharing

Current TPM abstractions offer guarantees about one single computer, and the TPM's hardware protection mechanisms do not extend across devices. For example, the TPM owner domain provides an isolation mechanism for only a single TPM. When a new owner takes ownership of the TPM, they cannot access the previous owner's TPM-protected secrets. When the same user owns two different TPMs (on two different devices), the owner domains of each TPM remain isolated and cannot jointly offer hardware-based protection of the user's keys and data. Thus, mobile services cannot rely on TPMs alone to enable secure data sharing across devices.

3.2.1 Secure Key Exchange

To better illustrate these challenges, we now describe in-depth how to perform secure key exchange between two TPM-equipped mobile devices, a critical building block in enabling secure data sharing across devices. Key exchange is a common bootstrapping step used in security protocols that provide authentication and encryption, such as SSL, SSH, VPNs, etc. The TPM offers hardware-protection for cryptographic keys. Thus, even if a system were compromised, the key itself would remain protected. A desirable property for secure key exchange between two TPM-equipped devices is the establishment of a secure communication channel even when both are infected by malware. This requires TPMs to perform the cryptographic steps for key exchange without leaking the key to the malware.

Unfortunately exchanging a key securely between two parties is notoriously challenging in practice because of the *identity problem* – one party needs to verify the correct identity of the other. One way to do this is to use a public key infrastructure (PKI) where each party applies to a certificate authority for a digital certificate, which serves for others as a non-tamperable authentication of identity.

Thus, the TPM specification does not directly provide an implementation of any secure key exchange protocol. Because TPMs lack the functionality of a key exchange protocol (e.g., Diffie-Hellman), two TPMs can exchange keys only by performing a *one-time key migration* from one device's TPM to another in the *absence of malware*. Without either of these properties, malware could either migrate the key to a malicious device or obtain a copy of it during migration.

Figure 2 shows the pseudo-code a device must execute to generate a key that can be shared with another device. This code requires use of the secure execution mode (i.e., Intel's TXT or AMD's SEM) to reduce its TCB and thereby reduce the likelihood of the presence of malware. To ensure that key migration is a one-time only operation, the code assigns a migration policy to the key based on a secret (denoted by S in the pseudo-code). Once S is destroyed, the key can no longer be migrated.

[1] See the Flicker [19] Web page for details on the difficulty of finding the appropriate hardware and software to use SEM (https://sparrow.ece.cmu.edu/group/flicker.html).

```
//Reduce the likelihood of malware.
1. Enter secure execution mode

//Create a shared key K. K is migrateable
//only by knowing a secret S.
2. Create symmetric key K
3. Create secret S
4. Set migration policy of K to a secret S

//Secure identity verification of device 2.
5. Verify identity of device 2

//Encrypt K with device 2's public key. Secret S
//is needed for this operation.
6. Using S, create migrateable copy of K for device 2

//Permanently disable K's ability to migrate.
7. Permanently destroy S

//Exit SEM and send encrypted K to device 2.
8. Leave secure execution mode
9. Send migrateable copy of K to device 2
```

Figure 2. **Pseudo-code for secure key exchange.**

3.3 Limitation 2: Trusted Clock

Today's TPMs do not offer a trusted real-time clock. Instead, the TPM combines a trusted timer with a secure, non-volatile counter. For every tick received, the TPM increments the value of a counter stored in memory. For every n increments of this counter, the counter value is persisted to the TPM's NV storage. The TPM has an estimate of the timer's frequency and thus has an approximate notion of time. However, this mechanism can keep track of time only when the TPM is running (and *not* when the platform is powered off). Because the counter value is persisted only every n increments, this mechanism does not even provide a guarantee of monotonicity. Upon a reboot, the timer is rolled back to the last persisted counter value violating monotonicity. The TPM's timer mechanism solely guarantees that as long as the platform does not reboot, the timer will move forward. As such, it can provide an approximate time-since-boot.

This mechanism is inadequate for offering real-time guarantees that would be useful for offline content access. For example, movie studios already charge a premium to make a movie available on home theaters on the day of release. Although TPMs can provide offline access securely, they cannot offer *make the following movie available for watching next Friday at midnight*.

3.4 Limitation 3: NV Storage

The TPM's NV storage is inadequate for applications that require frequent writes or require large amounts of trusted storage. For example, previous work [16] has shown that a trusted module offering a monotonic counter and a key solves several problems in distributed systems that stem from participants' ability to equivocate. Unfortunately, even though TPMs offer this functionality, their implementation of NV storage cannot meet the write frequency requirements of distributed systems protocols. The TPM specification dictates the inclusion of monotonic counters, but the spec requires only the ability to increment these counters at a very slow place (e.g., once every five seconds), which is insufficient for high-event applications such as networked games [16]. Similarly, although the TPM specification mandates access-controlled, non-volatile storage, most implementations provide only 1,280 bytes of NVRAM [26]. These limitations have led researchers to seek alternative designs for trusted devices [16].

3.5 How cTPM Overcomes These Limitations

To address these limitations, we propose cTPM, a modification to the TPM design that includes an additional cloud control domain. This domain offers the same functionality as the owner domain except that its primary seed is also shared with the cloud. Sharing the seed with the cloud allows both cTPM and the cloud to generate the same cloud root key (see Section 5.3 for details). Combining the cloud root key with remote storage lets cTPM: 1) better share data via the cloud, 2) have access to a trusted real-time clock, and 3) have access to remote NV storage that supports a large quantity of storage, and high frequency writes.

cTPM's design facilitates data sharing. The pre-shared primary seed lets the cloud effectively act as a PKI. The cloud and the device's TPM can use this shared secret to encrypt and authenticate their messages to each other. The identity problem has now been "pushed" to ensuring that the cloud primary seed is shared securely between cTPM and the cloud. This initial sharing step should be done at cTPM manufacturing time when the cTPM's three other primary seeds are provisioned.

The cloud domain also equips cTPM with a trusted clock using a protocol similar to the Time Protocol described in RFC 868 [27]. Once the clock value is obtained from the cloud, cTPM uses its local timer to advance the clock. It has a global variable that dictates how often it should re-synchronize the clock; the TPM owner sets this variable whose value default is one day.

Finally, cTPM uses the cloud for additional NV storage to overcome TPM NV storage limitations. There are no limits on how much additional NV storage the cloud can provide to a single cTPM. A portion of the physical cTPM chip's RAM is thus allocated as a local cache for the cloud-backed NV storage. The performance of cTPM cloud-backed NV storage exceeds that of the TPM because TPM NV accesses are no longer needed.

4 Trust Assumptions and Threat Model

4.1 Trusting the Cloud

All the new cTPM functionality associated with the cloud domain assumes the cloud is trustworthy and not compromised by malware. While everyone may not agree with this assumption, cloud providers have more incentives and resources to monitor and eliminate malware than average users. Security-conscious cloud providers could use secure hypervisors with a small TCB [18], narrow interfaces [24], or increased protection against cloud administrators [40, 28].

Whether using a TPM or not, a cloud compromise would already affect the security of a mobile service relying on the cloud for its functionality. However, even if the cloud were compromised, all secrets protected by the TPM-specific control domains other than the cloud domain would remain secure. For example, all device-specific secrets protected in the owner's control domain (i.e., using TPM's SRK) would remain uncompromised.

In the event that the cloud were compromised, cTPM could no longer offer its security guarantees. To recover would require changing the cloud seed (rekeying). To do so, we see only two options: issue a new device to the user, or implement a secure rekeying mechanism by visiting an authorized store (e.g., a mobile operator store such as AT&T) where the staff has specialized hardware to perform a secure rekey. A rekey would also be required whenever devices change ownership. While cTPM lets the owner *clear* the device (i.e., erase its cloud seed and all secrets protected by it), the new owner would need to physically visit a store to obtain a new seed.

One alternative to the current cTPM design is to have a trusted 3rd party offer the remote cTPM functionality rather than the cloud. For example, the cTPM could be offered by a TPM manufacturer rather than by the cloud. However, we have not fully pursued such an alternative cTPM design.

4.2 Threat Model

Our threat model resembles that of traditional TPMs: all software attacks are in scope (including side-channel attacks) because cTPM is isolated from the host platform and can therefore provide its security guarantees even if the host were compromised (e.g., infected with malware). However, physical attacks are out of scope. Such attacks include decapsulation, microprobing, or focused ion beaming the TPM chip [34], monitoring its internal buses [35], or inserting traffic on the bus between the CPU and the TPM. Furthermore, DoS attacks in which the (untrusted) operating system or applications deny access to the cTPM or to the cloud are out of scope. For example, a TPM can be put in lockout mode if an application attempts to "guess" an authorization value (e.g., a "password") to a secret it protects. During the lockout, the TPM refuses to serve any requests to protected secrets made by any application. Once the lockout timeout expires, the TPM exits lockout and can receive additional requests. TPMs today are thus susceptible to DoS attacks by applications that repeatedly attempt to guess the wrong authorization values until the TPM enters lockout and refuses to answer additional requests.

Another class of attacks specific to the cTPM stems from our use of remote cloud storage. The (untrusted) operating system could drop, corrupt, or re-order messages from the cloud. Even worse, it could delay messages from the cloud in an effort to serve stale data to the TPM. All such attacks are in scope and addressed by cTPM; for example, to ensure freshness, cTPM uses a local timer to timeout any pending requests not yet serviced.

cTPM has a dual relationship with the cloud. On one hand, it trusts the cloud with any keys and data the cloud stores in the cloud-backed NV storage. The cloud must offer increased assurance that these keys are not compromised; for example, cloud-stored keys should be protected against malware, malicious administrators [31], and side-channel attacks [41]. On the other hand, cTPM has additional local NV storage that protects its own secrets from the cloud, as needed. We believe that this dual relationship helps mobile services share data across devices, yet does not place unlimited trust in the cloud. The owner or privacy administrator can always use their own control domain to protect secrets from the cloud.

5 cTPM High-Level Design

The cTPM design extends the TPM 2.0 by adding: the ability to share a primary seed with the cloud, and the ability to access cloud-hosted non-volatile (NV) storage. This section describes the high-level design and the challenges we encountered when implementing these features. While our description is TPM 2.0-specific, our changes could be equally applied to TPM 1.2.

5.1 Cross-Device Usage Model

Each device has a unique cTPM with a unique primary seed shared with the cloud and used to derive additional keys (Section 5.3 describes the derived keys in more depth). All devices registered with the same owner have their keys tied to the owner's credentials. The cloud could then offer cTPM services that create a shared key across all devices owned by the same user. For example, when "bob@hotmail.com" calls this service, a shared key is automatically provisioned to the cTPM on each of Bob's devices. This shared key can bootstrap the data sharing scenarios described by this paper.

5.2 Architecture

cTPM consists of two different components, one running on the device and the other in the cloud. Both components implement the full TPM 2.0 software stack with the additional cTPM features. This ensures that all cloud

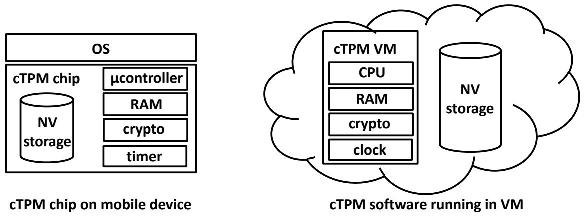

Figure 3. **cTPM High-level Architecture**.

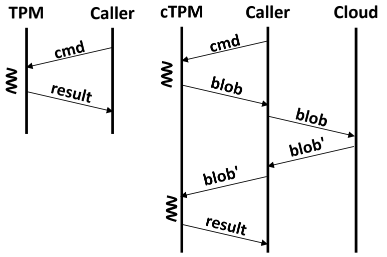

Figure 4. **The sequence of steps for issuing a synchronous command (left) versus an asynchronous command (right).** The cTPM remains responsive to other commands while the caller relays the blob to the cloud.

operations made to the cTPM strictly follow TPM semantics, and thus we do not need to re-verify their security properties. On the device-side, the cTPM software stack runs in the TPM chip, whereas the cloud runs the cTPM software inside a VM. On the cloud-side, the NV storage is regular cloud storage, and the timer offers a real-time clock function. The cloud-side cTPM software reads the local time upon every initialization and uses NTP to synchronize with a reference clock. When running in the cloud, cTPM resources (e.g., storage, clock) need not be encapsulated in hardware because the OS running in the VM is assumed to be trusted. In contrast, the device's OS is untrusted, and thus the cTPM chip itself must be able to offer these resources in isolation from the OS. Figure 3 illustrates the high-level architecture of the cTPM.

5.3 Shared Cloud Primary Seed

Upon starting, the local cTPM checks whether a shared cloud primary seed is present. If not, it disables its cloud control domain and all commands associated with it. A cTPM is provisioned with a cloud primary seed via a proprietary interface available only to the device manufacturer.

The cTPM uses the cloud primary seed to generate an asymmetric storage root key, called the *cloud root key* (CRK), and a symmetric communication key, called the *cloud communication key* (CCK). Both keys are derived from the cloud primary seed. These key derivations occur twice: once on the device-side and once on the cloud-side of the cTPM. Because the key derivations are deterministic, both the device and the cloud end up with identical key copies. The CRK's semantics are identical to those of the *storage root key* (SRK) controlled by the TPM's owner domain. The CRK encrypts all objects protected within the cloud control domain (similar to how SRK encrypts all objects within the owner domain). The CCK is specific to the cloud domain, and it protects all data exchanged with the cloud.

The cTPM uses the same mechanism to generate keys as TPM 2.0. In particular, the generation of a primary key from a seed is based on use of an approved key derivation function (KDF). TPM 2.0 uses the KDF from SP800-108 [25] in its specification.

We now examine the design challenges associated with exchanging data between the local cTPM and the cloud cTPM.

The Need for Secure Asynchronous Communication. cTPM cannot directly communicate with the cloud. Instead, it must rely on the OS for all its communication needs. Since the OS is untrusted, cTPM must protect the integrity and confidentiality of all data exchanged between the cTPM and the cloud-backed storage, as well as protect against rollback attacks. The OS is regarded merely as an insecure channel that forwards information to and from the cloud.

In addition to ensuring security, cTPM must support asynchronous communication between the local cTPM and the cloud. Today, the TPM is single-threaded, and all TPM commands are synchronous. When a command arrives, the caller blocks and the TPM cannot process any other commands until the command terminates. Making cTPM cloud communication synchronous would lead to unacceptable performance. For example, consider issuing a cTPM command that increments a counter in cloud-backed NV storage. This command would make the TPM unresponsive and block until the increment update propagates all the way to the cloud and the response returns to the local device.

Instead, we chose to make cloud communication asynchronous. Whenever a command that needs access to remote NV is received, cTPM returns to the caller an encrypted blob that needs to be sent remotely. The caller must send this blob to the cloud; if the cloud accepts the blob, it returns another encrypted blob reply to the caller. The caller then passes this reply to the cTPM, at which point the command completes. cTPM remains responsive to all other commands during this asynchronous communication with the cloud. Figure 4 illustrates these steps and contrasts them with a traditional simple TPM command. All cTPM commands that do not require access to remote NV storage remain synchronous, similar to TPMs today.

Dealing with Connectivity Loss. Loss of connectivity is transparent to the cTPM because all network signaling and communication is done by the operating system. However, the two-step nature of asynchronous commands requires the cTPM to maintain in-memory state between the steps. This introduces another potential resource allocation denial-of-service attack: a malicious OS could issue many asynchronous commands that cause the cTPM to fill up its RAM. Also, as mentioned in our threat model, an attacker could launch a staleness attack whereby artificial delays are introduced in the communication with the cloud.

To protect against these attacks, cTPM maintains a *global route timeout* (GRT) value. Whenever an asynchronous request is issued, cTPM starts a timer set to the GRT. Additionally, to free up RAM, cTPM scans all outstanding asynchronous commands and discards those whose timers have expired. The GRT can be set by the cTPM's owner and has a default value of 5 minutes.

5.4 Cloud-backed NV Storage

The TPM uses a special data structure, called an NV index, to store data values persistently to NV storage. When a persistent object is referenced in a TPM command, the TPM loads the object into its RAM. When allocating a new index, an application must specify its access control (read-only or read-write), its type, and size. There are four possible types of NV indexes: (1) *ordinary*, for storing data blobs, (2) *counters*, for storing secure monotonic counters, (3) *bit-fields*, which can be set individually, and (4) *extend*, which can be modified only by using an extend operation similar to PCRs.

At a high level, the cloud-backed NV storage is just a key-value store whose keys are NV indices. Accessing the remote NV index entries requires the OS to assist with the communication between the cTPM and the cloud. These operations are thus asynchronous and follow the same two-step model described in Figure 4. However, the remote nature of these NV indices raises additional design challenges.

Local NV Storage Cache. Remote NV entries can be cached locally in the cTPM's RAM. To do so, we add a *time-to-live* (TTL) to remote NV entries. The TTL specifies how long (in seconds) the cTPM can cache an NV entry in its local RAM. Once the TTL expires, the NV index is deleted from RAM and must be re-loaded from the remote cloud NV storage with a fresh, up-to-date copy. The local storage cache is *not persistent* – it is fully erased each time the computer reboots. We also add a *synchronization timestamp* (ST) set to the time the entry was last cached locally. If there is no in-memory cached entry of the NV index, this timestamp is null.

Caching's main benefits are performance and availability; remote NV read operations may not require a round-trip to the cloud if they can be read from the local cache. This enables the reading of NV storage entries even when the device is disconnected as long as their TTL has not expired. The trade-off is that locally cached entries could be stale. Cloud updates to a cloud-backed NV entry are reflected locally only after the TTL expires. The TTL controls the trade-off between performance and staleness for each NV index entry.

For writes, the local cache's policy is write back, and it relies on the caller to propagate the write to the cloud NV storage. A cTPM NV write command updates the cache first and returns an error code that indicates the write back to the NV storage is pending. The caller must initiate a write protocol to the cloud NV. If the caller fails to complete the write back, the write remains volatile, and the cTPM makes no guarantees about its persistence.

Trusted Clock. In cTPM, the trusted clock is an NV entry (with a pre-assigned NV index) that only the cloud can update. The local device can read the trusted clock simply by issuing an NV read command for this remote entry. Reading the entry is subject to a timeout much stricter than the regular global route timeout (GRT), called the *global clock timeout* (GCT). The trusted clock NV entry is cached in the on-chip RAM. In this way, the cTPM always has access to the current time by adding the current timer tick count to the synchronization timestamp (ST) of the clock NV entry.

$$maxClockError \leq TTL \times drift + GCT \qquad (1)$$

Equation (1) describes the upper bound of the local clock's accuracy as a function of TTL, drift and GCT. By default, the TTL is set to 1 day and the global clock timeout (GCT) to 1 second. A low GCT improves local clock accuracy, but may lead to unavailability if the device-to-cloud communication has high latency. We find that these values are sufficiently accurate for our mobile scenario (i.e., the release of movies on Fridays at midnight). However, setting the GCT even lower can further improve accuracy, while setting the TTL lower reduces the effect of drift.

5.5 Islanded Devices

Although connectivity loss is masked by the OS, devices could be offline for long periods of time. We refer to such devices as islanded devices. Islanded devices do not raise additional security concerns, even when they are out of sync with the cloud. Instead, when long periods of disconnection occur the cTPM functionality slowly degrades as entries in the local NV cache become stale. When devices reconnect, they need to re-sync their cloud-based cTPM state. However, we believe that most

```
NV_Read(NVIndex idx) {

// Garbage collect all local cache
foreach nvIdx in LocalCache
  if LocalCache[nvIdx].TTL Is Expired
    delete nvIdx from LocalCache
  endif
endforeach

// return NV entry if present
if idx in LocalCache return LocalCache[idx]

// return not found in cache
return ErrorCode.NotFoundInCache
}
```

Figure 5. **Reading NV entry from local cache.**

```
NV_Write(NVEntry entry) {

//Garbage collect all local cache
foreach nvIdx in LocalCache
  if LocalCache[nvIdx].TTL Is Expired
    delete nvIdx from LocalCache
  endif
endforeach

//Insert the entry in the cache
idx = LocalCache.Append(entry)

//Set the entry's TTL
LocalCache[idx].TTL = DefaultTTL
}
```

Figure 6. **Writing NV entry to local cache.**

mobile devices become islanded only when left unused. When used regularly, devices have ample opportunity to connect to the Internet and sync their cTPM state.

6 Detailed Design and Implementation

This section provides more detail on the cTPM's design and implementation. We describe how the cTPM shares TPM-protected keys between the cloud and the device, and we present the changes made to support NV reads and writes. We also describe the cloud/device synchronization protocol, and the three new TPM commands we added to implement synchronization.

6.1 Sharing TPM-protected Keys

The TPM 2.0 API facilitates the sharing of TPM-protected keys by decoupling key creation from key usage. **TPM2_Create()**, a TPM 2.0 command, creates a symmetric key or asymmetric key-pair. The TPM creates the key internally and encrypts any private (or symmetric) keys with its storage key before returning them to the caller. To use the key, the caller must issue a **TPM2_Load()** command, which passes in the public storage key and the encrypted private (or symmetric) key. The TPM decrypts the private key and loads it in RAM. The TPM can then begin to encrypt or decrypt using the key.

The separation between create and load is needed due to the limited RAM available on the TPM chip. It lets callers create many keys without having to load them all into RAM. As long as the storage root key (SRK) never leaves the chip, encrypting the new keys' private parts with the SRK guarantees their confidentiality.

This separation lets cTPM use cloud-created keys on the local device to gain two benefits. First, key sharing between devices becomes trivial. The cloud can perform the key sharing protocol between two cTPM VMs, as described earlier in Figure 2. Unlike TPM 2.0, this protocol does not need to use a PKI, nor does it need to run in a SEM. Once a shared key is created between two cloud cTPM VMs, both mobile devices can load the key in their chips separately by issuing **TPM2_Load()** commands. Second, key creation can be performed even when the mobile device is offline. This makes it simple for users to create shared keys across all their devices without having to ensure those devices are online first. We illustrate both these benefits in our extension of Pasture in Section 7.

6.2 Accessing Cloud NV Storage

The cTPM maintains a local cache of all reads and writes made to the cloud NV storage. A read returns a cache entry, and a write updates a cache entry only. The cTPM does not itself update remote cloud NV storage; instead the caller must synchronize the on-chip RAM cache with the cloud NV storage. This is done using a synchronization protocol.

Read Cloud NV. Upon an NV read command, the corresponding NV entry is returned from the local cache. If not found, cTPM returns an error code. The caller must now check the remote NV; to do so, it needs to initiate a *pull* synchronization operation (described in Section 6.3) to update the local cache. After synchronization completes, the caller must reissue the read TPM command, which will now be answered successfully from the cache. Figure 5 shows the pseudo-code for reading a remote NV entry from the local cache.

Write Cloud NV. An NV write command first updates the cache and returns an error code that indicates the write back to the remote NV storage is pending. The caller must initiate a *push* synchronization operation to the cloud NV (see Section 6.3). If the caller fails to complete the write back, the write remains volatile, and cTPM makes no guarantees about its persistence. Figure 6 shows the pseudo-code for writing an NV entry to the local cache.

6.3 Synchronization Protocol

The synchronization protocol serves to: (1) update the local cache with entries from the cloud-backed NV storage for NV reads) and (2) write updated cache entries back to the cloud-backed NV storage (for NV writes). On the device side, the caller performs the protocol using two new commands, **TPM2_Sync_Begin()** and

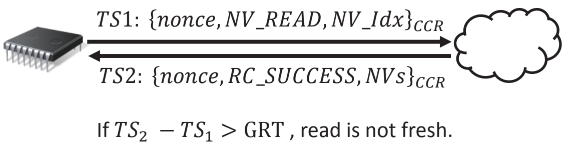
$TS1$: $\{nonce, NV\_READ, NV\_Idx\}_{CCR}$
$TS2$: $\{nonce, RC\_SUCCESS, NVs\}_{CCR}$
If $TS_2 - TS_1 >$ GRT , read is not fresh.

Figure 7. **Synchronization protocol: pull NV entry from cloud-backed NV storage.**

TPM2_Sync_End(). These commands take a parameter called *direction*, which can be set to either a *pull* or *push* to distinguish between reads and writes. All messages are encrypted with the cloud communication key (CCK), a symmetric key.

Pull from Cloud-backed NV Storage. The cTPM first records the value of its internal timer and sends a message that includes the requested NV index and a nonce. The nonce checks for freshness of the response and protect against replay attacks. Upon receipt, the cloud decrypts the message and checks its integrity. In response, the cloud sends back the nonce together with the value corresponding to the NV index requested. The cTPM decrypts the message, checks its integrity, and verifies the nonce. If these checks are successful, cTPM performs one last check to verify that the response's delay did not exceed its global read timeout (GRT) value. If all checks pass, cTPM processes the read successfully. Figure 7 shows the precise messages exchanged between the cTPM and the cloud to read the remote NV.

Push to Cloud-backed NV Storage. The protocol for writing back an NV entry is more complex because it must also handle the possibility that an attacker may try to reorder write operations. For example, a malicious OS or application can save an older write and attempt to reapply it later, effectively overwriting the up-to-date value. To overcome this, the protocol relies on a secure monotonic counter maintained by the cloud. Each write operation must present the current value of the counter to be applied; thus, stale writes cannot be replayed. cTPM can read the current value of the secure counter using the previously described pull protocol. Figure 8 shows the precise messages exchanged between the cTPM and the cloud to write a remote NV entry. Note that reading the secure counter need not be done on each write because the local cTPM caches the up-to-date value in RAM.

When the cloud receives an NV entry through the push synchronization protocol, it must update its NV storage. To do so, we equipped the cTPM with a third command, called **TPM2_Sync_Proc()** (for *process*). This command can be issued only by the cloud; the cloud takes the message received from the local device and calls sync process with it. The cloud cTPM decrypts the message and applies the NV update.

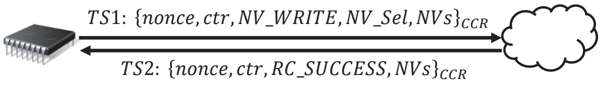
$TS1$: $\{nonce, ctr, NV\_WRITE, NV\_Sel, NVs\}_{CCR}$
$TS2$: $\{nonce, ctr, RC\_SUCCESS, NVs\}_{CCR}$

Figure 8. **Synchronization protocol: push NV entry to cloud-backed NV storage.**

6.4 Implementation

We implemented cTPM by modifying the TPM 2.0 (release 0.96) codebase; this codebase serves as both the TPM specification and a reference implementation. The original codebase was 23,163 lines of code; for cTPM, we added 1,304 lines of code, for a total of 24,467 lines of code. The bulk of this code implements the three new cTPM commands – sync begin, end, and process. We also made minor changes to the commands that update the TPM NV. These commands indicate whether NV access should be to the local on-chip NV or the cloud-backed NV. If the latter, the command can return additional error codes depending on whether the NV entry is found in the local cache (for reads) or whether the update must be written back to the cloud NV.

The TPM 2.0 codebase does not include a cryptography library. This is deliberate in order to reduce the hurdle of porting it to different OEM hardware environments. For example, one hardware OEM might want to use its own in-house crypto library, whereas another might want to use OpenSSL. The TPM 2.0 codebase just defines a crypto API. We used a Microsoft internal cryptography library for the TPM 2.0 needs.

TPM 2.0 also does not include platform resources, such as how to obtain entropy, how to receive a power-on/power-off signal, or how to access the underlying NV storage. For all platform needs, we used a library that provides the TPM platform resources from the underlying OS (Windows). Our platform implementation receives TPM commands via a network socket using a home-baked command/response protocol.

All our testing code and applications, such as Pasture and TrInc, were implemented in C#. All TPM commands are relayed via the network socket to and from cTPM.

7 Case Studies

This section presents two case studies on using cTPM to build trusted mobile services. In each case, we describe these services' current limitations and show how cTPM addresses them by improving performance or adding functionality.

7.1 Case Study 1: Pasture

Pasture [14] is a TPM-based protocol for secure offline data access. Using Pasture, the content receiver creates a TPM-bound encryption key, called a *bound key*, with a usage policy dictating that the TPM can use the key for decrypting content only when a certain PCR register contains a specified value. This policy provides *access undeniability* – once the key is used for decryption, the user cannot lie about its usage, and *verifiable revocation* – the user cannot use the key once revoked. Issuing a TPM extend operation on the PCR register to a predetermined value R represents the receiver's decision to consume the content. If the receiver decides to revoke the content instead, it extends the PCR to a different value. In this case, the PCR cannot be extended to R any longer. Over time, the PCR value represents a chain of decisions about whether to watch or revoke a sequence of movies.

Upon receiving a bound key, the content server checks that the key is bound to a correct usage policy, encrypts the content using it, and sends the encrypted content to the receiver. At any point in the future, the receiver can choose whether or not to decrypt the content; this choice can occur even in disconnected mode. Once made however, this choice cannot be undone.

The Pasture protocol also addresses computer reboots. An adversary could try to use such reboots to reset the PCR register, which opens the door to rollback attacks. This part of the Pasture protocol requires secure execution mode (SEM). In our cross-device Pasture implementation, we eliminated the need for SEM by leveraging a new TPM 2.0 type of NV index that has behavior similar to a PCR and is modified using *TPM2_NV_Extend()*.

Limitation 1: Lack of Sharing. The lack of sharing primitives in TPM prevents extending Pasture to a set of devices owned by a single user. Instead, each device must run its own version of Pasture, creating a TPM-specific bound key and uploading it to the server. The server then runs a Pasture session with each individual device. All devices must act separately despite being owned by the same user.

With cTPM, the cloud performs the Pasture protocol on behalf of all devices owned by a single user. A single bound key is shared by all devices, and a single copy of the content is encrypted with this bound key. The cloud can write this bound key directly to the cloud-backed NV storage and encrypt the content even when the client is disconnected. When the receiver connects to the cloud, it can then re-sync its nonvolatile state to receive a copy of the bound key and to start downloading encrypted content. As with the original Pasture, the policy specifies the PCR value necessary to use the key bound to it.

This multi-device version of Pasture complicates the process of accepting and revoking content. If any device accepts content and starts decrypting it, then the content can no longer be revoked. Thus, the content server accepts a revoke decision only when *all* of a user's devices have decided to revoke.

```
CreateBoundKey(hM):

//For each device, read current PCR and
// future PCR if decision is accept.
foreach dev in Owner.AllDevices() do
  $R_t^{\text{dev}}$ ← TPM_Read(PCR$_{\text{APP}}^{\text{dev}}$)
  $R_{t+1}^{\text{dev}}$ ← SHA2($R_t^{\text{dev}} || hM$)
endfor

//Create bound key with a disjunction of commit values.
E←TPM_CreateWrapKey({
  {
    PCR$_{\text{APP}} = R_{t+1}^{\text{dev1}}$ ||
    PCR$_{\text{APP}} = R_{t+1}^{\text{dev2}}$ ||
    ...
    PCR$_{\text{APP}} = R_{t+1}^{\text{devN}}$
  }
  &&
  PCR$_{\text{SEM}} = SemHappy$
})

//Create proof for the bound key.
EP← ( ``CreateBoundKey'', hM,
  $R_t^{\text{dev1}}$, $R_{t+1}^{\text{dev1}}$, ..., $R_t^{\text{devN}}$, $R_{t+1}^{\text{devN}}$, E, $\alpha$)
```

Figure 9. **Create bound key in multi-device Pasture.**

Another interesting challenge of multi-device Pasture is when one user makes conflicting decisions on different devices. This causes different values to be stored in their PCR registers. One solution is to insist that all devices owned by the same user share the same log of decisions about accepting or revoking the content. This ensures that the PCR registers on each device share the same value and work in sync. However, it is very difficult to enforce this coordination across devices when some are offline.

An alternate approach lets different devices maintain their own per-device log of decisions. This more flexible solution lets a user make different decisions for different devices without having to reconcile them. Because the per-device decisions can differ, the content server must ensure that a content revocation occurs only when all devices revoke. This approach requires the bound key to be attached to a policy that specifies a set of possible PCR values corresponding to each separate device.

We implemented this latter approach using the *TPM2_PolicyOR()* command, which creates a single policy as a disjunction of individual conditions (in our case, each condition corresponds to one PCR value). As long as a device's PCR value matches one condition, the bound key can decrypt the content. Note that extending the accept decision from one value to multiple values does not reduce protocol security even though it increases the chances of a hash collision. If hash collisions ever become a cause of concern, TPM 2.0 (and thus cTPM) permits the use of stronger hash functions (e.g., 192-bit SHA384). Figure 9 shows the multi-device CreateBoundKey implementation (CreateBoundKey in the original Pasture is shown in Figure 3 of [14]).

```
Attest(i, c', h, n):

1. Assert NV_Read(i) is a remote NV of type counter.
2. Let c be the value of that counter.
3. Assert no roll-over: $c \leq c'$.
4. $a \leftarrow < i, c, c', h >_{\text{TRINC}_{\text{PRIV}}}$.
5. Insert a into Q, ejecting the oldest value.
6. NV_Write(i, c').
7. Return a.
```

Figure 10. **Attest in cTPM TrInc [16].**

Limitation 2: Lack of Server-side Revocation. The original Pasture protocol lets a receiver revoke access to content in a verifiable manner. Once revoked, the receiver cannot further access the content. However, Pasture does not support server-side revocation. A Pasture movie server could use revocation to deny access to malicious clients, such as clients that paid using stolen credit cards.

Implementing server-side revocation in Pasture would prove very challenging because the client would have to agree to run code that would unload the bound key from the TPM. The client could always refuse to run such code and prevent a server from revoking the bound key.

With cTPM, the content server could simply ask the cloud to delete the bound key from the cloud-backed cTPM NV storage. This would not necessarily cause an immediate revocation because the device could be offline and store a cached copy of the key. However, the cached copy would eventually expire (based on its TTL), at which point the key is guaranteed to be revoked.

Limitation 3: Lack of Trusted Clock Guarantees. Because TPMs lack a trusted source of time, a Pasture movie server cannot offer time-based guarantees for its content (e.g., *make the following movie available for watching next Friday at midnight*). This scenario is quite attractive to consumers: a startup is currently selling proprietary technology for watching movies at home the same day they arrive in theaters. This hardware costs $35,000, and each movie release costs $500 [6].

With cTPM's trusted clock, a bound key could incorporate a clause specifying a future timestamp as an additional condition for using the key for decryption. The bound key could be revoked (either client-side or server-side) at any time; however its usage for decryption would remain restricted to only a future, pre-specified time value.

7.2 Case Study 2: TrInc

TrInc [16] is a trusted incrementer used to combat "equivocation", i.e., making conflicting statements to others in a distributed system. It uses a secure counter and a key, and was implemented on a smartcard due to its poor performance on TPMs.

TrInc's main API, a function called *Attest*, produces an attestation that the secure counter has been incremented from the current value c to a value c' not smaller than c. Each attestation covers the secure counter's interval $(c, c']$. TrInc uses these attestations to prove statements that prevent nodes from equivocating without being detected. In BitTorrent, for instance, the counter represents the number of blocks a peer has received, a value which is naturally monotonically increasing. Figure 10 illustrates our implementation of the Attest function using cTPM (the original Attest implementation is described in Section 3.5.1 of [16]).

TPM 2.0	**TPM 1.2**
TPM2_NV_Write()	TPM_NV_Write()
TPM2_NV_Read()	TPM_NV_Read()
TPM2_NV_Read (Counter)()	TPM_ReadCounter()
TPM2_PCR_Read()	TPM_PCRRead()
TPM2_PCR_Extend()	TPM_PCRWrite()
TPM2_Create()	TPM_Create()
TPM2_Load()	TPM_Load()
TPM2_Unseal(Unbind)()	TPM_Unbind()
TPM2_Sign()	TPM_Sign()
TPM2_Quote()	TPM_Quote()
TPM2_CertifyCreation()	TPM_Sign()
TPM2_Sync_Begin()	TPM_Unbind()
TPM2_Sync_End()	TPM_Unbind()

Table 1. **Mapping TPM 2.0 commands to their TPM 1.2 counterparts.**

8 Evaluation

8.1 Protocol Verification

We verified the correctness of our protocols using an automated theorem prover, ProVerif [3], which supports the specification of security protocols for distributed systems in concurrent process calculus (pi-calculus). We specified the synchronization protocol used by our system, both pull and push, in 98 lines of pi-calculus code. ProVerif verified the security of our protocols in the presence of an attacker with unrestricted access to the OS, applications, or network. The attacker could intercept, modify, replay and inject new messages into the network (similar to the Dolev-Yao model).

8.2 Performance Evaluation

Our main challenge in evaluating the performance of cTPM was the unavailability of a hardware TPM 2.0 chip. The TPM 2.0 specification, currently released for public review, is not yet available off-the-shelf. Through private conversations with TPM manufacturers, we learned that they are already porting the TPM 2.0 specification to their hardware, and that the hardware performance profile for TPM 2.0 will be similar to that of TPM 1.2. As a result, we used a TPM 1.2 chip to emulate the hardware performance of a future TPM 2.0 chip. To do so, we mapped TPM 2.0 commands used in our cTPM implementation to their equivalent TPM 1.2 counterparts, as shown in Table 1.

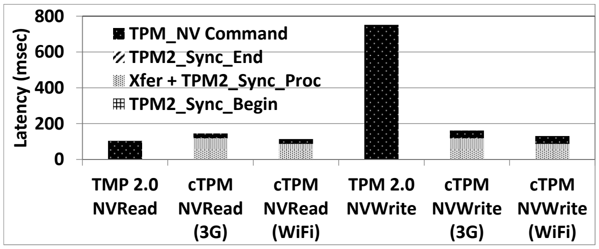

Figure 11. **cTPM NV performance:** 640 bytes.

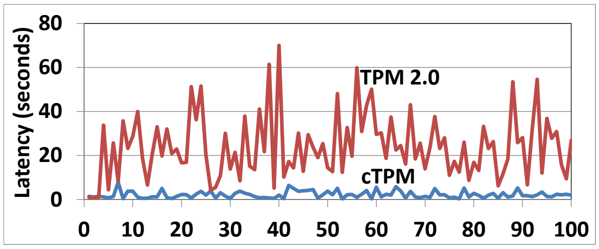

Figure 12. **Latency of TPM RSA 2048 key creation**.

Our experiments used the following setup. All applications that ran on the local cTPM used Windows 7 running on a PC with a 2 GHz Quad Intel Core i7. When a local cTPM command was issued, it was translated according to the map in Table 1 and executed against an Infineon TPM 1.2 chip. In the cloud, we ran the cTPM software in a Windows 7 VM (to emulate cloud behavior). By design, the cloud component of cTPM did not need to interact with a TPM chip.

Benchmarking NV Storage Performance. cTPM trades off the accessing of limited local TPM NV storage for the accessing of cloud-backed storage. While cloud-backed storage is very fast, it introduces latency between the device and the cloud. To evaluate this trade-off, we measured the latency of NV read and write operations for both TPM 2.0 and cTPM. We emulated Internet latencies using a standard network emulator [22] primed with 3G/4G and Wi-Fi Internet latency distributions from a recent measurement project [13].

We repeated our experiments with differently sized objects accessed in NV storage; sizes ranged from 256 bytes (corresponding to the size of a regular NV counter) to the maximum size allowed by the hardware TPM. Unfortunately, TPMs have low NV storage capacities: the largest write allowed by our TPM was only 640 bytes (whereas cTPM had no restrictions on the maximum size of its NV data). We present results using only 640-byte data objects; the results for the lower-sized objects are similar.

Figure 11 shows the access latencies for 640-byte NV objects. The local TPM 2.0 latencies are all due to the running of TPM NV commands. In contrast, cTPM latencies are the combination of four steps: (1) issuing a TPM2_Sync_Begin() command, (2) transferring the data to and from the cloud (labeled *Xfer*) and issuing a TPM2_Sync_Proc() command in the cloud, (3) issuing a TPM2_Sync_End() command, and (4) issuing a TPM NV command to access the data in memory. For NV reads, Internet latencies make the cTPM commands slightly slower in the case of 3G latencies and slightly faster in the case of Wi-Fi latencies. Note, however, that NV reads become much faster once cached locally.

Benchmarking Key Creation. When creating a key, the TPM uses local on-chip entropy, whereas cTPM can use any entropy source available to the cloud, such as a high performance hardware entropy source. For example, a company in Japan sells hardware able to produce 550 MBytes/sec of entropy for less than US$10K [15]. Our experiments used local hardware entropy found on commodity PCs. Even this entropy source was much faster than that found in TPMs.

Figure 12 shows the latencies of running 100 consecutive TPM commands to create an RSA key. The latencies of TPM commands are highly variable because the TPM blocks incoming commands to wait for the entropy source to generate random bits. The same is true for the hardware source of entropy on our PC, but this source is much faster, i.e., an average of an order of magnitude (factor of 12) faster than the TPM chip.

9 Related Work

cTPM draws inspiration from previous work on commodity trusted hardware and trusted applications.

Commodity Trusted Hardware Other than TPMs. The ARM architecture's solution for trusted computing is known as the ARM TrustZone [1]. ARM TrustZone provides a trusted execution environment on CPU cores, with hardware support for memory protection of the trusted environment, flexible control over interrupt delivery to the trusted environment, and the full power of the CPU for cryptographic operations. One could equip an ARM device with a TPM (or a cTPM) by running the TPM software stack inside the TrustZone.

Recent work from Intel has described Secure Guard Extensions (SGX) [20, 2, 12], a set of new instructions and architectures that support the concept of *enclaves*, which are isolated runtime environments similar to ARM TrustZone. Intel has shown the possibility of running secure applications inside of an enclave, such as a password manager, an enterprise rights management solution, and secure video conferencing [12]. It appears feasible for the TPM and cTPM software stacks to run inside an enclave, as well.

Trusted Applications. In addition to Pasture [14] and TrInc [16], several previous works have proposed the use of TPMs for building trusted mobile services. TruWallet described a TPM-based authentication tool for Web password protection [7]. It offered password sharing across devices owned by the same user (called *secure migration*). However, TruWallet needed to assume that the GUI and kernel were both trusted. Another project implemented a credentials manager in secure execution mode [4]. It encountered many of the performance challenges associated with this mode; for example, the network driver froze when running in SEM for more than 8 seconds. More recently, Windows 8 provided virtual smart cards, a way to use TPMs for remote authentication with server-side support [23]; these cards, bound to a single TPM, cannot migrate. For all these applications, cTPM would greatly ease sharing across devices.

Memoir describes a technique to protect the state of a trusted application while minimizing the number of NVRAM write operations [26]. With cTPM, applications could write their state to the cloud-backed NV storage and rely on Memoir-like techniques only when operating in disconnected mode.

10 Conclusions

This paper introduced cTPM, a cloud-enhanced design change to a traditional TPM that enables the simple sharing of keys and data across a user's many devices. We demonstrated cTPM's versatility by: 1) extending Pasture to support offline data access across multiple devices, server-side revocation, and real-time based guarantees for content availability, and 2) re-implementing TrInc without the need for extra hardware. We verified the protocols used to synchronize the cTPM's remote cloud storage and showed that cTPM's performance meets or exceeds that of a traditional TPM.

Acknowledgments: We are grateful to Ron Aigner, Ramakrishna Kotla, Jay Lorch, Bryan Parno, and Scott Shell for their feedback on this work and on the paper. We would like to thank Jonathan Smith and the anonymous reviewers for their feedback.

References

[1] ARM Security Technology – Building a Secure System using TrustZone Technology. ARM Technical White Paper, 2005-2009.

[2] I. Anati, S. Gueron, S. Johnson, and V. Scarlata. Innovative Technology for CPU Based Attestation and Sealing. In *Proc. of Workshop on Hardware and Architectural Support for Security and Privacy*, Tel-Aviv, Israel, 2013.

[3] B. Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *Proc. of 14th IEEE Computer Security Foundations Workshop (CSFW)*, Cape Breton, NS, 2001.

[4] S. Bugiel and J.-E. Ekberg. Implementing an Application-Specific Credential Platform Using Late-Launched Mobile Trusted Module. In *Proc. of 5th Annual Workshop on Scalable Trusted Computing*, Chicago, IL, 2010.

[5] D. Challener, K. Yoder, R. Catherman, D. Safford, , and L. V. Doorn. *A Practical Guide to Trusted Computing*. IBM Press, 2007.

[6] Digital Trends. Prima Cinema brings movies to home theaters on the day of the release for $500 a pop. http://www.digitaltrends.com/home-theater/prima-cinema-brings-movies-to-the-home-on-the-day-of-the-release/.

[7] S. Gajek, H. Lohr, and A.-R. Sadeghi. TruWallet: Trustworthy and Migratable Wallet-Based Web Authentication. In *Proc. of 4th Annual Workshop on Scalable Trusted Computing*, Chicago, IL, 2009.

[8] GigaOM. The average US subscriber owns 1.57 mobile devices. http://gigaom.com/2012/10/22/the-average-us-subscriber-owns-1-57-mobile-devices/.

[9] P. Gilbert, J. Jung, K. Lee, H. Qin, D. Sharkey, A. Sheth, and L. P. Cox. YouProve: Authenticity and Fideltiy in Mobile Sensing. In *Proc. of 10th International Conference on Mobile Systems, Applications, and Services (MobiSys)*, Lake District, UK, 2012.

[10] K. A. Goldman. IBM's Software Trusted Platform Module. http://sourceforge.net/projects/ibmswtpm/.

[11] Google. The Chromium Projects. http://www.chromium.org/developers/design-documents/tpm-usage.

[12] M. Hoekstra, R. Lal, P. Pappachan, C. Rozas, V. Phegade, and J. del Cuvillo. Using Innovative Instructions to Create Trustworthy Software Solutions. In *Proc. of Workshop on Hardware and Architectural Support for Security and Privacy*, Tel-Aviv, Israel, 2013.

[13] J. Huang, F. Qian, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck. A Close Examination of Performance Power Characteristics of 4G LTE Networks. In *Proc. of 10th International Conference on Mobile Systems, Applications, and Services (MobiSys)*, Lake District, UK, 2012.

[14] R. Kotla, T. Rodeheffer, I. Roy, P. Stuedi, and B. Wester. Pasture: Secure Offline Data Access Using Commodity Trusted Hardware. In *Proc. of 10th USENIX Symposium on Operating Systems*

Design and Implementation (OSDI), Hollywoood, CA, 2012.

[15] LETech. The Fastest True Random Number Generator with a real-time self-test function. http://www.letech.jpn.com/rng/grang_24ch_e.html.

[16] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda. TrInc: Small trusted hardware for large distributed systems. In *Proc. of 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, MA, 2009.

[17] H. Liu, S. Saroiu, A. Wolman, and H. Raj. Software Abstractions for Trusted Sensors. In *Proc. of 10th International Conference on Mobile Systems, Applications, and Services (MobiSys)*, Lake District, UK, 2012.

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

[19] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An Execution Infrastructure for TCB Minimization. In *Proc. of the ACM European Conference on Computer Systems (EuroSys)*, Glasgow, UK, 2008.

[20] F. Mckeen, I. Alexandrovich, A. Berenzon, C. Rozas, H. Shafi, V. Shanbhogue, and U. Savagaonkar. Innovative Instructions and Software Model for Isolated Execution. In *Proc. of Workshop on Hardware and Architectural Support for Security and Privacy*, Tel-Aviv, Israel, 2013.

[21] Microsoft. Help protect your files with BitLocker Driver Encryption. http://windows.microsoft.com/en-us/windows-8/using-bitlocker-drive-encryption.

[22] Microsoft. Standalone Network Emulator Tool. http://blogs.technet.com/b/juanand/archive/2010/03/05/standalone-network-emulator-tool.aspx.

[23] Microsoft. Understanding and Evaluating Virtual Smart Cards. http://www.microsoft.com/en-us/download/details.aspx?id=29076.

[24] A. Nguyen, H. Raj, S. Rayanchu, S. Saroiu, and A. Wolman. Delusional Boot: Securing Cloud Hypervisors without Massive Re-engineering. In *Proc. of the European Conference on Computer Systems (EuroSys)*, Bern, Switzerland, April 2012.

[25] NIST. Recommendation for Key Derivation Using Pseudorandom Functions. http://csrc.nist.gov/publications/nistpubs/800-108/sp800-108.pdf.

[26] B. Parno, J. R. Lorch, J. R. Douceur, J. Mickens, and J. M. McCune. Memoir: Practical State Continuity for Protected Modules. In *Proc. of IEEE Symposium on Security and Privacy*, Oakland, CA, 2011.

[27] J. Postel and K. Harrenstien. Time Protocol. http://tools.ietf.org/html/rfc868.

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

[29] M. Ryan. Introduction to the TPM 1.2. www.cs.bham.ac.uk/~mdr/research/papers/pdf/08-intro-TPM.pdf.

[30] N. Santos, H. Raj, S. Saroiu, and A. Wolman. Trusted Language Runtime (TLR): Enabling Trusted Applications on Smartphones. In *Proc. of 12th Workshop on Mobile Computing Systems and Applications (HotMobile)*, Phoenix, AZ, 2011.

[31] N. Santos, R. Rodrigues, K. P. Gummadi, and S. Saroiu. Policy-Sealed Data: A New Abstraction for Building Trusted Cloud Services. In *Proc. of the 21st USENIX Security Symposium*, Bellevue, WA, 2012.

[32] 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), 2011.

[33] E. G. Sirer, W. de Bruijn, P. Reynolds, A. Shieh, K. Walsh, D. Williams, , and F. B. Schneider. Logical Attestation: An Authorization Architecture For Trustworthy Computing. In *Proc. of Symposium on Operating Systems Principles (SOSP)*, Cascais, Portugal, 2011.

[34] S. Skorobogatov. Physical Attacks on Tamper Resistance: Progress and Lessons. In *Proc. of 2nd ARO Special Workshop on Hardware Assurance*, Washington, DC, 2011.

[35] C. Tarnovsky. Semiconductor Security Awareness, Today & Yesterday. BlackHat 2010 – http://www.youtube.com/watch?v=YzejlrGcnY8.

[36] The Economic Times (indiatimes). Singapore leads the world on mobile take up and marketing. http://articles.economictimes.indiatimes.com/2012-12-19/news/35912444_1_mobile-app-mobile-sales-singaporeans.

[37] The Register. Five mobile devices per person for 2040? http://www.theregister.co.uk/2012/07/18/acma_05_mobile_numbers/.

[38] Trusted Computing Group. TPM 2.0 Library Specification FAQ. http://www.trustedcomputinggroup.org/resources/tpm_20_library_specification_faq.

[39] Trusted Computing Group. TPM Main Specification Level 2 Version 1.2, Revision 116. http://www.trustedcomputinggroup.org/resources/tpm_main_specification.

[40] F. Zhang, J. Chen, H. Chen, and B. Zang. CloudVisor: Retrofitting Protection of Virtual Machines in Multi-tenant Cloud with Nested Virtualization. In *Proc. of Symposium on Operating Systems Principles (SOSP)*, Cascais, Portugal, 2011.

[41] Y. Zhang, A. Juels, M. Reiter, and T. Ristenpart. Cross-VM Side Channels and Their Use to Extract Private Keys. In *Proc. of the 19th ACM Conference on Computer and Communications Security*, Raleigh, NC, 2012.