cTPM: A Cloud TPM for Cross-Device Trusted Applications

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

Current Trusted Platform Modules (TPMs) are ill-suited for use in mobile services because they hinder sharing data across multiple devices seamlessly, they lack access to a trusted real-time clock, and their non-volatile storage performs poorly. This paper presents cloud TPM (cTPM), an extension of the TPM's design to address these problems. cTPM includes two features: 1) a cloud seed shared between the TPM and the cloud; and 2) remote storage in addition to the on-chip storage. cTPM allows the cloud to create and share TPM-protected keys across multiple devices, to manage a portion of a mobile device's TPM storage, and to provide each TPM with a trusted real-time clock and with high-performance non-volatile storage.

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; Singapore has 7.8 million mobile devices, which translates to 150% mobile penetration; and the average Australian will own five mobile devices by 2040. 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.

Mobile platforms, such as laptops, smartphones, and tablets are increasingly incorporating trusted computing hardware. For example, Google's Chromebooks use TPM to prevent firmware rollbacks and to store and attest user's data encryption keys. Windows 8 (on tablets and phones) offers BitLocker full-disk encryption and virtual smart cards using TPMs. Recent research leverages TPMs to build new trusted mobile services [3, 7], trusted cloud services [8], and operating systems [9].

Unfortunately, these two trends may be at odds: trusted hardware, such as the TPM, does not provide good support for cross-device functionality. Specifically, we have identified three limitations in the TPM design which hamper building cross-device trusted applications.

Limitation 1: Cross-Device Data Sharing. Current TPM abstractions offer guarantees about one single computer, and TPM's hardware protection mechanisms do not extend across devices. For example, TPMs 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. While, in theory, migrating a TPM-protected key from one TPM to another is possible, in practice, it requires using secure execution mode (SEM), such as Intel's TXT and AMD's SEM, and trusting a third-party PKI. Such requirements are very challenging. Our NSDI submission [2] describes in more depth the nature of these challenges.

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, volatile counter, which is periodically persisted to the TPM's NV storage. However, this mechanism can keep track of time only when the TPM is running (and *not* when the platform is powered off). Moreover, upon a unclean 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 *making the following movie available for watching next Friday at midnight*.

Limitation 3: Non-volatile (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 [3] 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 [3]. Similarly, although the TPM specification mandates access-controlled, non-volatile storage, most implementations provide only 1,280 bytes of NVRAM [7]. These limitations have led researchers to seek alternative designs for trusted devices [3].

Overcoming these limitations 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.

To address these limitations, we propose a single, simple modification to the TPM design, called cTPM: equipping the TPM with one primary seed that is shared with the cloud. Sharing the seed with the cloud allows both cTPM and the cloud to generate the same cloud root key. Combining the cloud root key with remote storage lets cTPM: 1) 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 pre-shared primary seed also equips cTPM with a trusted clock using a protocol similar to the Time Protocol described in RFC 868. 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.

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.

The TPM specification also mentions that a certificate demonstrating the authenticity of the TPM's embedded key pairs may be provided by the TPM's hardware manufacturer. In our experience, many TPMs (though not all) lack this certificate. The absence of this certificate makes it impossible for a third-party to determine whether a signed statement (e.g., a software attestation) is produced by a valid TPM or by an impersonating entity.

TPMs are equipped with a set of "extend-only" platform configuration registers (PCRs) guaranteed to be reset upon a computer reboot. PCRs are primarily used to store fingerprints of a portion of the booting software (e.g., the BIOS, firmware, and OS bootloader); Chromebooks and BitLocker 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. When invoked, the CPU disables interrupts (to ensure no other software is running) and a small bootloader starts executing. The bootloader then jumps to an address specified by the caller to execute any additional code. Flicker is an earlier project that demonstrated the use of secure execution mode [5].

The TPM spec does not provide minimum performance requirements, and, as a result, today's commodity TPMs are slow and inefficient. 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.

TPM 2.0. The Trusted Computing Group (TCG) is currently defining the specification for TPM version 2.0, the next version of the TPM. TPM 2.0 offers several improvements, including cryptographic algorithms agility. For example, SHA-2 and elliptic curve cryptography

(ECC) in addition to SHA-1 and RSA are offered by TPM 2.0. 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.

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. 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.

Trust Assumptions and Threat Model

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 [4], narrow interfaces [6], or increased protection against cloud administrators [10].

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.

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 and DoS attacks in which the (untrusted) operating system or applications deny access to the cTPM or to the cloud are out of scope.

Another class of attacks specific to the cTPM stems from our use of remote cloud storage. The (untrusted) OS 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 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 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.

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. 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.

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 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 1 illustrates the high-level architecture of the cTPM.

Shared Cloud Primary Seed

Upon starting, the local cTPM checks whether a shared cloud primary seed is present. If not, it disables its new cTPM functionality and all commands associated

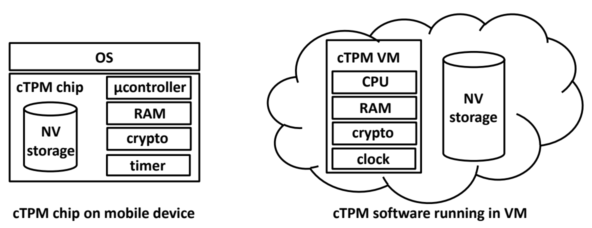

Figure 1. **cTPM High-level Architecture**.

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 based on use of an approved key derivation function (KDF). 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.

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

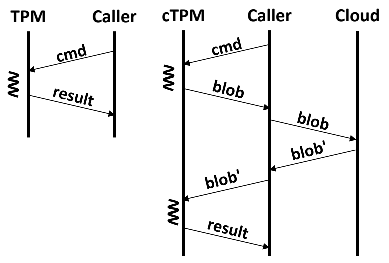

Figure 2. **The sequence of steps for issuing a synchronous command (left) versus an asynchronous command (right).** The cTPM remains responsive to other commands while the caller relays the blob to the cloud.

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 2 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.

Cloud-backed NV Storage

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 2. 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 locally cached 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 TTL controls the trade-off between performance and staleness for each NV index entry. Furthermore, the local storage cache is *not persistent* – it is fully erased each time the computer reboots.

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.

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 new TPM commands we added to implement synchronization.

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, loads it in RAM, and can begin to encrypt or decrypt using the key.

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. 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, 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 greatly simplifying creating a shared key.

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 the next section) 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.

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 the next section). If the caller fails to complete the write back, the write remains volatile, and cTPM makes no guarantees about its persistence.

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 **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.

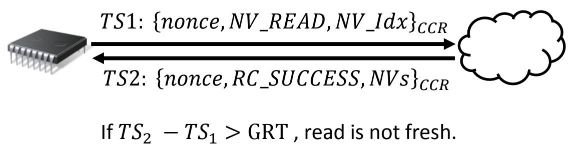
$TS1$: $\{nonce, NV\_READ, NV\_Idx\}_{CCR}$
$TS2$: $\{nonce, RC\_SUCCESS, NVs\}_{CCR}$
If $TS_2 - TS_1 >$ GRT , read is not fresh.

Figure 3. **Synchronization protocol: pull NV entry from cloud-backed NV storage.**

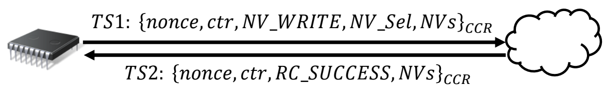
$TS1$: $\{nonce, ctr, NV\_WRITE, NV\_Sel, NVs\}_{CCR}$
$TS2$: $\{nonce, ctr, RC\_SUCCESS, NVs\}_{CCR}$

Figure 4. **Synchronization protocol: push NV entry to cloud-backed NV storage.**

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 3 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 4 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.

Protocol Verification We verified our protocols' correctness using an automated theorem prover, ProVerif [1], which supports the specification of security protocols for distributed systems in concurrent process calculus (pi-calculus). We specified our synchronization protocol, 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).

Conclusions

To summarize, the traditional TPM design fails to meet the requirement of today's cross-device trusted applications. This paper introduced cTPM, a cloud-enhanced design change to the traditional TPM design that enables: 1) sharing cryptographic keys and data across a user's many devices, 2) a trusted clock synced with the cloud, and 3) high-performance NV storage of unlimited size. cTPM accomplished these goals by only adding a cloud seed shared between the device and the cloud. Together with the asynchronous communication channel set up, the seed allows cTPM to interact with the cloud to provide better support for cross-device trusted applications.

References

[1] B. Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *Proc. of CSFW*, Cape Breton, NS, 2001.

[2] C. Chen, H. Raj, S. Saroiu, and A. Wolman. ctpm: a cloud tpm for cross-device trusted applications. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, pages 187–201. USENIX Association, 2014.

[3] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda. TrInc: Small trusted hardware for large distributed systems. In *Proc. of NSDI*, Boston, MA, 2009.

[4] 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.

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

[6] A. Nguyen, H. Raj, S. Rayanchu, S. Saroiu, and A. Wolman. Delusional Boot: Securing Cloud Hypervisors without Massive Re-engineering. In *Proc. of EuroSys*, Bern, Switzerland, April 2012.

[7] 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.

[8] 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.

[9] 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 SOSP*, Cascais, Portugal, 2011.

[10] 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 SOSP*, Cascais, Portugal, 2011.