Pasture: Secure Offline Data Access Using Commodity Trusted Hardware

Ramakrishna Kotla, Microsoft Research, Silicon Valley
Tom Rodeheffer, Microsoft Research, Silicon Valley
Indrajit Roy[*], HP Labs, Palo Alto
Patrick Stuedi[*], IBM Research, Zurich
Benjamin Wester[*], Facebook, Menlo Park

Abstract

This paper presents Pasture, a secure messaging and logging library that enables rich mobile experiences by providing secure offline data access. Without trusting users, applications, operating systems, or hypervisors, Pasture leverages commodity trusted hardware to provide two important safety properties: *access-undeniability* (a user cannot deny any offline data access obtained by his device without failing an audit) and *verifiable-revocation* (a user who generates a verifiable proof of revocation of unaccessed data can never access that data in the future).

For practical viability, Pasture moves costly trusted hardware operations from common data access actions to uncommon recovery and checkpoint actions. We used Pasture to augment three applications with secure offline data access to provide high availability, rich functionality, and improved consistency. Our evaluation suggests that Pasture overheads are acceptable for these applications.

1 Introduction

Mobile experiences are enriched by applications that support offline data access. Decentralized databases [50], file systems [24], storage systems [31], and email applications [36] support disconnected operation to provide better mobility and availability. With the increasing use of mobile devices—such as laptops, tablets, and smart phones—it is important that a user has access to data despite being disconnected.

However, support for disconnected operation is at odds with security when the user is not trusted. A disconnected untrusted user (assumed to be in full control of the user device) could perform arbitrary actions on whatever data was available and subsequently lie about it. This tension between mobility and security limits the use of disconnected operation in many potentially useful scenarios, for example:

- *Offline video rental services:* Most web-based video rental services require users to be online to watch a streaming video. Some allow users to download movies and watch them offline but the movies must be purchased ahead of time, with no refund possible for unwatched movies. It would be useful if a user could rent some movies, download them when online (for example, in an airport), selectively watch some of them offline (on the plane), delete the unwatched movies, and then get a refund later (when back online after landing). Similar offline services could be provided for electronic books.
- *Offline logging and revocation:* Secure logging of offline accesses is required by law in some cases. For example, HIPAA [48] mandates that (offline) accesses to confidential patient information be logged securely to enable security audits so as to detect and report privacy violations due to unauthorized accesses. Furthermore, accidental disclosures [23] to unauthorized entities are not exempt from civil penalties under HIPAA. Hence, it is also important to enable verifiable revocation of (unread) data from an unintended receiver's device in order to mitigate liability arising out of accidental disclosures.

Support for secure offline data access raises two important security problems: How do you know that an untrusted user is not lying about what data was accessed while offline? And, if the user claims data has been deleted, how do you know that he did not secretly keep a copy for later access?

These problems cannot be solved simply by using encryption because the untrusted user must be able to get offline access to the decryption keys in order to read the data. Rather, it is an issue of (1) securely detecting and logging access to the decryption keys when data is accessed and (2) securely retracting unused keys to prevent future access when unaccessed data is deleted. These problems are hard because the untrusted user is disconnected when he makes offline accesses and he is in full physical control of his device.

Indeed, these problems limit application functionality and deployability. Consequently, as observed above, current online services provide restricted offline functionality that does not allow a refund for downloaded but (allegedly) unaccessed movies or books.

Fortunately, recent advances have resulted in widespread availability of commodity trusted hardware in the

[*] Work done while at Microsoft Research, Silicon Valley.

form of Trusted Platform Modules (TPMs) [51] and secure execution mode (SEM) extensions (Intel's TXT [22] and AMD's SVM [1] technology) in many modern day laptops, tablets, and PCs [17].

Secure offline data access using trusted hardware. We present *Pasture*, a secure messaging and logging library, that leverages commodity trusted hardware to overcome the above problems by providing following safety properties:

- *Access undeniability:* If a user obtains access to data received from a correct sender, the user cannot lie about it without failing an audit. (Destruction or loss of the user device automatically fails an audit.)
- *Verifiable revocation:* If a user revokes access to unaccessed data received from a correct sender and generates a verifiable proof of revocation, then the user did not and cannot ever access the data.

Pasture uses a simple yet powerful *bound key* TPM primitive to provide its safety properties. This primitive ensures access undeniability by releasing the data decryption key only after the access operation is logged in the TPM. It provides verifiable revocation by permanently destroying access to the decryption key and generating a verifiable proof of revocation if a delete operation is logged in the TPM instead.

Making secure offline data access practical. Flicker [34] and Memoir [37] have demonstrated the use of TPMs and SEM to run trusted application code on untrusted devices in isolation from the OS and hypervisor. However, using SEM requires disabling interrupts and suspending all but one core, which can result in poor user responsiveness, resource underutilization, and higher overhead. Furthermore, these systems are vulnerable to memory and bus snooping attacks [16, 21]. Pasture avoids these drawbacks by carefully using trusted hardware operations to ensure practicality without compromising on safety.

First, unlike Flicker and Memoir, Pasture does not use SEM for the common case data access operations, and thus provides better user interaction and improved concurrency by enabling interrupts and cores. Perhaps surprisingly, we found that Pasture could maintain its safety properties even though SEM is limited to the uncommon operations of recovery and checkpoint.

Second, similar to Memoir, we significantly reduce overhead and improve durability by limiting NVRAM and non-volatile monotonic counter update operations (which are as slow as 500 ms and have limited lifetime write cycles [37]) to uncommon routines and not using them during the regular data access operations.

Contributions. We make two key contributions in this paper. First, we present the design and implementation of Pasture, providing secure and practical offline data access using commodity trusted hardware. We wrote a formal specification [40] of Pasture and its safety proofs in TLA+, checked the specification with the TLC model checker [27], and mechanically verified the proofs using the TLA+ proof system. Our evaluation of a Pasture prototype shows that common case Pasture operations such as offline data access takes about 470 ms (mainly due to the TPM decryption overhead) while offline revocation can be as fast as 20 ms.

Second, we demonstrate the benefits of Pasture by (a) providing rich offline experiences in video/book rental services, (b) providing secure logging and revocation in healthcare applications to better handle offline accesses to sensitive patient health information, and (c) improving consistency in decentralized data sharing applications [2, 31, 50] by preventing read-denial attacks.

2 Overview and Approach

The goal of Pasture is to improve mobility, availability, and functionality in applications and services by allowing untrusted users to download data when online and make secure offline accesses to data. In this section, we state our requirements, review features of commodity trusted hardware, define our adversary, and present our approach.

2.1 Requirements

(1) Access undeniability. The access-undeniability property prevents an untrusted node[1] from lying about offline accesses to data sent by *correct nodes*. It is beyond the scope of this paper to address a more general problem of detecting or preventing data exchanges between colluding, malicious nodes. (For example, a malicious user could leak data to another malicious user during an unmonitored phone conversation.)

(2) Verifiable revocation. While access undeniability prevents users from lying about past data accesses, the verifiable-revocation property allows a user to permanently revoke access to unaccessed data. Furthermore, the user can supply a proof that its access was indeed securely revoked. The user will not be able to decrypt and read the data at a later time even if he keeps a secret copy of the encrypted data.

(3) Minimal trusted computing base (TCB). To defend against an adversary who has full physical access to the device when offline, we want to have a small TCB. Hence, we do not trust the hypervisor, OS, or the application to provide Pasture's safety properties. We do not want to trust the bus or memory to not leak any information as hardware snooping attacks [16, 21] are possible in our setting. However, we have to trust something, since it is impossible to track a user's offline actions without trusting anything on the receiver device.

[1] We use the terms "node" and "user" interchangeably to refer to an untrusted entity that takes higher level actions on data.

(4) Low cost. We want to keep the costs low by using only commodity hardware components.

2.2 Commodity trusted hardware

Pasture exploits commodity trusted hardware to meet its requirements. Here we explain some of the key features we use and refer readers to the textbook [5] for details. TPMs [51] are commodity cryptographic coprocessors already shipped in more than 200 million PCs, tablets and laptops [17]. TPMs are designed to be tamper-resistant and cannot be compromised without employing sophisticated and costly hardware attacks. TPMs are currently used for secure disk encryption [4] and secure boot [8] in commodity OS.

Keys and security. Each TPM has a unique identity and a certificate from the manufacturer (Infineon for example) that binds the identity of the TPM to the public part of a unique endorsement key (EK). The private part of EK is securely stored within the TPM and never released outside. For privacy, TPMs use internally generated Attestation Identity Keys (AIKs) for attestations and AIKs are certified by trusted privacy CAs after confirming that they are generated by valid TPMs with proper EK certificates. It is cryptographically impossible to spoof hardware TPM attestations and emulate it in software. TPMs can securely generate and store other keys, make attestations of internal state, sign messages, and decrypt and encrypt data.

Registers and remote attestation. TPMs have volatile registers called Platform Configuration Registers (PCRs) which can be updated only via the TPM_Extend operation, which concatenates a value to the current value, and then overwrites the PCR with a SHA1 hash of the concatenated value. Intuitively, a PCR serves as a chained SHA1 hash of events in a log. TPMs also support remote attestation using a TPM_Quote operation which returns the signed contents of one or more PCRs along with a specified nonce. TPMs (v1.2) typically have 24 PCRs and we refer interested readers to other sources [5, 51] for details.

Bound keys. TPMs provide a TPM_CreateWrapkey operation that creates a bound public-private key pair. The encryption key can be used anywhere but the corresponding decryption key is shielded and can be used only in the TPM when specified PCRs contain specified values.

Transport sessions. TPMs support a form of attested execution using TPM transport sessions. Operations within a transport session are logged and signed so that a verifier can securely verify that a given TPM executed a specified sequence of TPM operations with specified inputs and outputs.

Non-volatile memory and counters. TPMs provide a limited amount of non-volatile memory (NVRAM) which can be used to persist state across reboots. A region of NVRAM can be allocated and protected so that it can be accessed only when specified PCRs contain specified values. TPMs also provide non-volatile monotonic counters which can be updated only by an increment operation (TPM_IncrementCounter).

Secure Execution Mode (SEM). AMD and Intel processors provide special processor instructions to securely launch and run trusted code in isolation from DMA devices, the hypervisor and the OS. Roughly speaking, the processor sets DMA memory protection so that DMA devices cannot access the trusted code, disables interrupts and other cores to prevent other software from taking control or accessing the trusted code, resets and extends a special $\mathrm{PCR}_{\mathrm{SEM}}$ register with the SHA1 hash of the trusted code, and then starts executing the trusted code.

The AMD and Intel details are different, but in each case the reset of $\mathrm{PCR}_{\mathrm{SEM}}$ produces a value that is not the same as the initial value produced by a reboot, and hence the result obtained by extending $\mathrm{PCR}_{\mathrm{SEM}}$ by the SHA1 hash of the trusted code is cryptographically impossible to produce in any other way. This PCR value can be specified to restrict access to secrets (that are encrypted by bound keys) or NVRAM locations to the trusted code only.

When the trusted application finishes its execution, the SEM exit code extends $\mathrm{PCR}_{\mathrm{SEM}}$ to yield access privileges, scrubs memory of any secrets it wants to hide, then reenables DMA, interrupts, and other cores. Flicker [34] demonstrated how to use SEM to execute arbitrary trusted code.

2.3 Adversary model

The adversary's goal is to violate safety. We do not consider violations of liveness, since the adversary could always conduct a denial-of-service attack by simply destroying the device. Violating safety means to violate either access undeniability or verifiable revocation. The adversary wins if he can obtain access to data from a correct sender and then subsequently survive an audit while claiming that access was never obtained. The adversary also wins if he can produce a valid, verifiable proof of revocation for data from a correct sender and yet at some time, either before or after producing the proof, obtain access to the data.

The adversary can run arbitrary code in the OS, hypervisor, or application. The adversary controls power to the device and consequently can cause reboots at arbitrary points in time, even when the processor is executing in secure execution mode. As opposed to Memoir [37], we assume that the adversary can perform hardware snooping attacks on the main memory [16] or CPU-memory bus [21] to deduce the contents of main memory at any time. We also assume that the adversary can snoop on the CPU-TPM bus.

However, we assume that the adversary cannot extract secrets from or violate the tamper-resistant TPM and compromise the processor's SEM functionality. We also assume that the adversary cannot break cryptographic secrets or find collisions or preimages of cryptographic hashes. We assume the correctness of processor's CPU and the trusted Pasture code that runs in SEM.

In Pasture, we ensure that a sender's data is accessed securely on a receiver node, and it is not our goal to keep track of who accessed the data or for what purpose. Other techniques [33, 34, 55] can be used to provide isolation across various entities on the receiver node based on the receiver's trust assumptions. Pasture's safety guarantees are for the sender and are independent of the receiver's trust assumptions.

Since an audit is possible only when the receiver is online, the adversary can prevent an audit by disconnecting or destroying the device. We let applications decide how to deal with long disconnections and device failures. In the offline video rental scenario, for example, the fact that a node might have been disconnected for a long time is irrelevant, because the user has already paid for the movies and can get refunds only by providing verifiable proofs of revocation.

To be conservative, an application cannot assume that data has been destroyed in a device failure or lost during a long network disconnection. In such situations it is impossible to tell comprehensively what data has been accessed. In spite of this, we guarantee that if there is a valid, verifiable proof of revocation of data received from a correct sender, then the adversary can never access that data.

2.4 Pasture's approach

We carefully use TPM and SEM operations to provide secure offline data access to meet Pasture's requirements.

(1) Summary log state in a PCR. Pasture uses a Platform Configuration Register $\mathrm{PCR}_{\mathrm{APP}}$ to capture a cryptographic summary of all decisions to obtain access or revoke access to data. Since a PCR can be updated only via the TPM_Extend operation, it is cryptographically impossible to produce a specified value in a PCR in any other way than by a specified sequence of extensions.

(2) Minimal use of SEM and NV operations. By carefully mapping the process of obtaining and revoking access onto TPM primitives, Pasture avoids using SEM or NV updates during normal operation and limits their use to bootup and shutdown.

(3) Prevent rollback and hardware snooping attacks. Since Pasture state is stored in a volatile PCR, an adversary could launch a rollback attack [37] to attempt to retract past offline actions. The rollback attack would proceed by rebooting the system, which resets $\mathrm{PCR}_{\mathrm{APP}}$ to its initial value, followed by re-extending $\mathrm{PCR}_{\mathrm{APP}}$ to a valid, but older state. The adversary would truncate the full (untrusted) log to match the state in $\mathrm{PCR}_{\mathrm{APP}}$.

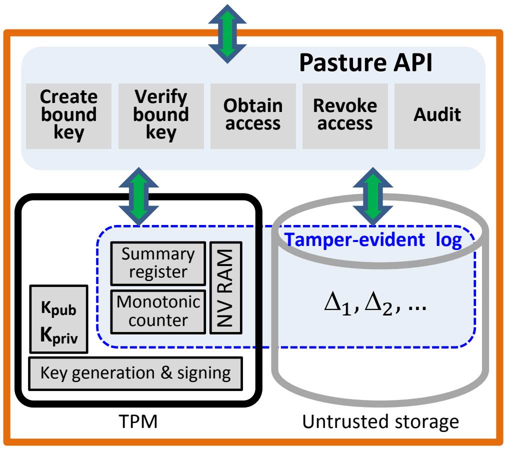


Figure 1: Pasture architecture.

Memoir prevents this attack by incorporating a secret into each PCR extension. The adversary cannot perform re-extensions without knowing the secret, which is shielded so that it is accessible only to the Memoir trusted code running in SEM. This is a reasonable approach in Memoir, which handles general applications and runs every operation in SEM. But since we grant the adversary the power to snoop on hardware busses, this defense is insufficient for Pasture.

Pasture prevents this attack by exploiting the fact that SEM is not needed for normal operations. Instead of making the required contents of $\mathrm{PCR}_{\mathrm{APP}}$ impossible for the adversary to reproduce, Pasture conjoins a requirement that $\mathrm{PCR}_{\mathrm{SEM}}$ contain a value that is cryptographically impossible to produce except by executing Pasture's trusted reboot recovery routine and verifying that $\mathrm{PCR}_{\mathrm{APP}}$ has been restored to the latest value.

Pasture uses Memoir's approach of saving the latest value of $\mathrm{PCR}_{\mathrm{APP}}$ in protected NVRAM on shutdown so that its correct restoration can be verified on the subsequent reboot.

3 Pasture Design

Figure 1 shows the high-level architecture of Pasture. Each node runs a Pasture instance which is uniquely identified by a public/private key pair securely generated and certified (using AIK) by its corresponding TPM. All proofs and messages generated by a Pasture instance are signed using the private part. Receivers verify signatures using the public part. Since the private part of the key is protected inside the TPM, it is impossible for an adversary to spoof Pasture's proofs or messages.

Each Pasture instance maintains a tamper-evident append-only log of decisions $\Delta_1, \Delta_2, \ldots$ about offline decisions to access or revoke a key. The full log is kept in the node's untrusted storage and a cryptographic summary of the log is maintained inside the TPM on a PCR. The application running on a Pasture instance uses the Pasture API to *Create* and *Verify* bound encryption keys

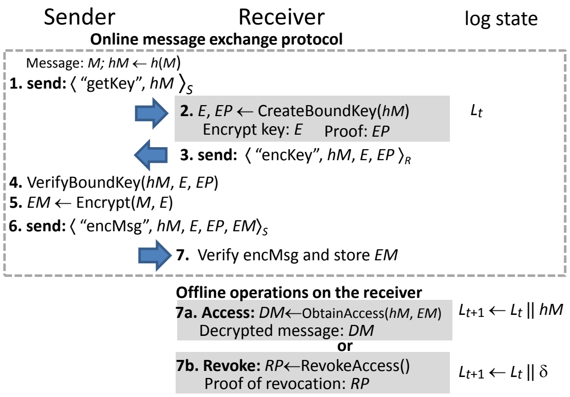

Sender
Receiver
log state
Online message exchange protocol
Message: M; hM ← h(M)
1. send: ⟨ "getKey", hM ⟩S
2. E, EP ← CreateBoundKey(hM)
Encrypt key: E Proof: EP
Lt
3. send: ⟨ "encKey", hM, E, EP ⟩R
4. VerifyBoundKey(hM, E, EP)
5. EM ← Encrypt(M, E)
6. send: ⟨ "encMsg", hM, E, EP, EM⟩S
7. Verify encMsg and store EM
Offline operations on the receiver
7a. Access: DM←ObtainAccess(hM, EM)
Decrypted message: DM
Lt+1 ← Lt || hM
or
7b. Revoke: RP←RevokeAccess()
Proof of revocation: RP
Lt+1 ← Lt || δ

Figure 2: Pasture data transfer protocol, offline operations, and log state. Shaded regions represent TPM operations. Protocol messages are signed by their corresponding message sender (S or R).

during the data transfer protocol, to *Obtain* and *Revoke* access to the corresponding decryption keys based on offline decisions, and to respond to an *Audit*.

Many of the following subsections contain **implementation details**. Skipping these details on the first reading may help the reader.

3.1 Data transfer protocol

Figure 2 shows the secure data transfer protocol used by a sender to transfer encrypted data to a receiver. When a sender wants to send data M to a receiver, the sender gets an (asymmetric) encryption key E generated by the receiver's TPM and sends the encrypted data *EM*. The receiver then can make offline decision to access the data M by decrypting *EM* or revoke access to M by permanently deleting access to the decryption key in its TPM and generating a proof of revocation.

We describe the protocol with a single sender below and defer discussion of concurrent data transfers from multiple senders to §3.6. At the beginning of the protocol, the receiver's log state is L_t. We use the subscript t to indicate the state before creating entry $t+1$ in the log. The subscript $t+1$ indicates the next state.

In step 1, the sender provides the cryptographic hash *hM* $= h(M)$ when requesting an encryption key.

In step 2, the receiver generates a key pair bound to a hypothetical future log state $L_t||$*hM*, in which the current log L_t is extended by a decision to obtain access to the bound decryption key. The cryptographic hash *hM* represents this decision. The TPM only permits decryption using the bound key when the log has the hypothesized state. Pasture creates a proof *EP* that the bound key was generated by the receiver's TPM in the proper manner using a transport session.

In step 3 the proof *EP* and the encryption key E are returned to the sender in the "encKey" message.

In step 4, The sender checks the proof to verify that the receiver acted properly. If the "encKey" message is malformed or if the proof is not valid then the sender aborts the protocol and does not interact further with the faulty receiver. In this way a correct sender can protect itself against a malicious receiver.

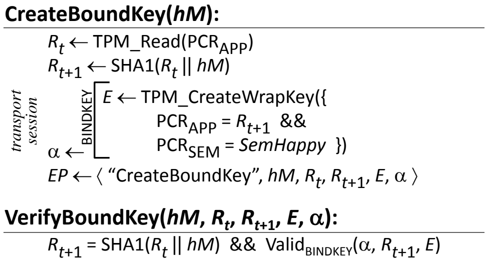


Figure 3: Create and verify bound key operations.

In step 5, the sender encrypts its message using E. In step 6, the encrypted data *EM* is sent to the receiver, along with the original hash *hM*, the encryption key E, and the receiver's proof *EP*.

In step 7, after receiving "encMsg" and verifying that it is properly signed by the sender, the receiver checks that *hM*, *E*, and *EP* match with the corresponding values it sent in its "encKey" message. Then the receiver stores the "encMsg" with the encrypted data *EM* on its local storage. The receiver can later make an offline decision to obtain access to the bound decryption key (and thus obtain access to the sender's data) or to revoke access.

In case of communication failures, neither the sender nor the receiver need to block because they can always discard their current state and restart.

Implementation details. For simplicity, we described the protocol as if M is the actual data X but we use the standard practice [35] of allowing the sender to choose a nonce symmetric key K_{Sym} to encrypt X and then encrypting $M = \langle K_{Sym}, h(X)\rangle$ using the asymmetric key acquired in this protocol. Securely obtaining or revoking access to M is equivalent to obtaining or revoking access to the actual data X. Note that K_{Sym} also acts as a unique and random nonce to prevent a faulty receiver from inferring M from the hash hM without decrypting the message even if the receiver has seen data X in prior exchanges with this or other senders.

Figure 3 shows the implementation of the `CreateBoundKey` and `VerifyBoundKey` operations. Pasture exploits the TPM primitive TPM_CreateWrapKey, which creates a key pair in which the decryption key is usable only when specified PCRs contain specified values. Pasture keeps a cryptographic summary of the log in a PCR called PCR_{APP}.

`CreateBoundKey` reads the current summary value R_t from PCR_{APP}, computes what the new summary value R_{t+1} would be if *hM* were appended to the log, and then invokes TPM_CreateWrapKey to create a key pair with the decryption key usable only when

```
ObtainAccess(hM, EM):
    append $hM$ to full log
    TPM_Extend(PCR$_{APP}$, $hM$)
    $DM \leftarrow$ TPM_Unbind($EM$)

RevokeAccess():
    $R_t \leftarrow$ TPM_Read(PCR$_{APP}$)
    append $\delta$ to full log
    TPM_Extend(PCR$_{APP}$, $\delta$)
    $R'_{t+1}, S'_{t+1}, \alpha \leftarrow$ TPM_Quote(PCR$_{APP}$, PCR$_{SEM}$)
    $RP \leftarrow \langle$ "RevokeAccess", $\delta, R_t, R'_{t+1}, S'_{t+1}, \alpha \rangle$
```

Figure 4: Obtain and revoke access operations.

$\mathrm{PCR}_{\mathrm{APP}}$ contains R_{t+1}. (The additional constraint that $\mathrm{PCR}_{\mathrm{SEM}}$ contains *SemHappy* is discussed in §3.4.) TPM_CreateWrapKey is invoked inside a TPM transport session, which provides an attestation α signed by the TPM that the BINDKEY sequence of TPM operations were performed with the indicated inputs and outputs.

The sender uses VerifyBoundKey to check the contents of the proof *EP*. Note that there is no need to verify that R_t is a correct cryptographic summary of the receiver's log. The attestation α proves that E is bound to R_{t+1}, which is the correct extension from R_t by *hM*. It is cryptographically impossible to find any other value from which an extension by *hM* would produce R_{t+1}.

3.2 Secure offline access and revocation

To obtain access, as shown in step 7a of Figure 2, the receiver irrevocably extends its log by *hM*. The TPM now permits decryption using the bound key, and the decrypted message *DM* can be obtained. A faulty sender could play a bait-and-switch trick by sending an encrypted text for a different data than initially referenced in its "getKey" message, but the receiver can catch this by noticing that $hM \neq h(DM)$. If the sender is faulty, the receiver can form a proof of misbehavior by exhibiting *DM* and the sender's signed "encMsg" message, which includes *hM*, E, *EP*, and *EM*. Any correct verifier first verifies that the receiver is not faulty by checking that $Encrypt(DM, E) = EM$. Then the verifier checks that $hM \neq h(DM)$, which proves that the sender is faulty. In this way a correct receiver can protect itself against a malicious sender.

To revoke access, in step 7b, the receiver irrevocably extends its log by δ, a value different from *hM*. This makes it cryptographically impossible ever to attain the state in which the TPM would permit decryption using the bound key. In effect, the bound decryption key has been revoked and the receiver will never be able to decrypt *EM*. Pasture constructs a proof *RP* of this revocation, which any correct verifier can verify.

Implementation details. Figure 4 shows the implementation of the ObtainAccess and RevokeAccess operations. Step 7a calls ObtainAccess to obtain offline data access. ObtainAccess appends *hM* to the full log on untrusted storage and extends the cryptographic summary maintained in $\mathrm{PCR}_{\mathrm{APP}}$. Since $\mathrm{PCR}_{\mathrm{APP}}$ now contains the required summary value, the TPM permits use of the decryption key to decrypt the data, which is performed via the TPM_Unbind primitive.

```
Audit(nonce):
    $R_t, S_t, \alpha \leftarrow$ TPM_Quote(PCR$_{APP}$, PCR$_{SEM}$, $nonce$)
    $AP \leftarrow \langle$ "Audit", full log, $R_t, S_t, nonce, \alpha \rangle$
```

Figure 5: Audit operation.

Step 7b calls RevokeAccess to revoke data access. RevokeAccess appends δ to the log and extends the cryptographic summary accordingly. Since $\delta \neq hM$, this produces a summary value $R'_{t+1} \neq R_{t+1}$. Since it is cryptographically impossible to determine any way of reaching R_{t+1} except extending from R_t by *hM*, this renders the decryption key permanently inaccessible. Pasture uses TPM_Quote to produce an attestation α that $\mathrm{PCR}_{\mathrm{APP}}$ contains R'_{t+1}. (The simultaneous attestation that $\mathrm{PCR}_{\mathrm{SEM}}$ contains S'_{t+1} is discussed in §3.4.) The exhibit of the prior summary value R_t along with R'_{t+1} and the attestation α proves that R_{t+1} is unreachable.

There are several ways in which the code in RevokeAccess can be optimized. First, Pasture can skip the TPM_Read operation by tracking updates to $\mathrm{PCR}_{\mathrm{APP}}$ with the CPU. Second, if the proof of revocation is not needed immediately, Pasture can delay executing the TPM_Quote until some later time, possibly coalescing it with a subsequent TPM_Quote. A multi-step sequence of extensions from R_t to the current attested value of $\mathrm{PCR}_{\mathrm{APP}}$, in which the first step differs from the bound key value R_{t+1}, is cryptographically just as valid as a proof of revocation as a single-step sequence. Coalescing is a good idea, since TPM_Quote involves an attestation and hence is fairly slow as TPM operations go.

3.3 Audit

A verifier can audit a receiver to determine what decisions the receiver has made. The receiver produces a copy of its full log along with a proof signed by the TPM that the copy is current and correct. Hence, a faulty receiver cannot lie about its past offline actions.

Implementation details. Figure 5 shows the implementation of the Audit operation, which computes the response *AP* to an audit request. *AP* contains a copy of the entire log along with an attestation α of the current summary value R_t contained in $\mathrm{PCR}_{\mathrm{APP}}$. The attestation also includes the nonce sent by the auditor to guarantee freshness. (The simultaneous attestation that $\mathrm{PCR}_{\mathrm{SEM}}$ contains S_t is discussed in §3.4.) Any correct verifier can check the attestation and check that R_t is the correct cryptographic summary value for the purported log contents, and thereby determine exactly what decisions $\Delta_1, \Delta_2, \ldots$ the audited node has made.

3.4 Dealing with reboots

The main difficulty faced by Pasture is dealing with reboots. Since decryption keys are bound to the log summary value contained in $\mathrm{PCR}_{\mathrm{APP}}$, anything that the adversary can do to break Pasture's guarantees of access undeniability and verifiable revocation must involve rebooting the node. Rebooting the node causes the TPM to reset PCRs to their initial values, which opens the door to rollback attacks.

Pasture's solution is inspired by Memoir-Opt [37] and in §2.4 we outlined the novel aspects of our approach. Since $\mathrm{PCR}_{\mathrm{APP}}$ is volatile, an adversarial reboot will cause its contents to be lost. So, like Memoir-Opt, Pasture uses a protected module containing a checkpoint routine that runs in SEM and saves the latest contents of $\mathrm{PCR}_{\mathrm{APP}}$ in a region of TPM NVRAM accessible only to the protected module. The checkpoint routine is installed to run during system shutdown and as part of a UPS or battery interrupt handler [37]. Note that our system does not assume correct operation of such mechanisms for safety.

As a node cycles through shutdown, reboot, recovery, and operation, it is important to keep track of where the current log summary value is located. During operation, it lives in $\mathrm{PCR}_{\mathrm{APP}}$. Shutdown moves it from $\mathrm{PCR}_{\mathrm{APP}}$ to Pasture's protected NVRAM region. Reboot and recovery moves it back into $\mathrm{PCR}_{\mathrm{APP}}$. To keep track of this, Pasture's protected NVRAM region contains two fields: R and *current*. R is used to hold a log summary value and *current* is a boolean flag indicating that the value in R is indeed current.

The checkpoint routine sets *current* to TRUE after it has copied $\mathrm{PCR}_{\mathrm{APP}}$ into R. On reboot, Pasture recovers by reading the full log $\Delta_1, \Delta_2, \ldots$ and extending $\mathrm{PCR}_{\mathrm{APP}}$ by each entry in turn. Then Pasture uses SEM to enter its protected module and check that the value recorded in the NVRAM is indeed current and that it matches the value contained in $\mathrm{PCR}_{\mathrm{APP}}$. If so, the recovery routine sets *current* to FALSE, indicating that the current log summary value now lives in $\mathrm{PCR}_{\mathrm{APP}}$.

Observe that failure to run the checkpoint routine before a reboot will erase $\mathrm{PCR}_{\mathrm{APP}}$ but leave *current* $=$ FALSE, a state from which recovery is impossible. This is a violation of liveness, but not of safety.

Implementation details. Figure 6 shows the implementation of the `Recover` and `Checkpoint` operations. `Recover` extends $\mathrm{PCR}_{\mathrm{APP}}$ with each entry Δ on the full log and then enters SEM to check that the saved copy in NVRAM is current and matches the value in $\mathrm{PCR}_{\mathrm{APP}}$. If a shutdown happens precisely at the wrong time in the middle of `ObtainAccess` or `RevokeAccess`, it is possible for the full log to contain one final entry not represented in the saved log summary value. In this case, that final decision was not committed and the implementation described here will fail to recover. However, it is a simple matter to add a SEM routine that merely reads nv.R so that `Recover` can tell whether or not it should remove the final entry from the full log. If recovery is successful, *current* is set to FALSE, indicating that the current log summary value now lives in $\mathrm{PCR}_{\mathrm{APP}}$.

```
Recover():
FOR EACH entry $\Delta$ on full log: TPM_Extend($\mathrm{PCR}_{\mathrm{APP}}$, $\Delta$)
secure execution mode
    IF nv.current && nv.$R$ = TPM_Read($\mathrm{PCR}_{\mathrm{APP}}$)
    THEN
        nv.current ← FALSE
        TPM_Extend($\mathrm{PCR}_{\mathrm{SEM}}$, Happy)
    ELSE
        TPM_Extend($\mathrm{PCR}_{\mathrm{SEM}}$, Unhappy)

Checkpoint():
transport session
    SEAL
        $R_t$ ← TPM_Read($\mathrm{PCR}_{\mathrm{APP}}$)
        $S_t$ ← TPM_Read($\mathrm{PCR}_{\mathrm{SEM}}$)
        $C_t$ ← TPM_ReadCounter(CTR)
    $\alpha$ ← TPM_Extend($\mathrm{PCR}_{\mathrm{SEM}}$, Unhappy)
secure execution mode
    nv.$R$ ← $R_t$
    IF $\mathrm{Valid}_{\mathrm{SEAL}}(\alpha, R_t, S_t, C_t)$ && $S_t$ = SemHappy
        && $C_t$ = TPM_ReadCounter(CTR)
    THEN
        TPM_IncrementCounter(CTR)
        nv.current ← TRUE
    TPM_Extend($\mathrm{PCR}_{\mathrm{SEM}}$, Unhappy)
```

Figure 6: Recover and checkpoint operations.

However, there is an additional important detail: how do normal operations know that the current log summary lives in $\mathrm{PCR}_{\mathrm{APP}}$. These operations do not use SEM, so they cannot access *current*. Moreover, checking *current* is not enough, because they need to know that $\mathrm{PCR}_{\mathrm{APP}}$ was correctly restored on the *most recent* reboot. An adversary could mount a rollback attack by crashing, rebooting and partially re-extending $\mathrm{PCR}_{\mathrm{APP}}$, which would leave *current* as FALSE from the prior reboot.

Pasture exploits secure execution mode to prevent this attack. When the CPU enters secure execution mode, the TPM resets $\mathrm{PCR}_{\mathrm{SEM}}$ to -1 (different from its reboot reset value of 0) and then extends $\mathrm{PCR}_{\mathrm{SEM}}$ by the cryptographic hash of the protected module. For Pasture's protected module, this produces a result *SemProtected* in $\mathrm{PCR}_{\mathrm{SEM}}$ that is cryptographically impossible to produce in any other way. The constraint that $\mathrm{PCR}_{\mathrm{SEM}}$ contains *SemProtected* is used to control access to Pasture's protected TPM NVRAM.The adversary cannot undetectably modify Pasture's NVRAM, because TPM_DefineSpace resets the protected NVRAM locations whenever their access control is changed.

When execution of Pasture's protected module is finished, it extends $\mathrm{PCR}_{\mathrm{SEM}}$ to disable access to the protected NVRAM. Pasture defines two constants that it may use for this extension. *Happy* is used during recovery after a reboot if $\mathrm{PCR}_{\mathrm{APP}}$ has been correctly restored. *Unhappy* is used in all other cases. We define

SemHappy $=$ SHA1(*SemProtected*$||$*Happy*), which is the value that $\mathrm{PCR_{SEM}}$ will contain if the recovery is correct. Since PCRs are volatile, any adversarial attempt to reboot the node and reinitialize $\mathrm{PCR_{APP}}$ will also reinitialize $\mathrm{PCR_{SEM}}$. Pasture maintains the invariant that whenever $\mathrm{PCR_{SEM}}$ $=$ *SemHappy*, $\mathrm{PCR_{APP}}$ contains the correct cryptographic summary of the log and decisions to obtain access or revoke access can be made. (Of course, adversarial entries in the log are permitted, but they cannot be rolled back.)

Since `CreateBoundKey` binds the key to require both $\mathrm{PCR_{APP}} = R_{t+1}$ and $\mathrm{PCR_{SEM}}$ $=$ *SemHappy*, the decryption key will only be useable if $\mathrm{PCR_{APP}}$ was correctly restored on the most recent reboot. Since `RevokeAccess` and `Audit` include $\mathrm{PCR_{SEM}}$ in their TPM_Quote, a verifier can verify that the quoted $\mathrm{PCR_{APP}}$ could only have been extended from a value that was correctly restored on the most recent reboot.

`Checkpoint` uses SEM to save the current value of $\mathrm{PCR_{APP}}$ in the NVRAM and set *current* to TRUE. However, there is a difficulty. Before `Checkpoint` saves the current value of $\mathrm{PCR_{APP}}$, it needs to know that $\mathrm{PCR_{APP}}$ was correctly restored on the most recent reboot; otherwise `Checkpoint` itself would be vulnerable to a rollback attack. `Checkpoint` has to perform its checking and saving activities in SEM, so that the adversary cannot tamper with them. But the way of knowing that $\mathrm{PCR_{APP}}$ was correctly restored is to check $\mathrm{PCR_{SEM}}$ $=$ *SemHappy*, and this information is erased by entering SEM.

The solution is to get a simultaneous attestation α of $\mathrm{PCR_{APP}}$ and $\mathrm{PCR_{SEM}}$ before entering SEM. Then, once in SEM, α can be checked, which proves that $\mathrm{PCR_{APP}}$ contained R_t when $\mathrm{PCR_{SEM}}$ contained *SemHappy*.

Unfortunately, there is another vulnerability: how do we know this is the *most recent* such α, and not some earlier one from the adversary. In defense, `Checkpoint` uses a TPM counter CTR whose value is read into α and then incremented in SEM once α is accepted as valid and current. This prevents any earlier α from being accepted.

There is yet a final vulnerability: how do we know that the adversary did not do anything bad between the time α was made and SEM was entered. For example, the adversary could make α, then extend $\mathrm{PCR_{APP}}$ to obtain access to a decryption key, then crash and reboot, re-extend $\mathrm{PCR_{APP}}$ back to where it was when α was made, and then finally enter SEM in `Checkpoint`. To prevent this, `Checkpoint` extends $\mathrm{PCR_{SEM}}$ inside α, which erases *SemHappy*, which makes $\mathrm{PCR_{APP}}$ useless for any attempt to obtain or revoke access until the next successful recovery. The above steps are performed in a transport session in the SEAL subroutine as shown in Figure 6.

3.5 Log truncation

Pasture allows applications to truncate logs in order to reduce storage, auditing, and recovery overheads. Truncation proceeds in several steps. (1) The subject node requests a trusted auditor to perform an audit and sign a certificate attesting to the subject node's current cryptographic log summary. The auditor has to be available at this time but an application could arrange for any desired number of trusted auditors. For example, the video service provider could act as an auditor for the offline video application. (2) The certificate is returned to the subject node, which discards all entries in its full log, creates a single log entry containing the certificate, and then performs a version of `Checkpoint` that verifies the certificate is current and if so saves the cryptographic summary of the new log in the NVRAM. (3) Then the subject node reboots to reinitialize $\mathrm{PCR_{APP}}$.

Observe that truncating the log implicitly revokes access to a decryption key whose decision was pending. However, since in such a case no explicit decision was made to revoke access, the normal way of obtaining a proof of revocation is not available. If such proofs are important to the application, it should make explicit revocation decisions before truncating the log.

3.6 Handling concurrent messages

The basic protocol (§3.1) constrains the receiver to handle one sender's message at a time. Here we describe how to support multiple outstanding messages.

First, it is easy to extend the design to employ multiple PCRs. Current TPMs support 13 unreserved PCRs that Pasture can use. The design is modified to track the usage of these PCRs, allocating a free one in `CreateBoundKey`, passing it through the protocol in the "encKey" and "encMsg" messages, and freeing it after the decision is made to obtain or revoke access. Logically, a separate log is used for each PCR. `Audit` is extended to copy all the logs and quote all of the PCRs.

Second, in situations where the number of PCRs is insufficient, there is nothing to prevent a receiver from performing steps 2-3 in parallel with any number of senders using the same PCR, creating keys bound to different values of the next state. Of course, with the same PCR, only one sender's message can be accessed at step 7a. The other messages effectively are revoked. The receiver can ask those senders to retry using a new bound key.

Given multiple concurrent "getKey" requests and only one available PCR, the receiver could form a speculative decision tree multiple steps into the future. For each message, the receiver would generate multiple keys, binding each key to a different sequence of possible prior decisions. Each sender encrypts its message (actually, just its message's symmetric key) with all the keys so that the receiver can decide offline about access to the messages in any order it desired. So that the receiver can prove that a key was revoked because the receiver failed to follow the speculated sequence of decisions, each key's revocation proof RP would have to show the entire sequence starting from the receiver's current log state. Of course, the number of keys per message explode with the number of pending decisions, so this approach would be viable for only a very small number of outstanding messages.

3.7 Correctness

Using the TLA+ language and proof system [6,27], we wrote a formal specification of Pasture and mechanically verified a proof of correctness [40]. The correctness of Pasture when there are no reboots is trivial as it follows from the properties of bound keys and PCRs. Preventing rollback attacks in the presence of reboots is critical. The following invariants ensure correctness:

- If $\mathrm{PCR}_{\mathrm{SEM}} = \mathit{SemHappy}$ then *current* $=$ FALSE, $\mathrm{PCR}_{\mathrm{APP}}$ contains the current log summary value, and there exists no acceptable SEAL attestation.
- If *current* $=$ TRUE then $\mathrm{PCR}_{\mathrm{SEM}} \neq \mathit{SemHappy}$, the NVRAM contains the current log summary value, and there exists no acceptable SEAL attestation.
- There exists at most one acceptable SEAL attestation.

These invariants are maintained by the use of *current*, the way $\mathrm{PCR}_{\mathrm{SEM}}$ is extended, and the reboot counter CTR.

Some comments on our methodology may be illuminating. After formulating a proof sketch to assure ourselves that Pasture was correct, we wrote a 19-page TLA+ specification. Several CPU-months spent on a large many-core server model-checking various configurations of this specification found no errors. To increase our confidence that we would have found errors if there were any, we then intentionally introduced some bugs and the resulting safety violations were easily detected.

Armed with confidence in the correctness of the specification, we then spent about two man-weeks writing a 68-page formal proof. The proof followed the reasoning of our original proof sketch, although with excruciating attention to detail, containing 1505 proof obligations. The TLA+ proof system checks them in half an hour.

Subsequently, we made a slight optimization to Pasture's operations, arriving at the version of Pasture described in this paper. It took only a few hours to revise the initial formal specification and proof to account for the optimization. The hierarchical structure of TLA+ proofs was a great benefit here, because the proof system highlighted precisely those few proof obligations that had to be revised in order to make the proof go through.

4 Applications

We use Pasture to prototype three applications with secure offline data access to (a) improve experience in a video rental service by allowing offline access to downloaded movies while enabling refunds for unwatched movies, (b) provide better security and mobility in a healthcare setting by allowing secure logging of offline accesses as required by HIPAA regulations, and (c) improve consistency in a decentralized storage system [2, 31, 50] by preventing read-denial attacks.

4.1 Video rental and healthcare

For wider deployability, we extended an email client application with Pasture to provide secure offline data access, and used the secure email as a transport and user interface mechanism to implement offline video rental and healthcare mockup application prototypes on top.

We implemented a generic Pasture add-in for Microsoft's Outlook email client [36]. Our video rental and health care applications are built on top of this secure email prototype to transfer data opportunistically to receivers when they are online, allow offline access to data, and permit remote auditing. The add-in calls the Pasture API (§3) to interact with the TPM.

The Pasture add-in allows senders to selectively encrypt sensitive attachments using the Pasture protocol. Users can choose messages that need the enhanced security properties while not paying overhead for the others. The add-in internally uses email to exchange Pasture protocol messages and acquires the encryption key from the receiver before sending the encrypted message.

On receiving a Pasture encrypted message, the user is given the option to access the attachment or delete it without accessing it. The user can use context (for example, the movie title, cast and a short description) included in the email body to make the decision. We assume that the sender is correct and motivated to include correct context. This assumption is reasonable for video rental and healthcare service providers. We also assume that the emails are signed to prevent spoofing attacks.

The user's decision to access or delete the attachment is permanently logged by Pasture. The Pasture add-in also provides an audit and truncate interface that a trusted entity can use to audit and truncate user logs

The Pasture-enhanced video rental service works as follows. When the user surfs the video rental service and selects movies for offline watching, he receives emails with encrypted symmetric keys as attachments. The encrypted movies are downloaded via https, which avoids sending the entire movie as an attachment. The user can watch movies offline by decrypting the attachments to extract the keys and then decrypting the movies. The user can revoke any movies not accessed. When the user comes back online, the video rental service provider audits the Pasture log to determine which movies were revoked and refunds the user accordingly.

Regulations [18, 48] in the healthcare industry impose fines [20] on healthcare providers if they allow unauthorized access to sensitive patient health information and they mandate that all accesses be logged securely. We used the Pasture email add-in to provide secure offline access to medical records, for example, when a nurse goes for a home visit.

Access undeniability ensures that offline accesses by the nurse are securely logged. Verifiable revocation allows the nurse to securely delete unaccessed records and prove it to the hospital. Verifiable revocation also helps hospitals in assessing and mitigating damages due to accidental disclosures [13, 23] by allowing them to retract emails after they are sent to unintended recipients by asking the receivers to delete the confidential email and send them back the proof of revocation.

4.2 Decentralized storage systems

Decentralized storage systems (such as Bayou [50] and Practi [2]) provide high availability by allowing nodes to perform disconnected update operations on local state when they are offline, send updates to other nodes opportunistically when there is connectivity, and read received updates from other nodes. Depot [31] builds upon Practi to provide a storage system that tolerates any number of malicious nodes at the cost of weakening the consistency guarantee. One attack Depot cannot prevent is a *read-denial* attack, in which a malicious node denies reading updates from correct nodes before making its own updates.

We used Pasture to build a decentralized storage system that prevents read-denial attacks, and thereby provides stronger consistency than Depot. Preventing read-denial is a simple consequence of access undeniability. In addition, where Depot detects *equivocation* [7, 28], in which a malicious node sends conflicting updates, Pasture prevents equivocation by attesting updates using the Pasture log (the same approach as A2M [7]). Formalizing our consistency guarantee is an avenue of future work.

5 Evaluation

We implemented Pasture in Windows 7 and evaluated the system on three different computers: an HP xw4600 3GHz Intel Core2 Duo with a Broadcom TPM, an HP Z400 2.67GHz Intel Quad Core with an Infineon TPM, and a Lenovo X300 1.2Ghz Intel Core2 Duo laptop with an Atmel TPM. We implemented everything except we were unable to port Flicker [34] and get SEM work on the HP machines because the HP BIOS disables the SENTER instruction, and we haven't completely ported Pasture to run on Atmel TPM although we ran some TPM microbenchmarks on it. Missing functionality of SEM does not affect our evaluation or conclusions because SEM is not needed for the common case data access operations. Furthermore, for the Pasture operations that would use SEM, as we show later, our measured overheads are already significantly greater than the cost of setting up the SEM environment [34, 37].

System	Isolation from malicious OS	Crash resilience	Invulnerable to snoop attack	No disabling of cores, interrupts per operation	No NV update per operation	Protected operations
Flicker	✓	×	×	×	✓	general
Memoir	✓	✓	×	×	✓	general
TrInc	✓	✓	✓	✓	×	attest
Pasture	✓	✓	✓	✓	✓	access revoke attest

Table 1: Comparison of trusted hardware based systems.

5.1 Qualitative analysis

Table 1 compares Pasture against some recent systems that use trusted hardware to protect the execution of application operations. Flicker [34] and Memoir [37] use SEM and TPMs to provide protected execution of trusted modules on untrusted devices. TrInc [28] uses trusted smart cards to securely attest messages and prevent equivocation attacks in distributed systems. Pasture provides secure offline data access using SEM and TPMs. (We discuss additional related work in §7.)

All of these systems provide isolation from a malicious OS, hypervisor, or other application code. Flicker provides a general abstraction but it is vulnerable to snoop attacks and does not provide crash resilience [37], and thus fails to ensure state continuity across crashes. Furthermore, its use of SEM disables cores and interrupts for every operation. Memoir suffers from the same drawbacks as Flicker except that it is resilient to crashes. TrInc is crash resilient but provides limited functionality of secure attestation, and it is less durable due to NV updates on attest operations. Pasture provides offline access, revocation, and attestation without needing SEM or NV writes with each operation while providing crash resilience and defending against snoop attacks.

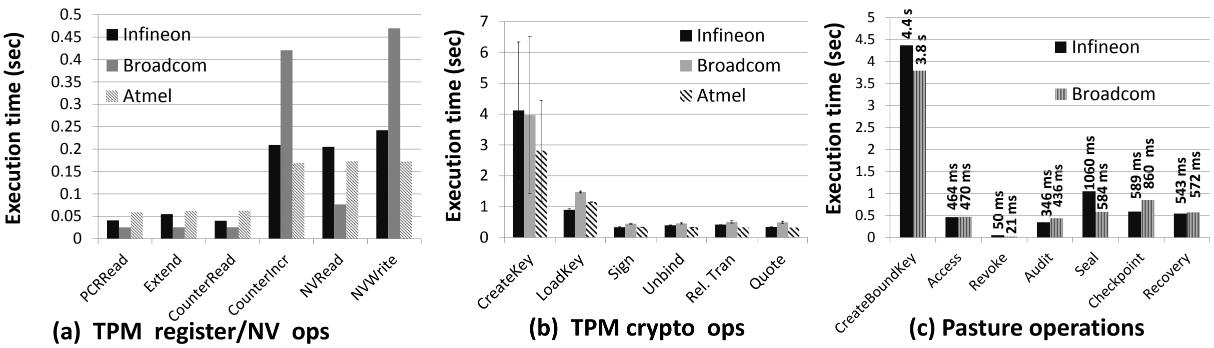
Minimal TCB. Pasture trusts just the `Checkpoint` and `Recover` routines which run during bootup and shutdown, and it does not trust any software during the common case data access operations. Pasture achieves this by exploiting the TPM primitives.

5.2 Pasture microbenchmarks

5.2.1 Computational overhead

Our first experiment measures the computational overheads imposed by TPM operations on Infineon, Broadcom and Atmel TPM chips. Figure 7(a) plots the execution time of register and NV operations; Figure 7(b) plots the execution time of TPM cryptographic operations to create and load a key, sign data, unbind a key, release a transport session, and quote PCR state; and Figure 7(c) plots the execution time of Pasture operations that build upon the TPM operations. We use 1024 bit RSA keys for our experiments. We make the following observations.

Slow NV updates. Incrementing an NV counter or writing NVRAM is far slower than reading or extending



(a) TPM register/NV ops

(b) TPM crypto ops

(c) Pasture operations

Figure 7: Computational overhead. Error bars represent one standard deviation of uncertainty over 10 samples.

a PCR. This supports our decision to avoid NV updates during common case operations.

Data exchange protocol. Creating a key in TPMs is enormously expensive, and it takes about 2.8 s (Atmel TPM) to 4 s (Broadcom TPM). This overhead accounts for almost all of Pasture's `CreateBoundKey` execution time. Key creation also has a huge variance. We study the impact of this overhead on perceived latencies to the end user in §5.3.1.

Offline operations. The offline operations to obtain or revoke access have acceptable performance. Revoking access is fast (21 ms) because it just requires a PCR extension (using the optimization discussed in §3.2 to defer the proof of revocation). Obtaining access, however, takes much longer (470 ms) as almost all of its time is spent in TPM_Unbind to decrypt and extract the symmetric key, which is fairly slow. We cannot hide this latency because the symmetric key is needed to decrypt the data. While the latency may be acceptable for an offline video or email application, our throughput is limited by these overheads. For some applications, batching can be used to amortize overheads across multiple data items as shown in §5.3.2.

Note that we hide the TPM overhead of LoadKey by loading it in the background while the data is being fetched from the sender and before the data is accessed offline.

Checkpoint and recover operations. (Note that our microbenchmarks do not include the 100 ms to 200 ms overhead required to set up the SEM environment [34].) `Checkpoint` takes about 1600 ms to seal and copy volatile state to the NVRAM, which is reasonable for a battery-backup shutdown routine. `Recover` takes about 600 ms to check the correctness of $\mathrm{PCR}_{\mathrm{APP}}$ when the system is booting up. Furthermore, before correctness is checked, `Recover` must read Pasture's log from the disk and extend $\mathrm{PCR}_{\mathrm{APP}}$ to its pre-shutdown state. Each TPM_Extend takes about 20 ms, so the additional overhead depends on the number of entries, which can be kept low by periodically truncating the log. If the log is truncated every 128 entries, at most 3 s would be spent extending $\mathrm{PCR}_{\mathrm{APP}}$. While the total overhead may seem high, modern operating systems already take tens to hundreds of seconds to boot up.

Audit. Pasture takes about 400 ms to generate a proof of revocation or a response to an audit request. Given that this operation is usually performed in the background, it does not much affect the user experience.

5.2.2 Network and storage overheads

Pasture incurs network and storage overhead due to (1) additional messages exchanged for fetching keys and (2) inclusion of an attested proof of execution in the protocol messages and the log. With 1024 bit encryption keys, Pasture exchanges about 1732 bytes of additional data in the data transfer protocol, and stores less than 540 bytes of data on disk for logging each operation.

Pasture's proofs contain hashes of messages instead of raw messages. Hence, network and storage overheads do not increase with data size. Pasture imposes considerable overhead when the message sizes are smaller but the overhead becomes insignificant for large messages.

5.3 Pasture applications

Pasture uses various optimizations—that hide latency and batch operations—to reduce overheads. Here we evaluate the end-to-end performance of the latency-oriented (offline video and healthcare) and throughput-oriented (shared folder application) applications.

5.3.1 Secure email transport

Our offline video rental and health care applications use Pasture-based secure email to allow secure offline accesses. For evaluating the overheads added by the Pasture system, we compare our applications with that of the corresponding applications that use regular email transport mechanism to send data (patient health records or the decryption key of the movie) but without secure offline data access guarantees.

Our Pasture-based applications incur an additional overhead in establishing encryption keys (one round trip delay) and generating keys in the receiver's TPM. Assuming there will be some *read slack time* between when an email is received in a user's inbox and when the user reads it, Pasture hides its latency by executing its pro-

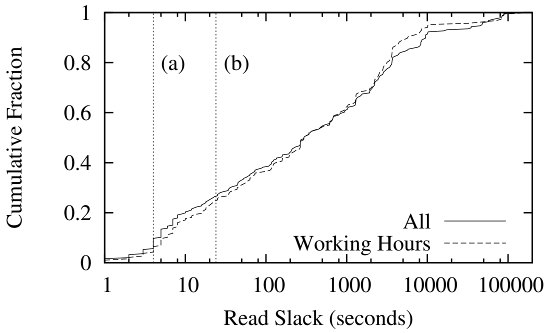


Figure 8: CDF of email read slack times.
Line (a) shows minimum Pasture overhead of 4s. Line (b) shows the total overhead of 24s including email server queuing and network delays. Only emails to the left of the lines are affected by Pasture.

tocol in the background. We evaluated the effectiveness of this approach via a small scale user study involving 5 users in a corporate setting. We logged the receive time and read time of the emails received over a period of one week for these users. To be conservative, we omitted unread emails.

Figure 8 shows the cumulative distribution of the read slack time of all the emails. We also include a separate line that considers only the emails that are received during work hours on a weekday. This removes any bias due to inactivity after work hours and over the weekend. For comparison, we measured the latency introduced by Pasture. The average additional latency introduced by Pasture was 24 s, with 4 s spent generating the encryption key at the receiver and about 20 s of network delay introduced by the processing queues of intermediate mail servers (the WAN network latencies to the mail servers was negligible, on the order of 40 ms).

We plot two vertical lines to show what fraction of emails are affected by the Pasture overhead. The affected emails are the ones whose recipients would have to wait. As shown in the figure, about 75% of emails are *not* affected by the total overhead introduced by Pasture, as their recipients checked their inbox more than 24 s after the email arrived. Even in the remaining 25% of emails which are affected, the Pasture protocol adds a delay of 15 s on average. If the processing and queuing delays at the mail servers are to be discounted, more than 90% (and 95% during the work time) of the emails are unaffected, and the rest experience an average delay of only 1 s. We conclude that Pasture can exploit the slack time of users to effectively hide the additional latency introduced for security. Furthermore, in the video rental application, Pasture latencies incurred can also be hidden while the encrypted movie downloads.

5.3.2 Decentralized storage system

We implemented a shared folder application on top of the Pasture decentralized storage system. The shared folder application allows nodes to update files locally when they are disconnected (similar to other weakly-connected decentralized storage systems [31,39,50]), attest updated files, and share file updates with other peers opportunistically when they are connected.

Scalable throughput using batching. Given that nodes are expected to read all the received updates, the Pasture shared folder application amortizes overheads by batching and performing secure attestation of updates, message exchange protocol operations, and read operations on an entire batch of received updates. Pasture scales linearly to about 1000 requests/sec (with a batch size of 460) because the overheads to attest updates (430 ms) and read (460 ms) received updates are independent of the batch size. This is because Pasture uses a constant-size hash of the message when performing an attest or read operation. We conclude that Pasture provides scalable throughput for applications where nodes have the opportunity to batch updates. Batching also reduces the recovery response time as we amortize the PCR extensions across multiple updates.

High durability. Pasture does not perform NVRAM writes or counter increments during common case operations. Only two NVRAM writes and one counter increment are required for each reboot cycle. Hence, assuming only one reboot per day, it would take Pasture more than 130 years to exhaust the 100K lifetime NVRAM and counter update limit of current TPMs [37]. Conversely, if a 5 year lifetime is acceptable, Pasture can perform a reboot cycle on an hourly basis and truncate the log to reduce audit and recover times.

6 Discussion

Pasture effectively hides and amortizes TPM overheads for applications with low concurrency. However, TPM's limited resources and high overheads hurt the scalability of Pasture with increasing concurrent requests. We discuss three key limitations and suggest improvements through modest enhancements in future TPMs.

First, Pasture's concurrency is limited by the number of available PCRs (13 or fewer) for binding keys. Increasing the number of PCRs would improve Pasture's ability to bind more keys to PCRs and have more requests in flight to support applications with high concurrency.

Second, Pasture spends a significant amount of time in TPM_CreateWrapKey to generate keys. This overhead could be reduced by simply separating key generation from the operation of key binding. For example, TPMs could generate keys whenever they are idle (or in parallel with other operations) and store them in an internal buffer for future use.

Third, TPMs allow storage of only a few keys, and we have to pay significant overheads to load a key (around 1 s) if it has to be brought from the stable storage every time. Increasing the buffer space to hold more keys would significantly reduce the overhead for highly concurrent applications.

7 Related work

Pasture builds upon related work in the areas of secure decentralized systems and trusted hardware.

Custom hardware. Many custom hardware architectures [30, 38, 46, 47, 53] have been proposed to protect trusted code from untrusted entities. These proposals have not gained widespread acceptance due to the high deployment cost and unavailability of custom hardware.

OS and VMM based approaches. A number of OS [26, 45, 49, 55], microkernel [25, 44] and VMM [11, 56] based approaches improve isolation and security by significantly reducing the TCB. A recent paper [45] provides an in depth review of the state of the art in this area. HiStar [55], Asbestos [52], and Flume [26] provide systemwide data confidentiality and integrity using decentralized information flow control techniques. Nexus [45] implements NAL logic [42] to provide general trustworthy assurances about the dynamic state of the system. While these systems provide powerful and general security primitives, they are vulnerable to hardware-snooping physical attacks [16, 21].

Commodity trusted hardware. Flicker [34] spurred research in secure execution by demonstrating how TPMs and SEM can be used to run trusted application code in isolation from the OS or hypervisor. TrustVisor [33] builds upon Flicker to protect sensitive application code in a single legacy guest VM. Memoir [37] prevents rollback attacks as described in §2.4. While these approaches provide strong isolation, they are vulnerable to hardware snooping attacks and also require frequent use of SEM, which can result in poor user responsiveness, resource underutilization and higher overhead.

Trusted hardware has been used to reduce email spam [14], in secure databases [32], for reasoning about network properties [43], for cloaked computation [9] by malware, and to provide trusted path [57] between a user's I/O device and trusted application code with minimal TCB. Recent approaches [3, 10] address an orthogonal issue of sharing TPM resources securely across multiple VMs. It is an avenue for future research to apply their approach to Pasture to share TPM resources across multiple receiver applications.

Preventing attacks in decentralized systems. Keypad [12] provides a simple online security mechanism for theft-prone devices by storing decryption keys at a server that client devices consult on every access attempt. A2M [7] prevents equivocation attacks [28, 29] by forcing attestation of updates into an append-only log using trusted hardware. TrInc [28] improves upon A2M by reducing the TCB to a trusted monotonic counter [41]. PeerReview [15], Nysiad [19], and other log-based replication systems [29, 31, 54] detect equivocation attacks without trusted hardware by using witness nodes and signed message logs. However, these systems do not prevent nor detect offline read-denial attacks.

8 Conclusion

Mobile user experiences are enriched by applications that support disconnected operations to provide better mobility, availability, and response time. However, offline data access is at odds with security when the user is not trusted, especially in the case of mobile devices, which must be assumed to be under the full control of the user.

Pasture provides secure disconnected access to data, enabling the untrusted user to obtain or revoke access to (previously downloaded) data and have his actions securely logged for later auditing. We implement Pasture using commodity trusted hardware, providing its security guarantees with an acceptable overhead using a small trusted computing base.

Acknowledgements

We thank Doug Terry, Ted Wobber, and Peter Haynes for providing considerable suggestions to the Pasture project, and the anonymous OSDI reviewers for their detailed reviews and excellent feedback. We thank Martin Abadi, Rama Ramasubramanian, Jay Lorch, Dahlia Malkhi, JP Martin, and Marcos Aguilera for their feedback on earlier drafts of this paper. We are grateful to Stefan Thom for helping us with the Windows code base.

References

[1] Advanced Micro Devices. *AMD64 virtualization: Secure virtual machine architecture reference manual*, 3.01 edition, 2005.

[2] N. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. PRACTI replication. In *NSDI*, 2006.

[3] S. Berger, R. Cáceres, K. A. Goldman, R. Perez, R. Sailer, and L. Doorn. vTPM: Virtualizing the trusted platform module. In *USENIX Security*, 2006.

[4] BitLocker Drive Encryption Overview. http://windows.microsoft.com/en-US/windows-vista/BitLocker-Drive-Encryption-Overview. Aug 31, 2012.

[5] D. Challener, K. Yoder, R. Catherman, D. Safford, and L. Van Doorn. *A Practical Guide to Trusted Computing*. IBM Press, Jan 2008.

[6] K. Chaudhuri, D. Doligez, L. Lamport, and S. Merz. Verifying safety properties with the TLA+ proof system. In *IJCAR*, 2010.

[7] B. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Attested append-only memory: Making adversaries stick to their word. In *SOSP*, 2007.

[8] Delivering a secure and fast boot experience with UEFI. http://channel9.msdn.com/events/BUILD/BUILD2011/HW-457T. Aug 31, 2012.

[9] A. M. Dunn, O. S. Hofmann, B. Waters, and E. Witchel. Cloaking malware with the trusted platform module. In *Proceedings of the 20th USENIX conference on Security*.

[10] P. England and J. Loeser. Para-virtualized TPM sharing. In *TRUST*, 2008.

[11] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A VM-based platform for trusted computing. In *SOSP*, 2003.

[12] R. Geambasu, J. P. John, S. D. Gribble, T. Kohno, and H. M. Levy. Keypad: An auditing file system for theft-prone devices. In *EuroSys*, 2011.

[13] Gmail delivery errors divulge confidential information. http://news.cnet.com/8301-13880_3-10438580-68.htm. Aug 31, 2012.

[14] R. Gummadi, H. Balakrishnan, P. Maniatis, and S. Ratnasamy. Not-a-Bot: Improving service availability in the face of botnet attacks. In *NSDI*, 2009.

[15] A. Haeberlen, P. Kouznetsov, and P. Druschel. PeerReview: Practical accountability for distributed systems. In *SOSP*, 2007.

[16] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest we remember: Cold boot attacks on encryption keys. In *USENIX Security Symposium*, 2008.

[17] T. Hardjono and G. Kazmierczak. Overview of the TPM Key Management Standard. http://www.trustedcomputinggroup.org/files/resource_files/ABEDDF95-1D09-3519-AD65431FC12992B4/Kazmierczak20Greg20-20TPM_Key_Management_KMS2008_v003.pdf. Aug 31, 2012.

[18] Health Information Technology for Economic and Clinical Health Act. http://en.wikipedia.org/wiki/HITECH_Act. Aug 31, 2012.

[19] C. Ho, R. van Renesse, M. Bickford, and D. Dolev. Nysiad: Practical protocol transformation to tolerate Byzantine failures. In *NSDI*, 2008.

[20] Hospital fined over privacy breaches in days after deaths of Jackson, Fawcett. http://www.phiprivacy.net/?p=2888. Aug 31, 2012.

[21] A. Huang. *Hacking the XBOX: An Introduction to Reverse Engineering*. No Starch Press, 2003.

[22] Intel Corporation. *LaGrande technology preliminary architecture specification.*, d52212 edition, 2006.

[23] Johns Hopkins University e-mail attachment error exposed personal info. http://www.phiprivacy.net/?p=4583. Aug 31, 2012.

[24] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Trans. Comput. Syst.*, 10:3–25, 1992.

[25] G. Klein et al. seL4: formal verification of an OS kernel. In *SOSP*, 2009.

[26] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard OS abstractions. In *SOSP*, 2007.

[27] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

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

[29] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *OSDI*, 2004.

[30] D. Lie, C. A. Thekkath, and M. Horowitz. Implementing an untrusted operating system on trusted hardware. In *SOSP*, 2003.

[31] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: Cloud storage with minimal trust. In *OSDI*, 2010.

[32] U. Maheshwari, R. Vingralek, and W. Shapiro. How to build a trusted database system on untrusted storage. In *OSDI*, 2000.

[33] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. TrustVisor: Efficient TCB reduction and attestation. In *IEEE Symposium on Security and Privacy*, 2010.

[34] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for TCB minimization. In *EuroSys*, 2008.

[35] A. Menezes, P. van Oorschot, and S. Vanstone, editors. *Handbook of Applied Cryptography (Discrete Mathematics and Its Applications)*. CRC Press, Dec. 1996.

[36] Microsoft Outlook 2012. http://office.microsoft.com/en-us/outlook/. Aug 31, 2012.

[37] B. Parno, J. R. Lorch, J. R. Douceur, J. Mickens, and J. M. McCune. Memoir: Practical state continuity for protected modules. In *IEEE Symposium on Security and Privacy*, 2011.

[38] A. Perrig, S. Smith, D. Song, and J. D. Tygar. SAM: A flexible and secure auction architecture using trusted hardware. In *IPDPS*, 1991.

[39] V. Ramasubramanian, T. L. Rodeheffer, D. B. Terry, M. Walraed-Sullivan, T. Wobber, C. C. Marshall, and A. Vahdat. Cimbiosys: A platform for content-based partial replication. In *NSDI*, 2009.

[40] T. L. Rodeheffer and R. Kotla. Pasture node state specification. Technical Report MSR-TR-2012-84, Microsoft Research, Aug 2012.

[41] L. F. G. Sarmenta, M. van Dijk, C. W. O'Donnell, J. Rhodes, and S. Devadas. Virtual monotonic counters and count-limited objects using a TPM without a trusted OS. In *ACM Workshop on Scalable Trusted Computing*, 2006.

[42] F. B. Schneider, K. Walsh, and E. G. Sirer. Nexus authorization logic (NAL): Design rationale and applications. *ACM Trans. Inf. Syst. Secur.*, 14(1):8:1–8:28, June 2011.

[43] A. Shieh, E. G. Sirer, and F. B. Schneider. NetQuery: A knowledge plane for reasoning about network properties. *SIGCOMM Comput. Commun. Rev.*, 41(4):278–289, Aug. 2011.

[44] L. Singaravelu, C. Pu, H. Härtig, and C. Helmuth. Reducing TCB complexity for security-sensitive applications: Three case studies. In *EuroSys*, 2006.

[45] 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 *SOSP*, 2011.

[46] S. W. Smith and J. D. Tygar. Security and privacy for partial order time. In *PDCS*, 1994.

[47] G. E. Suh, C. W. O'Donnell, I. Sachdev, and S. Devadas. Design and implementation of the AEGIS single-chip secure processor. *SIGARCH Comput. Archit. News*, 33(2):25–36, 2005.

[48] Summary of the HIPPA Security Rule. http://www.hhs.gov/ocr/privacy/hipaa/understanding/srsummary.html. Aug 31, 2012.

[49] R. Ta-min, L. Litty, and D. Lie. Splitting interfaces: Making trust between applications and operating systems configurable. In *OSDI*, 2006.

[50] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *SOSP*, 1995.

[51] TPM Main Specification Level 2 Version 1.2, Revision 116. http://www.trustedcomputinggroup.org/resources/tpm_main_specification. Aug 31, 2012.

[52] S. Vandebogart, P. Efstathopoulos, E. Kohler, M. Krohn, C. Frey, D. Ziegler, F. Kaashoek, R. Morris, and D. Mazières. Labels and event processes in the Asbestos operating system. *ACM Trans. Comput. Syst.*, 25(4), Dec. 2007.

[53] B. Yee and J. D. Tygar. Secure coprocessors in electronic commerce applications. In *USENIX Workshop on Electronic Commerce*, 1995.

[54] A. R. Yumerefendi and J. S. Chase. Strong accountability for network storage. *IIEEE Trans. on Storage*, 3(3), 2007.

[55] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *OSDI*, 2006.

[56] F. Zhang, J. Chen, H. Chen, and B. Zang. CloudVisor: Retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *SOSP*, 2011.

[57] Z. Zhou, V. D. Gligor, J. Newsome, and J. M. McCune. Building verifiable trusted path on commodity x86 computers. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2012.