Offline Untrusted Storage with Immediate Detection of Forking and Replay Attacks*

Marten van Dijk, Jonathan Rhodes, Luis F. G. Sarmenta, and Srinivas Devadas
Computer Science and Artificial Intelligence Laboratory (CSAIL)
Massachusetts Institute of Technology
Cambridge, MA 02139
{marten,jrhodes,lfgs,devadas}@mit.edu

ABSTRACT

We address the problem of using an untrusted server with only a trusted timestamping device (TTD) to provide trusted storage for a large number of clients, where each client may own and use several different devices that may be offline at different times and may not be able to communicate with each other except through the untrusted server (over an untrusted network). We show how a TTD can be implemented using currently available Trusted Platform Module TPM 1.2 technology without having to assume trust in the BIOS, CPU, or OS of the TPM's server. We show how the TTD can be used to implement tamper-evident storage where clients are guaranteed to ***immediately*** detect illegitimate modifications to their data (including replay attacks and forking attacks) whenever they wish to perform a critical operation that relies on the freshness and validity of the data. In particular, we introduce and analyze a log-based scheme in which the TTD is used to securely implement a large number of ***virtual monotonic counters***, which can then be used to time-stamp data and provide tamper-evident storage. We present performance results of an actual implementation using PlanetLab and a PC with a TPM 1.2 chip.

Categories and Subject Descriptors:
C.3 [Special-Purpose and Application-based Systems]: Smartcards
D.4.3 [File Systems Management]: Distributed File Systems

General Terms: Security

Keywords: virtual monotonic counters, untrusted storage, freshness, validity, replay attack, forking attack, integrity checking, TPM

1. INTRODUCTION

We address the problem of using an untrusted server with only a trusted timestamping device (TTD) to provide trusted storage for a large number of directories, where the files in each directory may be accessed and updated by several different devices that may be offline at different times and may not be able to communicate with each other except through the untrusted server (over an untrusted network). This functionality is highly relevant today as computing becomes increasingly mobile and pervasive. More and more users today, for example, regularly use several independent computing devices – such as a desktop at home, a laptop while traveling, a mobile phone, and another desktop at work – each of which may be offline or disconnected from the other devices at different times.

1.1 Problem Statement

Any trusted storage system using untrusted servers needs to address at least three security issues: privacy (i.e., a client's data must not be understandable by an adversary), authenticity (i.e., a client must be able to verify that the client's data originated from the client), and ***freshness*** (i.e., a client must be able to verify that the storage system is returning the most recent version of the client's data). Of these, freshness is the most challenging problem.

The privacy of a client's data can easily be achieved through encryption, while its authenticity can be accomplished by using digital signatures or message authentication codes (MACs). Neither of these techniques, however, can guarantee freshness. This is because even if the client encrypts the data and uses a signature or MAC, this does not prevent an adversary who can access and manipulate the memory, disk space, or software of the untrusted server, or who can intercept and manipulate messages transmitted over the untrusted network, from performing a ***replay attack***. That is, in response to a read request, the adversary can replace the most recent signed and encrypted version of the desired data with an older but likewise signed and encrypted version. The client can verify the signature on the data it receives, but cannot tell that it is not the most recent version available.

1.2 Existing Solutions

The traditional defense against replay attacks is to use timestamping [3, 9]. This technique assumes that the client is somehow able to maintain a trusted dedicated counter whose current (most recent) value is available to all the client's devices whenever they need it. Figure 1(a) shows one way to implement time-stamping assuming such a counter. Here, a client, Alice, first creates her own unique private-public key pair (SK_A, PK_A), and stores it in each of her many devices. Then, whenever one of her devices wishes to write or update her data, the device first increments the dedicated counter and then stores, in the storage server, a file record containing the updated data together with a timestamp, which consists of the dedicated counter's ID, value, and a signature (using Alice's private key, SK_A) of the hash of the client's data, and the counter's ID and value, that is,

$$\texttt{Sign}_{SK_A}(\texttt{H}(data_A)\|\texttt{ctrID}_A\|\texttt{ctrVal}_A).$$

*This work was done under the MIT-Quanta T-Party project, funded by Quanta Corporation.

STC'07, November 2, 2007, Alexandria, Virginia, USA.

When a device wishes to retrieve the data from the storage server, the storage server returns the file record with the timestamp, and the client device then verifies the corresponding signature and checks whether the signed counter value in the timestamp corresponds to the current counter value. If the values do not match, then the client device knows that the storage server has given it an older version of the file record – i.e., that the server is attempting a replay attack.

The dedicated counter required by this traditional approach can be maintained if one of the clients is known and guaranteed to be online all the time, or if at each moment at least a majority of the client's devices are online and reachable by any client device who needs access to the counter value. However, in the general case where the client's different devices may be ***offline*** at any possible time, it is impossible for the client's devices to reliably and securely maintain and agree on the current value of the dedicated counter.

Note in particular that other general-purpose, multi-user network file systems that use untrusted storage servers have been proposed, such as SUNDR [14, 11] and Plutus [10]. These provide privacy and authenticity, but can only provide limited guarantees for freshness because they do not utilize any trusted third parties. SUNDR, for example, offers protection against ***forking attacks***, a form of attack where a server uses a replay attack to give different users a different view of the current state of the system. However, it does not immediately detect forking attacks. Instead, it offers ***fork consistency***, which essentially ensures that the system server either behaves correctly, or that its failure or malicious behavior will be detected at a later moment when users are able to communicate with each other (for example, once a day during night time).

1.3 Our Solution

If we want to ***immediately*** detect forking attacks and replay attacks whenever a critical operation needs to be performed, then one solution is to use a trusted third party that is always online, and that can be trusted to correctly maintain the clients' dedicated counters. However, if we do not want to, or cannot, assume any trusted third parties, then this solution is not possible.

A solution is possible, however, if we assume an untrusted third party with a trusted module, such as a TPM. We present such a solution in this paper. As shown in Fig. 1(b), our scheme implements a ***virtual counter manager (VCM)*** using a small trusted computing base (TCB) in the form of a single ***trusted timestamping device (TTD)*** installed in an online untrusted third party machine. Using a ***log-based*** scheme that keeps track of the increment operations of this single TTD, we can securely implement an unlimited number of dedicated ***virtual monotonic counters***, which can each in turn be used by multiple clients to allow forking and replay attacks on their data to be immediately detected.[1]

Note that although more efficient solutions are possible by assuming more complex trusted hardware, our minimal scheme is significant because it allows implementation using a wider class of trusted hardware components, including components which are already in many existing machines available to the general public. Specifically, our scheme is implementable using the current version of the Trusted Platform Module (TPM 1.2) [25], an inexpensive secure coprocessor that is currently becoming a standard component in new commodity PCs and laptops today (and potentially mobile devices in the future as well [23]). Moreover, its security depends solely on the TPM itself, and remains secure even if the BIOS, CPU, OS, and administrators of the third party server that contains the TPM are compromised or malicious, or contain security bugs. This means that unlike other schemes that use the TPM to implement virtual monotonic counters [16, 24], our schemes are implementable today, using existing machines and existing off-the-shelf operating systems. Additionally, our scheme can also be implemented using smartcards and other secure devices that can implement the simple functionality of a TTD.

[1]In earlier work [19], we presented a simpler version of our log-based scheme. In this paper, we extend this scheme, develop a fully functional virtual counter manager around it, and present results on experiments involving hundreds of clients.

1.4 Outline

Our paper is arranged as follows: We begin in Sect. 2 by presenting our model of a trusted timestamping device, and then giving an overview of our log-based scheme for using such a device to implement a VCM. In Sect. 3, we present our solution in detail, describing the different protocols for reading and incrementing virtual monotonic counters. We then present our proof-of-concept implementation of these protocols using PlanetLab and a VCM implemented by using a PC with a TPM 1.2 chip in Sect. 4, and present a summary of some experimental results which show that acceptable performance can be achieved with such minimal trusted hardware. In Sect. 5, we discuss and address the issues of efficiency, reliability, replication, and physical security. We discuss related work in Sect. 6, and conclude in Sect. 7.

2. SOLUTION OVERVIEW

2.1 Trusted Timestamping Device (TTD)

Abstractly, a trusted timestamping device (TTD) is a device with the following key properties:

- It has an arithmetic monotonic counter, which is a variable t whose value can be made to go up (by 1) using "increment" operations, but which cannot be made to revert to an older value — even by the owner of the timestamping device.

- It has a unique private signing key (SK), which can be used in special timestamping operations to produce unforgeable signatures that can only be produced using the device itself (and which cannot be used to sign arbitrary data). This private signing key has a corresponding unique public verification key (PK), which is certified by a trusted certificate authority (using a traditional certificate), and which can be used by any third party to verify the signatures produced with the signing key.

- It supports the following timestamping operations:

  `ReadSign`(rec)**,** which outputs a ***read certificate*** of the form
  $(X = (\text{"Read"}, t, rec),\ \texttt{Sign}(X))$, and

  `IncSign`(rec)**,** which atomically increments t, and outputs an ***increment certificate*** of the form
  $(X = (\text{"Inc"}, t_{new}, rec),\ \texttt{Sign}(X))$,

  where rec is a record containing arbitrary data, and `Sign`$(...)$ indicates an unforgeable and verifiable signature produced by the device using its unique signing key.

- It is secure — i.e., there must not be any commands or attacks that would allow an adversary (even one that owns and can give arbitrary commands to the device) to successfully rewind the value of t, or produce valid `ReadSign` and `IncSign` outputs without actually invoking the `ReadSign` and `IncSign` commands themselves.

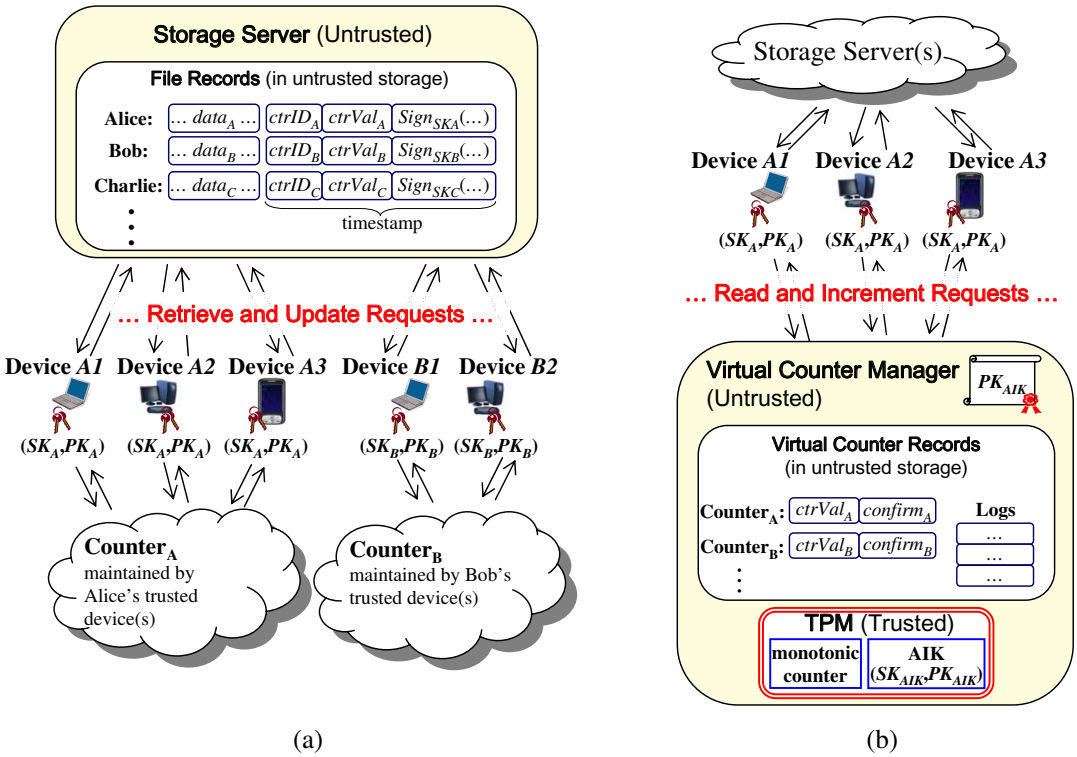


Figure 1: Trusted Storage on Untrusted Servers with Timestamping. (a) Traditional approach: assume a dedicated counter maintained by the client's devices (requires online communication between the client's devices). (b) Our approach: use an untrusted virtual counter manager with a trusted TPM chip (no trusted BIOS, OS, or CPU required). Note: the virtual counter manager and the storage server may or may not be the same machine.

2.2 Log-Based Scheme Overview

In order to implement an arbitrary number of dedicated virtual monotonic counters using only a single TTD, we first define the value of a specific virtual counter as the value of the TTD's (global) counter t at the last time the virtual counter's increment protocol was successfully invoked. Note that this somewhat unorthodox definition results in *non-deterministic* monotonic virtual counters. That is, the value of a particular virtual counter C is guaranteed to always increase, but can increase by unpredictable amounts, since operations on other virtual counters can cause the global counter to increment in between increment requests for C. This is not a problem, however, since for detecting forking and replay attacks as described in Sect. 1, a client needs only to be able to check that the value of the virtual counter for a file has not changed since the file record was timestamped.

Defining our virtual monotonic counters in this way allows us to implement a secure virtual counter manager (VCM) by using the TTD to timestamp the increment requests submitted by a client for each virtual counter, and keeping a log of such timestamps (i.e., increment certificates). Given such a scheme, a client can then read and check the freshness of a virtual counter C's value by asking the virtual counter manager to present a ***log*** of increment certificates produced by the TTD since the most recent increment request for C, and checking that there were indeed no other increment requests for C since the supposedly most recent request. Note that even if the VCM machine is running malicious software or using malicious hardware, it cannot fool the client into accepting an older value for a virtual counter since doing so would require forging increment certificates, which is infeasible if the TTD itself is secure.

2.3 Protocols Overview

Our solution provides four protocols: slower read and increment protocols which return proofs of validity (described below), as well as faster read and increment protocols which do not return proofs of validity. A client can use the slower protocols whenever it needs to perform a critical operation that depends on *immediately* verifying the freshness of the counter. If the client is performing a non-critical operation (i.e., one which can be reversed, aborted, and/or retried before a critical operation at a later time), then it can use the faster protocols instead.

These protocols are initiated by a client, and assume that each client has his own unique public-private key pair stored in each of his devices. The client's devices use the client's private key to sign increment requests, create timestamps, and create ***confirmation certificates***. Confirmation certificates are produced and given to the virtual counter manager whenever a client's device succesfully performs a read or increment with validation. It provides a trustworthy proof (trusted by the client's devices) of the valid value of the client's virtual counter at the time of the certificate's creation.

On the manager's side, the virtual counter manager has a TTD, and software which keeps track of:

1. an array of the ***most recent*** confirmation certificates for each virtual counter and
2. an array of each of the increment certificates which were generated since the generation of the ***oldest*** amongst the most recent confirmation certificates.

Using these, together with the TTD operations described earlier, the virtual counter manager implements four protocols for operating on individual virtual counters:

1. ***Increment-without-validation (Fast-Increment)***, in which a client's device requests to increment one of the client's virtual counters and which results in an increment certificate,

2. ***Read-without-validation (Fast-Read)***, in which the VCM returns the current value of a virtual counter,

3. ***Read-with-validation (Full-Read)***, in which not only the current value of a virtual counter but also a proof of the validity of this virtual counter is returned, and

4. ***Increment-with-validation (Full-Increment)***, which is the increment-without-validation and read-with-validation protocols combined into a single protocol.

The read and increment protocols with validation produce a ***validity proof***, which is composed of:

1. the most recent confirmation certificate of the corresponding virtual counter, together with

2. a list (or log) of each of the increment certificates which were generated since the creation of this most recent confirmation certificate and

3. a new read or increment certificate.

As detailed in Sect. 3.3 and Fig. 5, a validity proof enables a client to determine the ***validity*** of the output of the read or increment protocol, and to detect any malicious or erroneous behavior by the VCM in the past. That is, for each of the past increments by any of a client's devices, the client can check whether the increment was based on a retrieved counter value (received from the virtual counter manager during one of its protocols) that is equal to the current counter value just prior to the increment. (See [6] for a similar definition of valid storage). This check is made possible by having an increment certificate also certify the value on which the corresponding increment is based. This check is necessary for critical operations, since such operations rely on whether past increments were based on retrieved counters that were fresh and valid.

3. SOLUTION DETAILS

The description in Sect. 2 describes the general intuition behind our ideas. In this section, we present the actual protocols in more detail and we explain practical extensions to the protocols.

3.1 Increment-without-Validation

Figure 3 summarizes the increment-without-validation protocol in which Alice wants to increment a virtual monotonic counter of her choice without verifying whether the increment is valid but with the knowledge that she is able to check its validity by contacting the VCM at any given moment in the future by using the read-with-validation protocol. In case Alice wishes to perform a critical operation that depends on the virtual monotonic counter, then the increment-with-validation protocol of Sect. 3.4 can be used to immediately verify validity and freshness such that any form of replay attacks are *immediately* detected. The relationships between the different data values in the increment protocols are depicted in Fig. 2(a).

Suppose that Alice retrieves data from the untrusted storage server together with a corresponding timestamp which is based on a virtual monotonic counter with identity `ctrID`, value `ctrVal` (at the moment of timestamping), and public and secret key pair (`ctrPK`, `ctrSK`);

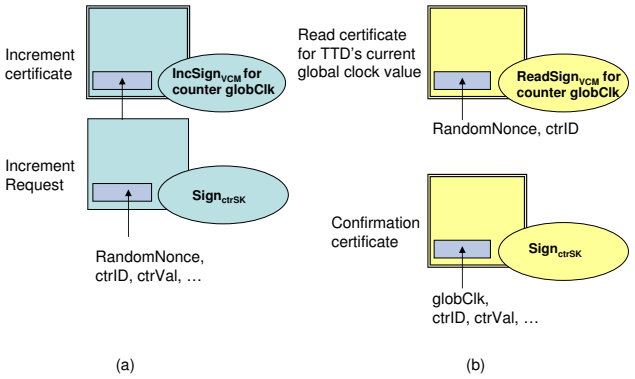


Figure 2: Relationships between the data values used in (a) the increment-without-validation protocol and (b) the read-with-validation protocol. Arrows represent containment (i.e., the lower data structures are contained in the upper ones). Ovals represent operations on the data in the rectangles, which produce a tuple containing the data and a signature.

$$\texttt{timestamp} = (\texttt{stamp} \| \texttt{Sign}_{\texttt{ctrSK}}(\texttt{stamp})), \text{ where}$$
$$\texttt{stamp} = (\texttt{Hash}(data) \| \texttt{ctrID} \| \texttt{ctrVal}).$$

We assume that Alice knows the counter's secret key, which give her the authority to modify and update the timestamped data.

In order to create a new timestamp with which Alice can sign an updated version of the retrieved data, Alice needs to increment the virtual counter with identity `ctrID`. Alice selects a random anti-replay nonce `antiReplay` and concatenates the anti-replay nonce, `ctrID`, and the current value `ctrVal` of this counter according to Alice's knowledge (from the retrieved timestamp). Alice computes the ***increment request*** `IncReq` as shown in step 1 of the protocol.

By verifying the signature in `IncReq` the VCM checks the authenticity of Alice's request. (This signature is also used in the read-with-validation protocol in order to prove to another authorized device or user Bob that the increment certificate for `ctrID` originated from an authentic request and not a fake increment.) Besides verifying the nonce's signature, the VCM checks whether the current counter's value is equal to `ctrVal`. If not, then the VCM should notify Alice about her out-of-date knowledge. If `ctrVal` does match the current counter value, then the VCM uses the TTD to compute the increment certificate `IncCert` of step 2. Notice that Alice's prior knowledge of the current counter value `ctrVal` is signed by the increment certificate. For this reason, as we will see in Sect. 3.3, the validity of `ctrID` can be verified in the read-with-validation protocol even in the presence of a malicious VCM (who may purposely not notify Alice about out-of-date knowledge), as long as the VCM has a trusted TTD .

By using the PK of the VCM's TTD, Alice verifies the increment certificate. Since the anti-replay nonce in `IncReq` is chosen at random, replay attacks of previously generated increment certificates (by, for example, a malicious VCM or a man-in-the-middle) will be detected by Alice. We do not protect against denial of service; if the increment certificate does not arrive within a certain time interval, then the client's device should retransmit its request with the same nonce. As soon as an increment certificate is veri-

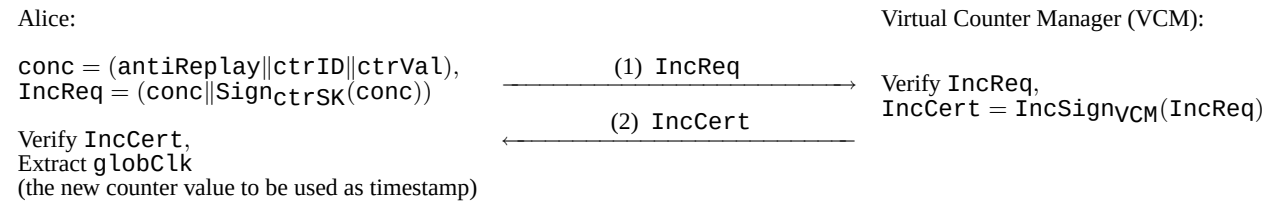


Figure 3: Increment-without-Validation protocol. Notice that `IncCert` contains the current global clock counter `globClk` of the TTD. This value will be the new counter value of the virtual counter with ID `ctrID`.

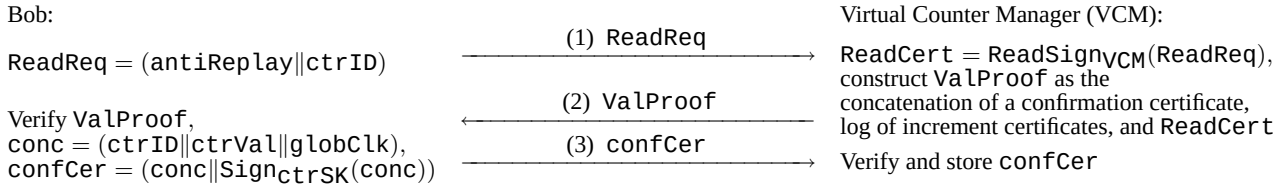


Figure 4: Read-with-Validation protocol.

fied, we say the increment has been ***successful*** and Alice may use the new counter value to timestamp data. The new counter value of `ctrID` is the global clock value `globClk` that the TTD used to create the increment certificate. As noted in Sect. 2, the signature of the increment certificate serves to unforgeably link the certificate to VCM's particular TTD, and a particular point in time on that TTD. (Note, however, that verifying the increment certificate is not the same as verifying the *validity* of the increment, which is explained in Sect. 3.3.)

3.2 Read-without-Validation

In the read-without-validation protocol, a client's device asks for the most recent value of a specific virtual counter. The VCM simply signs and returns the most recent counter value without making use of the TTD.

3.3 Read-with-Validation

Figures 4 and 5 depict and explain the read-with-validation protocol, which includes transmitting a validity proof. The relationships between the different data values are depicted in Fig. 2(b).

Suppose that Bob wishes to verify the freshness of a timestamp that corresponds to data from an untrusted storage server and which is based on a virtual counter with identity `ctrID`=D. In order to read and obtain a validity proof of the value `ctrVal` of D, Bob initiates the read-with-validation protocol with the VCM. Bob selects a random anti-replay nonce $antiReplay$ and computes the ***read request*** `readReq` as shown in step 1 of the protocol in Fig. 4.

In step 2, the VCM constructs a validity proof as shown in Fig. 5. Bob verifies the confirmation certificate by using the counter's public key `ctrPK`. This tells him that, when TTD's global clock value was equal to T, the virtual counter had the value t_0. By using the public key of the VCM's TTD, Bob verifies the log of increment certificates together with the read certificate. The read certificate records the TTD's current global clock counter value t_{now}. Bob then checks that no increment certificate between t_0 and t_{now} is missing. Afterwards, he extracts those increment certificates that signed a request with `ctrID`=D. He uses `ctrPK` to verify the authenticity of these requests. Finally, as depicted in Figure 5, Bob verifies whether each of the extracted increment certificates records the previous counter value. The most recent extracted increment certificate contains the current counter value `ctrVal` $= t_n$ which is signed by the confirmation certificate in step 3.

3.4 Increment-with-Validation

The increment-with-validation protocol first executes the increment-without-validation protocol. Then, the resulting increment certificate, which already has a signature over the value of the current global clock value as the incremented virtual counter value, is used in place of the read certificate in the read-with-validation protocol to produce a validity proof that proves the new value of the counter as well as the fact that the previous value of the counter was valid before the increment (and thus that the new value is valid).

3.5 Improvements

Here, we explain two techniques that can be used to improve the performance of the log-based scheme.

Sharing. One performance bottleneck is the fact that read and increment operations typically take a few seconds to execute on existing TPM 1.2 chips. Thus, as we will show in Sect. 4, if we only allow one virtual counter to be incremented for each increment of the global counter, then a single TPM can only handle a few virtual counters before the overall performance becomes unacceptably slow.

A solution to this problem is to allow multiple increment protocols of independent virtual counters to be executed at the same time, sharing a single TTD primitive. The general idea here is to collect the individual `IncReq`'s for each virtual counter to construct a single shared request (e.g. by using an authenticated search tree [5, 4]) which can then be used as an input to a single shared `IncSign` primitive. This increments all the corresponding virtual counter values to the same global clock value. Note that the same idea can also be used to share the `ReadSign` primitive.

Time-Multiplexing. The log-based scheme has another significant drawback: if a virtual counter v is not incremented while other counters are incremented many times, then the validity proof for v would need to include the log of all increments of all counters (not just v) since the last increment of v. The length of this log can easily grow very large.

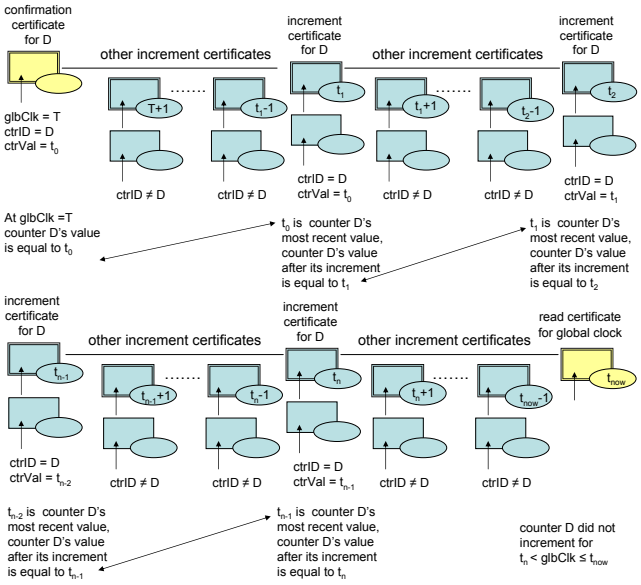


Figure 5: Validity proof for D containing a confirmation certificate, a log of each of the increment certificates which were generated since the creation of this confirmation certificate, and a read certificate (`ReadCert` in Fig. 4) with TTD's current global clock counter. After receiving the validity proof, Bob checks the signatures on all the increment certificates, extracts the certificates with increments for D, and verifies that each increment was done with the correct knowledge of the previous value of the virtual counter. If this validation succeeds, the client produces a new confirmation certificate.

A solution to this problem is to time-multiplex the global clock. That is, instead of allowing increments at each possible global clock value for each client, each client associates with each of his virtual counters a fixed schedule of global clock values that are allowed to be virtual counter values. The main advantage of time-multiplexing is that the log of increment certificates in a validity proof of a virtual counter can be reduced to those for which the corresponding verification algorithm retrieves a value which is allowed according to the schedule of the virtual counter. The disadvantage of time-multiplexing is a possible increase in the latency between the request and finish of an increment or read protocol. In an extended report [27], we introduce the idea of an ***adaptive schedule*** that allows the virtual counter's schedule to change (now increment-without-validation is not allowed, and increment and confirmation certificates should include the current version of the schedule at the time of their generation).

4. IMPLEMENTATION AND RESULTS

As a proof-of-concept, we have implemented our protocols and tested them using PlanetLab and a PC with a TPM 1.2 chip.

4.1 TPM 1.2 Implementation of TTD

We implemented the TTD described in Sect. 2 by: (1) using the TPM's ***built-in monotonic counter*** as the arithmetic monotonic counter, (2) using an ***attestation identity key (AIK)*** as the unique private signing key, and (3) implementing the `ReadSign`(rec) and `IncSign`(rec) operations by using the TPM's `TPM_Read_Counter` and `TPM_Increment_Counter` command (respectively) inside an ***exclusive and logged transport session***, using the AIK as the signing key, and the hash of rec as the input nonce. We used TPM/J [18], a cross-platform Java-based API for the TPM which provides support for using transport sessions and monotonic counters on TPM 1.2 chips. On the STMicro TPM 1.2 chip that we used, we found that a `ReadSign`(rec) or `IncSign`(rec) operation costs about 1.3 seconds total, and that the TPM throttles the monotonic counter increment operations such that we could only execute at most one `IncSign` every 2.15 seconds.

4.2 PlanetLab Simulation of VCM

To develop an intuition for the practical limits of a trusted storage application built around a TPM, we implemented a VCM on a PC with TPM 1.2 chip. We used the PlanetLab network [22] to simulate increment and read requests for different numbers of virtual monotonic counters. From this network, 150 machines were selected to act as simulation nodes. Each virtual counter was assigned to a single simulation node, and we evenly distributed the different counters across the simulation nodes. All communications were done using Java RMI.

Each simulation node performed the read-with-validation and increment-without-validation protocols with the VCM for its own assigned virtual counters. The read and increment requests were scheduled according to probabilistically determined intervals by using a Poisson distribution with a frequency of one request per virtual monotonic counter every 15, 30, or 60 seconds.

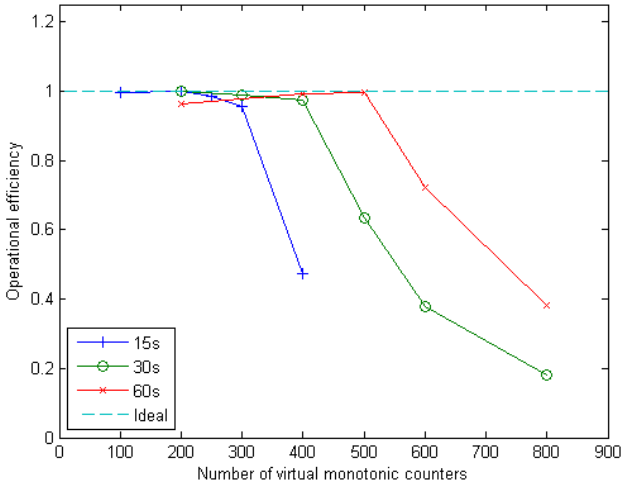
To determine when the system saturates, we use the idea of ***operational efficiency*** defined as the ratio of the number of operations actually completed to the expected number of operation requests, in an experiment. If the operational efficiency gets less than 1, then incomplete outstanding requests are being queued up. This means that the ***latency***, defined as the time from when an operation was requested to the time of its completion (this includes transmission and verification of proofs and certificates), grows impractically large for increasing simulation times. If the operational efficiency is ≈ 1, then the system is not saturated since all request can be handled in time.

Experimental results show that as expected, without sharing, only a few virtual monotonic counters can be maintained by the VCM. (For brevity, these results are not shown here.) Figure 6 shows experimental results with *both* sharing and time multiplexing. As shown, several hundred virtual monotonic counters can be managed before saturation. Before saturation, the latency of a read operation is around 10 seconds and the latency of an increment operaion scales with 2-3 seconds times the time multiplexing period. In general, as shown, the maximal number of virtual monotonic counters before saturation decreases with an increase in the frequency of requests and increases with an increase in the time multiplexing period. In practice, we believe the performance achieved here is practical if we assume that one virtual monotonic counter represents a set of files of one client (e.g., a directory).

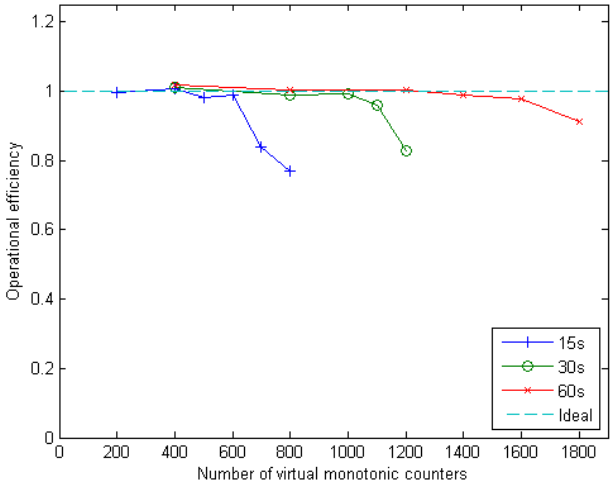
5. DISCUSSION

Our results demonstrate the potential for actually implementing a VCM using a commodity TPM-enabled server available today (without requiring any changes to the TPM). A number of issues, however, arise when considering the use of our technique.

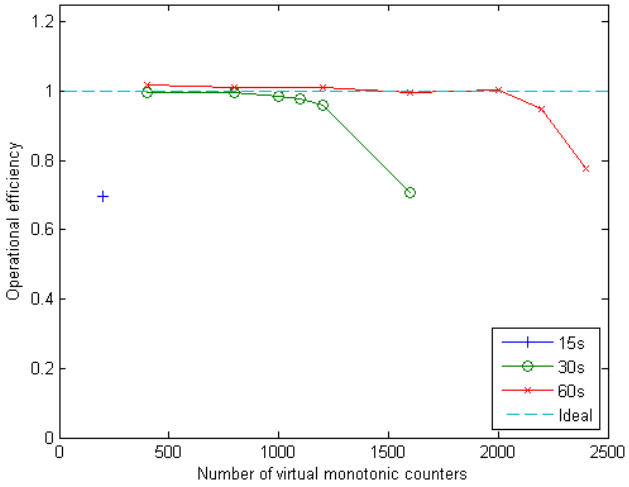
Efficiency. As shown, using sharing and multiplexing improves efficiency. Another technique that can also be used to improve latency is to have each client adopt the policy that at least one of its devices, or a third party that it trusts (but which does not have to be global and does not have to be online at the same time as



(a) No multiplexing



(b) Multiplexing with period 8



(c) Multiplexing with period 16

Figure 6: Performance Measurements at varying rates of request and multiplexing.

the client's devices) periodically performs the read-with-validation protocol, and produces a confirmation certificate. This removes the need to present increment certificates from times before the confirmation certificate, and thus reduces the log size and delay for read-with-validation and increment-with-validation operations performed by the client's devices.

Reliability. So far, our schemes provide ***tamper-evident*** trusted storage. That is, they guarantee that any incorrect behaviors by the storage server, virtual counter manager, or network – whether caused by random faults or malicious attacks – are guaranteed to at least be detected by the client's devices. Our schemes so far, however, do not by themselves actually prevent such incorrect behavior. That is, because we assume that the servers and the network are completely untrusted, it is always possible for these to simply fail or refuse to work correctly. In short, our schemes so far do not protect against simple ***denial-of-service*** attacks.

To tolerate such random failures and malicious attacks we can employ a ***replication*** scheme on top of our tamper-evident scheme. That is, for each data file that we want to store, we store several copies on separate storage servers, and use several different virtual counters managed by separate virtual counter managers. Then, if only a minority of managers is malicious and a sufficient number of managers can be connected, the correct data can still be retrieved from the replicated storage, and its freshness can be checked through the multiple virtual counter managers. In this way, we can build a ***tamper-tolerant*** trusted storage system over our tamper-evident one.

Tree-Based Scheme. In previous work [19, 26] we proposed, besides a simplified form of the log-based scheme presented here, a tree-based scheme for managing virtual monotonic counters. In this tree-based scheme, we propose a mechanism for the TPM to maintain an authentication tree with its root stored in the TPM, and propose new TPM commands which would allow this authentication tree to be used to securely implement an arbitrarily large number of dedicated deterministic virtual monotonic counters using only a small constant amount of trusted non-volatile storage in the TPM. Although this scheme cannot yet be implemented with existing TPM 1.2 chips, its ability to provide dedicated and ***deterministic*** counters would enable us to greatly simplify our protocols and reduce communication costs. It would also lead to many interesting application scenarios in which virtual counters can be linked to objects and operations [19]. We note though that even though the communication costs are much less in the tree-based scheme than in the log-based scheme, the load on the TPM would be more in the tree-based scheme [26]. Thus, in high-load applications where the same TPM is being used to serve a large number of clients at the same time, we expect that it would be best to employ a hybrid scheme that uses both the log-based and tree-based scheme.

Security. Note that our scheme offers strong security because it does not rely on any other component in the VCM other than the TTD itself. Specifically, in our case, we are able to implement a TTD not by using the TPM's trusted boot-related features, but by using TPM 1.2's built-in monotonic counter feature.

A possible problem worth noting is what happens if the power to the VCM fails some time after the `TPM_Increment_Counter` (in the `IncSign` primitive) but before the virtual counter manager is able to save the increment certificate to disk. This will lead to a gap in the log of increment certificates in proofs of validity. This problem cannot be used for a replay attack because users will at least detect the gap during a verification of a validity proof. However, it does make all the virtual counter values be-

fore the power failure untrustworthy (because client devices have no proof that these counters were not incremented during the time slot of the gap). This problem cannot easily be avoided because of the limitations of existing TPMs. Note, however, that recovery of a counter's value is still possible if all the corresponding authorized devices communicate together and agree on the last valid value of the counter which can then be signed in a confirmation certificate.

6. RELATED WORK

In this work, we reduce the trusted computing base to only a single TPM 1.2 chip [25, 15], which is a standard component on machines today. This differs from many other systems that require complex secure processors [20, 1, 12, 28, 21]. The TPM is a small inexpensive trusted chip with limited computational capabilities and a small amount of trusted volatile and non-volatile memory. One way to use the TPM is to use it to perform a ***trusted boot*** process, which enables a PC to ensure that only an unaltered trusted OS is loaded on a it, and be able to prove to an external party that the PC is in fact running such trusted code. Such a trusted boot process has been used, for example, to give clients stronger security guarantees when using web servers [13, 17], as well as to provide more security in distributed and peer-to-peer systems [7, 2]. Using such a trusted boot process with a TPM 1.2 chip, it is possible to implement a virtual monotonic counter manager. One way of doing this is briefly discussed by both TCG [24] and Microsoft [16]. The problem with techniques that rely on trusted boot, however, is that they require heavy and restrictive security assumptions. First, aside from requiring a TPM, trusted boot also requires at least a trusted BIOS component (called the Core Root-of-Trust for Measurement or CRTM), and may require other hardware-based security features as well [8]. Second, trusted boot is not robust against physical attacks on the host PC. If, for example, the adversary can read and modify memory directly without going through the CPU, then the trusted OS can be compromised. Third, trusted boot cannot protect the system from bugs in the trusted software code, and thus extreme care must be taken to ensure that the trusted OS is really secure and bug-free. Finally, all this requires the user to use the special trusted OS while using the machine, and thus does not allow us to take advantage of user machines that may not want to run this trusted OS. Thus, using a TPM with a trusted OS is still far from being a practical solution.

7. CONCLUSION

In this paper, we introduced, implemented, and analysed a virtual storage system using untrusted servers that allows immediate detection and prevention of forking and replay attacks, trusting only on a TPM 1.2 chip in the untrusted server. Experiments show that it can provide trusted storage for a large number of directories, where, for each directory, the devices that are authorized to use it may be offline at different times with respect to one another.

8. REFERENCES

[1] T. Arnold and L. van Doorn. The IBM PCIXCC: A new cryptograhic co-processor for the IBM eServer. *IBM Journal of Research and Development*, 48:475–487, 2004.

[2] S. Balfe, A. Lakhani, and K. Paterson. Securing peer-to-peer networks using trusted computing. In C. Mitchell, editor, *Trusted Computing*, chapter 10. IEE, 2005.

[3] D. Bayer, S. Haber, and W. Stornetta. Improving the Efficiency and Reliability of Digital Time-Stamping. In *Sequences II: Methods in Communication, Security, and Computer Science*, pages 329–334, 1993.

[4] A. Buldas, P. Laud, and H. Lipmaa. Accountable Certificate Management using Undeniable Attestations. In *Proceedings of the 7th ACM Conference on Computer and Communications Security*, pages 9–17, 2002.

[5] A. Buldas, P. Laud, and H. Lipmaa. Eliminating Counterevidence with Applications to Accountable Certificate Management. *Journal of Computer Security*, 10:273–296, 2002.

[6] D. Clarke, S. Devadas, M. van Dijk, B. Gassend, and G. E. Suh. Incremental Multiset Hash Functions and their Application to Memory Integrity Checking. In *Advances in Cryptology - Asiacrypt 2003 Proceedings*, volume 2894 of *LNCS*. Springer-Verlag, 2003.

[7] A. Dent and G. Price. Certificate management using distributed trusted third parties. In C. Mitchell, editor, *Trusted Computing*, chapter 9. IEE, 2005.

[8] E. Gallery. An overview of trusted computing technology. In C. Mitchell, editor, *Trusted Computing*, chapter 3. IEE, 2005.

[9] S. Haber and W. S. Stornetta. How to Time-Stamp a Digital Document. In *CRYPTO '90: Proceedings of the 10th Annual International Cryptology Conference on Advances in Cryptology*, pages 437–455, 1991.

[10] M. Kallahala, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus: Scalable Secure File Sharing on Untrusted Storage. In *Proceedings of the Second Conference on File and Storage Technologies (FAST 2003)*, 2003.

[11] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, 2004.

[12] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural Support for Copy and Tamper Resistant Software. In *Proceedings of the* 9^{th} *Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pages 168–177, November 2000.

[13] J. Marchesini, S. W. Smith, O. Wild, and R. MacDonald. Experimenting with TCPA/TCG Hardware, Or: How I Learned to Stop Worrying and Love The Bear. Technical Report TR2003-476, Dartmouth College, Computer Science, Hanover, NH, December 2003.

[14] D. Mazières and D. Shasha. Building Secure File Systems out of Byzantine Storage. In *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Distributed Computing*, pages 108–117, 2002.

[15] C. Mitchell, editor. *Trusted Computing*. The Institution of Electrical Engineers, 2005.

[16] M. Peinado, P. England, and Y. Chen. An overview of NGSCB. In C. Mitchell, editor, *Trusted Computing*, chapter 4. IEE, 2005.

[17] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and Implementation of a TCG-based Integrity Measurement Architecture. In *Proceedings 13th USENIX Security Symposium (San Diego, CA)*, 2004.

[18] L. F. G. Sarmenta and contributors. TPM/J: Java-based API for the Trusted Platform Module (TPM). http://projects.csail.mit.edu/tc/tpmj/, Dec. 2006.

[19] 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 *Proceedings of the 1st ACM CCS Workshop on Scalable Trusted Computing (STC'06)*, Nov. 2006.

[20] S. W. Smith and S. H. Weingart. Building a High-Performance, Programmable Secure Coprocessor. *Computer Networks (Special Issue on Computer Network Security)*, 31(8):831–860, April 1999.

[21] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing. In *Proceedings of the* 17^{th} *Int'l Conference on Supercomputing (MIT-CSAIL-CSG-Memo-474 is an updated version)*, New-York, June 2003. ACM.

[22] The Trustees of Princeton University. PlanetLab — An open platform for developing, deploying, and accessing planetary-scale services. https://www.planet-lab.org/, 2007.

[23] Trusted Computing Group. Mobile Phone Specifications. https://www.trustedcomputinggroup.org/specs/mobilephone/.

[24] Trusted Computing Group. TPM v1.2 specification changes. https://www.trustedcomputinggroup.org/groups/tpm/TPM_1_2_Changes_final.pdf, 2003.

[25] Trusted Computing Group. TCG TPM Specification version 1.2, Revisions 62-94 (Design Principles, Structures of the TPM, and Commands). https://www.trustedcomputinggroup.org/specs/TPM/, 2003-2006.

[26] M. van Dijk, L. Sarmenta, C. O'Donnell, J. Rhodes, and S. Devadas. Proof of Freshness: How to efficiently use on online single secure clock to secure shared untrusted memory. Technical report, 2006.

[27] M. van Dijk, L. F. G. Sarmenta, J. Rhodes, and S. Devadas. Securing Shared Untrusted Storage by using TPM 1.2 Without Requiring a Trusted OS. Technical report, MIT CSAIL CSG Technical Memo 498, May 2007.

[28] B. S. Yee. *Using Secure Coprocessors*. PhD thesis, Carnegie Mellon University, 1994.