Securing Shared Untrusted Storage by using TPM 1.2 Without Requiring a Trusted OS

Marten van Dijk, Luis F. G. Sarmenta, Jonathan Rhodes, and Srinivas Devadas

MIT Computer Science and Artificial Intelligence Laboratory (CSAIL) [*]

{marten,lfgs,jrhodes,devadas}@mit.edu

Abstract

We address the problem of using an untrusted server with a trusted platform module (TPM) 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). The clients only trust the server's TPM; the server's BIOS, CPU, and OS are not assumed to be trusted. We show how the currently available TPM 1.2 technology can be used to implement tamper-evident storage, where clients are guaranteed to at least detect illegitimate modifications to their data (including replay 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 built-in monotonic counter of a TPM 1.2 chip 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. Tamper-tolerant storage, which guarantees that a client can continue to retrieve its original data even after a malicious attack, is provided by using data replication on top of the tamper-evident storage system. As a separate application of our log-based scheme, we also show how these virtual monotonic counters can be used to implement ***one-time certificates***, which are certificates that can be spent at most once. One-time certificates can be used for one-time authentication and authorization, and can be useful in applications such as DRM, offline payments, and others. Finally, we implement these ideas using an actual PC with a TPM 1.2 chip and present preliminary performance results.

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

1 Introduction

In this paper, we address the problem of ***using an untrusted server with a trusted platform module (TPM) 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 will show how the TPM 1.2 technology that is currently available in many new PCs today [52] can be used to effectively implement ***tamper-evident*** storage, where clients are guaranteed to at least detect illegitimate modifications to their data, including replay attacks. Providing ***tamper-tolerant*** storage, which guarantees that a client can continue to retrieve its original data even after a malicious attack can also be achieved by using data replication on top of the tamper-evident storage system. Moreover, unlike other schemes using the TPM, we achieve all this while trusting only in the server's TPM. In particular, our schemes remain secure even if the server's BIOS, CPU, and OS are compromised by a malicious adversary, or contain security bugs.

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

The functionality provided by our techniques 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. If such a user wanted to make her data available to all her devices wherever she goes, one solution would be to employ a ***third party online storage service*** (such as Amazon S3 [1] or others) to store her data. At present, however, most (if not all) such third party online storage services require a high level of trust in the service provider and its servers, including the software running on these servers, and the administrators of these servers. Our techniques significantly reduce this requirement by only requiring that the user trust in the TPM 1.2 chips on the storage servers, without needing to trust the servers' BIOS, CPU, OS, and administrators. Aside from giving the user more security when using mainstream online storage services, this new ability would also enable a user to potentially make use of machines owned by ordinary users, such as in a ***peer-to-peer network***. As long as these other users' machines have a certified working TPM 1.2 chip, a user need not trust the owner of these machines, or the software running on these machines.

Furthermore, by specifically addressing the problem of ***replay attacks***, our techniques also enable new applications where the user herself may be considered the untrusted party. In Sect. 6, for example, we show how our techniques can be used to implement ***one-time certificates***. These are certificates which can only be successfully "spent" at most once. The spending of a one-time certificate generates a proof which includes the identity (or pseudonym) of the entity who collected the one-time certificate. The proof can be verified by an independent verifying party at a later time to conclude that the certificate is authentic and has been collected by the entity whose identity is included in the proof. Significantly, the verifying party does not require any contact with the party who issued the certificate or the (TPM of the) user who received the certificate from the issuing party. The verifying party only needs to trust the TPM 1.2 chip in the user's machine. A one-time certificate cannot be spent multiple times, that is, only one correct proof can be generated as a result of the spending of a one-time certificate and proofs cannot be forged even if the user is able to hack her machine or OS. One-time certificates enable many interesting and potentially very useful offline applications, including offline payment and DRM applications, as well as any other applications requiring offline one-time authentication, authorization, or access control.

This paper is organized as follows. In Sect. 2, we begin by presenting the various problems and challenges in achieving our goal. We also present our overall approach, which is the use of ***virtual monotonic counters*** implemented using a TPM 1.2 chip without relying on a trusted OS. Section 3 contains related work. Section 4 shows how TPM 1.2 can be used to implement two basic primitives. We explain how a large number of virtual counters can be managed by using these primitives. This results in a "log-based scheme", which can be improved by using "sharing" and "time-multiplexing". In Sect. 5 we present the results of both a theoretical performance analysis, as well as experimental results from an actual implementation using a commodity PC with a TPM 1.2 chip. Finally, in Sect. 6, we discuss the implementation of one-time certificates, before concluding our paper in Sect. 7. The Appendix of this paper contains more details, including security and performance analysis, and a replication scheme.

2 Problem Statement and Overview of Our Approach

2.1 The Problem

Any system that implements trusted storage 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 three issues, freshness is the most challenging problem for our scenario.

The privacy of a client's data can easily be achieved through encryption, while its authenticity can be ac-

complished 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 truly the most recent version available. This replay attack problem is particularly significant in applications where there is concrete (e.g., financial) benefit to be gained from a successful replay attack, such as, for example, if the data involved represents money, licenses, or some other kind of "stored value" such that rewinding this value to an older one can allow the adversary to gain more goods or services that he would otherwise be entitled to.

2.2 Limited Solutions

The traditional defense against replay attacks is the use of ***timestamping*** [7, 23]. 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 (i.e., $Sign_{SK_A}(H(data_A)\|ctrID_A\|ctrVal_A)$).[1] 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. Note that this technique also works with ***replication***. In this case, a client's device would store the same file record and the same time stamp to multiple storage servers, but would only need to retrieve data with a current timestamp from one of the servers.

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. In this paper, however, we wish to handle the general case where ***the client's different devices may be offline at any possible time***. Thus, it is possible, for example, that at each moment only at most one of the client's devices is online while all the other devices are offline or unreachable. This makes it impossible for the client's devices on their own to reliably and securely maintain and agree on the current value of the dedicated counter.

Ideally, what is needed in this case is a trusted third party that is always online, and that can be trusted to correctly maintain the client's dedicated counter(s). Note, however, that in the scenarios that we are interested in, ***the clients cannot rely on an online trusted third party***. Thus, this solution does not work either. A solution is possible, however, if we allow an online ***untrusted third party*** (whether it is the storage server itself, or another untrusted server that is online and reachable by the client devices) to at least have a ***trusted module*** with certain features. The TCG's TPM chip [50] is increasingly becoming available as a standard component in new commodity PCs. 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***

[1] The storage server can verify the client's signature as well as check whether the timestamp on the new version is newer than the timestamp on the version currently stored on the server. This protects the server from a "fake update" attack by devices not owned by the client. Note, however, that if the server does not perform this check, the client (Alice) would still be able to detect such attacks herself, and would consider the server responsible for failing to prevent the attacks.

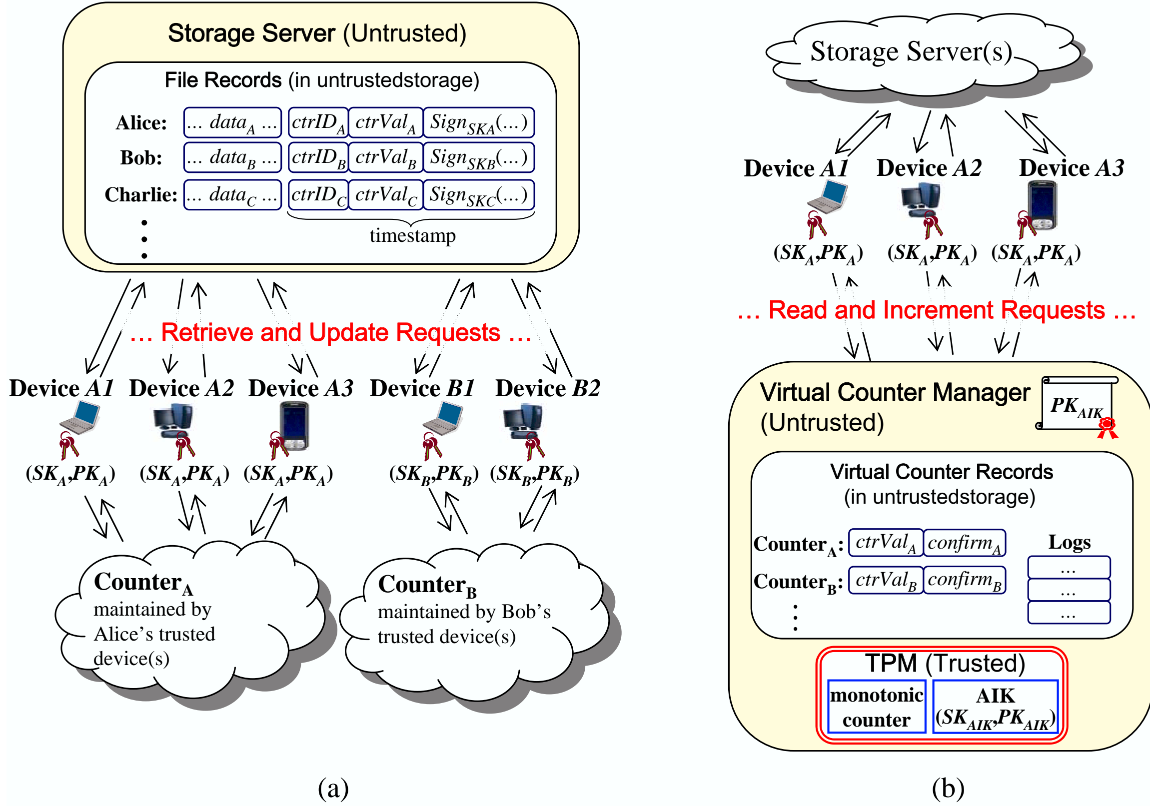

Figure 1: Trusted Storage on Untrusted Servers with Timestamping. (a) Traditional approach: assume a dedicated counter maintained by the client's devices. (b) Our approach: use an untrusted virtual counter manager with a trusted TPM chip (no trusted BIOS, OS, or CPU required). Note that the virtual counter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

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 [34, 42], as well as to provide more security in distributed and peer-to-peer systems [16, 6]. 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 [51] and Microsoft [41].

The problem with techniques that rely on trusted boot 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 [20]. 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.

2.3 Our Solution

In this paper, we present a new solution that, unlike previous solutions, relies only on the TPM 1.2 chip itself. As shown in Fig. 1.b our scheme allows a virtual counter manager to be implemented using a host machine where ***all the components including the BIOS, CPU, memory, storage, OS, and all software can be untrusted except for the TPM***. As we will show, we are able to do this not by using the TPM's trusted boot-related features, but by using TPM 1.2's built-in ***monotonic counter*** feature. Since the TPM effectively only has one built-in monotonic counter, however, using this monotonic counter to securely implement a virtual counter manager that can handle a potentially unlimited number of virtual counters is not straightforward.

If the TPM had the built-in ability to keep track of a large number of ***dedicated deterministic monotonic counters***, then we could directly use the TPM to keep track of each client's monotonic counter. In response to a read request from a client, for example, the host would simply invoke the TPM, which would then return a certificate with the desired counter's current value signed by the TPM's unique private key (i.e., its ***AIK***, described in Sect. 4.1). Similarly, in response to an increment request, the host would invoke the TPM, which would then increment the counter (by 1), and return a signed certificate with the counter's new value. In both cases, the resulting certificate and the TPM's signature on it (which the client can verify if the public key of the TPM's AIK is certified by a trusted authority) would be enough "proof" to convince the client that the counter values returned are in fact fresh (i.e., current). Additionally, in the case of an increment, the certificate also proves that the counter has actually been irreversibly incremented, and implicitly, that the *previous* counter value was equal to the new counter value minus 1.

Unfortunately, however, existing TPM 1.2 chips actually only support the use of ***only one built-in monotonic counter*** during a particular boot cycle.[2] The challenge, therefore, is to be able to produce similar "proofs of freshness" as provided by the read and increment certificates described above, for an unlimited number of virtual counters using only this single built-in monotonic counter. Our solution to this problem is the ***log-based scheme***, which we present in Sect. 4. As we will show, the log-based scheme is able to support an unlimited number of virtual monotonic counters using a single built-in monotonic counter by sacrificing the deterministic property of individual monotonic counters, and supporting only ***non-deterministic*** monotonic virtual counters, wherein the value of a particular virtual counter is guaranteed to always increase, but can increase by unpredictable amounts. Although non-deterministic monotonic counters do not have as strong properties as deterministic ones, they are still sufficient for use with the timestamping scheme we have outlined earlier.

In our solution we support, besides read and increment protocols, that do return proofs of freshness also faster read and increment protocols that do not return proofs of freshness. This is useful if the freshness and validity of a retrieved counter value only needs to be verified by a client's device if it needs to perform a critical operation that relies on the freshness and validity of the counter value (where a critical operation can be, for example, the commit point of a transaction). Since a critical operation also relies on whether past increments by the client's devices were based on retrieved counters that were fresh and valid, a client's device should be able to use the proofs as supplied by the read-with-proof and increment-with-proof protocols to check whether the history of increments was based on fresh counter values. Therefore, we need to satisfy the stronger requirement that a client's device is able to use the proofs in the read- and increment-with-proof protocols to verify whether the client's counter has behaved like a valid counter towards the client; ***a counter behaves like a valid counter towards client i if the counter value that a device of client i most recently retrieved from the virtual counter manager before initiating an increment protocol (with or without proof) is the same counter value that was the result of the most recently successfully executed increment protocol by any of i's devices*** (see [14] for a similar definition of valid storage).

[2] TPM 1.2 chips can keep track of at least 4 monotonic counters at the same time, but once one is incremented, the others cannot be incremented until the machine is rebooted [52].

3 Related Work

Protecting and validating the integrity of data storage has been a well studied topic due mainly to its high importance to such a wide range of applications. For this, cryptographic one-way hash functions [18] are often used by a client to create a small local checksum of large remote data. Merkle proposed hash trees (authentication trees) as a means to update and validate data hashes efficiently by maintaining a tree of hash values over the objects [37]. Recent systems [17, 21, 14, 30, 35] make a more distinct separation between untrusted storage and a trusted compute base (TCB), which can be a trusted machine or a trusted coprocessor. These systems run a trusted program on the TCB (usually a trusted machine or machine with a trusted coprocessor) that uses hash trees to maintain the integrity of data stored on an untrusted storage. The untrusted storage is typically some arbitrarily large, easily accessible, bulk store in which the program regularly stores and loads data which does not fit in a cache in the TCB.

The work on certificate authentication trees in [25] has led to the introduction of authenticated dictionaries [40] and authenticated search trees [9, 8]. In the model of authenticated dictionaries a trusted source maintains all the data which is replicated over multiple untrusted directories. Whenever the trusted source performs an update, it transmits the update to the untrusted directories. The data is maintained in an authenticated tree structure which root is signed together with a timestamp by the trusted source. If a user/client queries an untrusted directory, then it uses the tree structure and the signed root to verify the result. A persistent authenticated dictionary [2, 22, 31, 32] maintains multiple versions of its contents as it is modified. Timeline entanglement [33] creates a tamper-evident historic record of different persistent authenticated dictionaries maintained by mutually distrusting sources.

Byzantine-fault-tolerant file systems [12, 13] consist of storage state machines that are replicated across different nodes in a distributed system. The system provides tamper-evidence and recovery from tampering, but ***both*** properties rely on the assumption that at least two-thirds of the replicas will be honest. The expectation is that replicas are weakly protected but not hostile, so the difficulty of an adversary taking over k hosts increases significantly with k. Byzantine-fault-tolerant file systems distribute trust but assume a threshold fraction of honest servers.

SUNDR [36, 28] is another general-purpose, multi-user network file system that uses untrusted storage servers. SUNDR protects against ***forking attacks*** which is 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. SUNDR does not prevent forking attacks, but guarantees ***fork consistency***, which essentially ensures that the system server either behaves correctly, or that its failure or malicious behavior will be detected later ***when users are able to communicate with each other***. This is achieved by basing the authority to write a file on the public keys of each user. Plutus [24] is another efficient storage system for untrusted servers that cannot handle these forking attacks.

In our system, we place a small TCB in the form of a TPM at an untrusted third party for maintaining virtual counters for timestamping to allow a client's device to ***immediately*** (without the need to communicate to any of the other client's devices) detect misbehavior whenever a critical operation needs to be performed, this includes replay attacks. This prevents forking attacks and guarantees the freshness, integrity, and consistency of data.

We reduce the trusted computing base to only a single TPM 1.2 [52, 39], which is a standard component on machines today. This differs from many other systems that require complex secure processors [46, 5, 29, 53, 48, 49, 47]. (We notice that in [45], Shapiro and Vingralek address the problem of managing persistent state in Digital Rights Management (DRM) systems. They include volatile memory within their security perimeter, which increases the TCB.)

We use the TPM as a secure counter which can be used to timestamp events according to their causal relations. An order in logical time can be extracted if we know which events logically cause other events. Lamport clocks [26, 27] are conceptual devices for reasoning about event ordering. Our scenario with a centralized untrusted third party which implements a virtual counter manager with access to the TPM does not need Lamport clocks to reason about logical time. Our difficulty is how to reason about malicious behavior.

For completeness we mention that in accountable time-stamping systems [10, 11] all forgeries can be explicitly proven and all false accusations explicitly disproven; it is intractable for anybody to create a pair of contradictory attestations. Buldas et al. [9] introduced a primitive called ***undeniable attester***. Informally, the attester is such that it is intractable to generate both a positive and a negative attestation based on an element x and a set S such that the attester concludes $x \in S$ for the positive attestation and $x \notin S$ for the negative attestation. Their intention is to prove the existence or non-existence of objects in a database. An ***authenticated search tree*** [25, 40, 9, 8] can be used for the construction of an undeniable attester.

One technique also worth noting is that described by Schneier and Kelsey for securing audit logs on untrusted machines [43, 44]. Each log entry contains an element in a linear hash chain that serves to check the integrity of the values of all previous log entries. It is this element that is actually kept in trusted storage, which makes it possible to verify all previous log entries by trusting a single hash value. The technique is suitable for securing append-only data that is read sequentially by a verifying trusted computer.

Another concept worth noting is that of a certificate revocation list (CRL), which is a list which is signed by a certification authority (CA) together with a timestamp. By obtaining the most recent CRL one can easily verify whether a certificate is still valid. Based on CRLs, less communication intensive solutions are proposed in [38, 25, 40]. In [38] the idea is to sign a message for every certificate stating whether it is revoked or not and to use an off-line/on-line signature scheme [19] to improve the efficiency. The idea to use a log of certificates in which each of the certificates attests to a positive or a negative action is a general principle which we also use in our approach (we use a log of increment certificates in which each of the increment certificates attests to whether a certain virtual monotonic counter has been incremented or not).

4 Log-Based Scheme TPM 1.2 Implementation

In order to implement a virtual counter manager based on TPM 1.2 we use one of the TPM's built-in physical monotonic counters as a "***global clock***". Essentially, we use the TPM with its global clock as a timestamping device to timestamp each virtual counter. Whenever we wish to increment a virtual counter we increment the global clock and timestamp the virtual counter's ID with the incremented value of the global clock. We then define the value of a specific virtual counter as the value of its most recent timestamp. In other words, ***the value of a particular virtual counter is defined as the value of the global clock at the last time that the virtual counter's increment protocol was invoked***.

Note that this results in ***non-deterministic*** monotonic virtual counters. The unpredictability of the virtual counter's increments relative to the global clock means that a client needs to check every increment that is performed on the global clock.[3] That is, in order to verify whether a retrieved virtual counter value is fresh and valid, the client's device needs to examine a ***log*** of timestamps that resulted from increments on the global clock. Checking each timestamp determines whether the client's particular virtual counter has been incremented (i.e., timestamped by the global clock) more recently.

In this section, we explain the details of our ***log-based scheme*** based on this idea. We first explain how to use a TPM 1.2 chip to implement two global clock primitives: *IncAndSignClock* – which increments and signs the resulting global clock value together with an input nonce, and *ReadAndSignClock* – which reads and signs the current value of the global clock together with an input nonce. We then show how these primitives can be used to implement read and increment protocols for individual virtual counters.

4.1 Global Clock Operations using TPM 1.2

A TPM 1.2 chip has three features which are useful for our purposes. First, the TPM has the ability to hold an ***attestation identity key (AIK)*** which is a unique signing keypair, whose private key is never revealed outside the

[3] Except when using time-multiplexing, which is described in Sect. 4.4.

TPM, and whose public key is certified by a trusted third party (and can be verified through this certificate without contacting the trusted third party). Second, the TPM has at least one built-in (or "physical") ***monotonic counter*** whose value is non-volatile (i.e., it persists through reboots), and monotonic (i.e., it can be increased by 1, but it can never be reverted to an older value, even if one has complete physical access to the entire machine hosting and invoking the TPM). Third, the TPM supports ***exclusive and logged transport sessions***, which allows the TPM to prove to an external party that it has executed certain operations (atomically) by signing (with the AIK) a log of the operations performed on the TPM (together with their inputs and outputs), together with an anti-replay nonce. These features allow us to implement our schemes by using the TPM's built-in monotonic counter as the global clock, and using the AIK and transport sessions to produce trusted signatures, or timestamps, using this global clock.

Specifically, we implement the *IncAndSignClock*($nonce$) primitive by using the TPM's built-in TPM_Increment_Counter command (which increments the TPM's built-in monotonic counter) inside an exclusive and logged transport session using the AIK as the signing key. This produces a signature over a data structure that includes the anti-replay nonce and a hash of the transport session log, which consists of the inputs, commands, and outputs encountered during the entire transport session. This signature can then be used together with the input nonce $nonce$ and the transport session log, to construct an ***increment certificate***. Note that by making this transport session exclusive, we ensure that the TPM will not allow other exclusive transport sessions to successfully execute at the same time. This ensures the ***atomicity*** of the increment operation.

The verification algorithm for such an increment certificate is as follows: First, it checks that the nonce in the certificate is the same as the input nonce. If they are, the input nonce $nonce$ together with the transport log, the signed output, and the certified public key of the TPM's AIK is used to verify the certificate. Finally, if the certificate verifies as valid, the algorithm retrieves the global clock's value, which is included in the transport session log of inputs and outputs as part of the certificate.

The *ReadAndSignClock*($nonce$) primitive is implemented like the *IncAndSignClock*($nonce$) primitive where the TPM_Increment_Counter command in the transport session is replaced by the TPM's built-in TPM_Read_Counter command. In this case, instead of an increment certificate, we produce a ***current global clock certificate*** certifying the current value of the global clock (i.e., TPM's built-in monotonic counter).

4.2 Log-Based Scheme Protocols Overview

We begin by assuming that each client has her own unique public-private key pair stored in each of her devices. A 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.

On the manager-side, we assume that the ***virtual counter manager*** has a certified trusted TPM 1.2 chip, and some software, memory, and persistent (e.g., disk) storage, which can all be untrusted. The software on the virtual counter manager 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*** most recent confirmation certificate.

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

1. ***Increment-without-validation (aka 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 (aka Fast-Read)***, in which the virtual counter manager returns the current value of a virtual counter,

3. ***Read-with-validation (aka 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 (aka Full-Increment)***, which combines the increment-without-validation and read-with-validation protocols 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 current global clock certificate or a new increment certificate.

Given these, a client can reconstruct the global clock values at which the virtual counter was incremented since the creation of the most recent confirmation certificate. This reconstruction detects any malicious behavior in the past. Specifically, the reconstruction of past increments is used to determine whether the virtual counter values on which these increments were based are valid. That is, as described in Sect. 2, for each of the past increments by any of the client's devices it is checked 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. This check is made possible by having an increment certificate also certify the value on which the corresponding increment is based (hence, the consistency of the reconstructed list of increments can be verified). (We note that in order to verify the freshness of data it is sufficient to verify the validity of the counter values on which past increments were based since only newly incremented values are used for timestamping data in our virtual storage application.)

4.3 Protocols Details

We proceed with the details of the different protocols:

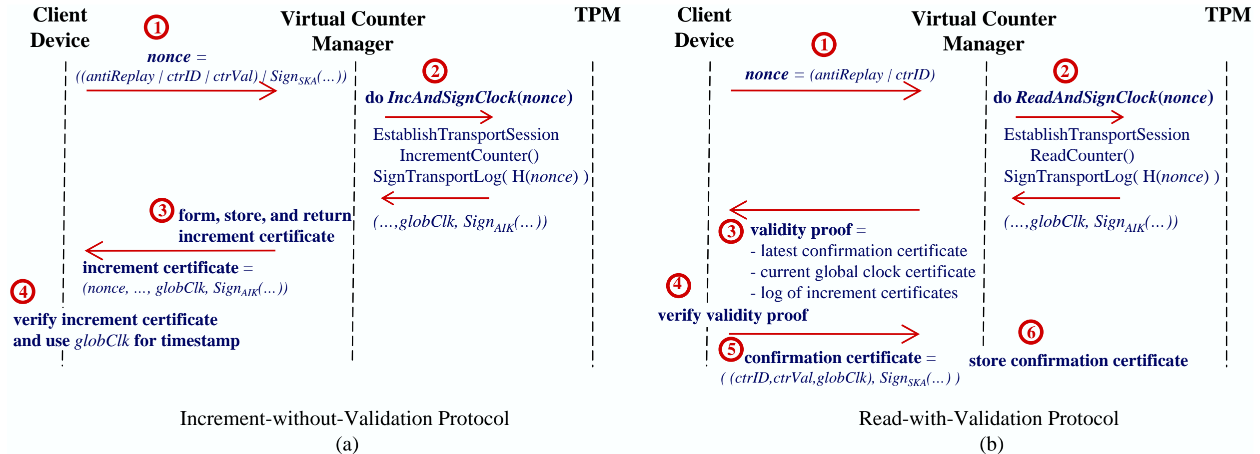

Figure 2: Protocols. (a) Increment-without-Validation. (b) Read-with-Validation. (Note: Increment-with-Validation is similar to Read-with-Validation, except that an increment operation is performed and the increment certificate is used in place of the current global clock certificate.)

Increment-without-Validation protocol: In Figure 2.a shows the interaction between a client device, the virtual counter manager, and its TPM during an increment-without-validity protocol. If a client's device wants to increment one of the client's virtual counters, then it selects a random anti-replay nonce $antiReplay$ and concatenates

the anti-replay nonce, the counter identity $ctrID$ of the virtual counter which needs to be incremented, and the current value $ctrVal$ of this counter according to the knowledge of the client's device. Let SK be the client's secret key. The device computes

$$nonce = (conc \| Sign_{SK}(conc)) \text{ where } conc = (antiReplay \| ctrID \| ctrVal). \quad (1)$$

The nonce is forwarded to the virtual counter manager with the request to use $nonce$ as the input nonce of the *IncAndSignClock* primitive (step 1 in Fig. 2.a). Besides verifying the nonce's signature, the virtual counter manager checks whether the current value of the counter with ID $ctrID$ is equal to $ctrVal$. If not, then the virtual counter manager notifi es the client's device about its out-of-date knowledge. If $ctrVal$ does match the current counter value, then the virtual counter manager uses the TPM to execute *IncAndSignClock*($nonce$) (step 2 in Fig. 2.a) and the resulting increment certifi cate is send back to the client's (step 3 in Fig. 2.a) device who verifi es the certifi cate. In this scheme we do not protect against denial of service; if the increment certifi cate 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 certifi cate is accepted (that is, its verifi cation passed in step 4 in Fig. 2.a), the client's device may use the new counter value to timestamp data. If the client's device accepts the increment certifi cate, then we call the increment ***successful***.

Since the anti-replay nonce is chosen at random, replay attacks of previously generated increment certifi cates (by, for example, a malicious virtual counter manager or a man-in-the-middle) will be detected by the client's device. We will show that the validity of a client's virtual counter can be verifi ed in the read-with-validation protocol (even in the presence of a malicious virtual counter manager but with a trusted TPM) because the client's device's knowledge of the current counter value $ctrVal$ is included in the input nonce of the *IncAndSignClock* primitive. The role of the counter ID in the input nonce is to distinguish which increment certifi cates correspond to which virtual counters. The nonce's signature proves the authenticity of the request towards the virtual counter manager. The signature is also used in the read-with-validation protocol to prove that each increment certifi cate originated from an authentic request and not a fake increment.

Read-without-Validation protocol: In the read-without-validation protocol, a client's device asks for the most recent value of a specifi c virtual counter. The virtual counter manager simply signs and returns the most recent counter value without making use of the TPM.

Read-with-Validation protocol: In Figure 2.b shows the interaction between a client device, the virtual counter manager, and its TPM during an read-with-validity protocol. If a client's device wants to read and obtain a validity proof of the value of one of the client's virtual counters, then it selects a random anti-replay nonce $antiReplay$. Let $ctrID$ be the counter ID of the virtual counter which current value needs to be returned by the protocol. The concatenation,

$$nonce = (antiReplay \| ctrID),$$

is forwarded to the virtual counter manager (step 1 in Fig. 2.b) with the request to use it as the input nonce of the *ReadAndSignClock* primitive. The virtual counter manager uses the TPM to execute *ReadAndSignClock*($nonce$) (step 2 in Fig. 2.b). The resulting current global clock certifi cate is transmitted to the client's device together with the most recent confi rmation certifi cate of the virtual counter with identity $ctrID$ and a log of each of the increment certifi cates which were generated since the creation of this most recent confi rmation certifi cate (step 3 in Fig. 2.b). The sequence of certifi cates in this transmission form a validity proof as depicted in Figure 3.

The most recent confi rmation certifi cate is a certifi cate generated by one the client's devices during a previous read-with-validation or increment-with-validation protocol. It is a certifi cate of the concatenation of $ctrID$, the value $globClk'$ of the global clock at the time when either protocol was executed, and the value $ctrVal'$ of the counter with identity $ctrID$ at the time when either protocol was executed. That is, the confi rmation certifi cate

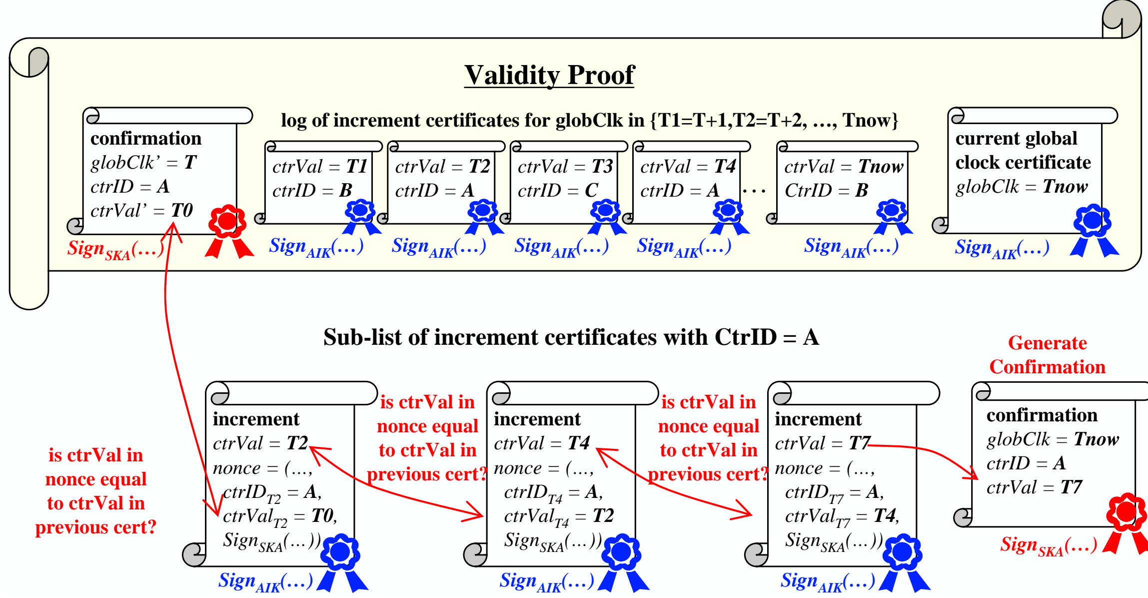

Figure 3: The Validation Process. After receiving the validity proof, the client checks the AIK signatures on all the increment certifi cates, extracts the certifi cates with increments for its counter ID, and verifi es that each increment was done with the correct knowledge of the previous value of the virtual counter. If this validation succeeds, the client produces a new confi rmation certifi cate.

is a pair

$$(confirm, Sign_{SK}(confirm)) \text{ where } confirm = (ctrID \| ctrVal' \| globClk') \tag{2}$$

and SK is the client's secret key.

Let $globClk$ be the global clock value as certifi ed by the current global clock certifi cate. Since the anti-replay nonce is chosen at random, replay attacks of previously generated current global clock certifi cates will be detected by the client's device. Therefore, we may assume that $globClk$ represents the current global clock value.

The client's device also verifi es each of the signatures that constitute the log of increment certifi cates (which were generated since the creation of this most recent confi rmation certifi cate) and uses the client's public key to verify the confi rmation certifi cate. Then, for each value t in the range

$$globClk' < t < globClk, \tag{3}$$

there should exist an increment certifi cate for which its verifi cation algorithm retrieves the value t (see the log of increment certifi cates in Fig. 3). See (1), let

$$conc_t = (antiReplay_t \| ctrID_t \| ctrVal_t) \tag{4}$$

be part of the input nonce of the increment certifi cate for t. The list of these input nonces contains a sublist of input nonces with $ctrID_t = ctrID$ (see the sublist of increment certifi cates in Fig. 3). The sublist of these input nonces corresponds to all the increments of the counter with identity $ctrID$ during the period where the global clock value ranged from $globClk'$ to the current global clock value $globClk$. See (1), each of the input nonces within the sublist contains within itself a signature that can be verifi ed by using the client's publlic key. This detects whether the corresponding increment certifi cates originated from an authentic request and not a fake increment.

The sublist can also be used by the client's device to detect whether the increments are based on valid counter values. See (4), let $ctrVal_t$ be one of the values in the input nonce within the sublist. Then, counter value t

resulted from an increment protocol which was initiated by one of the client's devices who thought that, just before the start of the increment protocol, the value of the counter is equal to $ctrVal_t$. Hence, if the counter with identity $ctrID$ has behaved like a valid counter, then, for each pair of consecutive input nonces with $ctrVal_t$ and $ctrVal_T$, $t < T$, within the sublist,

$$t \text{ should be equal to } ctrVal_T. \tag{5}$$

If this check passes, then the last value $ctrVal$ within the sublist is the current value of the virtual counter with identity $ctrID$ and the client's device transmits to the virtual counter manager (step 5 in Fig. 2.b) a new confirmation certificate as in (2) in which

$$confirm = (ctrID \| ctrVal \| globClk).$$

Increment-with-Validation protocol: The increment-with-validation protocol first executes the increment-without-validation protocol. The resulting increment certificate contains the current global clock value as the incremented virtual counter value. In this sense the increment certificate can also function as a current global clock certificate. Therefore, in order to provide a validity proof after the execution of the increment-without-validation protocol, the read-with-validation protocol can be executed without using the *ReadAndSignClock* primitive.

4.4 Improvements

In this subsection we explain two techniques that can be used to improve the performance of the log-based scheme. The details of both techniques are presented in Appendix A.

Sharing. One problem with the log-based scheme as described so far is that each read and increment primitive on a virtual counter requires the TPM to produce a signature using its AIK. As we will show in Sect. 5, such a signature operation typically takes around 1 s using existing TPM 1.2 chips today. Moreover, TPM 1.2 chips also ***throttle*** increment operations on the TPM's built-in monotonic counter to prevent wear-down of the TPM's non-volatile memory. This means that increment operations are only possible once every 2 to 5 seconds (depending on the manufacturer). As we will show in Sect. 5, 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 global clock primitive. The general idea here is to collect the individual nonces of each increment protocol and to construct a single shared nonce which can then be used as an input to a single shared $IncAndSignClock$ 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.

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 a an increment or read protocol. In order to reduce the effects of this problem we may use an ***adaptive schedule***.

Here, a virtual counter's schedule is allowed to change. For example, the client's devices may agree to a back-off strategy, that is, immediately after a successful increment, the allowed slots for a virtual counter are close to each other and as the virtual counter is not used, the allowed slots get spaced farther and farther apart according to a known deterministic formula.

Security. In Appendix B we sketch a proof of security of the log-based scheme with sharing and time-multiplexing. We also discuss what happens if the power to the virtual counter manager fails some time after the TPM_Increment_Counter (in the $IncAndSignClock$ primitive) but before the virtual counter manager is able to save the increment certificate to disk.

Replication. So far, our schemes provide ***tamper-evident*** trusted storage. That is, they guarantee that any incorrect behavior 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. One such scheme is outlined in App. C.

Tree-Based Scheme. For completeness, we mention that in a previous work [3, 4] 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 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 [4]. 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. Thus, in the end, we expect that 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.

5 Performance

This section outlines both theoretical and experimental results on the performance of our log-based scheme with and without sharing.

Theory. In Appendix D we analyse a theoretical scenario for the log-based scheme with sharing, with fixed schedules for time multiplexing, and with no other protocols than the increment-with-validation protocol, where we assume (for reasons of simplicity) that, given a specific virtual counter, its increment requests arrive according to a Poisson distribution which is the same for all virtual counters. The appendix characterizes the expected

latency between the request and finish of a shared increment-with-validation protocol in terms of the number of virtual monotonic counters, the speed at which requests arrive, the processing speed of the TPM, the time needed to set up a network connection, and the speed of transmission over the network. The appendix shows that if, for each virtual counter v, the TPM's computation of one $ReadAndSignClock$ or $IncAndSignClock$ primitive is fast enough with respect to the speed at which v's increment requests arrive, then the network bandwidth is the limiting factor, just as in any storage application. This means that our approach is efficient in that it has the potential to serve as many clients, just as in any storage application.

Experiment. We performed experiments that were designed to demonstrate how the latency and network bandwidth are effected as the number of managed virtual monotonic counters increases. We implemented a ***client host***, which simulates a number of clients, where each client is responsible for reading and incrementing one virtual counter, and a ***virtual counter manager***, which contains a TPM 1.2 chip and performs the read-with-validation and increment-without-validation protocols. The client host ran a variable number of clients, with each client running as a separate thread in the simulation software. Each client built a schedule of events for a particular virtual counter ahead of time according to a Poisson distribution with a period of 15 seconds. Each event was randomly chosen to be either a request of a read-with-validation or an increment-without-validation.

We used two client hosts, an HP Pavilion 762n, with a 2.26GHz Pentium 4 and 512MB of RAM, and a Gateway M-465 with a 1.83 GHz Core Duo and 2GB RAM. The counter server was an HP Compaq dc7600, with a Pentium 4 3.40GHz processor, 1GB of memory, and a Broadcom TPM 1.2 chip. Measurements showed that with this TPM a $ReadAndSignClock$ primitive can be executed by the server in 0.9 seconds, and an $IncAndSignClock$ primitive takes 1.3 seconds. Furthermore, we found that the Broadcom TPM 1.2 chip throttles the monotonic counter increment operations so that we could only execute one $IncrementAndSignClock$ primitive every 2.15 seconds. The simulation software was implemented using Sun's Java 5.0, and a Java API that we wrote which accesses the TPM via the `/dev/tpm0` device in Linux. Network communication was handled using Java RMI, with each simulated client connecting to the server through a separate RMI connection.

Figure 4 shows the results of several experiments which was each run for a total of 36 minutes (5 minutes for warm-up, 30 minutes for data collection). For each experiment, the virtual counter manager used either the log-based scheme without sharing (LB) or the log-based scheme with sharing (SLB). (In the case with sharing, we used a simple list of individual nonces, not an authenticated search tree, for the shared nonce.) We ran several experiments with different numbers of virtual counters, and measured the average total latency that each client perceived for read and write requests, as well as the size of the output generated from the counter server.

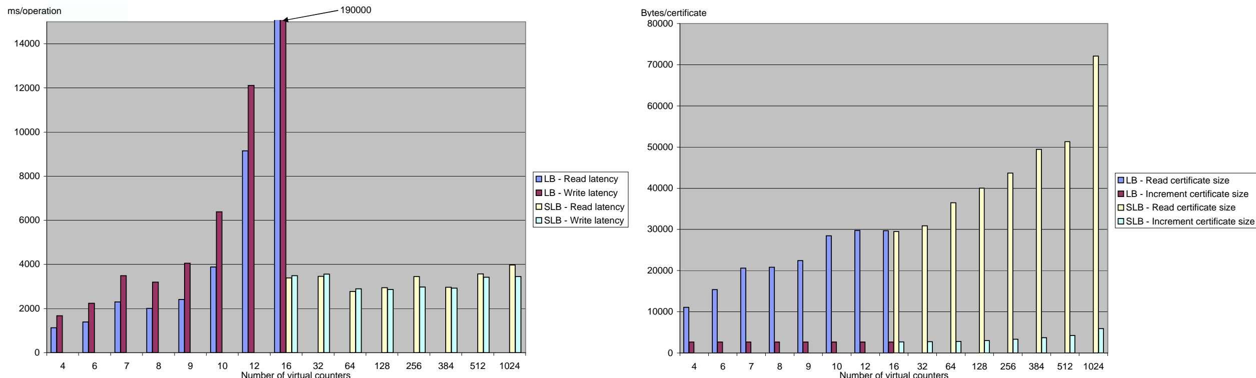

Figure 4: Simulation Results.

Note that for the ***log-based scheme without sharing***, the latency blows up around a total of 10 to 12 virtual counters. This is consistent with our expectation. That is, if no sharing is used, then the processing speed of the TPM is likely to be the bottleneck. In this case, we note that on average, the average amount of time for the TPM

to service a request (averaged over all read and increment requests received over the 30 minute period) is expected to be (and was) between ≥ 1.1 and ≤ 1.525.[4] Thus, since read and increment protocols for each virtual counter are requested every 15 seconds, we expect the virtual counter manager to support between $15/1.525 = 9.8$ and $15/1.1 = 13.6$ virtual counters.

In the log-based scheme ***with sharing***, we are limited by the speed at which we can process our requests or send replies over the network. As shown in Fig. 4, and as predicted by our theory, we did not reach the bottleneck of the system. That is, even for requests that arrive every 15 seconds per virtual counter, more than 1024 virtual counter were successfully managed using a single TPM, with each client only experiencing an average latency of less than 4 s per request. This shows that the log-based scheme with sharing has a much higher capacity for managing virtual counters than the log-based scheme without sharing, and has potential for actual practical use.

The graph on the right of Fig. 4 shows the a plot of the average sizes of the validity proof for the read requests (labeled here as "read certificate size'), and the increment certificates. Note that the size of the increment certificates generated by the log-based scheme without sharing is constant since they always contain a single nonce. In the log-based scheme with sharing, the size of an increment certificate grows linearly with the number of virtual counters, since it includes the different nonces that participate in the shared nonce. The size of the read certificates is notably bigger in both schemes, but, as expected, is roughly proportional to the expected number of global clock operations between increment requests of the same virtual counter.

6 One Time Certificates

Besides virtual storage, our log-based scheme can also be used to implement one-time certificates. These are certificates which can be spent at most once, and as soon as they are spent the collecting can prove his ownership to any third party. The spending protocol does not involve a trusted third party. One-time certificates can be used for one-time authentication which is useful in DRM applications and can be used for offline payment.

One-time certificates relate to our storage application as follows. In the current storage application data is timestamped by using a virtual counter. As soon as the virtual counter is incremented, the data is invalidated and the data or a new update of the data needs to be timestamped with the incremented virtual counter. The invalidation of the data happens at the moment of incrementing its corresponding virtual counter and is recorded in the log of increment certificates. This can be used to construct one-time certificates which get invalidated as soon as they are spent.

The main idea is that a user with TPM implements his own virtual counter manager in order to maintain virtual monotonic counters some of which corresponding to one-time certificates. The issuer of a one-time certificate asks the user's virtual counter manager to create a new virtual monotonic counter by using an increment protocol. The issuer signs the identity and value of the counter together with the content of the one-time certificate by using the issuer's secret key; the signature together with the content forms the one-time certificate. If the user wishes to spend the one-time certificate, then the entity who collects the certificate asks the user's virtual counter manager to execute an increment-with-validation protocol. The resulting proof of validity is used to extract the current value of the virtual counter. If this value matches the one-time certificate, then the entity accepts the one-time certificate (if it does not match, then the certificate has already been spend). The proof of validity contains the increment certificate of the increment-with-validation protocol which is based on a nonce that includes a signature with the secret key of the collecting entity. Therefore, the entity can use the proof of validity to prove to any third party that it corresponds to both the one-time certificate as well as his ownership.

[4] Note that the Poisson processes we are using would tend to generate an equal number of read and increment requests on average. If all the reads and write interleave perfectly, then a write would never need to wait, so the average time over all reads and increments together would be roughly $(0.9 + 1.3)/2 = 1.1$ s. The worst case is if all the increments happen consecutively without interleaving reads, in which case, an increment operation would need take 2.15 s each and the average time over all requests would be roughly $(0.9 + 2.15)/2 = 1.525$.

Migration of a one-time certificate can be accomplished by making the anti-replay nonce in the increment protocol that generates a migrated one-time certificate dependent on the proof of validity of a to be migrated one-time certificate. We will provide details and an implementation in a forthcoming paper. We also notice that our concept of one-time certificates can be extended to n-time certificates by using the same ideas.

7 Conclusion

We introduced, implemented, and analysed a new practical virtual storage system which security is solely based on trusting a TPM 1.2. As demonstrated by experiments and theory, it provides 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 with respect to one another. P2P distributed storage can be attained by using replication on top of our storage system. Our core technique is a log-based scheme which uses the TPM to manage a large number of virtual counters. We showed that the log-based scheme can also be used to implement one-time certificates, which are certificates that can be spent at most once.

A Appendix: Sharing and Time-Multiplexing

In this appendix we detail two methods that can be used by the virtual counter manager to schedule executions of the protocols in the log-based scheme.

Sharing: Multiple increment protocols can be executed in parallel by sharing a single *IncrementSignClock* primitive. The advantage of sharing is that the TPM only needs to execute the *IncrementSignClock* primitive once. We use the idea of an undeniable attester [9, 8] as explained in the related work section. An undeniable attester can be implemented as an authenticated search tree [9, 8]. A directed binary tree is a search tree [15] if every node n in the tree is associated to a unique search key $k[n]$ such that if n_L is the left child of n then $k[n_L] < k_n$ and if n_R is the right child of n then $k[n] < k[n_R]$ (here, $<$ is an ordering over keys). In an authenticated search tree the labels $l[n]$ of nodes n in the tree are recursively defined by $l[n] = H(v[n])$ with $v[n] = (l_L, k[n], l_R)$, where $l_R = l[n_R]$, if n's right child n_R exists, and $l_R = nil$, otherwise, and where $l_L = l[n_L]$, if n's left child n_L exists, and $l_L = nil$, otherwise (we call $v[n]$ the value of node n). Let k be the value of a search key. As is explained in [8], given the correct value of the root of an authentication tree, the ***(unique) existence or non-existence*** of a node in the authentication tree with search key k can be proved by giving the labels of a path from a leaf to the root.

If m client's devices request the execution of an increment protocol, then we propose to order their input nonces according to the counter IDs, see (1), in an authenticated search tree that has a depth of $\log m$ levels. This is possible if the virtual counter manager only accepts one request per counter ID. The virtual counter manager uses the root of the authenticated search tree as the input nonce of the *IncrementSignClock* primitive. The resulting increment certificate is transmitted to each of the devices. Also, each device receives a $\log m$ sized proof of the unique existence of a nonce within the authenticated search tree that corresponds to the counter ID corresponding to the device's request. Notice that the sharing of a single *IncrementSignClock* primitive increments all the corresponding virtual counter values to the same global clock value.

In the read-with-validation and increment-with-validation protocols, a proof of validity should not only include a log of increment certificates but also include, for each increment certificate, an existence proof stating that the corresponding authenticated search tree contains an input nonce with $ctrID$ or a non-existence proof stating that the corresponding authenticated search tree contains an input nonces with a counter ID different from $ctrID$. The existence and non-existence proofs determine the sublist of input nonces which correspond to the increments of the counter with identity $ctrID$.

The advantage of this approach is that the TPM only needs to execute the *IncrementSignClock* primitive once (instead of m times). We will show in our discussion on performance that this advantage of not having to use the slow TPM m times will outperform the disadvantage of the increased communication costs. We notice that authenticated search trees can also be used to share a single *ReadSignClock* primitive in the read-with-validation protocol.

Time-Multiplexing: Instead of allowing increments at each possible global clock value for each client, we propose to time-multiplex the global clock. That is, each client associates with each of his virtual counters a ***fixed schedule*** of global clock values that are allowed to be virtual counter values. So, each client's virtual counter has its own fixed schedule of allowable values which is distributed among the client's devices. During each increment protocol, the client's device checks whether the incremented counter value is allowed. If not (that is, the virtual counter manager did not appropriately wait with the execution of the increment protocol), then the increment certificate is rejected and the client's device retransmits its request with the same nonce. The main advantage of time-multiplexing is that the log of increment certificates in a proof of validity of a virtual counter, see (3), 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. A second advantage is that time-multiplexing controls into some extend the size of the authentication trees that are used in shared increment protocols.

In an ***adaptive schedule*** a virtual counter's schedule can change. Therefore, since a client's device should be able to verify whether a virtual counter's value is allowed as a result of an increment protocol, the client's device should be able to retrieve the current virtual counter's schedule during an increment protocol. Only an increment-with-validation protocol may possibly carry the information of the current schedule in its proof of validity. This means that the increment-without-validation protocol should not be initiated for virtual counters with adaptive schedules; even if a client's device needs to perform a non-critical operation, it needs to check the freshness and validity of the retrieved counter value.

For a virtual counter with adaptive schedule, we adapt the increment-with-validation protocol as follows. When requesting an increment, $conc$ in (1) should be replaced by

$$conc = (antiReplay \| ctrID \| ctrVal \| allowableSet),$$

where $allowableSet$ is (adaptively) chosen by the client's device and (is a sequence of parameters which) describes the set (or schedule) of counter values which are allowed as a next increment (after the execution of the increment protocol) of the counter with identity $ctrID$. The virtual counter manager should wait with the execution of the requested increment-with-validation protocol till the next allowable global clock value.

Adaptive scheduling shortens a proof of validity as follows. First, $allowableSet'$, the set of counter values which are allowed as a next increment at the time when the most recent confirmation certificate in the proof of validity was generated, should be included in this most recent confirmation certificate in $confirm$, see (2). We notice that no increment-without-validation protocols are executed for the virtual counter with identity $ctrID$. Therefore, the log of increment certificates which were generated since the creation of the most recent confirmation certificate does not contain increment certificates with input nonces that contain $ctrID$ (we do not take into account the the current-global-clock-counter certificate which is replaced with the newly generated increment certificate of the counter with identity $ctrID$ in the increment-with-validation protocol). The log of increment certficates in the proof of validity can be reduced to those increment certificates for which the corresponding verification algorithm retrieves a value t in the range (3) for which

$$t \in allowableSet'.$$

When verifying the proof of validity, the client's device only checks that the log is complete, that none of the increment certificates within the log corresponds to $ctrID$, and that in case of an increment-with-validation (as opposed to a read-with-validation) protocol the value corresponding to the newly generated increment certificate

is in $allowableSet'$. If these checks pass, then the client's device transmits to the virtual counter manager a new confirmation certificate that includes the current schedule.

Since no increment-without-validation protocols are executed for the counter with identity $ctrID$, we do not need to check the validity of the counter; we only need to check its freshness. Therefore, $ctrVal$ in $conc$, see (1), is not needed in the verification of a proof of validity and can be discarded.

B Appendix: Security Analysis

In this appendix we sketch a proof of security of the log-based scheme with sharing and time-multiplexing. We first consider a virtual counter which is maintained by the log-based scheme with sharing and a fixed schedule. We need to show that if a proof of validity verifies correctly, then the virtual counter is valid. A proof of validity consists of three parts; a confirmation certificate (since a malicious virtual counter manager may provide an old certificate, we do not assume that it is most recent), a log of increment certificates, and a current global clock certificate. The confirmation certificate is signed by the client's secret key and contains the value of a past global clock counter $globClk'$ at which a client's device correctly verified an earlier proof of validity. By assuming that the client's secret key has not leaked to an adversary and by using induction on the global clock counter value in confirmation certificates, we conclude that the virtual counter has been valid up to the moment when the global clock counter was equal to $globClk'$.

The current global clock certificate is the result of the $ReadSignClock$ primitive based on a random nonce to avoid replay attacks of older certificates. If this certificate verifies correctly and if we assume that the TPM is trusted and cannot be compromised, then this certificate contains the current global clock counter value $globClk$.

The increment certificates in the log of increment certificates correspond to values t in the virtual counter's fixed schedule with $globClk' < t < globClk$. By using the properties of authenticated search trees in the log-based scheme with sharing, a client's device can distill the sublist of increment certificates which record the increments of the virtual counter. Since client's devices check whether incremented values are in the schedule of the virtual counter, this sublist necessarily covers the most recent ***successful*** increment of the virtual counter. This means that a proof of validity is a proof of freshness of the virtual counter. Besides freshness, we need to show that each increment certificate in the sublist is the result of an authorized (no fake) increment based on a retrieved virtual counter value that is equal to the current counter value just prior to the increment. This follows directly from the check in (5).

Now consider a virtual counter which is maintained by the log-based scheme with sharing and an adaptive schedule. In this case the log of increment certificates correspond to values t, $globClk' < t < globClk$, which are in the virtual counter's adaptive schedule $allowableSet'$ as mentioned in the confirmation certificate of the proof of validity. Let $conCert'$ denote this confirmation certificate. As our induction hypothesis we assume that each previously generated confirmation certificate correctly records the value of the most recent increment up to the moment that the certificate was generated. Then, besides the adaptive schedule $allowableSet'$, $conCert'$ also records the value $ctrVal'$ ($\leq globClk'$) of the most recent increment up to the moment when $conCert'$ was generated (by our induction hypothesis). That is, no increments happened for global clock counter values t with $ctrVal' < t < globClk'$.

When verifying the log of increment certificates, a client's device verifies that none of the logged increment certificates corresponds to an increment of the virtual monotonic counter. This means that if there would have been a more recent ***succesful*** increment that results in an increment certificate for some value $ctrVal$ in the range $ctrVal' < ctrVal < globClk$ (we notice that, for each increment certificate of the virtual counter, some client's device has verified whether the corresponding authenticated search tree covers at most one increment for this virtual counter, hence, we do not need to consider the case $ctrVal = ctrVal'$), then $ctrVal$ lies outside the counter's adaptive schedule of $conCert'$ if it indicates a global clock counter value after the moment that $conCert'$ was generated, that is, $ctrVal \notin allowableSet'$ or $ctrVal < globClk'$. Since $ctrVal' < ctrVal <$

$globClk$ and no increments happened for global clock counter values t with $ctrVal' < t < globClk'$, we conclude that $ctrVal \notin allowableSet'$.

We notice that if there is a more recent successful increment, then there is also a more recent successful increment with the additional property that no increments happened for global clock counter values t with $ctrVal' < t < ctrVal$. Since the successful increment corresponds to a successfully executed increment-with-validation protocol, the corresponding proof of validity verified correctly and therefore includes a confirmation of proof certificate $conCert''$ with an adaptive schedule $allowableSet''$ with $ctrVal \in allowableSet''$. The successfully executed increment-with-validation protocol also leads to a confirmation certificate. Since this certificate was generated at the moment when the global clock counter value was equal to $ctrVal$, $allowableSet''$ corresponds to the schedule indicated at the most recent increment before $ctrVal$ (by our induction hypothesis). We remind the reader that no increments happened for $ctrVal' < t < ctrVal$, hence, $allowableSet'' = allowableSet'$. This implies a contradiction because we derived $ctrVal \notin allowableSet'$ as well as $ctrVal \in allowableSet''$. We conclude that a proof of validity is a proof of freshness. Therefore, since no increment-without-validation protocols are executed, each successful increment is based on a fresh counter value. Hence, the virtual counter is valid. For a more detailed proof in the random oracle model, we refer to [3] in which we modeled the TPM's functionality as a black box.

A possible problem worth noting is what happens if the power to the virtual counter manager fails some time after the TPM_Increment_Counter (in the $IncrementSignClock$ 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 client's devices will at least detect the gap during a verification of a proof of validity. However, it does make all the virtual counter values before the power failure untrustable (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 client's devices communicate together and agree on the last valid value of the counter which can then be signed in a confirmation certificate.

C Appendix: Replication

In order to enhance the robustness of our storage application we propose to timestamp data with a ***data specific counter***, which is in turn timestamped by virtual counters from multiple virtual counter managers. The reason for using a data specific arithmetic monotonic counter is the folowing.

Suppose that there are $2f + 1$ virtual counter managers. Let $dataCtrID$ indicate the identity of a data specific counter and let v_j, $1 \leq j \leq 2f + 1$, be the virtual monotonic counter value that is maintained by the jth virtual counter manager and which corresponds to $dataCtrID$. We assume that the jth virtual counter manager also stores the value d_j of $dataCtrID$ at the time when the corresponding virtual counter was incremented to v_j together with the timestamp $Sign_{SK}(d_j \| v_j \| j)$ where SK is the client's secret key. We assume that client's devices only use the increment-with-validation protocol with each virtual counter manager (a more complex scheme which uses the other protocols in order to allow lazy verification of the data specific counter only at critical operations is possible).

Whenever a client's device wishes to increment $dataCtrID$, it runs an increment-with-validation protocol with as many virtual counter managers untill it receives $f + 1$ correctly verified proofs of validity together with the corresponding timestamps of the data specific counter $dataCtrID$. Without loss of generality, let v_j be incremented to v'_j for $1 \leq j \leq f + 1$ during these increment-with-validation protocols. Since these $f + 1$ virtual counter values are fresh and valid and since there are in total $2f + 1$ virtual counter managers, these $f + 1$ virtual counter values overlap with at least one of the $f + 1$ virtual counter values that were incremented during the most recent increment of the data specific counter $dataCtrID$. This means that $d = \max\{d_1, \ldots, d_{f+1}\}$ is equal to

the most recent value of $dataCtrID$. The device increments the data specific counter to $d' = d + 1$ and for each j, $1 \leq j \leq f + 1$, it transmits to the jth virtual counter manager the confirmation certificate together with the timestamp $Sign_{SK}(d' \| v'_j \| j)$ of the incremented value with the incremented virtual counter v'_j. Each of the virtual counter managers that is offline maintains its timestamp of an older version ($\leq d$) of the data specific counter.

We note that P2P distributed storage can be achieved by using distributed servers each of which implement both a storage as well as a virtual counter manager.

D Appendix: Performance Analysis

In our theoretical analysis of the performance of our virtual storage application we assume that each request to start the increment-with-validation protocol arrives according to a fixed Poison distribution. In particular, if we define u_i as the probability that the increment-with-validation protocol for virtual counter i is started within one unit of real time by one of the devices of the client who owns virtual counter i, then $u_i = u$ equals the same constant for each client. In order to simplify our analysis, we only consider the log-based scheme with sharing, fixed time-multiplexing, and with only the increment-with-validation protocol. In this scenario a client's device retrieves timestamped data from the storage manager and runs the increment-with-validation protocol as soon as it wants to verify the retrieved timestamp and sign updated data with an incremented timestamp. The next theorem formulates the details of our result. We define one atomic piece of data as the value of a node in an authenticated search tree or a single signature.

Theorem 1 *Let γ be the number of time units needed by the TPM for the computation of one $ReadSignClock$ or $IncrementSignClock$ primitive, let β be the number of time units needed by the virtual counter manager for the transmission of one atomic piece of data, and let α be the number of time units needed by the virtual counter manager to set up a communication channel over the network. We define x as the solution of the equation $x = \alpha + 2\beta \log(\gamma/x)$. Suppose that, for each virtual monotonic counter, u is equal to the probability that one of the corresponding devices requests to start the increment-with-validation protocol within one time unit. We assume that $u\gamma \ll 1$. If the virtual counter manager uses the log-based scheme with sharing, fixed time-multiplexing, and with only the increment-with-validation protocol (and no other protocols) to manage $N \leq 1/(ux)$ virtual monotonic counters, then the expected latency between request and finish of an increment-with-validation protocol can be guaranteed to be at most $(\beta N/(2(1 - \alpha N u))) \log(N u \gamma)$ which is $\leq 1/(2u)$. For $N \geq 1/(ux)$, the expected latency converges fast to a value which is at least $1/u$ times the number of devices of the client who owns the virtual monotonic counter for which the increment-with-validation protocol is intended.*

Interpretation: The theorem states that if, for each virtual counter, the expected number of corresponding increment requests that arrive within the amount of time needed by the TPM to compute one of the primitives is < 1 (that is, $u\gamma < 1$), then the expected maximum number of virtual monotonic counter that the virtual counter manager can manage is approximately 1 divided by the expected number of increment requests that arrive within the amount of time needed by the virtual counter manager to set up a connection over the network and transmit several signatures and hashes (that is, $1/(ux)$, where $x \leq \alpha + 2\beta \log(\gamma/\alpha)$). In other words, if, for each virtual counter, the TPM's computation of the primitives is fast enough with respect to the speed at which the virtual counter's increment requests arrive, then the network bandwidth is the limiting factor as in any storage application.

Proof: Let N be the total number of virtual monotonic counters. Since increment requests arrive according to a Poisson distribution, every moment in time is similar, hence, fixed periodic scheduling leads to an optimal time multiplexing strategy. Suppose that for each virtual monotonic counter the corresponding client uses a fixed

scheduling algorithm which allows periodic global clock counter values with period q. Hence, the expected number of virtual monotonic counters which are scheduled for the same global clock counter value equals N/q.

Let $\mathbf{I}$ be the expected number of virtual counters whose increments are shared in a single increment-with-validation protocol. Given a virtual counter, let $\mathbf{T}$ be the expected amount of real time taken by the global clock in the log-based scheme to increment from one scheduled value to the next scheduled global clock counter value. The probability that for a given virtual counter none of the corresponding devices starts the increment-with-validation protocol during $\mathbf{T}$ real time units is equal to $(1-u)^{\mathbf{T}}$. Hence, out of the N/q counters who are scheduled for the same global clock counter value only a fraction $(1-(1-u)^{T})$ have requested to start the increment-with-validation protocol:

$$\mathbf{I} = \frac{N}{q}(1-(1-u)^{\mathbf{T}}). \tag{6}$$

The increment-with-validation protocol takes care of at most one request per virtual counter. Suppose that, for a given virtual counter, multiple devices request to start the increment-with-validation protocol during the expected $\mathbf{T}$ units of real time that the log-based scheme uses to increment the global clock from a first scheduled value to a next scheduled global clock counter value. Then only one device can take part of the increment-with-validation protocol for the virtual counter's next scheduled global clock counter value. The remaining devices will be served at the future scheduled global clock counter values. However, during the next time intervals, more devices who request to start the protocol will be disappointed. This cascading effect converges to the situation in which each device is forced to wait for all the other devices to finish their version of the increment procol of the same virtual counter. Only if $\mathbf{T}$ is less than the expected time $(1-u)/u$ between two consecutive requests of devices of the same client, this situation is avoided; for example, if we require

$$u\mathbf{T} \leq 1/2. \tag{7}$$

If (7) is satisfied, then the average latency between a request to start and finish of an increment-with-validation protocol is equal to $\mathbf{T}/2$. If (7) does not hold, for example, if $u\mathbf{T} \geq 2$, then the latency is approximately the number of devices D of the client who owns the virtual counter times $\mathbf{T}$, that is, the latency is $\geq D\mathbf{T} \geq 2D/u$.

Let $\mathbf{L}$ be the expected number of increment certificates that constitute the log in a proof of validity. Then, the expected size of a proof of validity (which is mainly the log of increment certificates of the proof of validity together with the increment certificate generated by the increment-with-validation protocol) is approximately equal to $(\mathbf{L}+1)\cdot\log\mathbf{I}$ atomic pieces of data (due to the proofs of unique existence and non-existence which are derived from the corresponding authenticated search trees). The virtual counter manager needs to transmit this number of atomic pieces of data to each of the $\mathbf{I}$ devices that share the increment-with-validation protocol. This takes

$$\mathbf{I}(\alpha+\beta(\mathbf{L}+1)\log\mathbf{I})$$

real time units. The probability that, for a given virtual counter, exactly h scheduled global clock values occur between two consecutive increments of the virtual counter is equal to

$$(1-(1-u)^{\mathbf{T}})(1-u)^{h\mathbf{T}}(1-(1-u)^{\mathbf{T}}).$$

Hence,

$$\mathbf{LI} = \frac{N}{q}\sum_{h\geq 0}(1-(1-u)^{\mathbf{T}})^2(1-u)^{h\mathbf{T}}h = \frac{N}{q}(1-u)^{\mathbf{T}}. \tag{8}$$

Notice that $\mathbf{LI}+\mathbf{I} = N/q$. We conclude that the virtual counter manager needs about

$$\alpha\mathbf{I} + \beta(N/q)\log\mathbf{I}$$

real time units to transmit the atomic pieces of data of one shared increment-with-validation protocol.

The TPM needs γ real time units for the computation of the single $IncrementSignClock$ primitive during the shared increment-with-validation protocol. Since the TPM's computation and the virtual counter manager's transmissions can be parallized, a single shared increment-with-validation protocol takes

$$\max\{\gamma, \alpha\mathbf{I} + \beta(N/q)\log\mathbf{I}\}$$

real time units. Hence, in the log-based scheme $\mathbf{T}$ is equal to this number times the period q:

$$\mathbf{T} = \max\{q\gamma, \alpha q\mathbf{I} + \beta N\log\mathbf{I}\}. \tag{9}$$

Given (7), (6) reduces to $\mathbf{I} \approx Nu\mathbf{T}/q$ (and reduces (8) to $\mathbf{LI} \approx N/q$, hence, $\mathbf{L} \approx 1/(u\mathbf{T})$ which is consistent with our intuition that $1/(u\mathbf{T})$ is approximately equal to the expected time $(1-u)/u$ between two consecutive increment requests of the same virtual counter divided by $\mathbf{T}$). This approximation combined with (9) yields

$$\mathbf{T} \approx \max\{q\gamma, \alpha Nu\mathbf{T} + \beta N\log(Nu\mathbf{T}/q)\}. \tag{10}$$

Equation (10) immediately shows that if $\alpha Nu \geq 1$, then $\mathbf{T}$ tends to infi nity, which contradicts (7). Hence, (7) implies $N < 1/(\alpha u)$.

Let q^* be the solution of q in the equation $q\gamma = \alpha Nu\mathbf{T} + \beta N\log(Nu\mathbf{T}/q)$. If $\mathbf{T} = q\gamma$, then $\frac{\mathrm{d}\mathbf{T}}{\mathrm{d}q} = \gamma > 0$. If $\mathbf{T} = \alpha Nu\mathbf{T} + \beta N\log(Nu\mathbf{T}/q)$, then

$$\mathbf{T} = \frac{\beta N}{1-\alpha Nu}\log(Nu\mathbf{T}/q) \tag{11}$$

and

$$\frac{\mathrm{d}\mathbf{T}}{\mathrm{d}q} \approx \frac{\beta Nq}{(1-\alpha Nu)\ln 2\mathbf{T}}\left\{\frac{1}{q}\frac{\mathrm{d}\mathbf{T}}{\mathrm{d}q} - \frac{\mathbf{T}}{q^2}\right\},$$

hence,

$$\frac{\mathrm{d}\mathbf{T}}{\mathrm{d}q} \approx \frac{-\beta N/((1-\alpha Nu)q\ln 2)}{1-\beta N/((1-\alpha Nu)\mathbf{T}\ln 2)}.$$

This shows that if $\beta N/((1-\alpha Nu)\mathbf{T}\ln 2) > 1$, then $\frac{\mathrm{d}\mathbf{T}}{\mathrm{d}q} > 0$ and $\mathbf{T}$ is minimized for $q = 1$ (the smallest possible period).

If $\beta N/((1-\alpha Nu)\mathbf{T}\ln 2) < 1$, then $\frac{\mathrm{d}\mathbf{T}}{\mathrm{d}q} < 0$ and $\mathbf{T}$ is minimized for $q = \lfloor q^*\rfloor$ or $q = \lceil q^*\rceil$. Let $\mathbf{T}$ denote this minimum. From our analysis of the derivative of $\mathbf{T}$ we infer that $\mathbf{T} = \alpha Nu\mathbf{T} + \beta N\log(Nu\mathbf{T}/q)$ for $q = \lfloor q^*\rfloor$ and $\mathbf{T} = q\gamma$ for $q = \lceil q^*\rceil$. Let $\mathbf{T}^*$ be the solution of $\mathbf{T}$ for $q = q^*$ if we were allowed to use real valued periods. Notice that $\mathbf{T}^* = q^*\gamma$ as well as $\mathbf{T}^* = \alpha Nu\mathbf{T}^* + \beta N\log(Nu\mathbf{T}^*/q^*)$. We derive

$$\mathbf{T}^* \leq \mathbf{T} \leq \lceil q^*\rceil\gamma \leq q^*\gamma + \gamma = \mathbf{T}^* + \gamma.$$

Together with the assumption $u\gamma \ll 1$, this shows that the upperbound $\mathbf{T} \leq 1/(2u)$ of (7) is approximately equivalent to $\mathbf{T}^* \leq 1/(2u)$. By substituting $\mathbf{T}^* = q^*\gamma$ in (11), we obtain $\mathbf{T}^* = (\beta N\log(Nu\gamma))/(1-\alpha Nu)$. Hence, $\mathbf{T}^* \leq 1/(2u)$ is equivalent to $(\beta N\log(Nu\gamma))/(1-\alpha Nu) < 1/(2u)$, which is equivalent to $N \leq 1/(ux)$ for x defi ned by the solution of the equation

$$x = \alpha + 2\beta\log(\gamma/x).$$

Notice that these values for N and $\mathbf{T}$ satisfy $\beta N/((1-\alpha Nu)\mathbf{T}\ln 2) \approx 1/(\log(Nu\gamma)\ln 2) < 1$.

References

[1] Amazon. Amazon Simple Storage Service (S3). http://aws.amazon.com/s3, 2006.

[2] A. Anagnostopoulos, M. Goodrich, and R. Tamassia. Persistent Authenticated Dictionaries and Their Applications. In *4th International Conference on Information Security (ISC)*, 2001.

[3] Anonymized. Proof of Freshness: How to efficiently use on online single secure clock to secure shared untrusted memory. Technical report, 2006.

[4] Anonymized. Virtual Monotonic Counters and Count-Limited Objects using a TPM without a Trusted OS. In *Proceedings of The First ACM Workshop on Scalable Trusted Computing (STC'06)*, pages 27–41, 2006.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

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

[10] A. Buldas, P. Laud, H. Lipmaa, and J. Villemson. Time-stamping with binary linking schemes. In *Advances in Cryptology - CRYPTO '98*, pages 486–501, 1998.

[11] A. Buldas, H. Lipmaa, and B. Schoenmakers. Optimally efficient accountable time-stamping. In *Public Key Cryptography '2000*, pages 293–305, 2000.

[12] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *Third Symposium on Operating Systems Design and Implementation*, February 1999.

[13] M. Castro and B. Liskov. Proactive recovery in a byzantine-fault-tolerant system. In *Fourth Symposium on Operating Systems Design and Implementation*, October 2000.

[14] 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.

[15] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. In *Introduction to Algorithms*. MIT Press/McGraw-Hill, 1990.

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

[17] P. T. Devanbu and S. G. Stubblebine. Stack and queue integrity on hostile platforms. *Software Engineering*, 28(1):100–108, 2002.

[18] D. Eastlake and P. Jones. RFC 3174: US secure hashing algorithm 1 (SHA1), Sept. 2001.

[19] S. Even, O. Goldreich, and S. Micali. On-line/off-line digital signatures. *Journal of Cryptology*, 9(1):35–67, 1996.

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

[21] B. Gassend, G. E. Suh, D. Clarke, M. van Dijk, and S. Devadas. Caches and Merkle Trees for Efficient Memory Integrity Verification. In *Proceedings of Ninth International Symposium on High Performance Computer Architecture*, New-York, February 2003. IEEE.

[22] M. Goodrich, R. Tamassia, and A. Schwerin. Implementation of an Authenticated Dictionary with Skip Lists and Commutative Hashing. In *Proceedings of DARPA Information Survivability Conference and Exposition*, pages 68–82, 2001.

[23] 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.

[24] 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.

[25] P. Kocher. On certificate revocation and validation. In *Proceedings of Financial Cryptography 1998*, pages 172–177, 1998.

[26] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, 1978.

[27] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):241–248, 1979.

[28] 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.

[29] 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* 9th *Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pages 168–177, November 2000.

[30] U. Maheshwari, R. Vingralek, and W. Shapiro. How to Build a Trusted Database System on Untrusted Storage. In *Proceedings of OSDI 2000*, 2000.

[31] P. Maniatis. *Historic Integrity in Distributed Systems*. PhD thesis, Stanford University, Aug. 2003.

[32] P. Maniatis and M. Baker. Enabling the Archival Storage of Signed Documents. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST 2002)*, pages 31–45, 2002.

[33] P. Maniatis and M. Baker. Secure History Preservation through Timeline Entanglement. In *Proceedings of the 11th USENIX SEcurity Symposium*, 2002.

[34] 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.

[35] D. Mazières and D. Shasha. Don't trust your fi le server. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems*, May 2001.

[36] 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.

[37] R. C. Merkle. Protocols for public key cryptography. In *IEEE Symposium on Security and Privacy*, pages 122–134, 1980.

[38] S. Micali. Effi cient certifi cate revocation. Technical Report MIT/LCS/TM-542b, 1996.

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

[40] M. Naor and K. Nissim. Certifi cate revocation and certifi cate update. In *Proceedings 7th USENIX Security Symposium (San Antonio, Texas)*, 1998.

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

[42] 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.

[43] B. Schneier and J. Kelsey. Cryptographic support for secure logs on untrusted machines. In *The Seventh USENIX Security Symposium Proceedings, USENIX Press*, pages 53–62, January 1998.

[44] B. Schneier and J. Kelsey. Secure Audit Logs to Support Computer Forensics. *ACM Transactions on Information and System Security (TISSEC)*, 2(2):159–176, 1998.

[45] W. Shapiro and R. Vingralek. How to manage persistent state in DRM systems. In *Digital Rights Management Workshop*, pages 176–191, 2001.

[46] 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.

[47] G. E. Suh. *AEGIS: A Single-Chip Secure Processor*. PhD thesis, Massachusetts Institute of Technology, Aug. 2005.

[48] 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* 17th *Int'l Conference on Supercomputing (MIT-CSAIL-CSG-Memo-474 is an updated version)*, New-York, June 2003. ACM.

[49] G. E. Suh, C. W. O'Donnell, I. Sachdev, and S. Devadas. Design and Implementation of the AEGIS Single-Chip Secure Processor Using Physical Random Functions. In *Proceedings of the* 32nd *Annual International Symposium on Computer Architecture (MIT-CSAIL-CSG-Memo-483 is an updated version available at http://csg.csail.mit.edu/pubs/memos/ Memo-483/Memo-483.pdf)*, New-York, June 2005. ACM.

[50] Trusted Computing Group. Trusted Platform Module (TPM) Specifi cations. https://www.trustedcomputinggroup.org/specs/TPM/.

[51] Trusted Computing Group. TPM v1.2 specifi cation changes. https://www.trustedcomputinggroup.org/groups/tpm/ TPM_1_2_Changes_fi nal.pdf, 2003.

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

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