Published in the *Proceedings of 3rd ACM Conference on Data and Application Security and Privacy (CODASPY), 2013, pp. 353–364.*

Multi-user Dynamic Proofs of Data Possession using Trusted Hardware

Stephen R. Tate
Dept. of Computer Science
UNC Greensboro
srtate@uncg.edu

Roopa Vishwanathan
Dept. of Computer Science
UNC Greensboro
r_vishwa@uncg.edu

Lance Everhart
Dept. of Computer Science
UNC Greensboro
lmeverh2@uncg.edu

ABSTRACT

In storage outsourcing services, clients store their data on a potentially untrusted server, which has more computational power and storage capacity than the individual clients. In this model, security properties such as integrity, authenticity, and freshness of stored data ought to be provided, while minimizing computational costs at the client, and communication costs between the client and the server. Using trusted computing technology on the server's side, we propose practical constructions in the provable data possession model that provide *integrity* and *freshness* in a dynamic, multi-user setting, where groups of users can update their shared files on the remote, untrusted server. Unlike previous solutions based on a single-user, single-device model, we consider a multi-user, multi-device model. Using trusted hardware on the server helps us to eliminate some of the previously known challenges with this model, such as *forking* and *rollback* attacks by the server. We logically separate bulk storage and data authentication issues to different untrusted remote services, which can be implemented either on the same or different physical servers. With only minor modifications to existing services, the bulk storage component can be provided by large-scale storage providers such as Google, CloudDrive, DropBox, and a smaller specialized server equipped with a trusted hardware chip can be used for providing data authentication. Our constructions eliminate client-side storage costs (clients do not need to maintain persistent state), and are suitable for situations in which multiple clients work collaboratively on remotely stored, outsourced data.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; H.3.2 [**Software**]: Security and Protection—*authentication, verification, cryptographic controls*

Keywords

Cloud storage; data outsourcing; trusted hardware

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CODASPY'13, February 18–20, 2013, San Antonio, Texas, USA.
Copyright 2013 ACM 978-1-4503-1890-7/13/02 ...$15.00.

1. INTRODUCTION

In this paper, we devise and present natural and important extensions of the Dynamic Provable Data Possession (DPDP) work of Erway *et al.* [8]. In DPDP, a client maintains a dataset on a remote untrusted server, and the system needs to support data access and modification requests, including data additions, modifications, and deletions. The DPDP protocol provides a proof of integrity when data is accessed, so that the client is assured that neither the untrusted server nor another malicious party has tampered with the data. The solution of Erway *et al.* relies on techniques from authenticated data structures, and in particular maintains an authenticated skip list over data blocks. All operations on the skip list are $O(\log n)$ expected time, and the proof of integrity consists of a sequence of hashes from the leaf or leaves corresponding to the data being accessed up to the root of the skip list, so the time and space required for such a proof is also $O(\log n)$. The client keeps a copy of the root hash value locally, and that is all that is required to authenticate any data access from the server.

The DPDP solution just described assumes a model in which there is a single client using a single client-side device, which does not allow for one of the main advantages of remote storage: the ability of users to access their data from anywhere, on a wide variety of devices, some of which may have only sporadic network access. Furthermore, we would like to support not only multiple client-side devices, but also multiple users, so that any authorized user can update the remotely stored data from any device, but unauthorized users (including a malicious server) cannot make any changes that will be presented as valid on subsequent accesses. Moving from a single-user, single-device client model to a multiple-user, multiple-device client model introduces some obvious problems, as well as some subtle ones, as we describe next.

Allowing multiple client-side devices in the DPDP model of Erway et al. [8] means that each device must have access to the root hash, so there is no single local storage area for root hashes. Considering a user with a desktop system and a mobile device, if the root hash is stored and updated on the desktop computer, and the user later accesses his data from a mobile device, that device may not have the most current root hash for verification. A naïve solution would be to store root hashes remotely with the data, signed by an authorized user's key for integrity — since the key would change far less often than the data hash, keys could be kept on devices in protected local storage. Alternatively, private signing keys could even be stored remotely, encrypted by a key that is

derived from a passphrase that can be entered locally on any device, similar to the way "Firefox Sync" stores passwords — this technique can be relatively secure, particularly if combined with some other method of access control.

The problem with the naïve solution is one of **freshness** [33]: If we store root hashes on a remote server, signed or not, when we retrieve them we have no way of knowing that the server is returning our most recent value. This is similar in concept to the well-known notion of a replay attack, and in the context of remote storage is sometimes called a "roll back attack" [10] or a "forking attack" [19, 32]. The notion of forking comes from the following scenario: consider what would happen if the server made a snapshot of the current data, including the root hash, and then accepted a change from the user's desktop system. The user (or perhaps even a different authorized user) then accesses the data from a mobile device, and is given the old copy from the snapshot. These two versions could then be associated with these two different client machines, and could evolve separately, forking the stored data contents into two (or more) histories of changes that form a consistent view for that device. If there is no authentic communication between the devices, there is no way to detect that they are working with different versions of the data. In the distributed computing community, it is accepted that this is impossible to avoid without either a trusted server or communication between different clients, and a notion of "fork consistency" has been defined for this: a solution has fork consistency if the view of any client device is internally consistent, even if globally there are multiple histories. Examples of work in the distributed computing community that explores this notion include the SUNDR networked file server [16] and the SPORC group collaboration system [9].

Since client devices may only occasionally be online, we cannot rely on direct device-to-device communication to detect forked storage state, so some form of trusted server must be used. Our focus in this paper is to minimize the amount of trust needed to create a system that avoids problems with forking, while maintaining integrity and freshness of data. In particular, we don't want to trust the parties running the server, nor do we want to trust an open computing system, regardless of who is running the system. Our solution utilizes lightweight trusted hardware, similar to the Trusted Platform Modules (TPMs) [31] that are being included in many current systems. While our technique could be implemented using current TPM standards, the resulting solution would be inefficient, so we instead consider TPM extensions proposed by Sarmenta *et al.* for efficient "virtual monotonic counters" (counters which can never repeat values, even across machine reboots) [25]. These extensions are consistent with existing TPM capabilities, but allow for better and more efficient support for applications that rely on monotonic counters. While not providing the same level of assurance as a purely-hardware based solution, one could also implement our technique using other forms of small-size trusted computing models such as TrustVisor [21], or its predecessor Flicker [22].

In addition to considering a server that maintains a single data store, we extend this to consider a hierarchy of stored objects like a standard tree-based filesystem. Our solution for this model avoids keeping a signed root hash for each stored object, but rather relies on the hierarchical structure to define "control points" which have associated signed values. These control points correspond to places in the hierarchy in which access control changes, and so are significantly less numerous than the number of objects. This both reduces the number of trusted values that must be managed through our trusted hardware-based solution and reduces the number of public key operations that must be performed to verify signatures when data is accessed. Control points also give a convenient way to increase throughput for a complete remote storage solution, by distributing management of different control points across separate servers. While certain aspects of our solution, such as the bulk storage server, can be further distributed for load balancing or resilience, properties of the authentication server create significant challenges for distributing further, and we leave these challenges open for future work.

Contributions: In summary, the work described in this paper provides the following contributions:

- We extend the DPDP model to allow for data updates from multiple devices as well as multiple users in an authorized group, and provide a solution that avoids client-to-client communication by using a small TPM-like trusted hardware component. Supporting this solution, we provide two specific implementations that have slightly different properties (one of which provides anonymity to users within a group).
- We develop a new, efficient way of validating many virtual monotonic counters at once, layered on top of the TPM extensions proposed by Sarmenta *et al.* [25]. This solution is several orders of magnitude faster than the straightforward solution, allowing the processing of requests at realistic server speeds.
- We extend the single data object model to support a hierarchical filesystem in which updates for different parts of the hierarchy are restricted to users or groups of users, and we provide a general framework for efficiently supporting this model.
- We provide some preliminary experimental results that explore the feasibility of these techniques, and how well they scale to multiple active users.

1.1 Paper Outline

In Section 2, we briefly review and compare relevant related work. In Section 3, we describe the multi-user DPDP scheme using trusted hardware, and describe two possible implementations of it that use either group signatures or user certificates. In Section 4, we extend our multi-user DPDP scheme to multiple objects in a hierarchical filesystem. In Section 5, we provide security definitions and analysis. In Section 6, we present some preliminary experimental results, and in Section 7, we conclude the paper.

2. RELATED WORK

Security issues in outsourced storage systems have led to the development of two main solution models: proofs of data possession (PDP) which offer tamper-evidence, and proofs of retrievability (PoR) which offer tamper-tolerance. Our work is designed to fit into the PDP model.

The concept of PDP was introduced by Ateniese *et al.* who presented an efficient protocol for static (read-only) data stores [1], and in later work designed a protocol for a somewhat limited dynamic setting [3]. Subsequently, Erway *et al.* presented a fully dynamic PDP protocol [8]. Our

paper is most closely aligned with this line of work, in an extended model with multiple users and multiple client devices, while maintaining the $O(\log n)$ efficiency of the best previous fully dynamic single-user, single-device solution due to Erway *et al.* [8]. Specific challenges in this extension were described in the Introduction.

Iris [28] offers practical solutions to most major security issues such as integrity, freshness, scalability, and retrievability, but requires a trusted third party that can perform involved computations and store fairly large quantities of cached data. Proofs of retrievability (PoRs) [13, 26, 35, 28] are a related concept in which the client is assured of data retrievability, in addition to just data integrity and freshness verification (we do not deal with retrievability). In recent work in secure storage, Xiong *et al.* [34] and Goodrich *et al.* [11] have independently proposed ways to securely store, share and distribute data through a public, untrusted cloud. These works tackle a different issue: they focus on providing *confidentiality* of data stored on the server, whereas we focus on providing *integrity* and *freshness* of data. Confidentiality is an orthogonal issue for us, and can be layered on top of our protocols (by using encryption, access control policies, private communication channels, etc.).

Although our work is in the PDP model, which has a different flavor than the distributed computing community's idea of remote file storage, there are some distributed computing works that are worth mentioning. One of the early influential works in secure remote file storage was SUNDR [16, 20] which is a network file system stored on an untrusted, possibly remote, server. SUNDR introduced the notion of "fork consistency" that gives any two clients the ability to detect inconsistencies in their views of data, only if the clients can communicate with each other (and also maintain local meta-data). It was widely accepted that fork consistency is the best notion of data consistency one can achieve in the presence of an untrusted server and non-communicating clients, and SUNDR's forking semantics were used by others to implement remote storage protocols [6, 17, 5]. Our contribution relative to SUNDR and related work is a solution in the DPDP model in which clients do not communicate with each other and in which the only trusted server component is an embedded TPM.

Other works that provide integrity and consistency guarantees include SPORC [9], Venus [27] (requires a core group of trusted clients to be always online), Depot [18], and works that use a trusted component such as A2M [7]. TrInc [15] proposes the use of monotonic counter-based attestation in a peer-to-peer distributed system. We consider monotonic counters in the context of remote untrusted storage, and analyze the issues that come up therein (e.g., concurrency, consistency). Also, as we discuss in Section 6, a direct application of monotonic counter-based attestation to remote storage would result in a system with an unacceptably slow throughput of between 1 and 3 requests per second, which is insufficient for practical applications.

All of these distributed computing works are robust, have many desirable properties, and are interesting in their own ways. But by and large, they require clients to store some form of local meta-data for verification purposes. Our use-case consists of multiple clients that access the server from non-communicating devices that can have sporadic and/or untrusted network connections. To avoid the need for all clients to keep all of their devices always up-to-date with the latest local meta-data, one of our main motivating goals is to eliminate client-side meta-data storage.

To increase attestation throughput, we introduce a novel technique for batching TPM counter attestations. A similar problem was addressed by Moyer *et al.* for the purpose of providing attestations for a general web server, and their solution shares some properties with ours including the use of Merkle hash trees [24]. However, while they also answer multiple client requests with a single TPM attestation operation, their solution relies on an external, trusted time server. By using a time server, Moyer *et al.* can provide attestations for static content using a cached recent attestation, which reduces latency for static requests; however, handling dynamic requests re-introduces a small latency, resulting in similar performance to our technique, while still requiring the trusted time server.

3. MULTI-USER DPDP SOLUTION

In this section, we provide a threat model for our work, and extend the single-user, single-device DPDP model to the multi-user, multi-device model.

3.1 Threat Model

Assets and Goals: In the most basic model, the asset we are concerned with is a data object that is stored on a remote, network-accessible resource, and associated metadata that is maintained to support the security requirements. The goal is to preserve *integrity* (including authentication of updates) and *freshness* of the data object as seen by remote clients. In particular, users accessing the data object should be able to tell if the data that they received is the latest data that was stored by an authorized user (see "parties" below). For the purposes of this paper, we do not consider other goals such as confidentiality and availability, both of which can be addressed independently (e.g., [34, 11, 4, 18]). Extensions to multiple objects with more involved group-oriented access control policies are considered in Section 4.
Parties: The main parties actively involved in the system are clients and the service providers. The service providers are the Storage Server (**SS**) and the Authentication Server (**AS**), the latter of which is equipped with a Trusted Platform Module (TPM) chip whose properties are described in Definition 1. Both can potentially be corrupt, could collaborate, and can collude with outside parties. Clients and other parties are divided into authorized and unauthorized users, and only authorized clients are allowed to modify the data object in a way that will be accepted by users on subsequent accesses. Other non-participating, peripheral parties include the TPM manufacturer and a Privacy CA that certifies public keys belonging to the TPM.

Definition 1. The *Trusted Platform Security Assumption* is the assumption that TPMs are built by a honest manufacturer according to an open specification, and satisfy the following properties:

1. *Tamper-resistant hardware:* It is infeasible to extract or modify secret data or keys that are stored in protected locations in the TPM, except through well-defined interfaces given in the TPM specification.
2. *Secure Encryption:* The asymmetric encryption algorithm used by the TPM is CCA-secure.

3. *Secure Signatures:* The digital signature algorithm used by the TPM is existentially unforgeable under adaptive chosen message attacks.
4. *One-way Functions:* For generating hash digests, the TPM uses strongly collision-resistant one-way functions.
5. *Trustworthy PrivacyCA:* Only valid TPM-bound keys are certified by a trusted PrivacyCA.

Note that we do not rely on system-level attestation properties used in a lot of trusted computing applications, but only on properties of the TPM chip.

Threat Vectors: Attackers can consist of any collusion between servers and non-authorized clients, who can intercept and tamper with communication in arbitrary ways. Beyond this, we do not enumerate specific attacks, but rather take the approach that is common in cryptography: we model all colluding attackers as a single computational adversary, and any attack that can be performed by this adversary in polynomial time is fair game. The specific sequence of actions followed by the adversary, and how it chooses to use its resources, are non-deterministic in nature. The formal security properties of our constructions are sketched in the longer version of this paper [30], and are addressed more fully in ongoing work.

3.2 DPDP Definitions

The intuition behind DPDP is simple: A server stores a data object, which can be read from or updated. While the intuition is simple, a full definition of the precise model was one of the contributions of Erway *et al.*, who provided careful definitions of core operations and required communication. In the simple data model of Erway *et al.*, there is a single data object with multiple blocks, and blocks can be read, modified, inserted, or deleted [8]. While page limitations do not allow for a full repetition of the DPDP model here, we provide a brief overview, and refer the reader to the original paper for full details.

The system is initialized using a KeyGen operation which establishes cryptographic keys. Information is stored on a server, while the client retains a small amount of verification information called the metadata. Authenticated retrievals work by the client evaluating a Challenge function to create a retrieval challenge message to send to the server, who executes a Prove procedure which produces the data being retrieved along with a proof P of its authenticity, which is sent back to the user. The user then uses a Verify procedure to check the proof against a local copy of the data store's current metadata M_c. Updates also involve three functions, where the client starts by executing a PrepareUpdate function to create messages to send to the server. The server executes a PerformUpdate function, which updates the data it is storing, calculates new metadata $M_{c'}$, and creates a proof P that the update was performed correctly. Upon receiving $M_{c'}$ and P, the client executes a VerifyUpdate function with the proof and its copy of current metadata M_c to verify that the update was performed correctly.

Since we are considering multiple users, we refer to the group of one or more users that is authorized to modify this data object as the "authorized modifiers." In Section 4 we will show how this can be extended to a hierarchical filesystem of objects, supporting different access control lists for different objects. As explained in the Introduction, the key to supporting access from multiple devices is the ability to store the authenticating metadata on a shared but potentially untrusted server, and this causes problems with freshness and forking attacks. Sarmenta *et al.* sketched a solution to a similar problem using trusted hardware to support monotonic counters [25], but assumed a single user making updates with access control on the server to enforce this. We use Sarmenta *et al.*'s work as a starting point, and devise a solution that allows us to deal effectively with groups of authorized modifiers, and we also provide significant efficiency improvements when compared with the original techniques.

3.3 Virtual Monotonic Counters

Sarmenta *et al.* consider the issue of freshness for remote storage [25], and propose the use of a TPM as a solution. Among other capabilities, a standard TPM contains monotonic counters, which are hardware-based counters that can only be incremented, and will never repeat a value, even over a reboot. Additionally, TPMs work with Attestation Identity Keys (AIKs), which are digital signature keypairs whose secret key is only usable inside the TPM and is only used for signing (attesting) values that come from inside the TPM. A local or remote user can obtain the current counter value with a high level of assurance by having the TPM sign the counter value along with a user-supplied nonce.

Many interesting applications can be designed using dedicated monotonic counters, but unfortunately TPMs support only a very limited number of monotonic counters in hardware, and have just one counter active in any boot cycle. The main contribution of Sarmenta *et al.* was the idea of "virtual monotonic counters," leveraging the limited protected hardware to support a very large number of independent monotonic counters, along with two implementations of this general idea. One solution, called the log-based scheme, can be implemented using current TPMs, but can be very inefficient in some situations. The other solution, the hash tree-based scheme, provides much better performance but requires small and practical extensions to TPMs.

In the context of a digital wallet application, Tate and Vishwanathan performed a thorough experimental comparison of the log-based scheme and the hash tree-based scheme under a variety of workloads [29]. A representative result from that paper showed that when maintaining 1024 virtual counters, the hash tree-based scheme performed around 900 times faster than the log-based scheme for round-robin counter requests. On realistic non-uniform distributions, the average gain decreased to around 15 times faster for the hash-tree based scheme, but the worst case has an amazing speed difference of over 2000 times. While our remote storage solution could in fact use either of these virtual monotonic counter implementations, due to these substantial speed improvements we focus on the hash tree-based scheme. We next give a quick review of virtual monotonic counters and the hash tree-based scheme, and then explain our technique for "batching" attestation requests that allows the server to attest counter values at a rate that is a couple of orders magnitude higher than straightforward application of prior techniques.

3.3.1 Basic Techniques

Hash trees are a well-known tool for authenticating large data sets [23], where some part of the data set can be authenticated without knowing or processing the rest of the

data set (contrasted with, for example, hashing the entire data set, requiring re-hashing and processing all data in order to very authenticity). Hash trees work by providing an *authenticating path* for any data in a leaf of the tree, along with a hash value from the root of the tree that can be compared against a known-good hash value.

Sarmenta *et al.* show how to support many virtual monotonic counters by using a hash tree of counter values with the root hash protected by trusted hardware (an enhanced TPM). While protecting a large number of virtual monotonic counters, the only value that must be closely protected is the root hash, which is well within the capability of even limited devices like TPMs. By carefully defining operations that update the root hash (which require the user to provide authenticating path information to the TPM), the necessary security properties of virtual monotonic counters can be assured. While not used by Sarmenta *et al.* for remote storage applications, they do allow for data to be associated with each counter at a hash tree leaf, and that data is tightly bound to the counter since it must be provided as leaf-data in any authentication operation. We make use of this associated data in a non-obvious way, to support access control in our multi-user DPDP solution. We briefly list operations defined by Sarmenta *et al.* on virtual monotonic counters — note that the result of all operations is provided in the form of an "execution certificate", which contains the user-supplied nonce and is signed by a TPM-bound AIK:

- CreateCounter$(pos, data, nonce) \rightarrow id$: Creates a new counter in hash tree position pos, with a random component of the id assigned by the TPM (this is so that hashtree positions can be reused for new counters, with negligible probability of reusing an actual id that would "reset" that counter). $data$ is the data associated with the counter.
- ReadCounter$(id, nonce) \rightarrow (count, data)$: Returns the current count value and associated data for counter id.
- IncrementCounter$(id, data, nonce) \rightarrow count$: Increments counter id and associates new $data$ with the new count value, returning the incremented counter value.

To these operations we make a simple addition to provide an attested copy of the root hash — note that we could do this by simply reading any counter, but providing a direct command simplifies our applications:

- AttestRoot$(nonce) \rightarrow root_hash$: Returns an attested copy of the root hash.

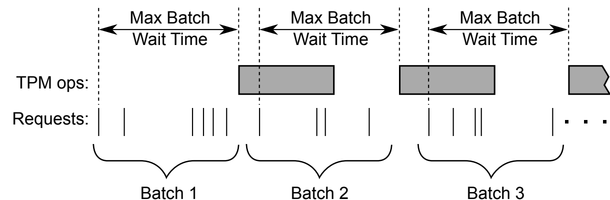
3.3.2 Batching Attestations

While virtual monotonic counters provide the functionality needed for our remote storage solution, the performance of the basic solutions from Sarmenta *et al.* would only be sufficient for low-volume applications. Every data read operation will require retrieving an attested counter value, which means the TPM must perform a digital signature operation for this attestation. Tests timing signature speed of a variety of version 1.2 TPMs found that digital signature speed varied from around 300ms to around 760ms [12], and so the fastest of these TPMs would only be able to make about three signatures per second. Since every attestation must include a user-supplied nonce in the signed counter value, it is not clear how we could satisfy client requests any faster than this three-per-second baseline; however, we next show how we can batch requests to greatly improve the rate at which counter verifications can be provided.

We once again rely on hash trees as a tool to enable the solution to our problem. Counter attestation requests are of the form ReadCounter$(id, nonce)$, as described in the previous section, so consider a sequence of n such requests given as $(id_1, nonce_1), (id_2, nonce_2), \ldots, (id_n, nonce_n)$. To answer these requests efficiently, we create a hash tree with the request nonces at the leaves, and call the root hash of this tree $TPMnonce$. We issue the TPM command AttestRoot$(TPMnonce)$, getting a signed copy of an execution certificate containing both the TPM-managed counter hashtree root and $TPMnonce$.

The server then assembles responses to attestation requests as follows: For request $(id_i, nonce_i)$, it sends out the authenticating path from the virtual monotonic counter hash tree for counter id_i (which includes the counter value and the root hash), the authenticating path from the nonce hash tree for $nonce_i$ (which includes $TPMnonce$), and the signed attestation that links the counter root hash with $TPMnonce$. We argue that this is secure, based on the following reasoning (which we formalize in the full version of this paper): $TPMnonce$ must have been computed after the server received $nonce_i$, since $nonce_i$ went into the computation of $TPMnonce$ through a secure hash function for which it is infeasible to find collisions. Furthermore, the TPM attestation must have taken place after $TPMnonce$ was computed, since that is included in the signed TPM response. These two facts taken together ensure that the counter root hash, which the TPM guarantees was current when the attestation was made, is no older than the time that $nonce_i$ was received by the server. Finally, since the counter hash tree guarantees that the reported value of counter id_i was current when the attestation was made, then that virtual monotonic counter value is no older than the time that $nonce_i$ was received — this is precisely the freshness guarantee that we are seeking.

The technique just described gives a way to answer multiple attestation requests with a single TPM attestation operation. Since the TPM operation is the bottleneck in responding to rapid attestation requests, this can give significant increases in the rate at which attestation requests can be answers. When responding to a stream of requests, we overlap request batching with TPM operations, as shown in this diagram:



Requests are denoted by vertical bars, and TPM operations are denoted by the boxes. At all times the server is either in an `idle` state, meaning no attestation requests are outstanding, or a `batching` state. We define a parameter called the "Max Batch Wait Time" to be amount of time the system waits after receiving an attestation request until it starts a TPM operation. If the server is `idle` and it receives a request, it sets a timer for "Max Batch Wait Time" and enters the `batching` state. When the timer runs out, the server sends attestation command to the TPM (using

$TPMnonce$), and re-enters the `idle` state. When the TPM command completes, the server constructs attestation responses, which it sends to clients. Selecting the "Max Batch Wait Time" is a trade-off between the rate at which requests can be serviced (the higher the wait time, the larger the batches, so the more efficient the service), and the maximum latency of a request.

It is important to note that the client interface for getting an attested counter value is almost identical to before: the client provides the counter id and a nonce, and receives a response that includes a TPM-signed attestation that depends in a strong way on the nonce that the client provided. Verification on the client side must add a verification of the nonce hash tree path, but that is the only difference from the straightforward use of virtual monotonic counters. As we will see in the Experiments section, this batching technique significantly increases the rate at which counter attestations can be produced.

3.4 Multi-User DPDP

To define multi-user DPDP, we first introduce notation for users and groups, as well as generic notation for testing group membership. We use U to denote the entire universe of users, both those with access to the data object and those without, and there is a designated group $G \subseteq U$ of users that are authorized to modify the data object. A special user, known as the group manager and denoted GM, is authorized to create the data object with the remote storage service and can designate which users are members of G. We assume a generic digital signature based service which can be used to authenticate whether a message was created by a member of G. In particular, there is a public key PK_G associated with the group, and each user u has an individual signing key $SK_{G,u}$ associated with this group. Two basic operations are supported:

- $GSign(SK_{G,u}, m) \rightarrow \sigma$: Used by user u to create a signature σ on message m.

- $GVerify(PK_G, m, \sigma) \rightarrow$ true/false: Verifies whether σ is a valid signature on m for group G (using G's public key PK_G).

We assume that any such signature scheme satisfies basic unforgeability properties, and in Section 3.6 we describe two different schemes to implement this service.

The key problem with extending earlier DPDP results, such as the work of Erway *et al.* [8], involves maintaining authentication metadata in a centrally accessible location so that clients have assurance that metadata that they retrieve from this central location is authentic and fresh. Our solution logically separates the problems of maintaining metadata from the problem of bulk storage into two servers, the "Authentication Server" (denoted **AS**) and the "Storage Server" (denoted **SS**), respectively. This is a logical separation, and the two servers could in fact be hosted on the same physical server, although the requirements are different. **AS** requires a TPM, but has low storage and throughput requirements, whereas **SS** has high storage and throughput requirements but does not need a TPM to authenticate operations. The two servers only interact with client devices, and never directly with each other, so could in fact be provided by two entirely separate entities.

3.4.1 AS: The Authentication Server

In this section we describe the authentication server **AS**. This server is a means for storing metadata associated with the stored data in such a way that the client can be sure that it is authentic and fresh. Metadata is stored in a pair $m = (vno, M_c)$ that ties it to a version number, and **AS** also stores a signature σ over m that can be verified as a signature from an authorized modifier (i.e., $GVerify(PK_G, m, \sigma)$ accepts the signature as coming from a member of the group G). When queried, **AS** can provide both m and the matching σ, and can also use its TPM to attest to the fact that the version number in m is the current version number. In the notation below, $auth_vno$ refers to this TPM attestation on the version number: specifically, $auth_vno$ will be produced as the result of an interactive protocol in which the client sends a random $nonce$ to the server, which executes the TPM command ReadCounter($id, nonce$) — $auth_vno$ is the resulting execution certificate that includes the version number and nonce signed by a TPM-bound AIK.

- Init($PK_G, nonce$) $\rightarrow id$: Access control on **AS** only allows the group manager to execute this command, which initializes **AS** to set up a new data store. **AS** assigns an id number to the new data store, and executes the TPM command CreateCounter($pos, PK_G, nonce$). Note that the second argument to CreateCounter is the data that is bound by the TPM to this new counter, and by setting this to be the group G's verification key PK_G, we bind that verification key to the counter in a strong, TPM-enforced way.
- GetVersion($id, nonce$) $\rightarrow (auth_vno, m, \sigma)$: This command provides the current version number and data identified by id, in a way that can be authenticated as authorized and fresh. In particular, σ is a valid signature on m (i.e., $GVerify(PK_G, m, \sigma)$ accepts), and the version number in m is attested to by the TPM as current by executing TPM command ReadCounter($id, nonce$) and returning the AIK-signed execution certificate as $auth_vno$ (along with m and σ).
- GetCurrNextVersion($id, nonce$) $\rightarrow (auth_vno, m, \sigma, nv)$: This is the similar to GetVersion, but sets an exclusive lock for this client for id (if some other client holds a lock on id, this operation fails), and returns the next available version number nv in addition to the attested current version number.
- ReleaseLock(id): This releases the lock that this client holds on id, and is used when the protocol must be aborted (e.g., due to a failure with **SS**).
- UpdateVersion($id, m, \sigma, nonce$) $\rightarrow auth_vno$: First, **AS** first checks to make sure that this client holds the update lock for this id, with the same "new version number" that is in m, and then retrieves the data associated with counter id (which was set in Init to be the verification key PK_G, and retained as this value on all subsequent operations). **AS** then runs $GVerify(PK_G, m, \sigma)$ to verify that this update is coming from an authorized group member. If these tests succeed, **AS** executes TPM command IncrementCounter($id, PK_G, nonce$) and returns the AIK-signed execution certificate showing the new version number. Whether successful or not, in any case, **AS** invalidates the update lock held by this client for id.

There are a few important things to note about the services provided by **AS**. First, since GM initializes the system by associating PK_G with the virtual monotonic counter id, and the only change to this counter is through UpdateVersion which keeps the same data PK_G, this binds the authorized group strongly to this counter, and only updates which are accompanied by a signature that verifies with this key are accepted.[1] Note that the clients do not have to trust **AS** on this point, since the counter-associated data PK_G is part of the signed execution certificate ($auth_vno$) that accompanies all responses to client requests — clients could immediately tell if this public verification key were changed by a malicious **AS**. Another important thing to notice is the use of the exclusive update lock: This allows the full update, including interaction with **SS**, to be atomic and sequenced with other requests from other concurrent clients. This lock should have some timeout value so that a client that fails before completing the update operation would not lock up the system (if a slow client responds after the timeout, the only consequence would be that the UpdateVersion operation fails).

3.4.2 SS: The Storage Server

The storage server **SS** is primarily an implementation of a basic single-client DPDP scheme, with the addition of version numbers and locks to handle concurrency and consistency issues. Due to page limitations we do not go into details here, but will instead focus on how **SS** is modified to address problems that are unique to the multi-client setting. Our two significant changes include the use of version numbers and the addition of an update lock.

Version numbers form a monotonically increasing sequence of values, and each time the data is updated a new version number is assigned. The server retains old versions for some amount of time, but may only update the most recent version, similar to partially persistent data structures [14]. Every client request to **SS** must contain a version number, but otherwise looks like a standard DPDP request. While the source of the version numbers is not relevant for the functioning of **SS**, we note here that our full protocol retrieves version numbers from **AS**, which in turn gets them from virtual monotonic counters managed by its TPM.

Since the requests to **SS** are coming through a single monotonic source, the requests should, for the most part, be monotonically sequenced. It is still possible for **SS** to receive out-of-sequence requests due to some clients being slower than others, and we keep old versions for some time so that slow clients are not indefinitely delayed due to failed requests. We discuss more fully how concurrency and consistency issues are resolved in Section 3.5, after the full protocol is defined. As far as **SS** is concerned, its behavior is controlled by the system parameter $accom_time$ (or "accommodation time"), which could be mutually agreed upon when the group manager sets up service with **SS**. Version numbers are managed by keeping track of the highest version number seen so far in a request ($curr_vno$) and a set old_vno_deaths that stores, for each old version number, a "time of death" after which requests for that version will no longer be answered. To update these values, when the server receives the first request for a new version number, beyond $curr_vno$, it puts $curr_vno$ in the old_vno_deaths sets with a "time of death" set to the current time plus $accom_time$, and then $curr_vno$ is updated with the newly-received version number. Therefore, slow clients have the amount of time specified by $accom_time$ to get read requests in before the old version dies. Note that $curr_vno$ is *not* updated when the DPDP operation PerformUpdate is performed — rather it is the next request from a client that contains the new version number that triggers the update. This is an important point which will be explained more fully in Section 3.5.

SS is also enhanced from a single-client DPDP server by the inclusion of update locking. When a data object is initially created on **SS**, the public key of the authorized modifier group is stored on the server, and lock requests must be signed to show that they came from an authorized user. In particular, at the beginning of an update the client requests an exclusive update lock using both the current version number and the next version number. Each lock request must be signed by the client identifying it as a member of an authorized group. If the signature verifies, **SS** allows the client to proceed with the update. Unlike read requests, there is no accommodation time for update requests, so the version number must match $curr_vno$ and there must be no other client locking that version in order for the request to succeed. To protect against clients that fail, a system parameter $lock_timeout$ determines how long the lock will be held before the lock is released and the overdue PerformUpdate request will not be accepted.

3.4.3 Complete Multi-User DPDP Protocol

In this section we describe how the client works with **AS** and **SS** in the complete multi-user DPDP protocol. First we present the protocol for authenticated data retrieval.

Retrieve protocol for user u:

1. **Get Version Number:** u generates a random $nonce$ and executes GetVersion$(id, nonce)$ with **AS** to get the current, fresh version number vno of the data object, and signature σ over a tuple $m = (vno, M_c)$, where M_c is in the form of metadata from the underlying DPDP system. u verifies the TPM-based freshness attestation on vno, checks that the version number in the freshness attestation matches the version number in the signed tuple, and computes $GVerify(PK_G, m, \sigma)$ to verify that M_c was authenticated by an authorized modifier (i.e., member of G).

2. **Get Data:** u and **SS** execute Challenge and Prove using version number vno, and u verifies the result using Verify and metadata M_c from the previous step. If all verifications succeed, u is assured that it has retrieved authentic data.

Next, we give the protocol for a user to update part or all of the stored data.

Update protocol for user u:

1. **Get Version Numbers:** u picks a random $nonce$ and executes GetCurrNextVersion$(id, nonce)$ with **AS** to get current version number vno and next version number nv, and an exclusive update lock registered with **AS**. If u fails to receive the lock from **AS**, it delays based on some backoff strategy and tries again. A successful execution of GetCurrNextVersion provides a signed tu-

[1]For key management purposes, it might be good to have a "change key" command that GM could execute, but this is a simple extension that we leave out of our current discussion.

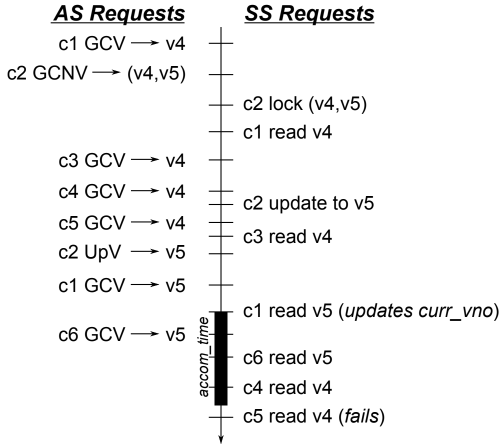
ple $m = (cvno, M_c)$ containing current metadata M_c, and is verified as it was in the retrieve protocol.

2. **Perform Update:** This is an interactive protocol run between C and **SS**. u sends $GSign(SK_{G,u}(cvno, nvno))$ to **SS** to obtain an update lock. **SS** verifies the signature with PK_G, and if verification is successful, grants an update lock to u. Once the lock is obtained, u can perform whatever data retrievals are necessary to prepare its update using $cvno$,[2] and then use PrepareUpdate to make the update request that u sends to **SS**. **SS** then executes PerformUpdate to produce updated metadata M_c' and proof $P_{M_c'}$ which it sends back to u. Finally, u uses the VerifyUpdate procedure from the DPDP scheme to check the information **SS** provided — if the update information is not accepted, then u should contact **AS** to release its update lock, and then follow error-handling procedures to decide whether to retry the update or just report an error to the user.

3. **Update Version Number:** If the update transaction with **SS** was accepted in the previous step, then we finish the transaction by updating the metadata storage on **AS**: First, u constructs the pair $m = (nvno, M_c')$ and computes $\sigma = GSign(SK_{G,u}, m)$. Then u executes UpdateVersion$(id, m, \sigma, nonce)$ with **AS**. If this succeeds, **AS** will complete the operation, which includes releasing its update lock for this transaction.

3.5 Concurrency, Consistency, and Freshness

In this section we address issues of concurrent updates, and show how multiple devices and users each get a consistent view of the data with strong freshness guarantees. We use the term "client" to refer to a user/device pair, so different clients could in fact represent the same user accessing the service from different devices. The following example shows the different possibilities for timing of requests from multiple clients, where **AS** interaction is shown on the left of the time-line (which goes down), and **SS** interaction is shown on the right. Clients are identified as c1, c2, and so on, and requests are abbreviated for compactness (e.g., GCV is for GetCurrentVersion, UpV is for UpdateVersion, etc.).



We consider a few interesting scenarios below that could potentially involve race conditions when there is an overlap between clients who try to read/update the data store at the same time. We do not consider clients whose read-only requests overlap with each other, or clients whose update requests do not overlap with other requests, since those are straightforward cases. In the diagram, client c2 is updating the data store, while all other clients are reading data.

Overlapping read/update requests: Client c1 places a request which starts before any c2 operations, and so will complete its operation with the version that was current when c1 made its request to **AS** (assuming it does so in time — see "slow clients" below for the alternative). c3 and c4 both start read requests by getting the initial version number (v4) from **AS**, and succeed with this version number despite the fact that c2 performs the update operation on **SS** before they issue the read request to **SS**. In our protocol, the current version number tracked by **SS** is not updated until a read request comes in with a new version number. The point of this is to guarantee that the version number on **AS** has been updated, without requiring any interaction between **SS** and **AS**. Therefore when c3's read request is made to **SS**, it proceeds as if there had been no update operation on **SS**. For c4, there is indeed a new version on **SS**, but the previous version is returned because the read request comes before the accommodation time for old versions expires.

Overlapping update requests: Update locks on **SS** and **AS** are granted exclusively to an authorized, requesting client for up until a maximum time limit (to prevent starvation), so it is not possible for two or more legitimate updates that correctly follow our protocol to overlap. Since lock requests are signed, it is impossible for an unauthorized user to be granted a lock and start an update sequence, so overlapping with unauthorized users is also not a possibility.

Slow clients: When c1's read request comes through for version v5, *curr_vno* is updated on **SS** and the accommodation time clock is started. At that point we have two clients who have not completed their read requests, with c5 being a particularly slow client that does not complete its read request until after the accommodation time expires. Therefore, c5's request fails, which is the only possible action in a system that enforces freshness. A slow client performing an update proceeds analogously, with the requirement that it must complete its update before the locks time out. While this isn't strictly necessary to enforce freshness, some form of lock timeout is necessary to recover from client failures. If handling slow client updates becomes a problem, either the lock timeout can be increased or the client could be allowed to complete its update operation if no other client has requested and captured the update lock.

3.6 Group Implementations

In this section we describe two ways of implementing the digital signature group membership verification service, with slightly different properties.

Using a Group Signatures Scheme: Group signature schemes are ideally suited to the system we have described. We consider the group signature scheme due to Ateniese *et al.* [2] — due to the tight correspondence between group signature operations and our required operations of $GSign$ and $GVerify$, we do not provide extensive details here. For readers unfamiliar with group signature schemes, we refer them to the Ateniese *et al.* paper for further information.

[2]Note that the authenticated version number provides a reliable way of validating data cached for added efficiency.

When a group signature scheme is set up by a group manager, the manager receives both a secret group manager key SK_{GM} and a group public key PK_G, and can use these values to admit individual users to the group through a `Join` operation. On admission, each member u receives a secret signing key $SK_{G,u}$. This maps quite directly to the definitions of $GSign$ and $GVerify$ that we require, so a group signature scheme is a natural fit for our protocol.

A group signature scheme provides several additional properties that are not necessary in the way we have defined our multi-user DPDP scheme, but might be useful in some situations. In particular, group signature schemes are designed so that signatures from different members of a group are indistinguishable without knowledge of a special secret or trapdoor. In our setting, this means that the data store could be updated so that anyone can verify that the update was made by a legitimate group member, but without additional information it would be infeasible to tell which group member made the update. Signature schemes provide various methods to `Open` a signature and determine who created a particular signature, but typically only the group manager or some other specially-designated user can do that. This property might be useful in situations where sensitive data is stored on a shared storage server, and tracking that information back to an individual user is undesirable.

Using a Standard Signatures Scheme and Certificates: Standard digital signature schemes such as RSA have been studied far more extensively than group signature schemes, so it might be desirable to rely on standard signatures when the stronger group anonymity properties of group signatures are not needed. In this case, the group manager serves as a certification authority (CA) and issues group-membership certificates to individual users to link their individual public verification keys to the group.

Mapping to our requirements, every signature σ produced by $GSign$ will include not only the user-produced RSA signature, but also a copy of the signing party's group membership certificate. The group public key PK_G is simply the verification key for certificates issued by the group manager, and the $GVerify$ operation checks the validity of the group membership certificate and the RSA signature, both of which are present in signature σ. This is a straightforward scheme the satisfies the requirements of our protocols, albeit without any additional anonymity properties.

4. HIERARCHICAL STORAGE

In this section, we extend the single-object DPDP protocol of Section 3 to a multi-object protocol, where the users update shared objects stored in a filesystem hierarchy. We could, of course, directly use the solution of the previous section where we treat each filesystem object (both files and directories) as a separate, protected data store, each with its own virtual monotonic counter and metadata, but that does not take advantage of savings that are possible due to the hierarchical structure of the filesystem.

Let client groups create files/directories (henceforth referred to as *objects*) on **SS**, and set permission levels for those objects by assigning each object a *label* ℓ, indicating whether the object has the same set of authorized modifiers as its parent. In typical filesystems, most objects will inherit the set of authorized modifiers from its parent, and in that case ℓ is set to "inherit"; otherwise, ℓ is set to "control point",

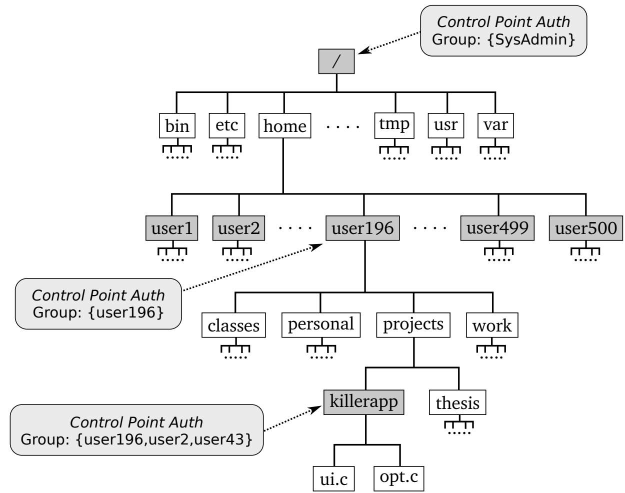


Figure 1: Sample file hierarchy with control points shaded grey, and three control points

and the object is designated as a control point. Authentication data is either signed or propagated up through "inherit" links to a signed directory, where we minimize the use of expensive signatures through inherit labels. An example of such a structure is illustrated in Figure 1.

In Figure 1, the system root directory is designated as a control point, users' home directories are designated as control points as well, since they are exclusively controlled by each user. The contents of each user's home directory is under the exclusive control of that user, but a user can assign read/write privileges selectively to other users if they so wish. In our example, user196 can allow a workgroup permission to modify a collaborative project directory, such as `killerapp/` in the figure. Since `killerapp/` does not inherit the exclusive permissions of its parent directory, it is designated as a control point.

We now describe protocols for retrieving (reading) an object and updating (writing) to an object.

Protocols for user u:

- **Retrieve:** To retrieve an object from the filesystem, the user performs an authenticated read of that object from the multi-user DPDP data store, which provides the proof and metadata for that retrieve. The user verifies the proof, and checks to see if the object is a control point. If the object is a control point, the user contacts **AS** to verify that the metadata is valid (i.e., has a valid group signature) and fresh. Otherwise, the verification continues recursively up the tree. The final result of a retrieve operation is either verified data or the special response `reject` indicating that the operation failed.

- **Update:** Update is similar to retrieve, in that the user first accesses and updates the object that she is interested in — if that object is a control point, she signs the metadata and sends it to **AS** to update the current version number. If the object is not a control point, then the user recursively updates the parent object with the new metadata (which continues until a control point is reached).

Referring to Figure 1 for an example, if a user196 is retrieving a file `/home/user196/classes/exam.txt`, then she will need to do an authenticated read of the file `exam.txt`, which produces metadata M_{c1} for that file. Since `exam.txt` is not a control point, there is no verification for M_{c1} on **AS**, but M_{c1} should be embedded in the `classes` directory object, so we do an authenticated read of that directory information which returns a proof and metadata for the `classes` directory, which we will call M_{c2}. However, since `classes` is also not a control point, we cannot verify M_{c2} directly, so do an authenticated retrieve of the metadata M_{c2} from the `user196` home directory, which returns the home directory metadata M_{c3}. This *is* a control point, so we verify that M_{c3} is the current value for `user196` control point using **AS**, and verification of this metadata authenticates all of the accesses below it, inductively authenticating the file `exam.txt`.

An update to `exam.txt` would work similarly to this, but performing updates along the path to the control point rather than just verifications.

The main difference between our hierarchical data storage solution and our original multi-user DPDP solution is in the use of control points. To make this more precise, we provide the following formal definitions of operations on control points.

Definition 2. Hierarchical Storage

1. `retrieveControlPoint`$((c_i, object), \alpha, P_\alpha) \rightarrow \eta$; $\eta \in$ ("accept","reject"): This is an interactive protocol run between the client, **AS**, and **SS**, where the client wishes to retrieve a control point stored on **SS**, and verify its integrity and freshness. The client first contacts **AS** to get and verify the current version number and metadata associated with this control point, and then challenges **SS** to prove possession of the control point. The client then verifies the proof given by **SS** using the meta-data obtained from **AS**. If the verification succeeds, the client is assured that it received authentic and fresh data.

2. `updateControlPoint`$((c_i, \text{object}), (c_i, \text{newObject}), Sign_{PK_G}(c_i, \text{newObject}), \alpha) \rightarrow ((c_i, \text{newObject}), \alpha')$: This is an interactive protocol run between the client, **AS** and **SS**, in which the client wishes to update a current control point (c_i, object) with $(c_i, \text{newObject})$. The client first gets the current version number of the control point, and the signature over the control point from **AS**, via the `GetVersionNumber` protocol, and verifies them. If the verification succeeds, the client executes `PerformUpdate` with **SS**, where **SS** will prove that it performed the update correctly. If the client accepts the proof, it will update the version information meta-data on **AS** by executing the `UpdateVersion` protocol.

5. SECURITY PROPERTIES

In this section we discuss the formal security properties of our scheme. We first give some intuition, and then move to a few technical details, with the complete details left to the full version of this paper [30]. Security requirements are defined using a game between an adversary and an oracle that plays the part of honest parties executing the protocol under question. The standard DPDP security model, as developed by Erway *et al.* [8], defines an "Adaptive Chosen File" (or ACF) attack on the system. We use this game as a starting point, adopting certain aspects of the POR security game defined by Juels and Kaliski [13] and extending to explicitly handle multiple objects and groups of users.

All colluding dishonest parties are modeled as a single adversarial entity, $\mathcal{A}$, which is only restricted by requiring that attacks run in polynomial time. Since $\mathcal{A}$ subsumes the functionality of the potentially corrupt **AS**, it has access to **AS**'s TPM, restricted by the Trusted Platform Security Assumption (so $\mathcal{A}$ has no access to any protected data such as secret keys managed by the TPM). On the other side of this game, we model all honest authorized modifiers as a "verifier" party $\mathcal{V}$, since these parties are verifying that the changes that they make are reflected by the servers (i.e., by $\mathcal{A}$). $\mathcal{V}$ operates as an oracle that $\mathcal{A}$ interacts with, maintaining secrets for authorized users such as group signing keys, and exposing an interface that includes oracle functions $\mathcal{O}^{update}$ and $\mathcal{O}^{retrieve}$ that provide ways for $\mathcal{A}$ to request that standard DPDP operations be initiated by $\mathcal{V}$. Keeping this high-level notation simple necessitates an unusual feature in our model: a client performing a retrieve or update operation performs several steps to prepare a request, perform the request by interacting with the servers, and verify the server responses. However, the servers are under control of the adversary, so $\mathcal{A}$ must provide a "callback" interface for the oracle to interact with the adversarially controlled servers.

The ACF game starts in a *setup phase*, where $\mathcal{A}$ interacts with the oracle to create arbitrary files, access and modify them, all while monitoring what $\mathcal{V}$ and **AS**'s TPM do for these requests. At the end of the setup phase, $\mathcal{A}$ returns a file identifier id for which it will try to fool the clients into accepting a bogus version. At this point we need to define the "good" version of file id; we use an intuitive description here, since a more rigorous treatment (which is in the full version of this paper) requires establishing non-trivial notation. The basic idea is straightforward: if we take the sequence of updates performed during the setup phase, and extract just the update operations made by users who can legitimately affect file id, then an honest execution of these update operations produces the "good" version of the file, which we denote as F_{id}.

So finally, the adversary wins this game if it can convince $\mathcal{O}^{retrieve}$ to accept anything other than F_{id}. More rigorously, in the game we execute the following experiment for adversary $\mathcal{A}$ and multi-user DPDP scheme MDPDP, where λ is a security parameter that determines lengths of keys and other cryptographic properties:

$$\begin{array}{l} \text{Experiment } \mathbf{Exp}^{\mathsf{setup}}_{\mathcal{A},MDPDP}(\lambda) \\ \quad id \leftarrow \mathcal{A}^{\mathcal{O}}(1^\lambda) \\ \quad f \leftarrow \mathcal{O}^{retrieve}(id) \\ \quad \text{output "win" if } f \neq \mathsf{reject} \text{ and } f \neq F_{id} \end{array}$$

Our final security definition follows directly from this experiment, saying that no polynomial time adversary can win this game with better than negligible probability.

Definition 3. Scheme MDPDP is a secure multi-user DPDP scheme for hierarchical storage if, for any polynomial time adversary $\mathcal{A}$ and any $c \geq 1$, there exists a $\lambda_0 > 0$ so that for all $\lambda \geq \lambda_0$,

$$Prob\left[\mathbf{Exp}^{\mathsf{setup}}_{\mathcal{A},MDPDP}(\lambda) = \text{"win"}\right] < \frac{1}{\lambda^c}\ .$$

The full theoretical analysis of the MDPDP scheme presented in this paper is beyond the scope of this conference paper. Details are provided in the full paper showing that our scheme is a secure multi-user DPDP scheme under this definition.

6. EXPERIMENTS

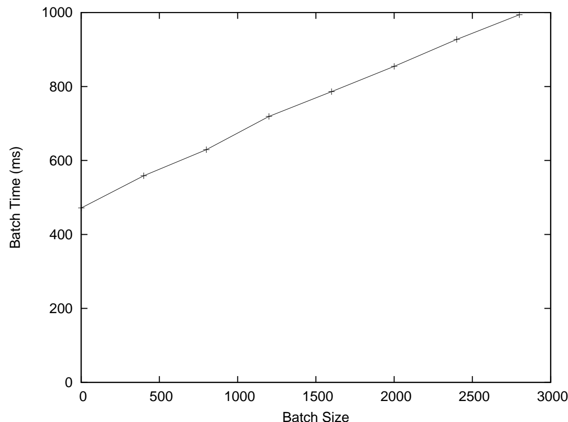
In this section, we present preliminary results of experiments into the practicality of the system we have described in this paper. The experiments in this paper are not intended to be an extensive performance evaluation, but are meant as a preliminary exploration of the feasibility of our approach. As mentioned earlier, prior work has provided extensive comparisons of the log-based and the hash tree-based virtual monotonic counter schemes [29], so we concentrate here on the more efficient hash tree-based scheme.

The storage server **SS** is a modified DPDP server, and performance is consistent with the experiments performed by Erway *et al.* [8] so we do not report those results here. Instead, we focus on our main innovation, the TPM-based authentication server **AS**, which is also the main bottleneck of the system due to speed limitations of the TPM.

Since our system requires modest extensions of a standard TPM, we could not perform our tests on TPM hardware, but instead use the timing-accurate TPM simulator of Gunupudi and Tate [12]. We use a timing profile of an Infineon TPM, which provides the fastest RSA signature time (308.88ms) of any TPM model, and perform tests on a server with an Intel Xeon X3353 processor and CPU-based computations written in Java and run under the IcedTea6 1.6 JVM. The TPM simulator supports plugins for TPM extensions, and we created a hashtree plugin to provide virtual monotonic counters using the techniques described in Section 3.

In this test setting, doing a single isolated virtual monotonic counter attestation, complete with server processing and setting up the authenticated command, takes 425ms; therefore, the maximum request throughput that could be achieved by directly applying the techniques of Sarmenta *et al.* [25] is approximately 2.35 requests per second.

Using our technique of batched attestation (described in Section 3.3.2), we achieve dramatically higher throughput rates, suitable for many practical applications. We tested multiple batch sizes, starting at 400 requests and going up to 2800 requests in steps of 400, running 50 tests at each size and computing the average time, giving the results below.



The times were very consistent, varying by at most 10% around the average over the 50 runs. The batch times show a linear increase as the batch size increases, which is expected since building the nonce hash tree and creating attestation responses are both linear time operations in the batch size.

To optimize our use of resources, we balance time of CPU time with the TPM time, as that keeps both computational resources at full utilization. Specifically, while a TPM request is outstanding, we are using the CPU to post-process the results of the previous TPM operation and construct the nonce tree for the next TPM request. If these times are balanced, we complete construction of the nonce tree at the same time the previous TPM request finishes, so we are immediately ready to issue the next command to the TPM. The TPM time is a constant 425ms regardless of batch size, and so TPM and CPU time are balanced when the total time is twice this, or 850ms. Zooming in on our graph, we see that this happens with a batch size of approximately 1975 requests. At that size, we process 1975 requests every 425 ms, giving *an optimized throughput of 4647 requests per second*. This is a dramatic improvement, with a throughput around 2000 times greater than the base rate of 2.35 requests per second. This is fast enough to satisfy even some very demanding applications.

7. CONCLUSION AND FUTURE WORK

Using lightweight trusted hardware on a remote storage server, we extend previous work in the area of dynamic provable data possession (DPDP) and construct DPDP protocols for the setting where multiple collaborative users work on remotely stored shared data. Our constructions provide proofs of integrity, freshness, and when used to implement a hierarchical filesystem they authenticate both the data itself and its place in the filesystem. A new method of batching attestations provides throughput for multiple attestations that is several orders of magnitude greater than the straightforward solution, bringing this well into the range of practical application.

In ongoing work, we are further developing the theoretical security model and performing more extensive performance evaluation. A practically relevant direction for future work would be to explore the idea of load-balancing across multiple servers. With respect to our model, it should be straightforward to distribute the storage server across multiple machines for load-balancing client requests, but it is harder to distribute the authentication server tasks since our current protocols depend on the the TPM and virtual monotonic counter being globally unique for each control point. One possible solution is to distribute control points to different **AS**es; however, it's not clear how to duplicate resources for an individual control point, and so distributing in this manner simply creates multiple points of failure when validating a path in the filesystem, making the system more brittle. We leave the search for a better solution as an open problem.

8. ACKNOWLEDGMENTS

The work reported in this paper is supported by the National Science Foundation under Grant No. 0915735. The authors would like to thank Adam Lee and the anonymous reviewers for their helpful comments.

9. REFERENCES

[1] G. Ateniese, R. C. Burns, R. Curtmola, J. Herring, L. Kissner, Z. N. J. Peterson, and D. X. Song. Provable data possession at untrusted stores. In *ACM CCS*, pages 598–609, 2007.

[2] G. Ateniese, J. Camenisch, M. Joye, and G. Tsudik. A practical and provably secure coalition resistant group signature scheme. In *CRYPTO*, pages 255–270, 2000.

[3] G. Ateniese, R. D. Pietro, L. V. Mancini, and G. Tsudik. Scalable and efficient provable data possession. In *SecureComm*, 2008. Article 9.

[4] K. D. Bowers, A. Juels, and A. Oprea. Hail: a high-availability and integrity layer for cloud storage. In *ACM CCS*, pages 187–198, 2009.

[5] C. Cachin. Integrity and consistency for untrusted services. In *SOFSEM*, pages 1–14, 2011.

[6] C. Cachin, I. Keidar, and A. Shraer. Fail-aware untrusted storage. In *DSN*, pages 494–503, 2009.

[7] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Attested append-only memory: making adversaries stick to their word. In *SOSP*, pages 189–204, 2007.

[8] C. C. Erway, A. Küpçü, C. Papamanthou, and R. Tamassia. Dynamic provable data possession. In *ACM CCS*, pages 213–222, 2009.

[9] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten. SPORC: Group collaboration using untrusted cloud resources. In *OSDI*, pages 337–350, 2010.

[10] J. Feng, Y. Chen, D. Summerville, W.-S. Ku, and Z. Su. Enhancing cloud storage security against roll-back attacks with a new fair multi-party non-repudiation protocol. In *Proc. IEEE Consumer Communications and Networking Conference (CCNC)*, pages 521–522, 2011.

[11] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Practical oblivious storage. In *ACM CODASPY*, pages 13–24, 2012.

[12] V. Gunupudi and S. R. Tate. Timing-accurate TPM simulation for what-if explorations in trusted computing. In *Proceedings of the International Symposium on Performance Evaluation of Computer and Telecommunicatoin Systems (SPECTS)*, pages 171–178, 2010.

[13] A. Juels and B. S. Kaliski, Jr. PORs: Proofs of retrievability for large files. In *ACM CCS*, pages 584–597, 2007.

[14] H. Kaplan. Persistent data structures. In *Handbook on Data Structures and Applications*. CRC Press, 2001.

[15] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda. TrInc: Small trusted hardware for large distributed systems. In *NSDI*, pages 1–14, 2009.

[16] J. Li, M. N. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *OSDI*, pages 121–136, 2004.

[17] J. Li and D. Mazières. Beyond one-third faulty replicas in Byzantine fault tolerant systems. In *NSDI*, 2007.

[18] P. Mahajan, S. T. V. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: Cloud storage with minimal trust. In *OSDI*, pages 307–322, 2010.

[19] D. Mazieres and D. Shasha. Don't trust your file server. In *Workshop on Hot Topics in Operating Systems*, pages 113–118, 2001.

[20] D. Mazières and D. Shasha. Building secure file systems out of Byzantine storage. In *ACM PODC*, pages 108–117, 2002.

[21] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. D. Gligor, and A. Perrig. TrustVisor: Efficient TCB reduction and attestation. In *IEEE Symposium on Security and Privacy*, pages 143–158, 2010.

[22] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for TCB minimization. In *EuroSys*, pages 315–328, 2008.

[23] R. C. Merkle. A digital signature based on a conventional encryption function. In *CRYPTO*, pages 369–378, 1987.

[24] T. Moyer, K. Butler, J. Schiffman, P. McDaniel, and T. Jaeger. Scalable web content attestation. In *ACSAC*, pages 95–104, 2009.

[25] L. F. G. Sarmenta, M. v. Dijk, C. W. O'Donnell, J. Rhodes, and S. Devadas. Virtual monotonic counters and count-limited objects using a TPM without a trusted OS. In *Proceedings of the First ACM Workshop on Scalable Trusted Computing*, pages 27–42, 2006.

[26] H. Shacham and B. Waters. Compact proofs of retrievability. In *ASIACRYPT*, pages 90–107, 2008.

[27] A. Shraer, C. Cachin, A. Cidon, I. Keidar, Y. Michalevsky, and D. Shaket. Venus: verification for untrusted cloud storage. In *CCSW*, pages 19–30, 2010.

[28] E. Stefanov, M. van Dijk, A. Oprea, and A. Juels. Iris: A scalable cloud file system with efficient integrity checks. *IACR Cryptology ePrint Archive*, 2011, 2011.

[29] S. R. Tate and R. Vishwanathan. Performance evaluation of TPM-based digital wallets. In *International Symposium on Performance Evaluation of Computer and Telecommunication Systems*, pages 179–186, 2010.

[30] S. R. Tate, R. Vishwanathan, and L. Everhart. Multi-user dynamic proofs of data possession using trusted hardware – expanded version. Available at http://span.uncg.edu/pubs, 2012.

[31] Trusted Computing Group. Trusted Platform Module Specifications – Parts 1–3. Available at https://www.trustedcomputinggroup.org/specs/TPM/.

[32] M. van Dijk, J. Rhodes, L. F. G. Sarmenta, and S. Devadas. Offline untrusted storage with immediate detection of forking and replay attacks. In *Workshop on Scalable Trusted Computing*, pages 41–48, 2007.

[33] M. van Dijk, L. Sarmenta, C. O'Donnell, and S. Devadas. Proof of freshness: How to efficiently use an online single secure clock to secure shared untrusted memory. Technical Report CSG Memo 496, MIT, 2006.

[34] H. Xiong, X. Zhang, D. Yao, X. Wu, and Y. Wen. Towards end-to-end secure content storage and delivery with public cloud. In *ACM CODASPY*, pages 257–266, 2012.

[35] Q. Zheng and S. Xu. Fair and dynamic proofs of retrievability. In *ACM CODASPY*, pages 237–248, 2011.