Strong Accountability for Network Storage

Aydan R. Yumerefendi and Jeffrey S. Chase
Duke University
{aydan,chase}@cs.duke.edu

Abstract

This paper presents the design, implementation, and evaluation of CATS, a network storage service with strong accountability properties. A CATS server annotates read and write responses with evidence of correct execution, and offers audit and challenge interfaces that enable clients to verify that the server is faithful. A faulty server cannot conceal its misbehavior, and evidence of misbehavior is independently verifiable by any participant. CATS clients are also accountable for their actions on the service. A client cannot deny its actions, and the server can prove the impact of those actions on the state views it presented to other clients.

Experiments with a CATS prototype evaluate the cost of accountability under a range of conditions and expose the primary factors influencing the level of assurance and the performance of a strongly accountable storage server. The results show that strong accountability is practical for network storage systems in settings with strong identity and modest degrees of write-sharing. The accountability concepts and techniques used in CATS generalize to a broader class of network services.

1 Introduction

A system is *accountable* if it provides a means to detect and expose misbehavior by its participants. Accountability provides powerful incentives to promote cooperation and discourage malicious and incorrect behavior.

This paper proposes and evaluates system support for *strongly accountable* storage protocols and services. A system is strongly accountable if it provides a means for each participant to determine for itself if others are behaving correctly, without trusting assertions of misbehavior by another participant who may itself be compromised.

To illustrate strong accountability properties and the means to provide them for shared storage, we present CATS, a *certified accountable tamper-evident storage* service. CATS is a rudimentary network storage service: it enables clients to read and write a shared directory of objects maintained by a CATS server. CATS provides clients with the means to verify that the server executes writes correctly, and that read responses are correct given the sequence of valid writes received at the server. Crucially, strong accountability of the server also extends to the clients: a correct server can prove that its state resulted from actions taken by specific clients. Clients cannot deny or repudiate their operations on a strongly accountable server, or the impact of those operations on the shared state. The CATS network storage service is based on a generic state storage toolkit. Our intent is that the CATS toolkit can serve as a substrate for a range of accountable services with richer semantics.

Various notions of accountability and end-to-end trust appear in many previous systems (see Section 6). Our approach is focused on assuring *semantically* faithful behavior, rather than performance accountability [15]. For example, a compromised CATS server can deny service, but it cannot misrepresent the shared state without risk of exposure that is provable to all participants. In this respect, CATS offers a strong form of tamper-evidence [17] at the server. More generally, strong accountability may be viewed as a form of Byzantine failure detection. It is a complementary alternative to consensus voting [8, 28], which is vulnerable to "groupthink" if an attack compromises multiple replicas.

The CATS toolkit embodies the state of the art from research in *authenticated data structures* [2, 6, 12, 20, 21, 24, 25]. As in many of these schemes, CATS servers can supply cryptographic proofs that their read responses resulted from valid write operations. CATS also incorporates new primitives for secure challenges and audits. Servers export a *challenge* interface to verify that writes are incorporated and visible by other clients. The *audit* interface enables an auditor to verify integrity of writes over arbitrary intervals of recent history, so that a faulty server cannot revert writes without risk of detection. Other participants may verify audit results without requiring trust in the auditor. Finally, we extend previous work on authenticated data structures with new capabilities for fast, reliable secondary storage access and caching, and investigate the impact of design choices on storage, access, and accountability costs.

This paper is organized as follows. Section 2 presents an overview of our approach, threat model, and assump-

tions. Section 3 describes the design of the CATS state management toolkit and storage service. Section 4 outlines the details of our prototype implementation, while Section 5 presents experimental results with the prototype. Section 6 sets our work in context with related work, and Section 7 concludes.

2 Overview

The CATS network storage service is a building block for distributed applications or services. Its purpose is to act as a safe repository of state shared among the participants (actors) of a distributed system, who are clients of the storage service. The shared state guides the behavior of the clients, and the clients may affect the group through updates to the shared state. The clients may represent a community of users, who are responsible for their updates.

CATS supports the core functions of a network file system. Clients can create, read, and write opaque objects identified by unique identifiers (*oids*). Each write request generates a new version of an object. Versions are named by monotonically increasing timestamps of the writes that created them. Reads retrieve the most recent version of the object. A client may also request a prior version of an object that was valid at a specified time. We have limited our focus to essential object storage functions: for example, in its present form, CATS does not support nested directories, symbolic names or links, renaming, partial reads or writes, truncates, or appends.

2.1 Threat Model

The strong accountability property provides clients with the means to verify that the server protects the integrity of the shared state: read responses correctly reflect the valid writes issued by the clients. Moreover, the system can assign provable responsibility for an object version to the client that issued the write for that version.

To define the threat model, we specify the properties of a correctly executing storage service.

- **Authenticity and undeniability.** The service executes writes issued only by properly authorized clients. A client cannot deny responsibility for a write that it has issued.
- **Freshness and consistency.** Writes are applied in order, and reads always return the object version that was valid at the requested time (i.e., the version created by the previous write).
- **Completeness or inclusion.** Writes are included in the service state and are visible to other authorized clients.

A faulty storage server could attempt to violate any of the above properties. For example, it could accept writes from unauthorized clients, improperly modify existing objects, or replay earlier writes. In a more subtle attack the server could acknowledge the completion of a write request but attempt to conceal it from other clients. The storage service or a client could attempt to deny that it executed or requested a completed operation. Table 1 lists possible attacks.

CATS does not prevent any of these forms of misbehavior. In particular, it does not attempt to remove the need for trust among clients and servers; trust is essential for sharing and cooperation. Rather, the philosophy of our approach is: "trust but verify" [35]. CATS holds the storage server and other participants accountable for their actions, so that a faulty actor is exposed and isolated. Accountability precludes an effective attack to subvert the behavior of the overall system without unavoidable risk of detection. Integrity also requires that an accountable system protect its participants from false accusations of guilt. In CATS, a participant's guilt is cryptographically provable to all participants.

CATS is compatible with read access controls. However, violations of confidentiality are not part of the threat model: a CATS server cannot be held accountable for serving reads that violate the policy.

2.2 Trust Assumptions

Table 2 lists the components of a CATS service. The accountability properties of the system rest on correct functioning of two core elements.

Asymmetric cryptography. Each actor can sign its requests using at least one asymmetric key pair bound to a principal; the public key is distributed securely to all actors, e.g., using a Public Key Infrastructure (PKI). Digital signatures ensure integrity, authenticity, and non-repudiation of actions.

Secure authentication is the core trust assumption in our system. If keys are compromised, then an actor may be falsely held accountable for actions it did not take. Note, however, that an attacker cannot misrepresent the actions of any actor whose key it does not possess, and any actions it takes with a stolen key can be traced to that key. Even so, a successful attack against a trusted PKI root would open CATS to subversion by the attacker; thus a PKI constitutes a Trusted Computing Base in the traditional sense.

External publishing medium. Each actor has a means to publish a *digest* of its state periodically. A digest is a secure hash over the actor's state at a point in time. Digests are digitally signed, so an actor cannot repudiate previous claims about its state. Only the storage server publishes digests in the accountable storage service example.

Each actor must have independent access to the history of published digests in order to validate proofs independently. Thus the publishing medium is a trusted

Attack	*Defense*
Server fails to execute write or object create	provable detection
Client repudiates object create or write	provable detection
Server denies write or object create	provable detection
Client writes out of order	rejected by server
Server returns invalid read response	provable detection
Fraudulent or unauthorized writes	provable detection with simple static ACLs; relies on trusted external authorization server for richer policies
Server replays or reverts valid writes, destroys objects, reorders writes, or accepts out-of-order writes	verifiable detection by challenge or audit
Tampering & forking, or other variants of above attacks on data integrity	verifiable detection by challenge or audit
Violation of privacy or read access policy	no defense

Table 1: **Summary of attacks and defenses.**

Component	*Trust Assumptions*
Clients and servers	Trusted but accountable: Cannot subvert the system without risk of provable detection. Incorrect or invalid actions taken with a stolen key are provably detectable.
Publishing medium	Trusted to render published digests visible to all participants. Accountable for forgeries or alterations to published digests. Cannot subvert the system without risk of provable detection.
Authorization service	Not required for simple static access control lists. Must be trusted to enforce richer access control policies if needed.
Trusted platform / trusted path	Necessary for individual user accountability, else no distinction between misbehavior of user and misbehavior of user software.
Public key infrastructure:	Trusted Computing Base: Compromise of PKI can subvert the system.

Table 2: **Summary of components and trust assumptions.**

component. Since digests are signed, a faulty publishing medium cannot forge or alter digests unilaterally, but it could mount a denial-of-accountability attack by concealing them. The medium could also collude with a participant to alter the digest history of that participant, but such an attack would be detectable and provable by another participant that caches previously published digests. In essence, a faulty publishing medium can destroy the accountability properties of the system, but it cannot itself subvert the system.

Due diligence. Accountability in CATS relies on voluntary actions by each actor to verify the behavior of the others. For example, if the clients of the CATS storage service do not request or check proofs that their writes were executed and persist (see 2.3), then a faulty server may escape detection. Of course, a lazy client may free-ride on the diligence of others, possibly leading to a classic tragedy of the commons. What is important is that an attacker cannot determine or control its risk of exposure.

2.3 Challenges and Audits

An important element of our approach is to incorporate *challenge* and *audit* interfaces into service protocols. Challenges force a server to provide a cryptographic proof certifying that its actions are correct and consistent relative to published state digests. An important form of challenge is an *audit* to verify consistent behavior across a sequence of actions or an interval of history. Challenges and audits do not require trust in the auditor; any actor may act as an auditor. If an actor's challenge or audit reveals misbehavior, the actor can present its case as a *plaintiff* to any other actor, which may verify the validity of the accusation.

Auditing defends against a freshness attack, in which a faulty server discards or reverts a valid write that it has previously accepted (Section 3.5). A client with authority to read an object can choose to audit the sequence of updates to that object through time to ensure freshness and consistency. Actors select a degree of auditing that balances overhead and the probability of detection of misbehavior. The server cannot change its history to conceal its misbehavior from an auditor.

For example, a CATS storage server may be challenged to prove that it has incorporated a recently completed write into its published state digest. It may be audited to prove that its current state resulted from a sequence of valid writes by authorized clients, and that its read responses reflect that state. A challenged or au-

dited CATS server must also justify that any accepted write complies with the existing access control policy (See Section 2.4). A faulty server cannot allow unauthorized writes or execute its own writes using a fraudulent identity. If an attacker subverts the server, the worst damage it can cause is to deny service or discard data; covert modifications (including reverted writes) are tamper-evident and can be exposed at any time through challenges and audits. In particular, the server can be held provably accountable for a forking attack [17].

2.4 Discussion and Limitations

User accountability, identity, and privacy. Ideally, the accountability of a system should extend to its users. For example, the CATS storage service could hold users accountable for actions they take within a community or organization, possibly exposing them to legal sanction or other social sanction.

However, user accountability requires strong identity bindings. There is a fundamental tension between accountability and privacy or anonymity. Strong identities exist today within user communities that interact using shared services in areas such as health care, infrastructure control, enterprise work flow, finance, and government, where accountability is particularly important. Many of these areas already use some form of PKI.

User accountability also requires a trusted path to the user so that software running with the user's identity cannot act without the user's consent. The questions of how to establish a trusted path to the user and ensure that the user authorizes actions taken with its identity are outside the scope of this paper. The solution is likely to require some trust in the platform (e.g., a Trusted Platform Module [33]).

More generally, a signed message or action is not a guarantee of intent by the principal: many security breaches occur when components are subverted or private keys are otherwise stolen [3]. Importantly, when such a breach occurs, our approach ensures that attackers cannot modify existing history, as noted above.

External publishing medium. The external publishing medium is a certified log that is globally visible. The publishing medium is much simpler than the storage service because it does not accept updates, does not support dynamic object creation, and does not allow write-sharing. There are several ways to implement an external publishing medium. Timestamping services [7] publish digests using write-once media such as newspapers. An alternative solution may use a trusted web site. Finally, the clients of the service can implement the publishing medium by using a peer-to-peer approach that leverages some form of gossip and secure broadcast.

Access control. As part of its operation a CATS service should demonstrate that a write complies with a write access control policy. It is easy to hold the server accountable for enforcing a simple and static policy for write access control; for example, if the creator of an object provides an immutable list of identities permitted to modify the object. We are investigating how to extend strong accountability to richer access control policies. A CATS server may use an external authorization server to govern access control policy and issue signed assertions endorsing specific clients to write specific objects. However, such an authorization server must be trusted, and it is an open question how to assure accountability for it in the general case.

Denial of service through challenges and audits. One concern is that challenges and audits could be used as the basis for a denial-of-service attack on the server. We view this problem as an issue of performance isolation and quality of service: resources expended to serve audits and challenges may be controlled in the same way as read and write requests. Community standards may define a "statute of limitations" that limits the history subject to audit, and bounds the rate at which each actor may legitimately request audits and challenges.

3 CATS Design

Figure 1 presents a high-level view of the CATS storage service and its components. The service and its clients communicate using the Simple Object Access Protocol (SOAP). Each message is digitally signed. The Public Key Infrastructure (PKI) and the publishing medium constitute the trusted base. Misbehavior of any other actor can be detected and proven.

The storage service has simple and well-defined operations (Table 3), which allows us to design it as a thin layer on top of a generic state store toolkit. The CATS toolkit incorporates common state management functions to organize service state to enable strong accountability. The CATS storage service uses the state store toolkit to ensure the tamper-evidence of its state and to prove to its clients that it maintains its state correctly.

The toolkit's core principle is to separate state management from state transformation due to application logic. The CATS toolkit represents service state as an indexed set of named, typed, elements. The state store accepts updates annotated with a request—digitally signed by the issuer—and links these action records to the updated data objects. The CATS accountable storage service is a simple application of the toolkit because each element corresponds to exactly one storage object, and each update modifies exactly one element. The toolkit provides foundational support for strongly accountable application services, although accountability for more advanced services must be "designed in" to the internal logic, state representations and protocols.

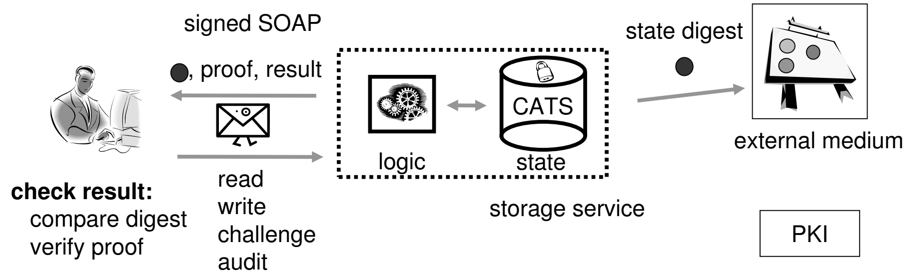

Figure 1: **Overview of a CATS service**, e.g., the accountable storage service.

Operation	*Description*
`read(oid)`	Returns the most recent version of the object with the specified *oid*. In our prototype, every read response includes a proof of correctness relative to the digest for the most recent epoch (implicit challenge).
`read(oid, time)`	Returns a signed response with the version of the specified object at given time (epoch). In our prototype, every read response includes a proof of correctness relative to the digest for the epoch (implicit challenge).
`write(oid, value, current)`	Overwrites the specified object with the given value. The server accepts the request only if at the time of the write the version stamp of the object is equal to *current*. Otherwise, returns the object's version stamp.
`challenge(oid, epoch)`	Requests that the server provide a proof that the object with the given *oid* is or is not included in the service state, relative to the published digest for the given time (epoch). The response includes a membership proof for the object and its value, or an exclusion proof if the *oid* was not valid.
`audit(oid, epoch, span, depth)`	Requests that the service substantiate the write history for the object named by *oid* over the time interval $[epoch, epoch + span]$. The request is equivalent to *depth* randomly selected challenge requests over the interval. The depth parameter controls the degree of assurance.

Table 3: **Accountable storage service operations.** The service accepts reads and writes to a set of named versioned objects. All client writes and all server responses are digitally signed so that actors can be held accountable for their actions. Challenges and audits provide the means for clients or third-party auditors to verify the consistency of a server's actions relative to periodic non-repudiable digests generated by that server and visible to all actors.

3.1 Action Histories

Strong accountability requires CATS to represent action histories whose integrity is protected, and provide primitives to retrieve, exchange, and certify those histories. We have to know who said what to whom, and how that information was used.

Every CATS request and response carries a digital signature that uniquely authenticates its sender and proves the integrity of the message. Specifically, the content of each request and response is encapsulated in a signed *action record*. Actors may retain a history of action records and transmit them to make provable statements about the behavior of other actors and the validity of their actions. The action history may be integrated into the internal service state.

For example, the CATS storage server links each element or object to the set of client requests that affected it or produced its value. A CATS server may justify its responses by presenting a sequence of cached, signed action records that show how its state resulted from the actions of its clients, starting from its published state at a given point in time, as described below.

Action records are verifiable by any receiver and are non-repudiable by the sender. Importantly, symmetric cryptography approaches, e.g., Message Authentication Codes or SSL, are not sufficient for accountability. Shared keys cannot guarantee non-repudiation of origin or content: any party in possession of the shared key can forge the actions of another using the key.

3.2 State Digests and Commitment

A CATS server periodically generates a signed digest over its local state. The state storage data structure provides a function for an actor to compute a compact, fixed-size hash over an atomic *snapshot* of its current state.

Servers publish their digests by including them in their responses to clients and publishing them to an external medium (Section 2.2). Publishing a digest commits the server to a unique view of its state and its history at a specific point in time. The server cannot lie about what its state was at that time without the risk of exposure. Any modification of the state affects the digest; any conflicts with previously published digests are detectable and provable. In particular, the server cannot safely present different views of its state to different clients—a *forking attack* [17, 23]. While nothing prevents the the server from returning different digest values to different clients, clients can detect such misbehavior by comparing digests among themselves or against the published copy.

An *epoch* is the time interval between generation of two consecutive state digests by some actor. An epoch is *committed* when the digest of its end-state is published. Epochs are numbered with consecutive timestamp values. To preserve the integrity of the timeline, the computation of each new state digest incorporates the digest from the previous epoch as an input. This form of chaining prevents the service from removing a committed snapshot or adding a new snapshot in the past.

3.3 Proofs

Servers can make provable statements about their states or operations relative to a published digest. A CATS server can generate two kinds of proofs:

- **Inclusion.** The server claims that a given $(oid, value)$ pair was recorded in the service state at the end of some epoch. The claim is backed by an unforgeable *membership proof* showing that the $(oid, value)$ pair was used to compute the digest for that epoch in a proper way.
- **Exclusion.** The server claims that no object with a given *oid* existed in its state at the end of some epoch. The claim is backed by a proof exhibiting a proper computation of the epoch digest that did not include an object with that *oid*. If a previous writer had written an object with that *oid*, it could have verified that the server incorporated the object into its digest in a proper way that would have precluded such a proof.

Section 3.6 describes these proofs and the data structures that support them in more detail.

A client may request a proof by issuing a *challenge* on a target *oid*. The server responds to a challenge with an inclusion proof if the *oid* was valid, or an exclusion proof if it was not. If the challenge is issued by an actor other than the last writer, an inclusion proof includes a signed copy of the write that generated the object's value.

Challenges enable a client to substantiate any *read* or *write* response. A writer can issue a challenge to verify that its write to an *oid* was incorporated into the server's state for an epoch, as represented in the server's published digest, in a proper way that precludes the server from denying or safely concealing the write at a later time. A reader can issue a challenge to verify that its view of the server's state is consistent with the server's digests, and therefore (by extension) consistent with the views verified by the writers and seen by other readers.

3.4 Request Ordering and Consistency

The state of a stored object results from an ordered sequence of writes to the object. Accountability for object values requires that the server and the object's writers agree on and certify a serial order of the writes.

Each stored object is annotated with a version stamp consisting of a monotonically increasing integer and a hash of the object's value. The action record for an object write must include the object's previous version stamp, certifying that the writer is aware of the object's state at the time of the write. The server rejects the write if the version stamp does not match, e.g., as a result of an intervening write unknown to the writer. The writer may retry its write, possibly after completing a read and adjusting its request.

Inspection of a sequence of write actions to an *oid* will reveal any attempt by the server to reorder writes, or any attempt by a writer to deny its knowledge of an earlier write.

3.5 Freshness and Auditing

It remains to provide a means to hold a server accountable for "freshness": an object's value should reflect its most recent update prior to the target time of the read. A faulty server might accept a write and later revert it. Freshness is a fundamental problem: a server can prove that it does not conceal writes to an object over some interval only by replaying the object's complete write history over the interval, including proofs that its value did not change during any epoch without a valid write action. Reverted writes are the most difficult form of the "forking attack" to defend against; in essence, the *fork consistency* assurance of SUNDR [17, 23] is that if the server conceals writes, then it must do so consistently or its clients will detect a fault.

Our approach in CATS is to provide a means for any participant to selectively audit a server's write histories relative to its non-repudiable state digests. The strong accountability property is that the server cannot conceal any corruption of an object's value from an audit, and any misbehavior detected by an audit is verifiable by a third party. Although reverted writes for objects or intervals that escape audit may go undetected, a server cannot

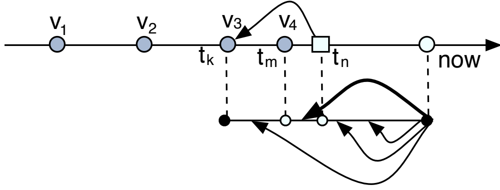

Figure 2: At time t_n the malicious operator reverts the effect of the update performed at time t_m and sets the corresponding value to the one valid at time t_k. An audit that tracks the value through a sequence of snapshots detects this violation, as long as a read is performed in the interval $[t_m; t_n)$.

predict or control what actions will be audited.

Figure 2 illustrates the auditing process. In this example, an object receives an update at time t_m, which is correctly applied and the value of the object becomes v_4. However, at a later time, t_n, the server reverts the value of the element to a previous value v_3, created at time t_k. As a result of this intervention, reads to the object produce an incorrect result (v_3 instead of v_4).

To certify freshness, a client must challenge and examine the object's value for every epoch in the interval since the last reported update to present. We will refer to this interval as the *span* of auditing ((t_k, now) in our example). If the audit confirms that the object and its reported value were incorporated into all digests for the interval, then the client can safely conclude that the reported value is indeed fresh. On the other hand, if the server cannot provide a membership proof for the object and its value relative to some subsequent digest, then the server is faulty.

Clients can reduce the overhead of certifying freshness at the expense of a weaker, probabilistic assurance. Instead of inspecting every snapshot in a span, the client may instead issue challenges for randomly selected snapshots in the span. We refer to the number of audited epochs as the *depth* of the audit. In the above example, the client performs an audit of depth 4. The audit successfully detects the misbehavior, since it inspects the service state in a snapshot created in the interval $[t_m, t_n)$, in which the object has the reverted value v_4.

The CATS toolkit and accountable storage service support probabilistic audits as described in the above example. Clients can issue audit requests and specify a list of randomly selected snapshots from a specified auditing span. CATS then constructs a membership proof for the object and its value in the specified snapshots and returns the result back to the client.

The probability of detecting a reverted update for a given state element depends on the lifetime of the reverted update relative to the span of auditing. Using the above example, as the number of snapshots in $[t_n, now)$ increases, the probability of selecting a snapshot in $[t_m, t_n)$ decreases. Thus, detecting an offense becomes more costly as more time passes since the offense took place. If an incorrect update is not noticed earlier, it may be difficult and expensive to detect it later.

One way to deal with this problem is to examine more snapshots in the span (increase the depth of auditing) when verifying older objects. More formally, if p is the probability of selecting a snapshot that reveals misbehavior, the probability of detecting misbehavior after d snapshot inspections is $P(p, d) = 1-(1-p)^d$. If the lifetime of a reverted update decreases relative to the span, then p decreases. Figure 3 shows the probability that audits of fixed depth detect that the value of an element is incorrect for different values of p and d.

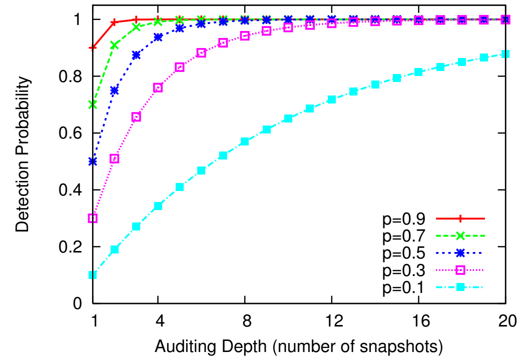

Figure 3: The longer a reverted update remains undetected (decreasing p), audits will have to inspect a larger number of snapshots to ensure a certain probability of detection.

Another possible solution is to have trusted auditors periodically inspect every object and ensure its freshness. The auditors publish commit records visible to all clients, certifying that the service state until the point of inspection is consistent. An alternative, less expensive form of auditing can impose a probabilistic bound on the number of reverted updates. The core idea is to choose objects at random and inspect their values in randomly selected state snapshots. Using a sampling theory result stating that the successful selection of n $(n > 0)$ white balls from an urn ensures that with probability $1 - 1/e$ there are no more than an $1/n$ fraction of black balls in the urn, we can prove the following two theorems:

Theorem 1. *Examining the complete execution history of n $(n > 0)$ objects chosen uniformly at random from all state objects present in a service's state, and ensuring the correctness of each update operation to these ob-*

jects, guarantees that with probability at least $1 - 1/e$ the service has maintained correctly at least $(1 - 1/n)$ fraction of all objects over its complete execution history.

Theorem 2. *Examining n ($n > 0$) state update operations and ensuring that none of them reverts a previous update, ensures that with probability at least $1 - 1/e$, no more than an $1/n$ fraction of all state updates have been reverted.*

We can further improve the effectiveness of audits using repetition. It takes $O(\log \delta)$ repetitions of an auditing algorithm to boost the probability of detection to $(1 - \delta)$ for any $\delta \in (0, 1)$.

Finally, another complementary approach is for each writer to challenge periodically objects it has written to ensure that its updates persist until another authorized client overwrites them.

3.6 Data Structure Considerations

We considered two ways to compute a compact digest over a dynamic set of elements and demonstrate that an element with a given *oid* and value is or is not part of the set: cryptographic accumulators [5] and authenticated dictionaries [2, 21, 24]. Of the two, authenticated dictionaries promise to be more practical as they have logarithmic update cost and produce proofs logarithmic in the size of the set (both for inclusion and exclusion). The advantages of authenticated dictionaries prompted us to use them as a basis for the CATS toolkit.

3.6.1 Authenticated Dictionaries

An authenticated dictionary is a map of elements with unique identifiers, and allows for efficient identifier-based updates and retrievals. Most proposed schemes are based on Merkle trees [24]. They organize the elements of the set in a binary search tree. Each node of the tree is augmented with an *authentication label*—the output of a *one-way*, *collision-resistant* hash function applied on the contents on the node's sub-tree. More specifically, the label of a leaf node is equal to the hash of its data and identifier, and the label of an internal node is equal to the hash of the concatenation of the labels of its left and right children. The recursive scheme labels the root of the tree with an *authenticator* covering the entire tree—its value is a digest computed over every element of the set.

A membership proof for an element constructed by an authenticated dictionary consists of a sequence of sibling hashes: one for each node on the path from the root of the tree to the leaf node containing the element. Using the sibling hashes and the hash of the element's data, an actor can recompute the root authenticator and compare it to the known authenticator. It is computationally infeasible to fabricate a set of sibling labels that combine with the element hash to yield the known root authenticator (e.g., an epoch digest). Exclusion proofs require that the hashes incorporate the key values for each node in the search tree, so that each membership proof also certifies that the path to the object reflects a proper sorted order for the tree. If the tree is in sorted order, then an exclusion proof for a given *oid* is given by the set of hashes for any subtree that covers the portion of the key space containing *oid*, but that does not contain that *oid* itself.

3.6.2 Practical Issues

A number of organizational issues impact the performance of authenticated dictionaries. In this subsection we describe the cost dimensions, the way they impact performance, and the specific choices we made to design the state store.

Tree degree. It is possible to authenticate any search tree simply by allocating extra space within each node to store its authentication label. A naive application of the same idea to a high-degree tree such as a B-Tree [4], however, can have an adverse impact on authentication performance. The high degree used by a B-Tree increases the size of the sibling set needed to compute an authentication label: a membership proof for a B-Tree storing 1,000,000 four-byte identifiers with out-degree of 100 is more than 7 times larger than a proof generated by a binary tree storing the same data.

Theorem 3. *Balanced binary search trees generate membership proofs of minimal size.*

Proof. A balanced tree of degree B storing N objects has height at least $\log_B N$. A membership proof consists of a constant size component for each sibling of every tree node along a root to leaf path. Each tree node has $B - 1$ siblings. Therefore, a membership proof has at least $f(B) = (B - 1) \log_B N$ components. This function obtains its smallest value for $B = 2$. □

The above result exposes a dilemma for building a scalable authenticated state store: indexing large collections requires the use of an I/O efficient data structure, e.g., a B-Tree, however, it will be costly to compute digests and to generate, transmit, and verify proofs in such a data structure. We address this challenge with a hybrid approach consisting of a balanced binary tree mapped to the nodes of a B-Tree for efficient disk storage (Figure 4). While this approach ensures the optimality of search operations, updates can trigger a large number of I/O operations to balance the binary tree. We relax the requirement of global binary tree balance and trade it for improved I/O performance: only the part of the tree that is stored within a single B-Tree block should

be balanced. This restriction may increase the height of the binary tree but offers an acceptable tradeoff between membership proof size and I/O efficiency. This solution is similar to the one of Maniatis [21]. Theorem 3 shows that his choice of a binary tree is indeed optimal.

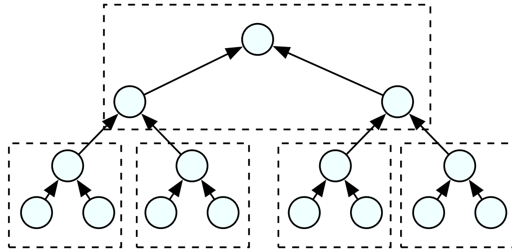

Figure 4: Mapping a binary search tree to disk blocks for efficient storage.

Data layout. Data placement also impacts the performance of authenticated search trees. Search trees provide two options for storing data. Node-oriented organizations store data at all nodes, while leaf-oriented designs store data only at the leaves. These two choices impact the space requirement, size of membership proof, and authentication cost. Leaf-oriented trees require more space but need less data and processing to compute an authentication label [32]. In the context of a binary tree mapped to a B-Tree, the smaller size of internal tree nodes in a leaf-oriented design results in a large portion of the binary tree being mapped to a single B-tree node. This organization reduces the I/O and authentication cost and we adopt it in our design.

Persistence. Since it is expensive to generate, publish, store, and audit epoch digests, it is desirable to minimize the number of epochs for a write stream by maximizing the *epoch length*, the average number of updates within an epoch. While there exist methods for building space-efficient persistent data structures [9] with *constant* space overhead for epoch length of one update, persistent authenticated search trees incur *logarithmic* space overhead; each update invalidates a logarithmic number of authentication labels. Using longer epochs reduces a constant fraction from this overhead by amortizing the cost of tree maintenance (including regenerating the authenticators) across multiple update operations. For example, since labels close to the root of the tree are affected by every update operation, delaying label regeneration to the end of the epoch ensures that each label is computed exactly once. Processing more updates within a single epoch also decreases space utilization and the disk write rate.

However, there are limits to the practical epoch length. Longer epochs include more writes and modify more disk blocks; the number of dirtied blocks for a write set also grows logarithmically with the size of the tree. If the set of dirty blocks does not fit within the I/O cache, then the system will incur an additional I/O cost to regenerate the labels at the end of the epoch. Longer epochs also increase the time to create a state digest and acknowledge the completion of an operation. Epoch length is an adjustable parameter in our design. in order to reflect these complexities.

4 Implementation

We implemented the state store toolkit and the storage service using C#, .NET framework 2.0, and Web Services Enhancements (WSE) 2.0 for SOAP/Direct Internet Message Encapsulation (DIME) communication. The storage service toolkit consists of 12,413 lines of code, and the service implementation took another 6,084 lines. The toolkit implementation consists of several layered modules. The lowest layer exports a block-level abstraction with a unified interface to pluggable underlying storage implementations. The current implementation uses files for storage media so it can easily use parallel or replicated storage supported by the file system.

An intermediary block cache module controls access to storage and buffering. The block cache provides interfaces to pin and unpin blocks in memory. An asynchronous background process monitors the block cache and drains the cache to keep the number of blocks in use between configured high-water and low-water marks. Each block can be locked in shared or exclusive mode. The current implementation does not perform deadlock detection, and it maintains all versions of a given object.

The authenticated dictionary implementation uses the block cache module to access storage blocks. The dictionary is implemented as a B-Tree with the keys within each block organized in a binary search tree. The current implementation uses Red-Black trees and it allows to use other algorithms if necessary. We use Lehman and Yao's B-link tree concurrency algorithm [16] to ensure optimal concurrency. The state store can correctly recover after a failure using write-ahead logging.

The storage service is log-structured [27]. It consists of an append-only log and an index for locating data on the log. We use the toolkit for the index, and a variant of the toolkit's write-ahead log for the append-only log. Each log entry consists of two portions: the original XML request, and the request payload. Clients identify objects using 16-byte key identifiers.

The service is structured as a collection of stages (Figure 5) using a custom staging framework inspired by SEDA [34]. A pool of worker threads services requests for each stage. The size of the pool changes dynamically in response to offered load. Stages have queues of fixed capacity to absorb variations in load. Once queues fill

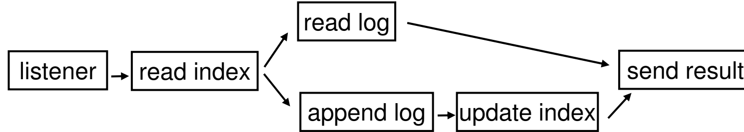

Figure 5: **Storage service implementation.** The storage service consists of an index and an append-only log. The service is implemented as a collection of stages. Each stages is associated with a pool of worker threads. Pools can grow and shrink depending on load.

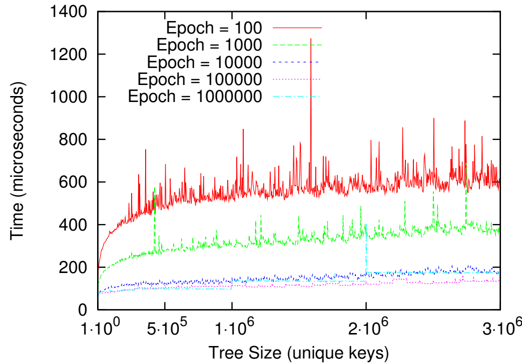

Figure 6: The time to apply an update to the authenticated dictionary is logarithmic relative to the number of unique keys in the index and increases as epoch length decreases.

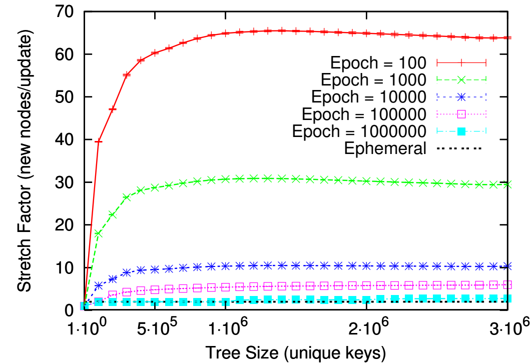

Figure 7: The average number of new nodes introduced per update operation, *stretch factor*, is logarithmic relative to the epoch length. Smaller epochs have significantly higher stretch factors resulting in larger indexes.

up, the service starts rejecting requests. Clients resubmit rejected requests at a later time.

The service and its clients communicate via SOAP over TCP. To limit the overhead of XML, we transfer binary data using DIME attachments. Requests and responses are digitally signed. The current implementation uses a custom XML canonicalization scheme and RSA for signatures. Future releases will integrate the service with web services standards such as XML-Signatures and WS-Security.

The service processes update requests in epochs. The service creates a new epoch as soon as it receives a valid write request for an object that has already been modified in the current epoch. This approach allows for the best granularity of accountability as it preserves every version of an object. However, it has impact on performance if workload has high contention (Section 5.4). An alternative implementation choice is to limit the rate at which new epochs are created by imposing a minimum bound on the length of an epoch. To enforce this bound, the service must reject a write if it will generate an epoch ahead of its due time. Clients must buffer and coalesce their writes, which will result in decreased data sharing.

5 Evaluation

In this section we present an evaluation study of our prototype. Our goal is to understand how accountability impacts performance under varying conditions. In particular, we study the impact of public key cryptography, object size, request rate, workload contention, epoch length, and auditing.

5.1 Methodology

We run all tests on IBM x335 servers running Windows 2003 Server Standard Edition operating system. Each machine has 2 Pentium IV Xeon processors running at 2.8GHz, 2GB of RAM, 1GB Ethernet, and IBM SCSI hard disks with rotation speed of 10,000 RPM, average latency and seek time of 3 ms and 4.7 ms respectively.

We use a population of 1,000,000 unique objects and pre-populate the server to have one version of each object. Since the cost of index operations grows logarithmically with the size of the state store, a state store with 1,000,000 objects has already reached a steady state in which increasing size has minor impact on performance (doubling the size of the store will increase the cost of operations by approximately 3%).

We use a community of clients to issue requests to the service using the SOAP interface. Our workloads consist of synthetic reads/writes of a controlled size. We vary

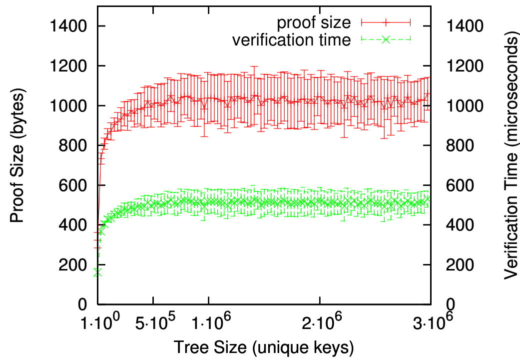

Figure 8: Membership proof size is logarithmic relative to the number of unique keys in the index and is on the order of 1KB. The time to verify a membership proof is on the order of 550 μseconds.

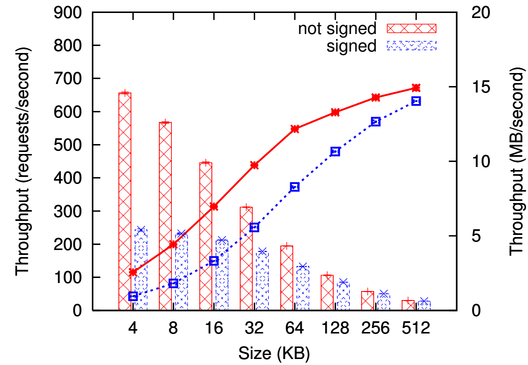

Figure 9: Maximum achievable data bandwidth for storage service RPC interface based on SOAP/DIME and WSE 2.0. The results suggest that this interface has high overhead and services using it will be potentially communication-bound.

the "heat" of the workload by biasing the probability of selecting a given object as the target of a read/write operation. We denote heat using the notation $X : Y$, which we interpret to mean: $X\%$ of the requests go to $Y\%$ of the objects. For example 100:100 is a uniform workload, and 80:20 is the typical hot/cold workload. All tests last 3 minutes with initial 30 seconds for warming the caches. We report averages and standard deviations from 10 runs. Our test workloads are a rough approximation of real storage workload and the magnitude of the performance results we report may differ from that of real systems. However, the focus of our study is the cost of accountability and our workloads allow us to vary the key parameters to quantify those effects.

We use SHA-1 to compute authentication labels. Object keys are assigned randomly in the identifier space (16 bytes). This is a conservative approach intended to limit spatial locality and stress our implementation. In practice, spatial locality can reduce the overhead of persistence and the cost of state digest computation. We store the index and the log in blocks of size 64KB. The index cache has capacity 2000 blocks and the log cache has capacity 8000 blocks. The cache sizes are chosen so that neither the index, nor the append-only log fit in memory. For storage medium we use files residing on the local NTFS file system. The files are accessed using the standard .NET I/O API. For improved performance we place the append-only log and the index on separate disks. We commit dirty blocks to disk every 2 seconds.

5.2 Toolkit Behavior

We first examine the behavior of the CATS toolkit. In particular, we study membership proofs (size and time to verify), and the impact of epoch length on update and storage costs. We use a set of 3,000,000 random keys of size 16 bytes and insert them consecutively in a CATS index. We create new epochs at a controlled rate.

Figure 6 shows the time to apply an update to the authenticated index as a function of the unique keys in the index and epoch length. We observe that the update cost is logarithmic with respect to the number of unique keys. The cost also grows as epoch length decreases. There are two primary reasons to explain this behavior. First, smaller epochs amortize the cost of maintaining the index and computing authentication labels among a smaller number of updates. Second, smaller epochs produce larger indexes and cause higher write rate per update operation.

Next we examine the impact of epoch length on storage costs. We define the *stretch factor* for a given epoch length to be the average number of new (Red-Black tree) nodes introduces by an update operation. Ephemeral trees have epoch length of infinity and stretch factor of 2. Figure 7 shows how the stretch factor changes with epoch length. The stretch factor is logarithmic relative to the number of unique objects in the index and grows as epoch length decreases. Thus very small epochs can produce very large indexes.

Finally, we examine membership proofs. Figure 8 shows the size of membership proofs and the time to verify a membership proof as a function of the number of unique keys in the index. Importantly, these metrics do not depend on epoch size. As expected, membership proof size is logarithmic relative to the unique keys in the index. For our configuration, a proof is on the order of 1KB and it takes about 550 μseconds to verify.

The important point to take away from these experiments is that membership proofs are compact and relatively cheap to verify. Update operations in our unoptimized prototype have acceptable performance. Epoch

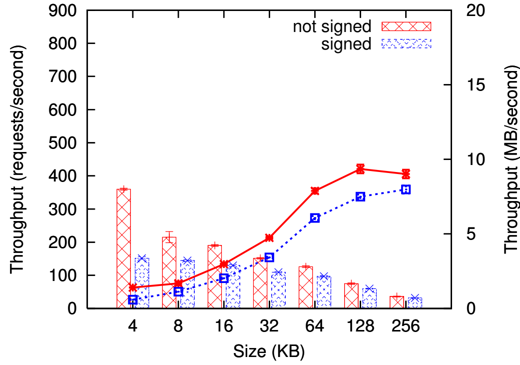

Figure 10: Write saturation throughput for objects of different size using 80:20 workload for configurations with and without digital signatures. As object size increases, the relative cost of digital signatures decreases.

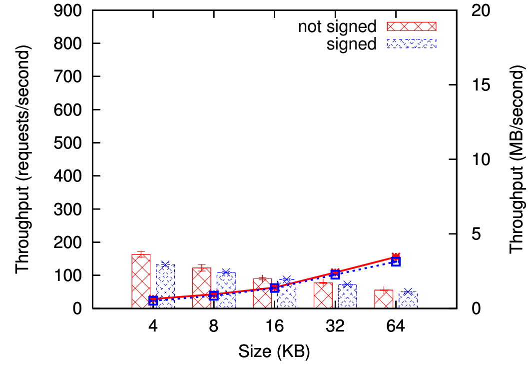

Figure 11: Read saturation throughput for objects of different size using 80:20 workload for configurations with and without digital signatures. Performance is seek limited due to the log-structured design.

length has direct impact on space overhead and update cost and is the primary factor that determines performance.

5.3 Saturation Throughput

We now measure the saturation throughput of the storage service prototype. The first experiment evaluates the maximum bandwidth a storage service using SOAP/DIME based on WSE 2.0 can achieve. We issue write requests to the service to store objects of controlled sizes. The client follows the full protocol, while the server only extracts the object from the SOAP message, validates it (when requests are signed) and sends a response to acknowledge the operation. No storage takes place. The server's CPU is saturated. Figure 9 shows the saturation request rate and the resulting data bandwidth for objects of different size with and without using digital signatures. Although we are using 1Gbps LAN, the observed data bandwidth peeks at approximately 15MB/s. This is the best possible request rate that our storage service can achieve. Importantly, the results show that digital signatures are expensive, however, the relative cost signing a request decreases as object size increases.

In the next experiments we measure the read and write saturation throughput of our implementation under a traditional hot/cold workload (80:20). We use the load generator to fully saturate the server and measure the sustained throughput. We vary object size to study its impact. Figure 10 shows the performance for write and Figure 11 for read operations. As object size increases, the request rate decreases and the transfer rate increases. With the increase of object size, write performance increases relatively to the maximum achievable request rate. The service offers better write than read performance, which is due to its log-structured design.

As object size increases the relative penalty of using digitally signed communication decreases. 4KB signed writes achieve 42% of the request rate of writes without signatures. This number increases to 86% for object of size 256KB. The ratios are better for reads: 81% for 4KB, and 90% for 64KB. The log-structured design explains this observation: the read workload is seek-dominated and the time spent signing a request is smaller relative to the time spent obtaining the data from the storage medium.

Overall, the experimental results show that our implementation offers reasonable performance for both read and write operations and that the cost of digital signatures is not prohibitive.

5.4 Workload Contention

In Section 5.2 we concluded that epoch length has significant impact on performance. The storage service creates a new epoch if it receives more than one update operation to the same object in the same epoch. As a result of this design choice, the performance of the service will be dependent on the contention present in its workload. In the next experiments we study this relationship.

We use workloads of different heat, ranging from uniform (100:100) to highly skewed (80:1) and generate requests at controlled rates bellow the server saturation level. Figure 12 shows the observed epoch creation rate. For a given workload heat, processing requests at a higher rate creates new epochs more frequently. As a result, well-provisioned services processing high volumes of client requests will produce a large number of state snapshots. Similarly, more skewed workloads create new epochs at a higher rate. Clearly, mechanisms are needed to allow services to discard state snapshots to

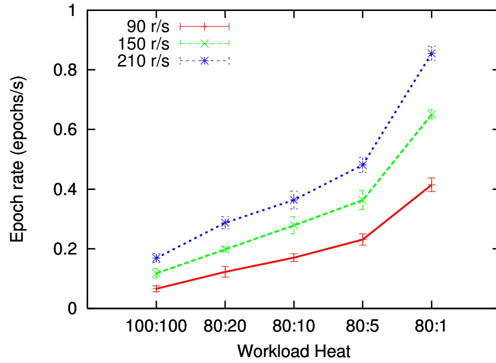

Figure 12: Rate of epoch creation as a function of request rate and workload contention. Higher request rates and contention create new epochs faster and can affect service longevity.

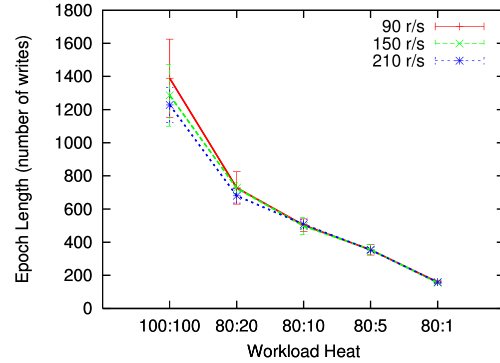

Figure 13: Epoch length as a function of request rate and workload contention. Epoch length is independent on request rate and is determined by workload contention.

keep space overhead and auditing costs at an acceptable level. Dealing with this problem is part of our future work. Figure 13 shows the resulting epoch lengths. Importantly, epoch length does not depend on request rate, however, it can vary from 1400 updates for a uniform workload, to 200 updates for a highly skewed workload. These results suggest that for typical usage scenarios index update operations (the primary cost component of accountability) will be in the range from 300 to 600 μseconds (Figure 6).

5.5 Challenges and Audits

Challenges and audits are the primary mechanisms to ensure a service behaves correctly over time. In the next experiments we study the impact of these mechanisms on service performance. The metric of concern is saturation throughput. For these experiments we populate the service so that the 1,000,000 objects are distributed among 100 epochs of length 10,000 updates. The resulting index is approximately 15 times larger than the index with all objects stored in an epoch of length 1,000,000. To isolate the impact of storage access on auditing, communication is not signed.

The cost of auditing operations depends on their *depth*, the number of inspected snapshots, *span*, the total number of snapshots in the examined interval, and *scope*, the fraction of objects that are likely to be audited, e.g., in our framework objects are audited only on access, and stale objects may never be audited. Using this terminology, a challenge is an audit of depth 1 and span 1.

Figure 14 shows the auditing saturation throughput for different combinations of span, scope, and depth. For a fixed scope, auditing becomes more expensive as span increases. Similarly, higher auditing depths are more expensive. Larger scope is also more expensive, however, its impact is less pronounced. We can explain the above results with the fact that the span and scope determine the locality of auditing operations: smaller spans and scopes concentrate audits on a portion of the index. Increasing the depth queries more snapshots and reduces the total audits/seconds rate.

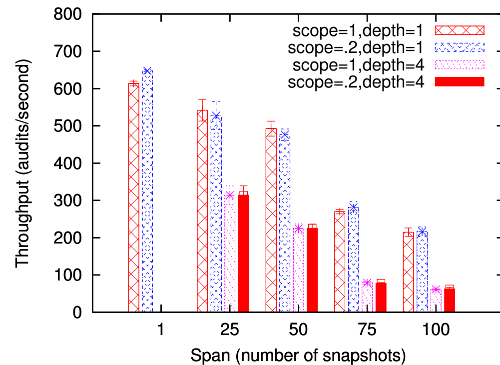

Figure 14: Auditing saturation throughput for different depths, spans, and scopes. Smaller spans show better performance due to increased locality. The impact of scope is less pronounced.

Controlling the rate of auditing is an important problem as it can be used to mount denial of service attacks. This is a resource control problem and solutions from the resource management domains are applicable. For example, clients can be allocated auditing budgets and the service can reject audits from a client if the client has exhausted her budget. Importantly, the services can be challenged for such rejections, to which it can reply with a collection of signed client requests to show that the client has exhausted its budget.

6 Related Work

Butler Lampson summarizes accountability as follows: "Accountability means that you can punish misbehavior". Our community often uses the term to refer to a means of enforcing an authorization policy by detecting and punishing violations after the fact, as an alternative to incurring the cost of preventing them in the first place [14]. More recent work seeks to enforce accountability for performance behavior as essential to effective functioning of networking markets [15].

Our notion of "semantic" accountability is different and complementary. We consider actors accountable if they can demonstrate that their actions are *semantically* correct as well as being properly authorized. An accountable server can be held responsible if its responses violate semantic expectations for the service that it provides. Accountability is "strong" if evidence of misbehavior is independently verifiable by a third party. Clients of a strongly accountable server are also strongly accountable for their actions, since they cannot deny their actions or blame the server.

Designing networked systems for accountability complements classical approaches to building trustworthy systems such as Byzantine Fault Tolerance [8, 28], secure hardware [31], and secure perimeters [14].

Secure perimeters attempt to *prevent* misbehavior by means of authorization and authentication. However, when actors reside in different administrative domains, there is no common perimeter to defend. Once the integrity of the perimeter is violated, it offers no protection, and even authorized actors may abuse the system to their own advantage.

Byzantine fault tolerance (BFT) is a general and powerful technique to *tolerate* misbehavior through replication and voting. However, BFT is vulnerable to collusion: a majority of replicas may collude to "frame" another. Also, in its pure form, BFT assumes that failures are independent, and thus it is vulnerable to "groupthink" in which an attack succeeds against a quorum of replicas. BFT offers limited protection when a service (and hence its replica set) is itself acting on behalf of an entity that controls its behavior (typical of web services). Strongly accountable systems can produce proofs of misbehavior that do not depend on voting or consensus, which makes statements of correctness or misbehavior provable to an external actor.

Secure hardware offers foundational mechanisms to ensure untampered execution at the hardware level. While a secure hardware platform protects the integrity of a programs code, it cannot ensure that its actions are semantically correct. Secure hardware can help provide a trusted path to the user by avoiding corruption of the software stack trusted by the user to correctly represent the user's intent.

Secure logging and auditing can also be used to preserve a tamper-evident record of events [29, 30]. In addition to maintaining a tamper-evident record of service events, we address the question of representing service state to make state management operations verifiable.

CATS embodies the state of the art from research in *authenticated data structures* [2, 6, 21, 25], beginning with the well-known technique of Merkle trees [24]. Relative to that work, it incorporates new primitives for secure auditing and integrates concurrency and recovery techniques from the database community. Our approach is inspired by Maniatis [21].

Our work incorporates these techniques into a toolkit that can act as a substrate for a range of accountable services. The elements of CATS have been used in many previous systems for similar goals of semantic accountability. Many systems require digitally signed communication and some maintain some form of signed action histories [11, 21]. Long-term historic state trails help establish provable causality in distributed systems [22]. Storage systems often reference data using cryptographic hashes to ensure tamper-evidence [26]. State digests are used to prove authenticity in file systems [10], applications running on untrusted environments [19], and time-stamping services [7].

The CATS toolkit is based on the storage abstraction of a binary search tree. The Boxwood Project [18] also provides a toolkit for building storage infrastructures from foundational search tree abstractions. Boxwood does not address the problem of accountability.

SUNDR [17, 23] and Plutus [13] are two recent network storage systems that defend against attacks mounted by a faulty storage server. These services are safe in the sense that clients may protect data from the server, and the clients can detect if the server modifies or misrepresents the data. Plutus emphasizes efficient support for encrypted sharing, while SUNDR defends against attacks on data integrity, the most difficult of which is a "forking attack". In both Plutus and SUNDR file system logic is implemented by clients: the server only stores opaque blocks of data. In SUNDR the server also participates in the consistency protocol.

Both SUNDR and Plutus can detect various attempts by the server to tamper with the contents of stored blocks. However, the papers do not attempt to define precise accountability properties. In general, they blame any inconsistencies on the server: it is not clear to what extent clients are accountable for their actions or for managing file system metadata correctly. For example, SUNDR does not show how to defend against a client that covertly deletes a file created by another user in the same group.

We argue for a stronger notion of accountability in which the guilt or innocence of the server *or its clients*

is provable to a third party. This is a subtle change of focus as compared to SUNDR, where the primary concern is to build a tamper-evident system and terminate its use once misbehavior is detected. CATS file semantics are implemented and enforced by the server in the conventional way, rather than implemented entirely by the clients. Clients can independently verify correct execution relative to published digests, rather than "comparing notes" to detect inconsistencies. CATS uses similar techniques to SUNDR and Plutus to make the stored data tamper-evident, but it also defends against false accusations against a server , e.g., a malicious client that "frames" a server by claiming falsely that it accepted writes and later reverted them. Accountability extends to the clients: for example, if a client corrupts shared data, the server can identify the client and prove its guilt even after accepting writes to other parts of the tree.

The BAR model [1] addresses the problem of building cooperative peer-to-peer services across multiple administrative domains. This is a Byzantine Fault Tolerance approach optimized for the presence of rational peers. Since the underlying primitives depend on voting and replication, BAR-based services are vulnerable to collusion. However, BAR does not maintain history and does not use auditing to deal with problems that may arise if peers collude. Our approach shares similar goals with BAR and is complementary. We address the problem of building an accountable service, within the control of a single authority, that is accessed by multiple clients, who may reside in different administrative domains. In this context, the service implements all logic, and voting and replication are not sufficient. Our methodology forces the service to preserve a faithful record of its history to enable auditing in the future. BAR-based services may be able to enforce performance guarantees, something we currently do not support, although accusations are not provable outside the system.

7 Summary and Conclusions

This paper presents the design, implementation, and evaluation of CATS—a storage service and state storage toolkit with strong accountability properties. A distributed system is strongly accountable when participants have the means to verify semantically correct operation of other participants, and assertions or proofs of misbehavior are independently verifiable by a third party based on cryptographically assured knowledge, without reliance on voting or consensus.

CATS builds on a history of work on authenticated data structures, and incorporates the state of the art in that area into a prototype network storage service. Challenge and audit interfaces allow clients, peers, or auditors to verify the integrity of the storage service state through time, providing probabilistic defense against reverted or replayed writes or replayed requests and other attacks on the integrity of the data or the write stream. The probability of detection of a reverted state update is a function of the depth and rate of auditing.

Experimental results with the prototype show that the CATS approach to strong accountability has potential for practical use in mission-critical distributed services with strong identity. The cost of authenticated communication is not prohibitive and its relative overhead decreases with the size of the object set and the stored objects. Workload write contention and write-sharing are the primary factors that affect the cost of accountable storage in our prototype: contention and sharing increase the storage, computation, and auditing costs. The cost of auditing depends on object age and the desired level of assurance.

The accountable storage service described in this paper is a simple example of a strongly accountable service built using the CATS state storage toolkit. It illustrates a state-based approach to building accountable network services that separates service logic from internal state representation and management, associates state elements with the digitally signed requests of the clients responsible for them, and supports verifiably faithful execution of state management operations. In future work we plan to explore use of the state-based approach to build more advanced file services and application services with strong accountability.

Acknowledgments

We thank the reviewers for their helpful comments, and especially our shepherd, Erik Riedel. We also thank Petros Maniatis for his comments and assistance in understanding some subtleties of previous work on authenticated data structures, including his own. Partial support for this research was provided by Duke University and by related funding from IBM and Network Appliance. This research was not supported by the National Science Foundation.

References

[1] A. S. Aiyer, L. Alvisi, A. Clement, M. Dahlin, J.-P. Martin, and C. Porth. BAR Fault Tolerance for Cooperative Services. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, October 2005.

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

[3] R. J. Anderson. Why Cryptosystems Fail. *Communications of the Association for Computing Machinery*, 37(11):32–40, November 1994.

[4] R. Bayer and E. M. McCreight. Organization and Maintenance of Large Ordered Indices. *Acta Informatica*, 1:173–189, 1972.

[5] J. Benaloh and M. de Mare. One-way Accumulators: A Decentralized Alternative to Digital Signatures (Extended Abstract). In *Proceedings of Advances on Cryptology*, pages 480–494, May 1997.

[6] A. Buldas, P. Laud, and H. Lipmaa. Accountable Certificate Management Using Undeniable Attestations. In *Proceedings of the 7th ACM Conference of Computer and Communications Security*, pages 9–17, November 2000.

[7] A. Buldas, P. Laud, H. Lipmaa, and J. Villemson. Timestamping with Binary Linking Schemes. In *Proceedings of Advances on Cryptology*, August 1998.

[8] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance. In *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation*, February 1999.

[9] J. Driscoll, N. Sarnak, D. D. Sleator, and R. Tarjan. Making Data Structures Persistent. *Journal of Computer and System Sciences*, 38:86–124, 1989.

[10] K. Fu, M. F. Kaashoek, and D. Maziéres. Fast and Secure Distributed Read-only File System. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation*, pages 181–196, October 2000.

[11] Y. Fu, J. Chase, B. Chun, S. Schwab, and A. Vahdat. SHARP: An Architecture for Secure Resource Peering. In *Proceedings of the 19th ACM Symposium on Operating System Principles*, pages 133–148, October 2003.

[12] 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, January 2001.

[13] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus: Scalable Secure File Sharing on Untrusted Storage. In *Proceedings of the Second USENIX Conference on File and Storage Technologies*, March 2003.

[14] B. W. Lampson. Computer Security in the Real World. In *Proceedings of the Annual Computer Security Applications Conference*, December 2000.

[15] P. Laskowski and J. Chuang. Network Monitors and Contracting Systems: Competition and Innovation. In *Proceedings of SIGCOMM*, September 2006.

[16] P. L. Lehman and S. B. Yao. Efficient Locking for Concurrent Operations on B-Trees. *ACM Transactions on Database Systems*, 6:650–670, December 1981.

[17] J. Li, M. N. Krohn, D. Mazières, and D. Shasha. Secure Untrusted Data Repository (SUNDR). In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation*, pages 91–106, December 2004.

[18] J. MacCormick, N. Murphy, M. Najork, C. A. Thekkath, and L. Zhou. Boxwood: Abstractions as the Foundation for Storage Infrastructure. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation*, pages 105–120, December 2004.

[19] U. Maheshwari, R. Vingralek, and W. Shapiro. How to Build a Trusted Database System on Untrusted Storage. In *Proceedings of the 4th USENIX Symposium on Operating System Design and Implementation*, October 2000.

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

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

[22] P. Maniatis and M. Baker. Secure History Preservation through Timeline Entanglement. In *Proceedings of the 11th USENIX Security Sysmposium*, August 2002.

[23] D. Mazières and D. Shasha. Building Secure File Systems out of Byzantine Storage. In *Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing*, pages 108–117, July 2002.

[24] R. C. Merkle. Protocols for Public Key Cryptosystems. In *Proceedings of the 1980 Sysmposium on Security and Privacy*, pages 122–133, April 1980.

[25] M. Naor and K. Nissim. Certificate Revocation and Certificate Update. *IEEE Journal on Selected Areas in Communications*, 18(4):561–570, 2000.

[26] S. Quinlan and S. Dorward. Venti: a New Approach to Archival Storage. In *First USENIX conference on File and Storage Technologies*, January 2002.

[27] M. Rosenblum and J. K. Ousterhout. The Design and Implementation of a Log-Structured File System. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, July 1991.

[28] F. B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[29] B. Schneier and J. Kelsey. Cryptographic Support for Secure Logs on Untrusted Machines. In *Proceedings of the 7th USENIX Security Symposium*, pages 53–62, January 1998.

[30] B. Schneier and J. Kelsey. Secure Audit Logs to Support Computer Forensics. *ACM Transactions on Information and System Security (TISSEC)*, 2(2):159–176, May 1999.

[31] S. W. Smith, E. R. Palmer, and S. Weingart. Using a High-Performance, Programmable Secure Coprocessor. In *Financial Cryptography*, pages 73–89, February 1998.

[32] R. Tamassia and N. Triandopoulos. On the Cost of Authenticated Data Structures, 2003.

[33] Trusted Computing Group. Trusted platform module specification. `https://www.trustedcomputinggroup.org/groups/tpm/`.

[34] M. Welsh, D. E. Culler, and E. A. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, October 2001.

[35] A. Yumerefendi and J. S. Chase. Trust but Verify: Accountability for Network Services. In *Proceedings of the 11th ACM SIGOPS European Workshop*, September 2004.