

# Configurable Security for Scavenged Storage Systems

Abdullah Gharaibeh, Samer Al-Kiswany, Matei Ripeanu  
Electrical and Computer Engineering Department, The University of British Columbia  
{abdullah, samera, matei}@ece.ubc.ca

## ABSTRACT

Scavenged storage systems harness unused disk space from individual workstations the same way idle CPU cycles are harnessed by desktop grid applications like Seti@Home. These systems provide a promising low-cost, high-performance storage solution in certain high-end-computing scenarios. However, selecting the security level and designing the security mechanisms for such systems is challenging as scavenging idle storage opens the door for security threats absent in traditional storage systems that use dedicated nodes under a single administrative domain. Moreover, increased security often comes at the price of performance and scalability. This paper develops a general threat model for systems that use scavenged storage, presents the design of a protocol that addresses these threats and is optimized for throughput, and evaluates the overheads brought by the new security protocol when configured to provide a number of different security properties.

## Categories and Subject Descriptors

D.4.3 [Operating Systems]: File Systems Management – *Distributed File Systems*. D.4.6 [Operating Systems]: Security and Protection – *Authentication, Access Control*. D.4.8 [Operating Systems]: Performance – *Measurements*.

## General Terms

Performance, Design, Experimentation, Security.

## Keywords

Secure storage systems, performance evaluation.

## 1. INTRODUCTION

Data-intensive scientific applications, (e.g., in astrophysics, climate simulation, or biological research) create large datasets that need to be archived, retrieved, and processed. To support the high IO throughput requirements of these applications, high-cost, specialized storage systems such as Storage Area Networks (SAN) are often used. While these solutions provide excellent I/O throughput, high costs limit their deployment. In some application scenarios, an alternative that offers an excellent price/performance ratio is scavenged storage: namely, systems that opportunistically aggregate idle storage space available on desktops or other compute nodes available in organizations and present it as a unified, high-performance data store. In addition to aggregating idle storage space, scavenged storage systems typically stripe data among multiple nodes to provide a ‘fatter’ I/O channel, and use

replication or erasure codes to increase data availability and durability., data and metadata management are usually decoupled to improve scalability and performance by keeping metadata management out of the critical I/O path.

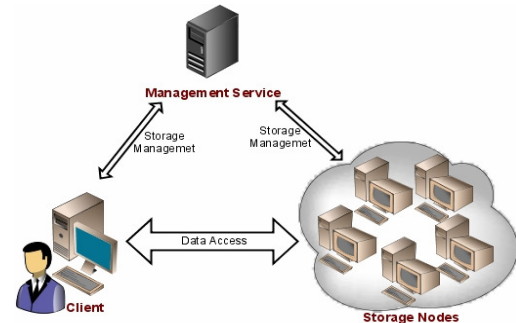


Figure 1. A typical scavenging storage system configuration

In general, a scavenged storage system consists of three main components: a metadata service, a set of storage nodes and the clients that access the system. Figure 1 presents one possible configuration in which each of these components runs on different, sometimes dedicated, nodes. In other configurations, the same node may play more than one role, e.g., both client and storage node at the same time. Several distributed storage systems employ this generic architecture (e.g., OceanStore [1], GoogleFS [2], PVFS [3], stdchk [4], Lustre [5], Ceph [6], FreeLoader [7], and Farsite [8]) that offers excellent performance and sufficient parallelism support. However, security in such systems is a key issue and different systems assume different deployment environments that drive their security requirements. We identify these deployment environments as: (a) *completely trusted*, (b) *partially trusted*, and (c) *untrusted*.

In a *completely trusted environment* all three system components and the communication channels among them are trusted. Generally these systems are optimized for performance and reliability and are hosted on dedicated clusters. Examples include GoogleFS [2], PVFS [3] and Lustre [5].

In an *untrusted environment*, all system components and communication channels are untrusted. Storage systems that operate in this environment aim to provide a storage service that can be deployed over wide area networks. Storage appliances like OceanStore [1], Farsite [8] and round-trip privacy with NFSv4 [9] fall in this group.

Finally, a *partially trusted environment* includes a combination of trusted and untrusted components. Systems that assume such an environment usually operate entirely or in part within a single administrative domain where a specific level of trust can be assumed. Ceph [6], for example, assumes a trusted metadata service and storage nodes and untrusted clients and communication channels. FreeLoader [7] extends Ceph’s set of untrusted components to include the storage nodes. Plutus [10]

assumes trusted client machines and untrusted storage nodes. These systems attempt to get acceptable performance while addressing specific threats perceived in the target deployment environment.

This work makes two contributions. First, we develop a threat model for scavenged storage systems operating in a partially trusted environment similar to that assumed by FreeLoader [7]. We assume that only the management service is trusted while the clients, the storage nodes and the communication channels are not. Second, we design a protocol to protect against the four attacks we identify: confidentiality, integrity, availability and authentication attacks. We evaluate the performance of our proof-of-concept implementation and the overheads related to different security features. Although a naïve implementation of the protocol shows severe performance degradation, we show that intelligent caching techniques can reduce this performance degradation while preserving the protocol’s security characteristics. Our prototype shows less than 4% performance degradation when enabling the authentication and integrity features, and 16% degradation when employing the full solution.

## 2. THE THREAT MODEL

This section identifies a set of possible attacks that can be mounted against scavenged storage systems in a partially trusted environment. The resulting threat model is developed in terms of four main attacks: confidentiality, integrity, authentication, and storage service availability. While we focus on threat modeling, we briefly provide references to possible protection techniques.

### 2.1 Confidentiality Attacks

Confidentiality attacks expose confidential data to unauthorized readers. There are two possible ways to compromise confidentiality: sniffing network traffic on untrusted channels and attacks on data stored on participating storage nodes.

Although resulting in high overheads, the common solution to address these attacks is to use encryption to secure the communication channels (e.g., TLS [RFC2246]) and stored data. (Note that if data is encrypted at the application level and stored encrypted only the system’s control channels need to be secured.)

### 2.2 Integrity Attacks

Integrity attacks attempt to modify information in the system (during transmission or once stored) without proper authorization. A possible integrity attack, common to all distributed systems, is a man-in-the-middle attack where the attacker places itself between two components in the system and relays messages. For example, an attacker can intercept request messages sent by a client to the manager and change their parameters. A second type of integrity attacks are data modifications at storage nodes (e.g., a malicious user with write access to the scavenged storage).

Cryptographic checksums (e.g., Message Integrity Codes) can be used to achieve message integrity. TLS implicitly employs such techniques to maintain channel integrity. Additionally, clients can compute per-block checksums and store them at the trusted manager to enable integrity verification for stored data.

## 2.3 Availability Attacks

Availability attacks (or Denial of Service attacks) attempt to make system services unavailable or slower to legitimate users for a period of time. Common denial of service attacks include consuming finite system resources (e.g. exhausting the file identifier space), disrupting system state information such as an attacker introducing inconsistency between the state at the manager and the state of storage nodes, and disrupting system configuration such as a malicious client trying to lower system’s performance by exhausting the disk space of a specific set of storage nodes to reduce parallelism.

The first attack above is difficult to defend against as it takes the form of a legitimate content with bad intent; however, system-level quotas can be used to mitigate its impact. The second one can be handled by employing protocols that require clients to prove that they have updated the state on storage nodes before changing the state at the manager accordingly. Finally, enforcing quotas per storage node can be used to cope with the last attack.

## 2.4 Authentication Attacks

Authentication attacks involve an attacker that masquerades as a legitimate end-user. Offline/online dictionary attacks as well as replay attacks fall in this category. In an offline dictionary attack the adversary records a successful authentication session then goes offline and tests passwords/keys against the recorded exchange without contacting the server. In online dictionary attacks, the adversary tries online possible passwords/keys. Finally, in replay attacks legitimate messages are recorded then replayed to trick the system into unauthorized operations.

Directly addressing authentication attacks beyond employing widely used authentication solutions (e.g., Kerberos [11] or TLS with X.509 certificates) is beyond the scope of this work.

## 3. THE SECURITY PROTOCOL

This section introduces our security protocol in the context of a scavenged storage system that assumes a partially trusted environment.

### 3.1 Storage System Overview

Starting from the original FreeLoader [7] scavenged storage system, we aim to build MosaStore, a highly configurable storage system that can operate in a variety of deployment environments (e.g., from desktop grids to clusters), to provide maximum performance for a variety of workloads (e.g., checkpointing, access to scientific datasets) and security requirements.

Our system follows the generic scavenged storage system architecture presented in Figure 1: a management service, a set of storage donors (benefactors), and a set of clients that access the system. Files are divided into chunks distributed among the benefactors. The manager maintains storage system metadata: available storage space, system’s namespace, file attributes, mappings from files to chunks and from chunks to benefactors. On the client side, we provide a traditional file system interface via a user-level implementation using FUSE Linux module [12].

The storage system design follows three guidelines the security system should also follow: component decoupling, lazy interaction, and statelessness. First, components are decoupled to

the extent possible (e.g., clients never read/write data through the manager) to improve scalability and reliability. Second, to enable high-performance, components interactions are moved outside the critical I/O path (e.g., benefactors and the management service never interact on the critical I/O path). Third, the system is stateless as much as possible in order to minimize failure effects and recovery overhead.

A write operation is performed as follows: First, the client asks the manager for a set of benefactors that may provide the required storage space. The manager replies with a set of candidate benefactors (selected according to system's striping policy). This step is usually performed per group of files rather than per file. Next, for each new file, the client requests a file handle (a numerical identifier that uniquely identifies each file) from the manager. Once this is received, the client divides the file into chunks and pushes them to the benefactors. Finally, the client puts together the chunkmap, i.e., the list that maps chunks to benefactors, and commits it to the manager which updates the file's metadata and the system's available storage space status.

A read operation is similar: the client asks the manager for the file's chunkmap. Once this is obtained, the client pulls the data from the benefactors directly.

### 3.2 Assumptions

Before discussing security requirements, we first introduce our design assumptions: First, we assume that the manager is trusted: deployed on a trusted machine and using a trusted code-base. Second, the system is not responsible for data confidentiality, i.e. data will be stored unencrypted at the benefactors (however, higher-level applications that require confidentiality can always encrypt the data before pushing it to storage). Third, since this is an open source project, we assume that an adversary can modify and deploy a client or a benefactor for malicious purposes. Forth, a malicious adversary can spoof messages. The first two assumptions result in a good tradeoff between simplicity and performance; while the last two are inherent in open systems.

### 3.3 Design Requirements

MosaStore's security requirements can be grouped into two main categories:

*Requirements related to security services:*

1. Authentication and authorization: Clients should be authenticated and authorized to access system resources.
2. Data integrity: A client should be able to verify that data stored at the benefactors is consistent and intact.
3. Accountability: The system should be able to assign blame in case of inconsistency, i.e., the system should be able to verify responsibility for corrupted data: a client or a benefactor.

*Requirements related to system characteristics:*

4. Acceptable performance degradation: Security overheads must be minimized to preserve system's high performance characteristics and its scalability.
5. Usability: First, the system should support access to stored data through a traditional POSIX file system API. Second, the set of security features provided by the system should be configurable to offer, in each deployment environment, the best choice between overheads and security levels.

6. Easy integration with Grid environments: We aim to facilitate MosaStore's use in Grid deployments as a high performance storage resource.

### 3.4 The Supporting Mechanism

Global Security Services API (GSS-API) [RFCs 2743, 2744] provides a mechanism-agnostic API that enables application portability across deployment environments using different authentication solutions. GSS-API provides access to security services (authentication and channel protection) and defines services and primitives at a level independent of the underlying security mechanism and programming language. Kerberos [11] or TLS with X.509 certificates can be used to support the GSS-API. GSS-API use involves four steps [13] (Figure 2):

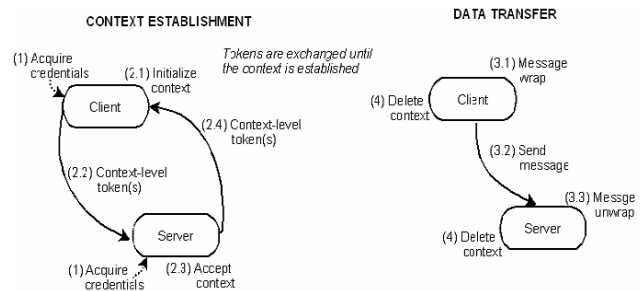


Figure 2. GSS-API protocol

*Context Establishment Phase* (steps 1 and 2 in Figure 2): First, Each communicating process acquires a set of credentials (e.g., a X.509 certificate or a Kerberos ticket) with which it may prove its identity to other processes. After this, two processes use these credentials to authenticate to each other and establish a security context that includes a shared symmetric session key. Authentication can be either one-way or two-way (mutual).

*Data Transfer Phase* (steps 3 and 4 in Figure 2): Once the context is established, processes can exchange messages with levels of protection (i.e., integrity and confidentiality) configured by the application. At the end of the communication session, both processes delete the security context.

GSS-API offers two advantages: First, the ability to port applications across environments that use different security mechanisms. Second, the Grid Security Infrastructure (GSI) [14], used by the Globus toolkit [15] (a toolkit to build computing grids), adopts the GSS-API as a standard interface to access its security services; hence, using GSS-API will make it easier to present MosaStore as a storage resource in Globus deployments.

### 3.5 Security Design

We use GSS-API to address three the security requirements: authentication, message integrity and confidentiality (if not application level data is not stored encrypted). We assume that all parties have certificates issued by a common trusted certificate authority, a reasonable assumption since most Grid deployments already have certificate authorities in place.

The other requirements presented in Section 3.3 are addressed by the MosaStore protocol presented below and in Figure 3 in the context of a write operation (The read protocol does not differ significantly from the write protocol. We omit its description for the sake of brevity). The steps are:

1. *Authentication.* The write operation starts with mutual authentication between the client and the manager. At the end of this step, a security context is established and all messages further exchanged between the client and the manager are protected within this context, achieving transmission integrity and confidentiality depending on configuration.
2. *Space reservation.* The client sends a request to the manager asking for free space from the benefactor pool. Note that this space is not related to a specific file (space reservation is typically done per group of files). The manager replies with a set of benefactors, along side a signed authorization ticket that contains the client's name and a timestamp that limits the validity of the ticket (we assume loosely synchronized clocks).
3. *Obtaining a fileID.* For each new file, the client asks the manager for a handle that will uniquely identify the file.
4. *Authentication with storage nodes.* The client mutually authenticates with the target benefactors and establishes a security context with each. Subsequent messages are protected within this context.
5. *Ticket verification.* The client sends the authorization ticket previously obtained to all benefactors selected to store the file. Each benefactor verifies the ticket validity (issuer, expiration time, and that the client is the ticket owner). Once successfully done, a positive acknowledgment is sent by each benefactor to the client (otherwise, the write fails).
6. *Data transfer.* The client divides the data into chunks, computes and stores locally a hash value for each chunk, signs each chunk, and pushes the chunks and their signatures to the benefactors. For each received chunk, the benefactor verifies the client's signature on the chunk and sends back a signed chunk-receipt (containing the benefactor's name, the chunk's hash, and the corresponding fileID) to the client. The client in turn verifies that the receipt corresponds to the original chunk sent by comparing the receipt chunk-hash and the local hash.
7. *Metadata commit.* After pushing all chunks, the client creates a chunkmap for the file. The chunkmap, together with the fileID and the intended file path, are sent to the manager to commit the write operation. The manager consults the access control policy module to check the client's permissions to write to the specified path. If authorized, the chunkmap is checked for consistency by verifying the signatures of each chunk receipt and that all chunk receipts belong to the same fileID. If successfully verified, the chunkmap is stored as part of the file's metadata. If not, the manager rejects the commit.

## 4. EVALUATION

### 4.1 Security Analysis

The protocol provides authentication, authorization, data transport and storage integrity, accountability and usability. Also, the system addresses a number of possible DoS attacks.

*Authentication* is achieved by requiring any two communicating entities to mutually authenticate before starting a communication session. The details of the authentication protocol depend on the mechanism used to support GSS-API. We experimented with an implementation based on TLS protocol and X.509 certificates.

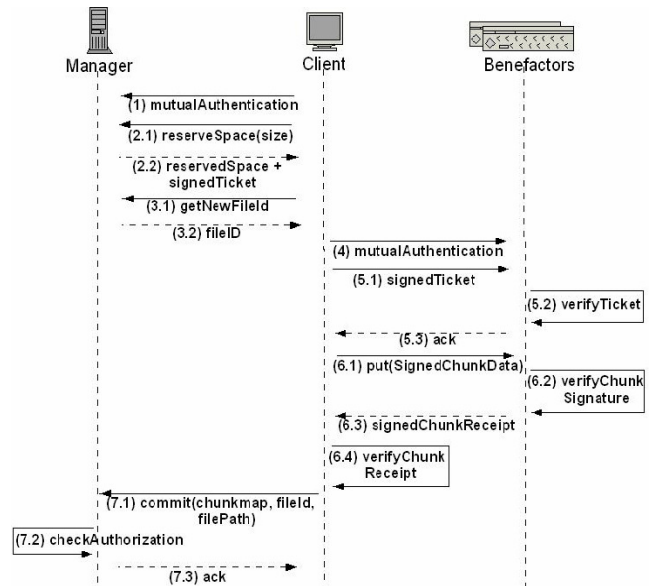


Figure 3. MosaStore secure write protocol

*Authorization* is provided by an independent access control module consulted by the manager. Once the manager obtains a positive authorization decision it allows access to locally stored metadata and, when needed, it provides the client with a ticket that allows clients to prove to benefactors their access rights. Our protocol does not assume any specific implementation for the access control module which can be represented as an ACL or an external authorization service such as Globus' Community Authorization Service [16].

*Transport integrity* is obtained by sending all traffic within the security context established using GSS-API mechanisms.

*Stored data integrity* is obtained by having the manager maintain chunk hashes received as part of the chunk receipt in the write commit phase. Therefore, a reading client can verify integrity of a chunk provided by a benefactor by checking its hash against the hash provided by the manager. Additionally, our system can be configured as a content addressable file system and naming chunks by the hash of their content (as in SFS [17]) thus providing the integrity verification information implicitly.

*Accountability*, that is, the ability to determine and prove to a third party who is guilty of data integrity violations, is obtained using chunk receipts and chunk hashes verifications. A chunk receipt represents a proof that a benefactor has accepted a specific chunk. Note that since neither clients nor benefactors are trusted, either party may maliciously accuse the other of integrity violations. If a benefactor is malicious, and maliciously alters the stored data, a client can use the chunk receipt to prove that the data has been maliciously altered. On the other side, a potentially malicious client can not falsely accuse a benefactor since: first, it is infeasible to fake a chunk receipt; second, benefactors maintain the clients' signature for each received chunk.

From *usability* standpoint, the additional security layer preserves traditional file system usability features. The only added new requirement is that all system components (manager, benefactors, and clients) have valid certificates and trust arrangements – a realistic assumption in the deployment environments we target, Grid environments.

*Availability.* Our design relies on two MosaStore features to minimize the effect of availability (DoS) attacks. First, a lazy garbage collection protocol that serves to synchronize the manager metadata with the benefactors' state. At each benefactor, the garbage collection process runs periodically: it informs the manager of the chunks stored locally and the manager replies with the set of chunks that need to be garbage collected (e.g., because files have been deleted or replication constraints relaxed). This lazy garbage collection mechanism maintains the eventual consistency of the system and preserves scalability as it reduces the number of components on the critical I/O path. The garbage collection mechanism, as well as the fact that write authorizations are time limited, bounds the ability of malicious clients to mount DoS attack by continuously dumping data on benefactors. Similarly, the fact that the manager decides on the set of benefactors when issuing write authorizations reduces the ability of malicious clients from targeting a specific group of benefactors to reduce the parallelism of the system. If a malicious client chooses to use only one or a few benefactors to write data to the manager will be able to detect it at commit time.

## 4.2 Performance Evaluation

We evaluate our prototype on a cluster of 2.33GHz Intel Xeon Quad-core CPU, 4GB memory nodes connected at 1Gbps. For all experiments we report averages and standard deviations over 15 runs. Except where explicitly mentioned, we use 1GB files split into 1MB chunks, one client and eight benefactors.

To quantify the performance impact for each of the security features (authentication, data integrity, accountability), we evaluate the write throughput while incrementally enabling each security feature. In all our tests below we compare three different security configurations: (i) authentication; (ii) authentication and integrity; (iii) authentication, integrity and accountability.

Figure 4 presents the achieved write throughput for various security configurations. The white bars (left) in Figure 4 demonstrate significant performance degradation (about 70%) when security is enabled. The culprit is the expensive mutual authentication operation, especially between the client and the benefactors, which incorporates a large number of costly public key cryptography operations and three additional round trips. The experiment demonstrates that the authentication overhead dominates the other overheads of the three security features rendering them to present close performance.

Our solution to address this problem is to cache the security context established between any two entities after the first mutual authentication. The caching period can be set to an absolute value (e.g., half an hour) or for the duration of streaming a single file (as in all experiments presented here). This optimization reduces the number of redundant authentications while maintaining the protocol's characteristics. Figure 4 (black bars) presents the effect of caching: up to three fold throughput improvement when security contexts are cached.

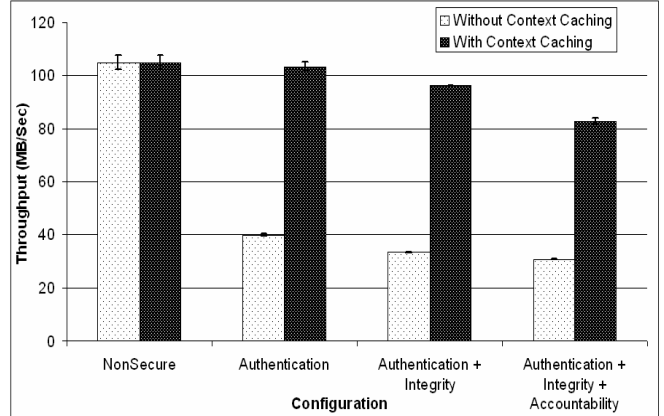


Figure 4. Achieved write throughput (average and stdev).

## 4.3 Effect of File Size

Figure 5 shows the effect of file size on the write throughput. The bars show the percentage throughput compared to the non-secure version, while the line shows the raw throughput for the non-secure version. When the number of chunks that needs to be stored in each benefactor becomes large, the relative performance becomes much lower (about 50% for a 128MB file size). Once the non-secure version saturates, all other configurations start to close the throughput gap on the expense of more CPU utilization.

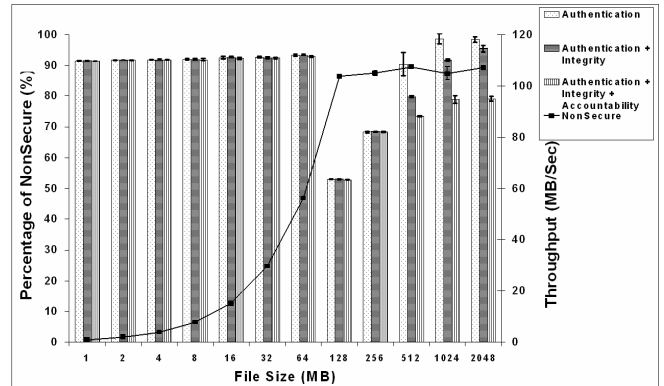
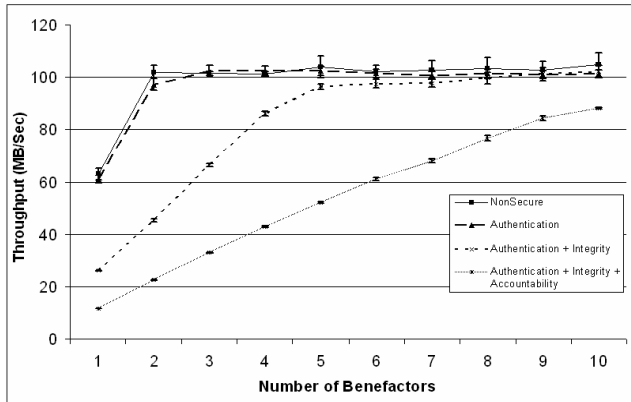


Figure 5. Average achieved write throughput while varying the file size. The line (right axis) shows the raw throughput of the non-secure version. Bars (left axis) show the percentage throughput achieved compared to the non-secure version.

## 4.4 System Scalability

We evaluate the ability of the system to efficiently incorporate additional capacity. Figure 6 presents write throughput when varying the number of benefactors. After adding the second benefactor, the achieved throughput for the non-secure version saturates as the client is bottlenecked by its network card (1Gbps). When security is enabled, throughput increases with the number of benefactors, and a system with ten benefactors achieves less than 16% performance degradation when all security features are enabled, and around 4% degradation when only authentication is used.



**Figure 6. Average achieved write throughput while varying the number of benefactors.**

## 5. RELATED WORK

Section 1 briefly categorized scavenged storage systems according to one differentiating factor: the trust assumptions in the deployment environment. While space constraint prevent us from deeply expand this classification, we mention two systems that are closely related to this work.

Gobioff [18] proposes a basic cryptographic capability system that meets the requirements of distributed storage systems architected similarly to our system. Gobioff's design enables the management service to control access to storage nodes asynchronously: the trusted manager enforces security policies while outside the critical IO path. While we have a similar design, we assume untrusted storage nodes; also, we use public key security, which simplifies key management considerably.

Leung and Miller [19] introduce a coarse-grained capability-based security protocol for object-based storage systems (e.g., Ceph [6]). While this work evaluates the performance and scalability of the proposed protocol, it assumes trusted storage nodes. Our protocol relaxes this assumption and uses a general security services layer (GSS-API) to enable easy portability across different deployment environments.

## 6. DISCUSSION

1.) *Advantages of our design.* Our design offers a number of advantages. First it preserves system's scalability and high-performance through component decoupling, lazy interaction, and use of stateless components where possible. Second the design allows enabling security features incrementally, to address the concerns of each specific deployment. Finally, we use standard security services offered by GSS-API enabling system portability.

2.) *Preserving accountability when replicating chunks among benefactors.* MosaStore uses data replication to improve availability and performance. To maintain accountability when replicating chunks, the replication protocol uses the chunk receipts stored at the manager. To trigger replication, the manager first sends a signed 'replicate' command to benefactor A the target for the new chunk replica. The 'replicate' command contains the source benefactor B and the corresponding chunk receipt previously stored at the manager. Second, A copies the chunk from B and verifies it against the receipt provided. Finally, A

generates a chunk receipt and sends it back to the manager thus assuming responsibility for the chunk.

## 7. SUMMARY

This paper presents a threat model for scavenged storage systems that operate in a partially trusted environment where the only trusted component is the management service. We designed a secure protocol that addresses the attacks presented in the threat model, implemented the protocol in the context of MosaStore storage system, and evaluated it for different security levels. Our protocol presents low performance degradation in small deployments, and close to original performance in larger deployments that offer more parallelism demonstrating that scavenged storage systems can offer security without compromising performance or scalability.

## 8. ACKNOWLEDGEMENTS

We thank Konstantin Beznosov, Sudharshan Vazhkudai and the anonymous reviewers for their valuable comments on this paper.

## 9. REFERENCES

1. Kubiawicz, J., et al. *OceanStore: An Architecture for Global-Scale Persistent Storage*. ASPLOS. 2000.
2. Ghemawat, S., H. Gobioff, and S.-T. Leung. *The Google File System*. in *SOSP'03*. 2003. Lake George, NY.
3. Carns, P.H., et al. *PVFS: A Parallel File System for Linux Clusters*. in *4th Annual Linux Showcase and Conference*. 2000. Atlanta, GA.
4. Al-Kiswany, S., et al. *stdchk: A Checkpoint Storage System for Desktop Grid Computing*. ICDCS. 2008. Beijing, China.
5. *Lustre*, <http://www.lustre.org/>. 2007.
6. Sage Weil, S.A.B., Ethan L. Miller, Darrell D. E. Long, Carlos Maltzahn. *Ceph: A Scalable, High-Performance Distributed File System*. OSDI. 2006.
7. Vazhkudai, S.S., et al., *Constructing collaborative desktop storage caches for large scientific datasets*. ACM Transaction on Storage (TOS), 2006. 2(3): p. 221 - 254.
8. Bolosky, W.J., et al. *Feasibility of a Serverless Distributed File System Deployed on an Existing Set of Desktop PCs*. SIGMETRICS. 2000.
9. Traeger, A., K. Thangavelu, and E. Zadok. *Round-trip privacy with nfsv4*. in *ACM Workshop on Storage Security and Survivability (StorageSS)*. 2007. Alexandria, Virginia.
10. Mahesh Kallahalla, E.R., Ram Swaminathan, Qian Wang, Kevin Fu. *Plutus: Scalable Secure File Sharing on Untrusted Storage*. FAST. 2003.
11. Ts'o, C.N.a.T., *Kerberos: an authentication service for computer networks*. IEEE Communications Magazine, 1994. 32(9): p. 33-38.
12. *FUSE, Filesystem in Userspace*, <http://fuse.sourceforge.net/>. 2007.
13. *The GNU Generic Security Services Library*: <http://www.gnu.org/software/gss/manual/>. 2008.
14. Butler, R., et al., *A National-Scale Authentication Infrastructure*. IEEE Computer, 2000. 33(12): p. 60-66.
15. Foster, I. and C. Kesselman, *Globus: A Metacomputing Infrastructure Toolkit*. International Journal of Supercomputer Applications, 1997. 11(2): p. 115-128.

16. Pearlman, L., et al. *A Community Authorization Service for Group Collaboration*. in *IEEE 3rd International Workshop on Policies for Distributed Systems and Networks*. 2002.
17. Gifford, D.K., et al. *Semantic file systems*. in *CM SIGOPS Operating Systems*. 1991.
18. Gobiuff, H.B., *Security for a High Performance Commodity Storage Subsystem*. 1999, Carnegie Mellon University.
19. Leung, A.W. and E.L. Miller. *Scalable security for large, high performance storage systems* *ACM Workshop on Storage Security and Survivability*. 2006