Spring 3-27-2010

DSFS: Decentralized Security for Large Parallel File Systems

Zhongying Niu
Huazhong University of Science and Technology, niuzhy@gmail.com

Hong Jiang
University of Nebraska-Lincoln, jiang@cse.unl.edu

Ke Zhou
Huazhong University of Science and Technology, k.zhou@hust.edu.cn

Dan Feng
Huazhong University of Science and Technology, dfeng@hust.edu.cn

Tianming Yang
Huazhong University of Science and Technology, ytmzqyy@yahoo.cn

See next page for additional authors

DSFS: Decentralized Security for Large Parallel File Systems

Zhongying Niu†, Hong Jiang‡, Ke Zhou†, Dan Feng†
Tianming Yang†, Dongliang Lei†, Anli Chen†
†*College of Computer Science & Technology, Huazhong University of Science & Technology*
Wuhan National Laboratory for Optoelectronics, China
‡*Department of Computer Science & Engineering, University of Nebraska-Lincoln, USA*
Email: niuzhy@gmail.com, jiang@cse.unl.edu, {k.zhou,dfeng}@hust.edu.cn

Abstract

This paper describes DSFS, a decentralized security system for large parallel file system. DSFS stores global access control lists (ACLs) in a centralized decision-making server and pushes pre-authorization lists (PALs) into storage devices. Thus DSFS allows users to flexibly set any access control policy for the global ACL or even change the global ACL system without having to upgrade the security code in their storage devices. With pre-authorization lists, DSFS enables a network-attached storage device to immediately authorize I/O, instead of demanding a client to acquire an authorization from a centralized authorization server at a crucial time. The client needs to acquire only an identity key from an authentication server to access any devices she wants. Experimental results show that DSFS achieves higher performance and scalability than traditional capability-based security protocols.

1. Introduction

Large-scale and high-performance storage systems have gained increasing importance in science, engineering and business applications. High-performance computing (HPC) applications of today and tomorrow, such as High-Energy Physics, Biosciences, Astrophysics, and Geophysics call for high-performance data access and Petabytes to Extabytes of data storage. Large business services such as Google and Yahoo! may service millions of users and store tens to hundreds of petabytes of data, a storage capacity that may soon reach the Exabyte scale. Securing such large-scale and high-performance storage systems is an important challenge because of the highly parallel and dense data accesses from a large number of users and the massive amounts of personal and sensitive data stored on them.

Unfortunately, the design concept of existing large-scale storage systems makes their security solutions more challenging and urgent. Recent studies [1–6] on large, high-performance storage systems have enabled direct interaction between clients and storage devices. A key concept behind this design is the decoupling of metadata and data paths. The client's data is typically striped across multiple devices to achieve high parallelism. Before accessing the devices, the clients acquire the necessary metadata information from a metadata server (MDS), and then directly interact with the storage devices. MDS is thus completely bypassed during the data transfer phase, which improves the performance and scalability of the system. However, attaching storage devices to the client-network renders these devices vulnerable to various network attacks, such as eavesdropping, masquerading and replaying. The storage devices must be intelligent to authenticate users and restrict illegal accesses rather than connected to and protected by an individual server. However, separating metadata from data has resulted in the loss of implicit knowledge of access privileges or authorizations at the storage devices because the information is now stored at MDS.

To secure I/O, MDS must communicate authorizations to storage devices. In the existing security schemes [7–13], a client wishing to access storage devices must acquire a capability authorizing the I/O from MDS and then present it to the storage device with the I/O request. However, when incorporated into large parallel file systems, these solutions based on capabilities either degrade performance or strongly depend on the workload and application environment, thus limiting the scope of these approaches. For example, a capability-based scheme may issue a capability for every block or object being accessed, i.e., a capability authorizes a single block or object I/O [9]. This requires the generation of tens or even hundreds of millions of capabilities, which imposes a substantial overhead on the authorization server and does not scale well in large-scale storage systems [11]. To reduce the number of capabilities, coarse-grained access control [10] allows a capability to authorize I/O to a set of objects, but constrains the granularity of access control. Maat [11], a scalable security protocol developed in Ceph [6], employs an extended capability to authorize I/O for any number of clients to any number of files, thus significantly decreasing the number of capabilities that MDS and OSD (object-based storage device) need to generate

and verify. However, the strategy in Maat to group multiple I/O authorizations into extended capabilities is strongly dependent on the workload and application environment, thus limiting the scope of this approach.

This paper introduces DSFS, a **d**ecentralized **s**ecurity system for large parallel **f**ile **s**ystem, designed to provide decentralized access control by pushing access control decisions into storage devices. Our goal for the DSFS system is to enable a network-attached storage device to immediately authorize I/O and benefit from any high-level access control policies without imposing any restrictions on data placement and management in existing parallel file systems.

The salient feature of DSFS lies in the fact that it stores a global access control list (ACL) in a centralized decision-making server, i.e., MDS, and pushes the server's access control decisions into storage devices. The storage devices can thus use the pre-stored decisions, in the form of local pre-authorization list (PAL), to make authorization decisions. The advantage of doing this is that (a) DSFS allows users to flexibly set any policy for the global ACL or even change the global ACL system but without impacting on the implementation of local PALs on storage devices, because all the polices or changes will be pre-interpreted to a simple authorization list that can be stored as the accessed object's security attributes on storage devices. (b) With pre-stored authorization decisions, DSFS enables a network-attached storage device to immediately authorize I/O, instead of demanding a client to acquire a capability from MDS for each object she wants to access. This is a more scalable solution as the cryptographic cost for MDS to generate capabilities has been removed from the critical I/O path. Similarly to the SCARED authentication approach [12], the client can thus use an identity key obtained from a centralized authentication server to access any devices, including MDS, she wishes to access.

We demonstrate and prove the DSFS concept on an object-based storage system (OBS) and implement a DSFS prototype in the HUST OSD project [14] that complies to the T10 standard [13]. Our implementation requires minimal changes to the current standard, which includes only an extended security attribute page and a collection object required, but enables the standard to support decentralized access control. We evaluate DSFS through trace-driven experiments and high-bandwidth benchmarks and compare it with the T10 OSD-2 security protocol and the state-of-the-art Maat security protocol. Experimental results show that DSFS achieves significantly higher performance and scalability than the T10 OSD-2 and Maat security protocols. To achieve the same performance as DSFS does, the latter two must make the capability cache hit rate approach to an impractical 100%.

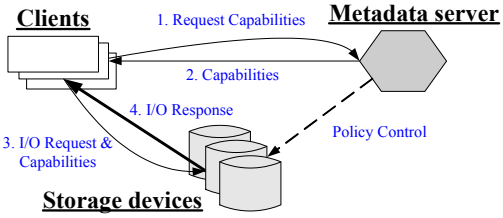


Figure 1. Parallel file system architecture and security flow. Clients request capabilities from MDS and use them to request I/O from storage devices.

The rest of this paper is organized as follows. Background and related work are presented in Section 2. Section 3 describes the DSFS architecture. Section 4 details the design and implementation of DSFS on an object-based storage system. Section 5 evaluates the effectiveness of DSFS through trace-driven experiments and high-bandwidth benchmarks on the DSFS prototype. Finally, Section 6 concludes the paper.

2. Background and Related Work

2.1. Decentralized vs. Centralized Access Control

Most of parallel file systems [1–6] that have been developed recently consist of three main components: clients, a metadata server cluster (MDS), and a cluster of storage devices, such as network-attached disks or object-based storage devices (OSD). Decentralized access control refers to access controls where ACLs are stored and users are authenticated and authorized at the storage devices, as opposed to centralized access control where ACLs are stored and users are authenticated at a centralized metadata server, where the users are then granted capabilities.

The main idea behind the centralized access control scheme is that maintaining ACLs at a centralized server makes modification to ACLs easier and reduces the complexity on storage devices. The storage devices can thus have an optimized simple security check on the capability. They can dedicate themselves to busy data moving and are oblivious to how users are authenticated and ACLs are implemented. Figure 1 shows the architecture and security flow in most parallel file systems. Clients request capabilities from the MDS and use them to request I/O from the storage devices. The capabilities are cryptographically hardened with a digital signature or HMAC [15] through the use of MDS's public key or the shared keys between MDS and storage devices respectively. In most cases, capabilities can be attained at a negligible cryptographic cost at the same time when the client gets the metadata with object location information. However, for large files striped across hundreds or even thousands of devices,

a simple access to a single file would require MDS to generate hundreds or even thousands of capabilities. In addition, for security purposes, the capability is usually specified a valid period as short as possible and the sharing keys between MDS and storage devices should be refreshed regularly. Once the capabilities being used are invalidated because of expiry or revocation, the client will have to incur an extra round trip to get new capabilities. Most capability-based schemes use cache to optimize the system performance. For example, MDS can cache pre-computed capabilities and clients can cache unexpired capabilities. But the efficiency of cached capabilities is strongly dependent on the workload and application environment.

The main idea behind the decentralized access control scheme is to remove the authorization process, i.e., capability generation and dispatch, from the critical I/O path through offloading the access privileges to storage devices. The storage devices can thus directly authenticate and authorize users without needing any capabilities, that is, without needing any capability cache and the cost of capability generation and verification in the critical I/O path. As a result, the same number of MDSs can service more storage devices, implying higher parallel performance. However, straightforwardly storing ACLs along with the accessed objects means that storage devices must be able to make authorization decisions according to object's ACLs and high-level access control policies, such as ACL inheritance, an important means to simplify management of ACLs in large hierarchical systems. It requires the storage devices to store and know the relationship between objects in a file system, which implies complex security implementation and a high cost of security check. This contradicts the design concept of parallel file systems in which the storage devices usually do not know how the object it holds fits into the file system. Therefore, ideal decentralized security solutions for parallel file systems should keep the existing system architecture and file placement and impose a minimal cost of security check on storage devices. Based on this design principle, the proposed DSFS in this paper is intended to take advantage of decentralized access control but without imposing any restrictions on the performance and file placement of parallel file systems.

2.2. Related Work

There have been numerous efforts to secure parallel and distributed storage systems [7–13, 16], most of which are based on capabilities. A fine-grained capability [7, 9] may authorize access privileges at the granularity of a block or object. In a large-scale system, a hotspot file containing terabytes of data may be accessed by thousands of clients. The system may stripe the file across hundreds or even thousands of devices to achieve data redundancy and highly parallel I/O. A single access to such a file would need hundreds or thousands of fine-grained capabilities. Especially, the I/O pattern in such systems is usually highly parallel and very bursty [17]; it is impractical to generate and return tens of thousands of capabilities in a timely manner. To cut down the number of capabilities in the system, NASD [9], SnapDragon [7], LWFS [10], and the T10 OSD security protocol [13] all allow a capability to grant access to a group of objects . But coarse-grained capabilities constrain the granularity of access control. Maat [11] employs extended capabilities to provide scalable I/O security for lager-scale parallel file systems. However, an effective grouping strategy to group multiple I/O authorizations into an extended capability in Maat implies a relatively large space and time penalty to store past behaviors and compute predictions. As a result, Maat's "authorization grouping strategy has a dramatic impact on the performance improvements gained by extended capabilities" [11].

SCARED [12] extends NASD [9] to provide mutual authentication between clients and devices. It supports authentication based on capabilities (as in NASD) as well as identity keys. In the latter case the SCARED device also stores ACLs along with the object. The SCARED identity key is usually derived from a shared secret key between the administrator and the devices. A client can use an identity key obtained from the administrator to access the SCARED device that then authorizes the request according to the local ACLs stored on it. The goal of the SCARED project was to develop a distributed serverless file system by enabling a device to store and manage the storage of file system data and metadata. This implies that the access control policy and enforcement of the whole file system must be built on SCARED devices, though in fact existing publicly available literatures [12, 18] do not detail how high-level access control policies are implemented on the SCARED devices. Similar to SCARED, other traditional distributed file systems, like AFS [19], SFS [20] and NFS V4 [21], store and manage the whole or portion of the file system data and metadata in file servers. They have different security requirements with existing parallel file systems.

Kher and Kim [16] use role-based access control (RBAC) in their system. They store role-based access control lists with each object on object-based storage devices. Since changes to role permissions (i.e., to the role-based access control list) are infrequent as compared to changes to role memberships, RBAC can reduce the complexity and cost of security administration in large parallel file systems. In most of the cases, a client needs to acquire only an identity key from the

file manager, which can be used by the client to further derive role keys. The device verifies the identity of the client by verifying the role key and then authorizes the request according to the role-based access control list. Unfortunately, Kher and Kim do not address the design and implementation issues of role-based access control lists, a key to developing decentralized access control in large parallel file systems.

3. DSFS Architecture

For DSFS we aim to be able to have many of the benefits of the decentralized access control without having any restrictions on data placement and management in existing parallel file systems. Our goal is to be able to allow users to set arbitrary access control policies without having to change the security code in their storage devices.

To achieve these goals, DSFS stores a global ACL at MDS, i.e., the centralized decision-making server. MDS makes access control decisions according to the global ACL and then pushes them into storage devices. The access control decisions are stored along with the accessed objects in the form of local pre-authorization lists (PALs) at storage devices. Next, the client authenticates herself to the authentication server that then issues her an identity key. Since the identity key is derived from the shared key between the authentication server and storage devices, the storage devices can verify the client's identity by verifying the identity key. With the client's identity and local PALs, the storage devices can determine whether to authorize the request. Figure 2 shows the DSFS's architecture and security flow. Since all access control decisions are made at MDS and the storage devices store only the final decisions, users can set arbitrary policies for the global ACL or even change the global ACL system without impacting on the implementation of local PALs. And because the final pre-authorization lists can be directly associated with the accessed objects as security attributes, there is no change to the data layout in the storage devices. Security check is also simple, that is, only a matching in PAL is required at the storage devices.

3.1. Authentication

DSFS' goal for authentication in parallel file systems is to make authentication fit into existing security infrastructure while keep the authentication process as simple as possible to ensure a high-performance I/O. To achieve this goal, DSFS separates authentication services from the file system. Like the SCARED authentication approach [12], the client is authenticated at a dedicated authentication server that then issues her an identity key for the subsequent access to the file system. The authentication server can authenticate clients independently or in collaboration with existing

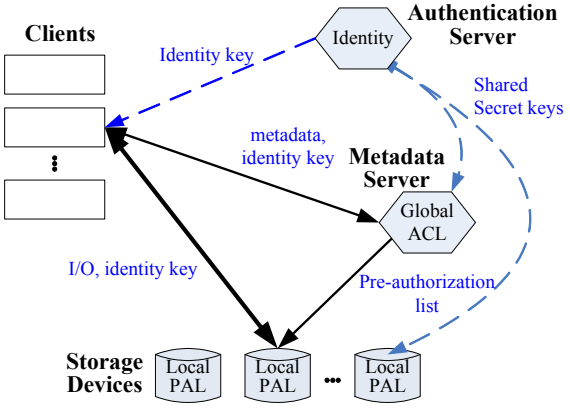


Figure 2. DSFS architecture and security flow. MDS stores a global ACL and pushes PALs into storage devices.

security infrastructure. In the former case, the server itself must store and maintain users' identity information, such as user identifiers and the user-to-role associations. In the latter case, for example, a server in collaboration with an X.509 certificate authority simply validates the client's certificate and then encodes the identities from the certificate into the identity key.

Access to a storage device is controlled using a single secret K, which is shared by the storage device and authentication server. An identity key can be generate as $idkey = MAC_K(K_{data})$, where MAC is a secure message authentication code such as HMAC [15]. K_{data} will be encoded as a concatenation of three values: $K_{data} = \{K_{id}, identity, expiration\}$. K_{id} is a unique identifier indicating the K used to generate the $idkey$. Multiple keys can be shared at the same time, thus refreshing a shared key would not impact the unexpired identity keys generated by other shared keys. $identity$ indicating the user's identity is a concatenation of two values: $identity = \{U_{id}, R_{id}\}$, where U_{id} is the user identifier and R_{id} is the identifier of the role assumed by the user. Though a user can assume multiple roles in a practical application, we only encode one role into the identity key to reduce the cost of PAL check. $expiration$ is the time limit of the validity of the $idkey$.

The generated $idkey$ and K_{data} are sent to the user. Note that $idkey$ must be kept secret, but K_{data} is not considered secret. Once the user has $idkey$ and K_{data}, she sends K_{data} to the storage device. With the shared secret K and K_{data}, the device can calculate $idkey$, which is then used as a shared secret between the user and device. Besides establishing a shared secret, the derivation using K_{data} binds the information encoded in K_{data} to $idkey$. Thus the storage device can make access control decision based on the identities encoded in K_{data} and local PALs stored on it.

Authentication to MDS is almost the same as authentication to the storage device. The only difference is that

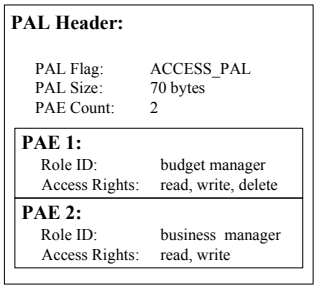


Figure 3. A sample PAL on an object with two entries (PAEs).

$idkey$ is derived from the shared key between the MDS and authentication server.

3.2. Authorization

DSFS stores pre-authorization lists along with each object to enable the storage device to immediately authorize I/O. A pre-authorization list is similar in form to the ACL in common use. A sample PAL with two pre-authorization entries (PAEs) is shown in Figure 3. A PAL is made up of a header and an arbitrary number of PAEs. A PAE defines access privileges of a user or a role. The former grants only one user access to the object with which the pre-authorization entry is associated, while the latter authorizes a group of users belonging to the role. Though user PAEs are supported, we prefer to use role PAEs for the whole authorization framework since the number of roles is usually less than the number of users and a change to the role's privileges is infrequent relative to a change to a user's privileges.

The PAL flag specifies the type of this PAL. Two types of PALs, namely, *access PAL* and *super PAL*, are proposed for two different purposes. The access PAL defines the usual access permissions of protected objects. The super PAL defines the permissions capable of being shared by a group of objects since in practice there will not likely be many different PALs when compared to the number of objects.

Figure 4 illustrates the pre-authorization model in a DSFS system. A pre-authorization is triggered when an object is created, or the permission is modified, or a PAL inconsistency occurs. In the case of object creation, the new object will be assigned a default permission, such as the object owner's permission, or automatically share a super PAL. In the cases of permission modification and PAL inconsistency, MDS makes new access control decisions and writes the latest pre-authorization list into the storage device. For example, assuming that the ACL for the file "/jim/foo" is C. The inherited ACL of her parent directory "/jim" is I. Then the object containing the data of "/jim/foo" will have the pre-authorization list $C|I$. To reduce the number of the occurrences of PAL inconsistency, MDS should ensure

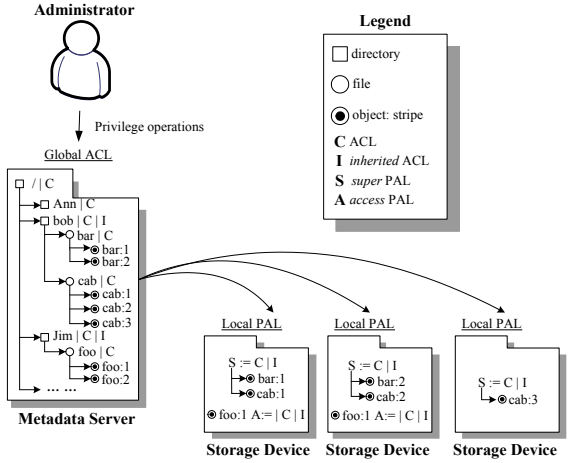


Figure 4. Pre-authorization model in a DSFS system

that the pre-authorization list is successfully written into the storage device. As the figure shows, a multiple-level inheritance model at MDS will be interpreted to a single-level sharing model on the storage devices.

3.3. Revocation

In addition to traditional revocation mechanisms built on certificates and ACLs, DSFS enables immediate revocation via invalidating an identity key. Revoking an identity key is possible in two ways. One method is *key expiration*. When giving an identity key to a client, the authentication server includes an expiration time in the key data. When the identity key held by a user expires, the user has to acquire a new key to declare its own identity. This approach works well when identities are used in a transient manner. For example, in order to authorize a temporary access to the system, the authentication server can define a short lifetime for the user's identity key. Another method is *key exchange*. By changing the shared key, all previous identity keys the authentication server had generated for a particular storage device are now invalid.

4. DSFS Design on OBS

As one of the most promising technological solutions to next-generation storage systems, object storage has received increasing attention in the past few years. This section details the design of DSFS on an object-based storage system, which is compliant with the current T10 OSD standard [13].

4.1. PAL Storage

DSFS stores global ACLs along with the metadata of file and directory in the form of database records at MDS. All changes to the global ACLs will be mapped to the database records. The existing policies applied to metadata can also be applied to the global ACLs. The local pre-authorization lists are stored as object

Table 1. The extended PAL attribute page

Attribute Number	Length (bytes)	Attribute	Client Settable	OSD Logical Unit Provided
0h	40	Page identification	No	Yes
1h	1	List flag	Yes	No
2h to FFh		Reserved	No	
0100h to BFFF FFFFh	13	User PAE	Yes	No
C000 0000h to FFFF FF00h	13	Role PAE	Yes	No
FFFF FF01h to FFFF FFFEh		Reserved	No	

Table 2. Collection type codes

Code	Name	Description
00h	LINKED	T10 specific
01h	TRACKING	T10 specific
02h	PAL	User objects may be added to or removed from the collection using the Collections attributes page.
03h to EEh		Reserved
EFh	SPONTANEOUS	T10 specific
F0h to FFh		Reserved

attributes along with each object at the storage device. Since many studies have addressed the storage and maintenance issues of metadata and for the sake of space constraint, this paper mainly describes the design and implementation of local PALs for object-based file systems (OBFS) at OSDs.

The T10 OSD standard stores and maintains object attributes in the form of attribute page. We propose an extended PAL attribute page that defines the fundamental PAL attributes and values shown in Table 1. An extended PAL attribute page includes a list of PAE attributes, including user PAE attribute (0100h to BFFF FFFFh) and role PAE attribute (C000 0000h to FFFF FF00h). The list flag attribute specifies the type of this PAL, *access PAL* or *super PAL*. An advantage of storing PALs in the form of attribute page is that the existing commands of setting attributes defined in the standard can be applied to PALs.

According to the T10 OSD standard, an OSD storage device contains one or more OSD logical units, each with four types of stored data objects: root object, partition, collection and user object. Each OSD logical unit has exactly one root object, as well as a variety of partitions. Each partition contains a set of collections and user objects. The user object contains end-user data (e.g., file or database data), while the collection is used for fast indexing of user objects. The standard requires each attribute page stored at OSD to be associated with an object. Access PAL page can be associated with user objects and provides access control to the latter. We propose an extended PAL collection shown in Table 2. Thus super PAL page can be associated with

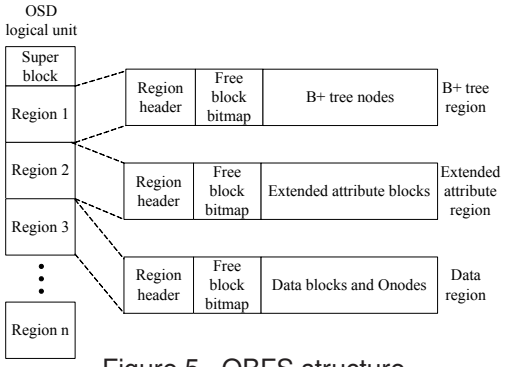


Figure 5. OBFS structure.

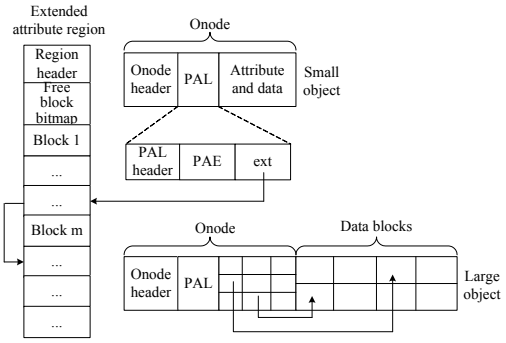


Figure 6. Object structure and PAL layout.

PAL collections. The user objects referenced by a PAL collection can share the super PAL associated with that PAL collection. To avoid confusion, a user object can be a member of one and only one PAL collection.

4.2. PAL Layout in OBFS

Since attributes in an OSD are organized in pages, and the same number and kind of pages are usually associated with the same kind of objects, we store these relatively fixed attribute pages in the forepart of objects to achieve fast accesses to attributes. To meet the requirement of storing a variable-length attribute page, such as the PAL attribute page, we use an *extended attribute* region to store an overrunning attribute.

As shown in Figure 5, the OBFS implementation divides an OSD logical unit into different regions for different functions. The super block records the characteristics of OBFS, including its size, the data block size, the empty and the filled data blocks and their respective counts, the size and location of the different regions, etc. The B+ tree region stores all B+ trees in OBFS. OBFS uses B+ trees to retrieve three types of data: 1) objects' onodes in data regions, 2) data blocks of large objects, and 3) free blocks in data regions. The extended attribute region stores the overrunning part of a variable-length attribute that fails to completely fit in a fixed-length attribute page. The data region stores objects' onodes and data. All of the blocks in a region have the same size, but the block sizes in different

regions may be different. Regions are initialized when there are insufficient free blocks in any initialized region to satisfy a write request. In this case, OBFS allocates a free region and initializes all of its blocks to the desired block size. In the current OBFS implementation, the block size in the B+ tree and data regions is set to 4KB, but the block size in the extended attribute region is set to 512bytes. Thus a block in the extended attribute region can hold about $512/13 = 39$ PAEs.

OBFS stores a PAL attribute page in an object' onode, which occupies a block in data regions. An onode, i.e., object metadata, is used to track the status of an object, similar to an inodes in a Linux file system, such as ext2 and ext3 file systems. Figure 6 shows an object structure and PAL layout. The data and attributes of a small object are directly stored in its onode, where the PAL attribute comes after the onode header. The data and attributes (except PALs) of a large object are stored in data blocks, which are referenced by the extended pointers in the object's onode. Each extended pointer is a two-tuple (address, length), which can specify any number of sequential blocks. The PAL in an onode is limited to 32 PAEs. Thus a PAL including the PAL header and extended pointer occupies $9 + 32 \times 13 + 8 = 433$ bytes in an onode, where the PAL header consists of the page number, the list flag and the number of PAEs in the PAL. The overrunning PAEs will be stored in the extended attribute region. A rational limit, such as 32 PAEs, enables an onode to hold all PAEs in most PALs, each of which can thus be read into memory with the onode in a lump.

5. Performance Evaluation

We ran extensive experiments to evaluate DSFS in three aspects: 1) DSFS's performance under trace-driven benchmarks and benefits relative to the T10 OSD-2 security protocol and the state-of-the-art Maat security protocol; 2) the overhead of security administration in DSFS and capability-based security storage systems; and 3) system throughput and scalability of different security setups under a high-bandwidth workload.

5.1. Prototype Implementation

We prototyped DSFS in the HUST OSD project [14]. The prototype stores global ACLs along with other metadata in a Berkeley database. The SET ATTRIBUTES command defined in the T10 standard can be used to set the pre-authorization lists. Apart from proving the identity of a client, the identity key can also be used to protect the OSD command from various network attacks, similar to the use of the capability key for securing an OSD command with the CMDRSP security method, one of the three security methods defined in the T10 OSD standard. We also implemented the Maat security protocol in our prototype system. With the initial OSD-2 security protocol implementation, our new OSD implementation supports three security schemes: DSFS, OSD-2 and Maat.

5.2. Experimental Setup

The experiments were conducted on 3 to 17 Linux hosts, each with one Intel Xeon 3.0 GHz processor and a total of 512MB physical memory (except MDS with 4GB physical memory). Each node was connected to a Highpoint Rocket 2240 Raid controller attached to 7 SATA disks (7200RPM, 300GB each). We used a separate SATA disk to house the operating system (Fedora Core 4, kernel version 2.6.12) and configured the other 6 disks into a RAID0 array to store data. All machines were connected by 1-Gbits Ethernet. In each experiment, one machine acted as MDS, while others acted as OSDs or clients.

In our experiments, we assumed for DSFS that the client had authenticated herself to the authentication server and gotten an identity key. For the Maat security protocol, we assumed that there has been a shared key between the client and OSD, which is used to protect the OSD command and fulfill the Maat security protocol. For the OSD-2 security protocol, the CMDRSP security method was used because this method provides the same security as DSFS does. In addition, we pre-stored 1 million records of metadata in the Berkeley DB for all experiments. We also assumed that the number of PAEs in a PAL is not more than 32, that is, the PAL can be stored into the object's onode. We identify the evaluation of overrun PALs as our future work.

5.3. Operation Latency

To determine the performance impact of the DSFS, Maat and OSD-2 security schemes, we timed open() operations at the MDS, and write() and read() operations at the client under several scenarios and compared these times to the times required in a non-secure system. The experiment was run on a collection of 512 files, each of size 4KB. Two kinds of system configurations are evaluated for the OSD-2 security protocol. One stored a file in a single OSD and another striped a file across 10 OSDs to achieve high parallel performance. We limited an I/O to be striped up to 10 OSDs, that is, MDS generates at most 10 fine-grained capabilities for an I/O request, since the experimental results on the Panasas file system [4] have discovered that striping a single I/O across more OSDs can cause the potential incast behavior on the network that results in aggregate bandwidth actually decreasing when increasing the number of OSDs past 14 in a single I/O. But it should be noted that limiting an I/O up to 10 OSDs does not mean a file can be striped across only 10 OSDs. In fact, the Panasas file system uses a two level

Table 3. MDS latencies of open() operations in microseconds (μs).

Latency (μs)	NonSecure	DSFS	OSD-2(1)		OSD-2(10)		Maat		
			cache miss	cache hit	cache miss	cache hit	cache miss	cache hit	renewal
open()	91	128	173	134	578	136	7369	132	50

Table 4. Latencies of write() and read() operations in microseconds (μs).

system	client		write()			read()		
	metadata	capability	MetadataGet	DataWrite	Total	MetadataGet	DataRead	Total
NonSecure	known	none	0	721	721	0	807	807
DSFS	known	none	0	942	942	0	1028	1028
OSD-2(1)	known	cache hit	0	940	940	0	1025	1025
OSD-2(10)	known	cache hit	0	940	940	0	1025	1025
Maat	known	valid hit	0	1016	1016	0	1102	1102
Maat	known	false hit	355	1022	1377	355	1107	1462
NonSecure	unknown	none	359	721	1080	359	807	1166
DSFS	unknown	none	433	942	1375	433	1028	1461
OSD-2(1)	unknown	cache miss	478	984	1462	478	1070	1548
OSD-2(10)	unknown	cache miss	886	984	1870	886	1070	1956
Maat	unknown	cache miss	7678	8222	15900	7678	8308	15986

striping pattern to stripe a file across all available OSDs but avoids massive retransmissions caused by an incast traffic pattern. This implies that the client in the Panasas file system has to acquire enough capabilities as the number of OSDs housing the data of the file when the file is opened, or cost an extra round trip to acquire another 10 capabilities from MDS when a parallel I/O is finished.

Table 3 lists the MDS latencies of open() operations in microseconds (μs). There are two cases, i.e., cache hit and miss, for the OSD-2 and Maat security protocols, since MDS can pre-computed a capability for the coming I/O request by predicting future file accesses. However, DSFS does not experience cache hit or miss because there is no any capability required in a DSFS system. When an open() request hits the cache at MDS, both OSD-2 and Maat result in costs comparable to DSFS. However, once a request misses the cache, the OSD-2 security protocol incurs an additional cost of $45\mu s$ for a 1-OSD system and $450\mu s$ for a 10-OSD system that respectively generate one and ten fine-grained capabilities. Requesting an extended capability in Maat is extremely slow, roughly 81 times slower than the non-secure system, 90% of which can be attributed to the capability generation and 9% to the database access for multiple metadata records. In this experiment, an extended capability grouped 8 clients and 8 files, thus needing $646\mu s$ to read 8 records of authorization information from the Berkeley DB. Maat uses a renewal protocol to extend the lifetime of expired capabilities, which incurs an additional penalty of $50\mu s$ at MDS.

Table 4 shows the latencies of write() and read() operations in microseconds (μs). For the Maat evaluation, we define a capability hit during its lifetime as a *valid hit* but a hit upon an expired capability as a *false hit*. Similar results to the open() operations of Table 3 can be observed in this table, that is, with a hit in OSD-2 and a valid hit in Maat, the latencies of OSD-2 and Maat are comparable to DSFS. When a false hit occurs, read and write operations in Maat run 1.91 and 1.81 times slower than the non-secure system. This is because the client usually has stored the location information of the file but must request a renewal token for the expired capability, which requires an additional round trip from the client to MDS. Whenever a cache miss occurs in a 1-OSD system, the OSD-2 security protocol is 1.35 and 1.33 times slower than the non-secure system for the write() and read() operations respectively, because of an additional cost associated with generating and verifying the fine-grained capability. In a 10-OSD system, the OSD-2 security protocol is 1.73 and 1.68 times slower because the client must request 10 fine-grained capabilities to open a file. With a cache miss in Maat, these two types of operations run respectively 15 and 14 times slower than the non-secure system, because the client must request a new extended capability from MDS, implying a high cost to generate and verify the extended capability on MDS and OSD respectively.

These numbers demonstrate that DSFS achieves lower operation latencies than the T10 OSD-2 and Maat security protocols. Both OSD-2 and Maat need to maximize capability cache hits to achieve the same performance.

5.4. Trace-based Evaluation

In this section, we further examine the performance and scalability of DSFS against the OSD-2 and Maat security protocols by using two publicly available traces: YouTube [22] and CAMPUS [23]. The YouTube trace is collected by monitoring YouTube traffic in a large university campus network. The CAMPUS trace is gathered from Harvard's main campus server and is dominated by email traffic. Though these traces are not special to security design, by analyzing the client's behaviors we can evaluate the performance of current security protocols in terms of hit rates, request delays,

Table 5. A summary statistics for a 24-hour trace.

Trace	Request Statistics			Req. Files Per Client		
	Client(IP)	file	requests	max	min	avg
YouTube	18936	52695	81278	1176	1	4
CAMPUS	13	15541	873345	8418	2	2216

Table 6. The capability hit rate as a function of lifetime in the T10 OSD-2 security protocol.

Lifetime	YouTube			CAMPUS		
	hit	miss	hit rate	hit	miss	hit rate
0.5h	3062	78216	3.77%	799998	73347	91.6%
1h	3211	78067	3.95%	809092	64253	92.6%
2h	3339	77939	4.11%	818070	55275	93.7%
6h	3479	77799	4.28%	831320	42025	95.2%
12h	3600	77678	4.43%	837684	35661	95.9%

Table 7. The capability hit rate as a function of lifetime in the Maat security protocol.

Lifetime	YouTube			
	valid hit	miss	false hit	valid hit rate
3minute	15126	23318	42834	18.6%
5minute	18651	23318	39309	22.9%
10minute	23831	23318	34129	29.3%
20minute	29811	23318	28149	36.7%
30minute	33415	23318	24545	41.1%
Lifetime	CAMPUS			
	valid hit	miss	false hit	valid hit rate
3minute	868748	2889	1708	99.5%
5minute	869398	2889	1058	99.5%
10minute	869918	2889	538	99.6%
20minute	870185	2889	271	99.6%
30minute	870276	2889	180	99.6%

etc.

Table 5 shows the summary statistics for the traces over a 24-hour period. For the YouTube trace, we selected all trace records that contain client requests to the YuTube server (anonymized IP: 63.22.65.73) from Wednesday March 12 04:00AM through Thursday March 13 04:00AM 2008. The CAMPUS trace is taken from the home02 server at Harvard from Thursday September 13 04:00AM through Thursday September 13 04:00AM 2001. Since the current OSD-2 and Maat security protocols authorize clients at the granularity of a single object and file, we pick out data-related operations, such as create, read and write, from the CAMPUS trace and group a series of read or write requests having sequential addresses from the same client to the same file into a big file read or write. The metadata operations, such as lookup and getattr, are filtered out, since they do not require a capability. As the table shows, the 24-hour CAMPUS trace has only 13 clients, with each requesting 2216 files, because this trace captures only the traffic between the email and general login servers and the disk arrays. Thus the CAMPUS trace represents traditional direct-attached storage environments, where the storage system is directly attached to the server or workstation. In contrast, the video server in the YouTube environment directly provides services to the client that must first acquire the necessary video information from the YouTube server. Such a system architecture is also used in most of current large-scale parallel file systems. Our statistics show that the YouTube system services a large number of clients, up to 18936 clients in the 24-hour trace on a YouTube server.

Since the lifetime of capabilities has a direct impact on the capability hit rate, before selecting an appropriate lifetime to evaluate the performance of the OSD-2 and Maat security protocols we first examine the capability hit rate using different capability lifetimes and in different trace environments. The results are shown in Tables 6 and 7. Since there is no privilege information recorded in these two traces, we assumed that all clients have the same permissions to all files. For OSD-2, we define a capability hit as a repeated access from the same client to the same file in a given period, i.e., the lifetime of capability, and a capability miss as a first-time access from a client to a file. For Maat, we assume that the system can accurately predict a client's future accesses. We can thus group the client and ten files that the client will access into an extended capability, that is, once the client requests a file she will get an extended capability for her to access ten different files in the future. We define the request by which the client gets the extended capability initially as a capability miss, and valid hit and false hit conform to the forgoing definitions. Every time a false hit occurs, the client renewals all extended capabilities that she holds. We also assume that the cache in the client and server is large enough to cache all capabilities generated in 24 hours. For example, a 2GB memory buffer can cache roughly 20 million fine-grained capabilities or 0.5 million extended capabilities, whereas according to our statistics there are 78216 OSD-2 capabilities and 23318 Maat capabilities at most in the 24-hour traces. In fact, these assumptions potentially increase the capability hit rate. As the tables show that prolonging the lifetime of fine-grained capabilities do not significantly increase the capability hit rate, because the relatively long lifetime allows the client to finish most of her operations before the capability expires. In contrast, a long lifetime of extended capabilities can significantly increase the capability hit rate under the YouTube environment. But prolonging the lifetime of extended capabilities has no remarkable impact on the capability hit rate under the CAMPUS environment, because the CAMPUS client may hold a large number of extended capabilities, a renewal can extend all capabilities that she caches. Although longer lifetime means higher hit rate for extended capabilities, the automatic revocation mechanism used by Maat requires each extended capability to have a short lifetime. In a large-scale storage system with a large number of clients, short-lived capabilities force clients to frequently contact MDS to extend the

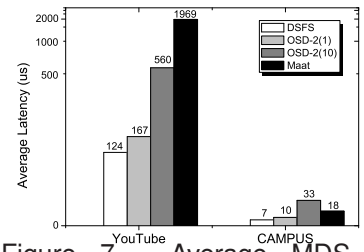


Figure 7. Average MDS open() latency.

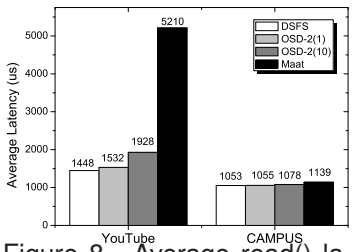


Figure 8. Average read() latency.

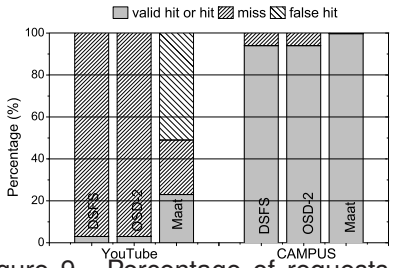


Figure 9. Percentage of requests with different metadata or capability hits.

lifetimes of capabilities thus compromising the performance benefit achieved by extended capabilities, as evidenced by the numbers of the YouTube trace in Table 7, where the number of false hits is 2.1 times bigger than that of valid hits.

We replay an hour-long trace from 19:00PM through 20:00PM on our prototype system and measure average request latency. We use a 1-hour lifetime for fine-grained capabilities and a 5-minute lifetime, the default value in the Maat literature, for extended capabilities. To give prominence to security overhead, for each data operation we read and write only 4KB data, roughly the size of an object node, from/to each storage device. As a common means to improve performance, our current prototype caches the file-system metadata at clients. When a request hits the metadata cache, the client need not acquire metadata from MDS for this request. In this 1-hour experiment the metadata cache has the same hit rate as the capability cache for fine-grained capabilities.

Figure 7 shows the average open() latency measured at MDS and Figure 8 shows the average read() latency measured at clients. (The write() latency results are similar and thus omitted.) With a high capability hit rate of up to 94% for OSD-2 and valid hit rate of up to 99.69% for Maat (shown in Figure 9) under the CAMPUS workload, the OSD-2 and Maat security protocols achieve latencies comparable to DSFS. However, under the YouTube workload OSD-2 is 4.5 times slower than DSFS in opening a file in a 10-OSD system while Maat is 16 times slower than DSFS. With the write() operations these slowdowns are 1.3 and 3.6 times for OSD-2 and Maat respectively. Even storing a file to a single OSD, OSD-2 is still slower than DSFS. This is because with a large number of clients in the YouTube system, the capability hit rate and valid hit rate is only 3% for OSD-2 and 23% for Maat respectively, as well as a false hit rate of 51% for Maat.

5.5. Security Administration Overhead

We measured the latency of the chmod operation to evaluate the overhead of security administration in DSFS and capability-based security systems (including OSD-2 and Maat), respectively. For the chmod operation, DSFS requires two round trips from clients to MDS and then from the MDS to OSD respectively, while capability-based security systems require only one round trip from clients to MDS, because in the capability-based systems the chmod operation only needs to modify the privilege information in MDS, but in the DSFS security system, in addition to modifying the global ACL, the chmod operation also needs to store pre-authorization lists to OSD.

Table 8 shows the overhead for the chmod operation. The system drivers were instrumented to report the time spent in fine-grained sub-operations. The latency is divided into the following categories: command encapsulation (CmdEncap), communication to MDS (CommToMds), communication to OSD (CommToOsd), database access, disk access, and security (including command en/decryption and privilege verification for the MDS command; or MAC computation and privilege verification for the OSD command). As shown in Table 8, a chmod operation costs a total of $433\mu s$ and $1144\mu s$ for capability-based and DSFS security systems respectively. The latter is 2.6 times slower than the former. But it should be noted that permission operations is much less frequent than the demanding I/O operation, such as read and write.

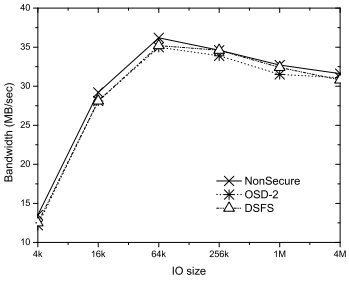
5.6. Throughput and Scalability

In this subsection, we evaluate under a high-bandwidth workload how much performance degradation is incurred when DSFS is added to an unsecured object-based storage system and compare it with the OSD-2 setup. The Maat setup is not evaluated in this experiment because its grouping strategy and benefits strongly depend on the workload and application environment.

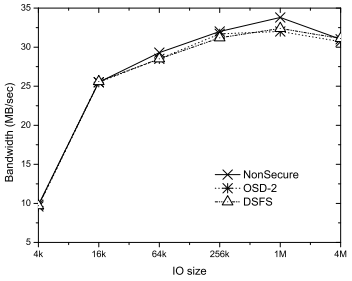
We ran a benchmark to measure the raw read, write performance with multiple transfer sizes (the transferred bytes per OSD command). A 512MB file is sequentially read and written from/to a 1-client and 1-OSD system. Figure 10 shows the bandwidth as a function of transfer size for the write and read benchmarks. For both non-secure and secure setups, the throughput increases with the transfer size. This is because the overall overhead of OSD command building and validation is less for higher transfer sizes when compared to lower transfer sizes.

Table 8. Overhead of chmod operations.

Latency (μs)	*MDS Command*				*OSD Command*				*TotalCost*
	CmdEncap	*CommToMds*	*Security*	*Database*	*CmdEncap*	*CommToOsd*	*Security*	*Disk*	
Capability	38	227	86	82	-	-	-	-	433
DSFS	38	227	86	82	42	258	231	180	1144



(a) Read bandwidth



(b) Write bandwidth

Figure 10. Read/write bandwidth with 1 OSD for the non-secure, OSD-2 and DSFS setups.

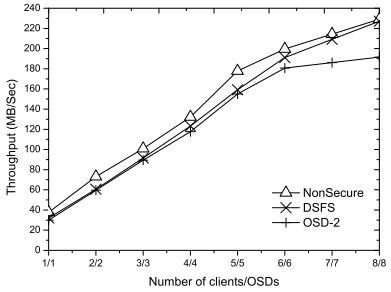


Figure 11. Aggregate read bandwidth with multiple clients/OSDs.

Another important observation from the figure is that both OSD-2 and DSFS setups have comparable bandwidth with the non-secure setup because the benchmark hardly yields overhead on MDS in a single-client to single-OSD system. Compared to the non-secure setup, the read and write performances with both the OSD-2 and DSFS setups decrease by less than 5%.

We ran another benchmark to measure the aggregate throughput and MDS's idle CPU time of the non-secure, OSD-2 and DSFS implementations with 1 through 8 OSDs and clients. Each client read and wrote files on an OSD and each OSD was accessed by exactly 1 client. In each run, each client created 512 files, each of size 256KB, on an OSD and sequentially read and wrote these files in 64KB chunks.

Figure 11 shows the aggregate throughput as a function of the number of OSDs/Clients for the read benchmarks. (The write benchmark results have similar trends and are not shown.) The results show that the aggregate read throughput of all clients for the non-secure and DSFS setups scales linearly with the number of clients/OSDs, which indicates that the MDS imposes very low overhead on a high-bandwidth workload and has not become a bottleneck in the non-secure and DSFS systems with up to 8 clients/OSDs. However, at the number of 6 clients/OSDs, there is an inflexion in the curve for the OSD-2 setup. As the number of clients/OSDs scales up above 6, the aggregate read throughput in the OSD-2 system starts to flatten out. This is because substantive capability requests in the OSD-2 system start to overwhelm the MDS that then restrains the whole system performance. The average percentage of idle CPU time, shown in Figure 12, also clearly illustrates the MDS's load with different security setups.

As the figure shows, the MDS's idle CPU time

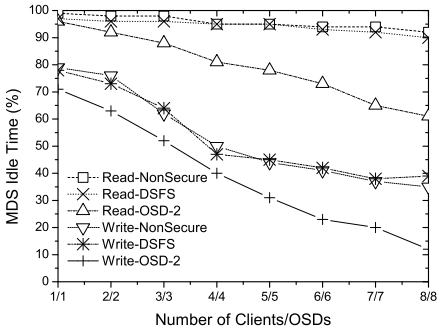


Figure 12. Average percentage of idle CPU time on the metadata server.

declines as the number of clients/OSDs increases. However, relative to the non-secure system, DSFS does not impose any additional overhead on the MDS because no capabilities are required for the read and write operations in the DSFS system. Thus the MDS's idle CPU time of the DSFS system is comparable to that of the non-secure system. On the contrary, the MDS's idle CPU time with the OSD-2 system declines faster than that of the non-secure and DSFS systems because the MDS must prepare a capability for each read or write request with OSD-2. Note also that the MDS's idle CPU time for the write operation is lower than that for the read operation, because in our current system implementations the MDS must synchronously update the file length for each successful write command, which not only increases the latency of a single write operation but also overloads the MDS when a larger number of write operations arise.

6. Conclusions

This paper described DSFS, a decentralized security system for large parallel file system, designed to provide decentralized access control by introducing

pre-authorization lists (PALs) to storage devices. With PALs, DSFS enables a network-attached storage device to directly authorize I/O rather than demanding a client to acquire an authorization from a centralized authorization server. As long as the client authenticates to a single authentication server and acquires an identity key, she can access any devices she wants. By maintaining a global ACL at a centralized making server, DSFS allows users to set any access control policy for the global ACL or even change the global ACL system but without impacting on the file placement in parallel file systems, without needing to update the security code in their storage devices.

We prototyped DSFS in the HUST OSD project. Our implementation requires minimal changes to the current T10 OSD standard but enables the standard to support decentralized access control. Experimental Results from trace-driven experiments and high-bandwidth workloads show that DSFS achieves lower operation latencies than the OSD-2 and Maat security protocols. Both OSD-2 and Maat need to maximize capability cache hits to achieve the same performance. The cost of security checks at storage devices for DSFS (up to 32 PAEs in a PAL in our experiments) is comparable to the overhead of verifying a fine-grained capability. However, DSFS completely removes the performance bottleneck on MDS caused by security overhead by avoiding capabilities for the demanding I/O operations.

References

[1] P. J. Braam, "The Lustre storage architecture," *http://www.lustre.org/documentation.html, Cluster File Systems, Inc.*, Aug. 2004.

[2] S. Ghemawat, H. Gobioff, and S. Leung, "The Google file system," in *Proc. of SOSP'03*, Oct. 2003.

[3] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka, "A cost-effective, high-bandwidth storage architecture," in *Proc. of 8th ASPLOS*, San Jose, CA, USA, Oct. 1998, pp. 92–103.

[4] D. Nagle, D. Serenyi, and A. Matthews, "The Panasas activescale storage cluster-delivering scalable high bandwidth storage," in *Proceedings of the ACM/IEEE SC2004 Conference*, Pittsburgh, PA, USA, Nov. 2004.

[5] F. Schmuck and R. Haskin, "GPFS: A shared-disk file system for large computing clusters," in *Proc. of FAST'02*, Jan. 2002.

[6] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proc. of OSDI '06*, Seattle, WA, November 2006, pp. 307–320.

[7] M. K. Aguilera, M. Ji, M. Lillibridge, J. MacCormick, E. Oertli, D. Andersen, M. Burrows, T. Mann, and C. A. Thekkath, "Block-level security for network-attached disks," in *Proc. of FAST '03*, San Francisco, CA, 2003, pp. 159–174.

[8] A. Azagury, R. Canetti, M. Factor, S. Halevi, E. Henis, D. Naor, N. Rinetzky, O. Rodeh, and J. Satran, "A two layered approach for securing an object store network," in *Proc of IEEE Security in Storage Workshop*, Greenbelt, MD, 2002, pp. 10–23.

[9] H. Gobioff, "Security for a high performance commodity storage subsystem," Ph.D. dissertation, Carnegie Mellon University, July 1999.

[10] R. A. Oldfield, A. B. Maccabe, S. Arunagiri, T. Kordenbrock, R. Riesen, L. Ward, and P. Widener, "Lightweight I/O for scientific applications," Sandia National Lab, Tech. Rep. 2006-3057, May 2006.

[11] A. W. Leung, E. L. Miller, and S. Jones, "Scalable security for petascale parallel file systems," in *Proc. of SC07*, Reno, NV, USA, Nov. 2007.

[12] B. C. Reed, E. G. Chron, R. C. Burns, and D. D. E. Long, "Authenticating network-attached storage," in *Proc. of 7th IEEE Hot Interconnects Symposium*, Aug. 1999.

[13] *SCSI Object-Based Storage Device Commands -2 (OSD-2)*, Project t10/1729-d, revision 5 ed., T10 Technical Committee, NCITS, January 2009.

[14] Z. Niu, K. Zhou, D. Feng, H. Jiang, F. Wang, H. Chai, W. Xiao, and C. Li, "Implementing and evaluating security controls for an object-based storage system," in *Proc. of MSST'07*, San Diego, CA, USA, Sep. 2007, pp. 87–99.

[15] M. Bellare, R. Canetti, and H. Krawczyk, "Keying hash functions for message authentication," in *CRYPTO*, 1996, pp. 1–15.

[16] V. Kher and Y. Kim, "Decentralized authentication mechanisms for object-based storage devices," in *Proc. of the Second IEEE International Security In Storage Workshop*, Washington, DC, 2003, pp. 1–10.

[17] F. Wang, Q. Xin, B. Hong, S. A. Brandt, E. L. Miller, and D. D. E. Long, "File system workload analysis for large scale scientific computing applications," in *Proc. of the Conference on Mass Storage Systems and Technologies*, Apr. 2004.

[18] B. C. Reed, M. A. Smith, and D. Diklic, "Security considerations when designing a distributed file system using object storage devices," in *Proc. of the First IEEE International Security In Storage Workshop*, Greenbelt, MD, Dec. 2002, pp. 24–34.

[19] J. H. Howard, M. L. Kazar, S. G. Menees, A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West, "Scale and performance in a distributed file system," *ACM Transactions on Computer Systems*, vol. 6, no. 1, pp. 51–81, February 1988.

[20] D. Maziéres, M. Kaminsky, M. F. Kaashoek, and E. Witchel, "Separating key management from file system security," in *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP)*, Kiawah Island, SC, 1999, pp. 124–139.

[21] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck, *Network File System (NFS) version 4 Protocol*, RFC 3530, IETF, April 2003.

[22] M. Zink, K. Suh, Y. Gu, and J. Kurose, "Watch global, cache local: YouTube network traffic at a campus network - measurements and implications," in *Proceedings of SPIE/ACM Conference on Multimedia Computing Networking*, 2008.

[23] D. Ellard, J. Ledlie, P. Malkani, and M. Seltzer, "Passive NFS tracing of email and research workloads," in *Proceedings of the Second Annual USENIX File and Storage Technologies Conference (FAST'03)*, San Francisco, California, March 2003, pp. 203–216.