Secure Capabilities for a Petabyte-Scale Object-Based Distributed File System[*]

Christopher Olson and Ethan L. Miller
Storage Systems Research Center
Computer Science Department
University of California, Santa Cruz
{topher,elm}@cs.ucsc.edu

ABSTRACT

Recently, the Network-Attached Secure Disk (NASD) model has become a more widely used technique for constructing large-scale storage systems. However, the security system proposed for NASD assumes that each client will contact the server to get a capability to access one object on a server. While this approach works well in smaller-scale systems in which each file is composed of a few objects, it fails for large-scale systems in which thousands of clients make accesses to a single file composed of thousands of objects spread across thousands of disks. The file system we are building, Ceph, distributes files across many objects and disks to distribute load and improve reliability. In such a system, the metadata server cluster will sometimes see thousands of open requests for the same file within seconds. To address this bottleneck, we propose new authentication protocols for object-based storage systems in which a sequence of fixed-size objects comprise a file and flash crowds are likely. We qualitatively evaluated the security and risks of each protocol, and, using traces of a scientific application, compared the overhead of each protocol. We found that, surprisingly, a protocol using public key cryptography incurred little extra cost while providing greater security than a protocol using only symmetric key cryptography.

Categories and Subject Descriptors:
D.4.3 [File Systems Management]: Distributed file systems;
D.4.6 [Security and Protection]: Cryptographic controls

General Terms: security, performance

Keywords: scalability, object-based storage, capabilities

[*]This work was supported in part by Lawrence Livermore National Laboratory, Los Alamos National Laboratory, and Sandia National Laboratory under contract B520714, and by the sponsors of the Storage Systems Research Center: Engenio, Hewlett Packard Laboratories, Hitachi Global Storage Technologies, IBM Research, Intel, Microsoft Research, Network Appliance, Rocksoft, Veritas, and Yahoo.

1. INTRODUCTION

Network attached storage offers a way to improve performance by separating the metadata path from the file data path. However, it creates a security problem not present in centralized file systems: the metadata server must securely communicate authorizations to the storage devices. NASD [4] proposed the use capabilities for this purpose. The metadata server could issue unforgeable capabilities to the client, and the client could then submit them with each request to the network-attached disk. The disk could then use the capability to authorize access to file data without needing to track users, groups, pathnames and file permissions.

Traditional NASD capabilities are specific to a single object on one disk. This approach works well for systems in which files are composed of a small number of objects, and files are accessed by one client at a time. This type of system is common, including most workstation environments and other small-scale computing environments. In such systems, individual clients send requests to open a file to a metadata server; in response, they are given capabilities that they can then present to the disk that stores the object containing the file data.

Unfortunately, this approach does not work well in an environment with terabyte-scale files composed of thousands of objects spread across thousands of network-attached disks. In order to open a single file, a client would have to request thousands of capabilities from the metadata server, one for each object. Worse, systems with terabyte-scale files often require that the files be accessed simultaneously by tens of thousands of individual clients cooperating on a scientific computation. If each client needs its own unique set of capabilities—necessary to ensure that non-authorized clients can't access the file—the metadata servers might have to hand out hundreds of millions of capabilities to access a single very large file.

Our research demonstrates an alternative approach, showing how the metadata server can generate a capability of fixed size, independent of the number of objects, that must be protected from forgery. Using symmetric encryption requires that all disks know the decryption key, opening vulnerabilities should an attacker compromise a storage device. On the other hand, using signatures and public key encryption may be computationally expensive. We present several secure protocols for such an object based file system, and estimate their overheads based on the file traces of a scientific computation. We found that the public key options do not add unbearable overhead but do offer greater protection should an attacker compromise a storage device.

2. BACKGROUND

Traditional distributed file systems such as NFS [19] and AFS [9] serve all files from a central host which mediates between clients and the disk. The central server handles small frequent metadata requests, but it creates a bottleneck for large bulk transfers of file data. To increase bandwidth for file data, network attached secure disks (NASD) [4] separate the metadata server from the file data server. A small cluster—generally fewer than ten computers—of metadata servers (MDS) manages the directory tree, file permissions and timestamps. The MDS associate a file with an object on a particular object storage device (OSD). Each OSD manages a flat namespace of objects and services only read and write requests for the data in those objects; there are potentially thousands of OSDs in a single system. The NASD design separates small metadata operations from high bandwidth data transfers, which can potentially use different network paths to further reduce contention. The design localizes management of the directory tree, which requires significant coordination to keep consistent, among a small number of hosts. Finally, the design involves a large number of hosts only for data requests, which requires little or no consistency coordination.

2.1 Ceph

We are designing a distributed file system, Ceph, that uses network attached object storage devices to hold file data for applications ranging from workstation-style individual file access to coordinated access for high-performance parallel applications. Unlike NASD, which places the entire contents of a file in one object on one OSD, Ceph distributes and replicates a file across a sequence of objects on many OSD. Traditional NASD techniques would bottleneck at the OSD if multiple hosts access the same file. Many scientific computations open, read and write one file from thousands of nodes in a coordinated way; by distributing a file to multiple objects on multiple OSDs, Ceph will distribute load under such conditions. NASD will lose file data if the replication internal to an OSD or a whole OSD fails, whereas Ceph can fall back on other OSDs.

In Ceph, metadata servers (MDS) manage the directory hierarchy, permissions and file to object mapping. To efficiently balance load, the MDS partition the directory tree across the cluster [21]. A client guesses which metadata server is responsible for a file, and contacts that server to open the file. That MDS will forward the request to the correct MDS if necessary. The responsible MDS will reply with a file handle, and information about which MDS manages each component along the full pathname. The client may use this information to improve future guesses about which MDS to contact for a particular file.

The file handle describes which objects on which OSD contain the file data. The RUSH algorithm maps a sequence index to the OSD holding the object at that position in the sequence, distributing the objects in a uniform way [8]. Unlike a simple hash function modulo the number of available OSD, the RUSH algorithm requires Ceph to redistribute only a small portion of the file data when adding or removing OSDs from the system. The RUSH algorithm also allows Ceph to replicate files in different patterns rather than replicate an entire OSD. Thus, when one OSD fails, multiple OSDs participate in its restoration. This narrows Ceph's window of vulnerability to a second failure that could cause irrecoverable data loss [22].

Ceph limits objects to a maximum size (*e. g.*, 1 MB), so files are a sequence of bytes broken into chunks on the maximum object size boundary. Since only the MDS hold the directory tree, OSDs do not have directory information to suggest layout hints for file data, such as in FFS [13]. Instead, the OSDs organize objects into small and large object regions, using small block sizes (*e. g.*, 4 KB or 8 KB) for small objects and large block sizes (*e. g.* 50–100% of the maximum object size) for large objects [20]. This layout guarantees that bulk reads and writes for a large object are likely to be contiguous on disk.

Security is an important issue in this environment, because with tens of thousands of clients and thousands of disks, it may take only a single compromised client or OSD to allow an intruder to read data from any OSD in the system. However, as described below, a naive implementation using the basic NASD security approach simply will not scale to hundreds of millions of $\langle client, OSD \rangle$ pairs.

2.2 Network Attached Secure Disks

Gobioff, *et al.* proposed capabilities for Network Attached Secure Disks, but a capability was good for only one object [5], as was the case for SCARED [17]. Both used a symmetric MAC to sign the capability, requiring the MDS to share a key with each OSD. SNAD [14] improved upon this approach, allowing capabilities to apply to multiple files by using public-key encryption to secure the capabilities, making it possible for an MDS to generate capabilities that could be used by multiple OSDs without concern that an OSD could forge capabilities. Aguilera, *et al.* [1] further improved on standard NASD capabilities by allowing them to specify multiple extents of blocks to which the capability applies, though they used a symmetric MAC rather than a public key signature to make the capability unforgeable.

None of these approaches, however, will suffice for Ceph, in which a file may contain thousands of objects spread across thousands of OSDs. We do not want to require that the MDS return thousands of capabilities, nor do we want a subverted OSD to be able to use a capability to issue requests to other OSDs. We will show how a fixed size capability can describe an indefinite number of objects, and we will show how to derive session keys specific to each OSD.

2.3 Secure Distributed File Systems

SFS [12] includes a hash of a host's public key to create a self-certifying pathname, and provides a consistent directory namespace to users regardless of which host they use to access SFS. CapaFS [18] adds access control to the self-certifying pathnames of SFS. However, SFS and CapaFS do not address replication of data or distribution of load. SFS-RO replicates directory trees, such as package distribution sites, assuring the user of the authenticity of the replicated data, but SFS-RO is a read-only file system [3].

SiRiUS, SUNDR, and Plutus all provide integrity and confidentiality for file systems stored in untrusted repositories [7, 10, 11]. SiRiUS uses a public/private key pair per client to secure lock boxes for files and directories. SUNDR clients sign revision histories so that a server cannot easily substitute old authenticated data for new authenticated data; at worst, the server may fork histories. Plutus uses lockbox techniques similar to those in SNAD, adding the use of Merkle hash trees and the ability to revoke permissions when a file is modified by using public-key encryption techniques to generate new lockbox keys. However, none of SiRiUS, SUNDR and Plutus addresses replication or large scale distribution. Plutus, in particular, has relatively low performance, and would likely perform poorly in a highly scalable environment because of bottlenecks in distributing and managing a large number of keys.

3. THE PROTOCOLS

In this section, we first describe the basic protocol by which a client accesses an object in Ceph. We then use this basic protocol as a baseline to describe three secure protocols for accessing multiple objects on multiple OSDs from multiple clients. These protocols are specified somewhat formally so that we can analyze their security. The principals in all of the protocols include a client machine C acting on behalf of a user U, a metadata server M, and an OSD D. If there is more than one of a given principal, the principal identifier may be subscripted.

We write $A \rightarrow B : M$ when we intend for the host A to send the message M to the host B. However an attacker may intercept, mangle and replay messages. If A sends M, it may arrive at B, a new message M' may arrive in its place, or no message may arrive at all. If B receives a message Q, it cannot automatically conclude that A recently sent Q.

We use K_A^U and K_A^R to represent the public and private keys for host A. We use K_{AB} to stand for a symmetric key known to A and B; a trusted server which generated the key may also know it, but for any other host to know K_{AB} constitutes a compromise of the key.

We write $\{M\}K_A^U$ to denote the encryption of the message M under the public key K_A^U; assuming that only A holds the private key K_A^R, M is readable only by A. We write $\{M\}K_{AB}$ for the authentication and symmetric encryption of M. M can be formed or read only by hosts that possess K_{AB}.

We denote the message M together with its public key signature by $\langle M \rangle K_A^R$; only A may sign the message, but any host can verify the signature using K_A^U.

We timestamp all messages so all encrypted and signed messages expire, and we assume that the symmetric and public key cryptography are sufficiently strong to thwart chosen plaintext attacks until well after legitimate messages expire.

3.1 Basic Operation

This section describes the exchange of messages in Ceph without any security to show how a client interacts with the MDS and OSDs. This protocol describes how a client opens a file and reads and writes its contents. It forms a baseline by which we evaluate the overhead of our security mechanisms.

PROTOCOL 1. *Accessing a file in a protected environment.*

Trust assumptions: The MDS cluster will properly maintain the file system metadata, and it will serve correct information in response to client requests. The MDS will check that the user is authorized to access the file in the requested mode before granting a capability authorizing access to the file's objects. Upon a read of an object the OSD will return the data most recently written to that object.

Message exchanges:

$$\begin{array}{lcll}
C \rightarrow M_1 & : & U, \mathrm{open}(\mathit{path}, \mathit{mode}) & (1.1) \\
M_1 \rightarrow M_2 & : & U, \mathrm{open}(\mathit{path}, \mathit{mode}) & (1.2) \\
M_2 \rightarrow C & : & H & (1.3) \\
C \rightarrow D(H,i) & : & \mathrm{read}(\mathit{oid}(H,i), \mathit{bno}) & (1.4) \\
D(H,i) \rightarrow C & : & \mathit{data} & (1.5) \\
C \rightarrow D(H,i) & : & \mathrm{write}(\mathit{oid}(H,i), \mathit{bno}, \mathit{data}) & (1.6) \\
D(H,i) \rightarrow C & : & \mathrm{okay} & (1.7)
\end{array}$$

Risks: An attacker may impersonate any participant, read any data, and forge data. □

In message 1, the client C requests to open a file on behalf of user U. The client makes its best guess of which metadata server to contact based on *path*, and sends the request to that server. The server may determine that it does not manage the partition containing the path. In message 2, the server forwards the open request to the correct metadata server. Hereafter, we elide message 2, with the understanding that, in future protocols, the MDS may forward an open request some number of times, so message 3 may originate from a metadata server other than the original recipient of the open request.

In message 3, the authoritative metadata server sends the client the *file handle* H, which lists the $\langle \mathit{osd}, \mathit{oid} \rangle$ pairs of the objects that compose the file. The response also indicates which metadata server handles each name along the path, and the client will cache this hint to improve its guesses of which metadata server to contact for future requests [21].

In message 4, the client requests block *bno* of object i of the file. The function $D(H,i)$ yields the identifier of the OSD that contains the i^{th} object; $\mathit{oid}(H,i)$ is its object identifier on that device. We intend that H represents seed values, and the D and *oid* are generator functions. In Ceph, H contains initialization data for the RUSH placement function, and $D(H,i)$ represents using RUSH to find the i^{th} OSD [8]. In Ceph the *oid* is simply the 64 bit value obtained by concatenating a 32 bit unique file identifier found in H with the 32 bit value i.

In message 5, the OSD returns the requested block. Messages 6 and 7 are similar to messages 4 and 5, but show a write request and its acknowledgment.

3.2 Handling Vulnerable Clients

We now investigate one way to secure the above protocol. We assume some strong physical and network protections for the MDS and OSDs. For example, these hosts may be locked in a room, guards with guns and vicious dogs may patrol the premises, a network firewall may permit only certain network traffic, and they may run only sanctioned software. We do not assume the clients have such strong protections. When a user enters a name and password into a workstation, he implies that he trusts the machine. When the user runs software, reads email or surfs the web, he may believe his actions are safe. However, that machine may have been compromised, or the user may unintentionally run some malware. We do not address protecting the user from his own actions here, but this protocol does protect other users from the compromise of one user. It does not permit a subverted machine to obtain access to data which the user of that machine could not ordinarily obtain. However, a machine that has been subverted by, for example, a keystroke logger, could capture a user's password and access any files to which that user had access.

The protocol uses symmetric cryptography to ensure privacy and integrity for all communications, including authentication of capabilities. We require that the client and MDS arrange a shared key K_{CM} *a priori*, which may come from Kerberos [15] or some security service that may already be deployed. Protocol 1 used U in the first message to identify the user opening the file; hereafter the protocols use C, which represents the token from such a service. The metadata server uses to select the key K_{CM}, and it associates that token with the user U. Although an existing security service authenticates the user to the MDS, the MDS provides session keys for the client to use with the disks; we do not burden the administrator with registering thousands of OSDs with an existing security service.

PROTOCOL 2. *Protecting honest clients from malicious or subverted clients.*

Trust assumptions: The MDS will keep the keys K_{CM} and K_{CD} secret. The MDS and OSD will keep K_{MD} secret. The MDS will generate "good" session keys, which are sufficiently random so that an attacker cannot guess a key sequence. The OSD will check that a request is authorized by the capability. Additionally, the trust assumptions of Protocol 1 apply.
Assumptions of honest participants: The client will keep the keys K_{CM} and K_{CD} secret.
Message exchanges:

$$C \rightarrow M \quad : \quad C, \{\text{open}(\mathit{path}, \mathit{mode})\}K_{CM} \qquad (2.1)$$

$$M \rightarrow C \quad : \quad \{H, K_{CD}, T_s, T_e\}K_{CM}, \mathcal{C} \qquad (2.2)$$

$$C \rightarrow D(H,i) \quad : \quad \mathcal{C}, \{\text{read}(i, \mathit{bno}), T_1\}K_{CD} \qquad (2.3)$$

$$D(H,i) \rightarrow C \quad : \quad \{T_1, \mathit{data}\}K_{CD} \qquad (2.4)$$

$$C \rightarrow D(H,i) \quad : \quad \mathcal{C}, \{\text{write}(i, \mathit{bno}, \mathit{data}), T_2\}K_{CD} \qquad (2.5)$$

$$D(H,i) \rightarrow C \quad : \quad \{\mathit{hash}+1\}K_{CD} \qquad (2.6)$$

where $\mathcal{C} = \{H, \mathit{perm}, K_{CD}, T_s, T_e\}K_{MD}$

Risks: We examine the abilities an attacker can obtain by compromising a key, though we do not exhaustively list all the possibilities. We designed the protocol to protect honest clients from malicious or subverted clients, so our risk analysis here focuses on failure to satisfy the honesty assumptions.

Should the user log into a compromised workstation, or download malware via email or the web, the attacker may acquire the rights of the user, but the protocol continues to protect other users from the breech. The attacker may use K_{CM} to impersonate C to the metadata server and access any file which C could ordinarily access.

The attacker may use K_{CD} to impersonate the client to the OSD and issue any request he chooses, or he may use K_{CD} to read requests and responses for that client. If the OSD properly verifies the capability before servicing a request, then the attacker may only use K_{CD} in such ways from time T_s to time T_e. □

Although the attacker could not use the information in message 1 to access any file data, we encrypt it in case the pathname leaks any sensitive information. After verifying that the user is authorized for the requested file and mode, the MDS generates a capability that states, "any client that knows K_{CD} may access the objects in H according to *perm* from time T_s to T_e." To prevent forgery of the capability, the metadata server encrypts it with K_{MD}, which is a symmetric encryption key known to the MDS and all OSD. The MDS also shares the new session key with the client.

The client includes the capability with every request, as in messages 3 and 5. This way, the OSD does not need to keep session state for each client. The OSD may use K_{MD} to decrypt the capability where it will find the session key K_{CD}. It can then use K_{CD} to decrypt the request, but before expending the processing for that, it first verifies that the current time is between T_s and T_e, and that the request is compatible with *perm*.

In the basic protocol, the client asked to read a particular object. However, in this version the client asks to read the i^{th} object. The OSD must verify that it indeed serves the i^{th} object; if the client included the *oid* in the request, the OSD would need to verify that as well. We do not assume that the functions $D(H,i)$ or $\mathit{oid}(H,i)$ are invertible, so the client requests the i^{th} object. The OSD computes $D(H,i)$ to verify that it is the correct OSD for the request, and then it computes the appropriate object $\mathit{oid}(H,i)$.

We did not design Protocol 2 to offer reasonable guarantees if the attacker compromises the MDS or OSD, as indicated by our not listing any honesty requirements on them. Nonetheless, we feel the protocol is weak since it requires all disks to share the key K_{MD}. If the attacker obtains K_{MD}, they he may create any capability he chooses and issue any request he chooses to the OSD. The attacker may also use the key to decrypt the capability in any request, obtain the session key, and then decrypt the request and response.

This protocol resembles the capabilities of NASD [5], and also shares the same weaknesses. Protocol 2 uses encrypted capabilities to carry the session key, whereas the NASD protocol derived the session key from the symmetric MAC used to authenticate a cleartext capability. We describe Protocol 2 here so that we may evaluate the cost of the stronger protocols.

3.3 Handling Vulnerable OSDs

With 10^3–10^4 OSDs in a system, it may be infeasible to provide strong physical and network protections for all of them. Thus, their exposure to attack may be greater than the MDS. Should an attacker compromise an OSD, he may obtain the critical key K_{MD} in the previous protocol and wreak havoc in the system.

This next protocol does not rely on a broadly shared secret; rather each OSD D shares its own K_{MD} with the MDS. That is, for any two distinct OSD D_1 and D_2, we probably have that $K_{MD_1} \neq K_{MD_2}$. In the previous protocol, the MDS could protect a capability from forgery by symmetrically encrypting it with K_{MD}. Now that there is no global symmetric key, the MDS must use a public key signature to authenticate the capability. Furthermore the MDS must provide a session key per OSD; otherwise a subverted OSD could use a global session key to impersonate the client to another OSD.

PROTOCOL 3. *Protecting honest clients and OSD from malicious or subverted clients and OSD.*

Trust assumptions: The MDS will keep the private key K_M^R and the symmetric key K_{CM} secret. The MDS will keep K_{MD} and K_{CD} secret for all OSD D. The MDS will generate good session keys. Additionally, the trust assumptions of the MDS in Protocol 1 apply.
Assumptions of honest participants: The client will keep the key K_{CM} secret. For each OSD D, the client and D will keep the session key K_{CD} secret. Each OSD D will keep the key K_{MD} secret. The OSD will check that a request is authorized by the capability. Upon a read of an object the OSD will return the data most recently written to that object.
Message exchanges:

$$C \to M \quad : \quad C, \{\text{open}(\mathit{path}, \mathit{mode})\}K_{CM} \tag{3.1}$$
$$M \to C \quad : \quad \{\mathcal{C}\}K_{CM} \tag{3.2}$$
$$C \to M_{D(H,i)} \quad : \quad C, D(H,i) \tag{3.3}$$
$$M_{D(H,i)} \to C \quad : \quad \{K_{C,D(H,i)}, D(H,i), T'_s, T'_e\}K_{CM}, \mathfrak{T} \tag{3.4}$$
$$C \to D(H,i) \quad : \quad \mathfrak{T}, \{\mathcal{C}, \text{read}(i, \mathit{bno}), T_1\}K_{C,D(H,i)} \tag{3.5}$$
$$D(H,i) \to C \quad : \quad \{T_1, \mathit{data}\}K_{C,D(H,i)} \tag{3.6}$$

$$\text{where} \quad \mathcal{C} = \langle P, H, \mathit{perm}, T_s, T_e\rangle K_M^R$$
$$\mathfrak{T} = \{C, G, K_{C,D(H,i)}, T'_s, T'_e\}K_{M,D(H,i)}$$

Risks: As before, we examine only the risks presented by dishonest participants. For a client to leak K_{CM} or any K_{CD} carries similar risks to those for Protocol 2. If the attacker acquires the key K_{MD} for some OSD D, then he may read traffic between any client and that OSD, however he may not read traffic between clients and other OSDs. With K_{MD} the attacker may also impersonate that OSD to any client, but he may not impersonate any other OSD. If an attacker acquires the session key K_{CD} for some OSD D,then he gains similar abilities as when acquiring K_{MD}, however those abilities are restricted to the particular client and only last for as long as the client uses K_{CD}, which should only be from the time T'_s to T'_e. □

In message 2, the MDS provides a signed capability stating, "any client that satisfies P may access the objects in H according to *perm* from time T_s to T_e." P is a principal descriptor that states in conjunctive normal form all users and groups who may access the file. In general, the principal descriptor could be as long as the directory tree is deep, but in practice 99% of files need two or fewer conjuncts [16]. P is a short statement of all users authorized to access the file, so if many different users open the file within a few minutes, the MDS may return the same capability to all of them. This way the MDS needs to expend processing time for signing the capability only once. Furthermore the OSD may cache the results of verifying the signature, which also benefits all clients that opened the file within minutes of each other.

Unlike our previous protocols, the MDS does not return a session key in the second message. In message 3, the client must contact the MDS to obtain a session key. The MDS returns a ticket stating, "any client that knows $K_{C,D(H,i)}$ is the user C and a member of the groups G from time T'_s to time T'_e." The MDS prevents forgery of the ticket by encrypting it with a symmetric key known only to the MDS and the particular OSD. The user and group membership items link the ticket to all capabilities the user may carry, which makes the ticket valid for any file's object stored on that OSD.

The client provides both the ticket and the capability in each request, so the OSD does not need to store any per client state. However the disk may cache the result of verifying a capability's signature for performance reasons. It can at anytime evict a cached verification without affecting proper operation of the security mechanism. The OSD may use $K_{M,D(H,i)}$ to decrypt the ticket. If the ticket is still valid according to its timestamps then the OSD can use the session key inside to decrypt the request. If the capability is still valid according to its timestamps then the disk may verify the signature, and it may check that the user and group membership from the ticket satisfy the principal descriptor.

3.4 Handling Vulnerable OSDs Scalably

The previous protocol needed two messages in addition to those of the basic protocol, which introduces additional latency. A flash crowd may flood the few MDS servers with such requests, which presents a scalability concern. In the next protocol we eliminate the extra messages by computing OSD-specific keys on the client and verifying them on each OSD. This computation is distributed across thousands of clients and OSDs, and will better scale to large numbers of participants.

PROTOCOL 4. *Protecting honest clients and OSD from malicious or subverted clients and OSD, without additional messages.*

Trust assumptions: The MDS will keep the private key K_M^R and the symmetric key K_{CM} secret. The MDS will generate good session keys. Additionally, the trust assumptions of the MDS in Protocol 1 apply.
Assumptions of honest participants: The client will keep the private key K_C^R and the symmetric key K_{CM} secret. The client will all keep $\{K_0\}K_C^R$ secret. Each OSD D will keep its private key K_D^R secret, and the client and OSD will keep the session key $K_{C,D(H,i)}$ secret. The OSD will check that a request is authorized by the capability. Upon a read of an object the OSD will return the data most recently written to that object.
Message exchanges:

$$C \to M \quad : \quad C, \text{request ticket} \tag{4.1}$$
$$M \to C \quad : \quad \{\mathfrak{T}\}K_{CM} \tag{4.2}$$
$$C \to M \quad : \quad C, \{\text{open}(\mathit{path}, \mathit{mode})\}K_{CM} \tag{4.3}$$
$$M \to C \quad : \quad \{\mathcal{C}\}K_{CM} \tag{4.4}$$
$$C \to D(H,i) \quad : \quad \begin{array}{l}\{K_{C,D(H,i)}\}K^U_{D(H,i)},\\ \quad \{\mathfrak{T}, \mathcal{C}, \text{read}(i, \mathit{bno}), T_1\}K_{C,D(H,i)}\end{array} \tag{4.5}$$
$$D(H,i) \to C \quad : \quad \{T_1, \mathit{data}\}K_{C,D(H,i)} \tag{4.6}$$

$$\text{where} \quad \mathcal{C} = \langle P, H, \mathit{perm}, T_s, T_e\rangle K_M^R$$
$$\mathfrak{T} = \langle U, G, K_C^U, K_0, T'_s, T'_e\rangle K_M^R$$
$$K_{C,D(H,i)} = \{\{K_0\}K^U_{D(H,i)}\}K_C^R$$

Risks: For a client to leak K_{CM} or any K_{CD} carries similar risks as in Protocol 2. For the attacker to acquire the key K_D^R for some OSD D carries the same risk as if the attacker acquired K_{MD} in Protocol 2. □

In this protocol, the client asks once for a ticket to speak with all OSDs. The ticket certifies the client's identity, group membership and public key. It also carries a key initializer used to derive OSD specific session keys without further involvement from the MDS. We encrypt the capability and ticket throughout the protocol in case the file handle, user identity or group membership carry sensitive information. However, should the client submit a request to a subverted OSD, that OSD may reveal the information anyway. Encrypting these items at least protects the system from traffic analysis by a attacker that has not subverted an OSD.

To secure traffic with a given OSD D, the client uses his private key K_C^R and the OSD's public key K_D^U to derive K_{CD} from the key initializer K_0. Before sending this key with the request, the client encrypts it a second time with K_D^U so an eavesdropper may not learn the session key. Upon receiving a request, the OSD uses the client's public key then its private key to decrypt the alleged K_{CD}. The OSD checks that the result matches K_0 to verify that the creator of this session is indeed C.

4. TRACE-BASED EVALUATION

The three protocols for securing communications in a scalable object-based storage system all have different performance characteristics. Our next goal was to compare protocol performance using real storage system traces gathered from a workload traced at Lawrence Livermore National Lab. By applying measured operation overheads for different cryptographic and network operations, we were able to simulate the performance of each protocol on the traced workload.

We identified five contributors to total cost in our protocols: the number of files opened, the number of opens, the number of requests, the number of disk-node pairs and the number of disk-file pairs. There may be more open requests than unique files opened, if some file is opened multiple times. The first open of a file costs one amount, but the cost of subsequent opens may be different. For example, the MDS in Protocol 3 needs time on the first open to sign the capability, but it can cache the signed capability and quickly provide it on subsequent opens. Each disk request and matching response needs time for encryption, decryption, network transfer, and disk transfer. In Protocol 3 the client must obtain an OSD-specific ticket for each new OSD it contacts; the number of disk-node pairs counts how many times this occurs. In Protocol 3 the OSD must verify the capability once for each file it sees; the number of disk-file pairs counts how many times this occurs. Protocol 4 also incurs per disk-node and per disk-file costs.

To estimate and compare the overheads of the different protocols, we used traces of a scientific application running on 256 processors. We counted open, read and write requests performed by each of 256 processes. We also counted how many distinct OSDs each node interacted with, and how many distinct files each node opened. To estimate the processing time of cryptographic operations, we timed the `libgcrypt` implementation running on a 3.2 GHz Pentium 4. We used AES for symmetric encryption, DSA for signing and RSA for public key encryption. As a best guess for MDS response time, we used the throughput numbers reported by Weil, *et al.* [21]. To estimate OSD response time, we used the throughput reported by Wang, *et al.* [20] for a random read/write workload. Table 1 reports the timings we used as a basis for our calculations. Note that symmetrically encrypting 1 MB of data requires nearly as much time as the disk transfer. This cost, which appears in all our secure protocols, drowns any difference one might detect from the public key operations. If performance is critical, however, it might be possible to use an optimized cryptographic library or hardware-accelerated encryption.

Operation	Time
Symmetrically Encrypt 1KB (Capability & Request)	43μs
Symmetrically Encrypt 1MB (Object Data)	44 ms
Sign 1KB (Capability)	3.3 ms
Verify Signature on 1KB	4.0 ms
Generate a 128 bit key	120 μs
MDS authorize access and lookup handle	330 μs
OSD Read/Write 1MB	50 ms

Table 1: Timings of basic operations used in estimating the overhead of the security protocols. For cryptographic operations, we timed `libgcrypt` on a 3.2 GHz Pentium 4. We obtained other timings from [20] and [21].

Number of		Count of				
P	Disks	Opens	Reqs	Files	DN	DF
1	10	1,538	540,081	4	2,549	27
16	100	1,538	537,386	34	11,945	314
256	10,000	1,538	536,634	514	17,891	3,265

Table 2: Sample counts from traces. P is the number of partitions. DN (disk-node pairs) counts the total number of distinct disks accessed by each node. DF (disk-file pairs) counts the total number of distinct disks on which each file stores some objects.

The scientific application ran on 256 processors, but divided the nodes into N partitions accessing N files. The nodes of a partition access different regions of one file; this behavior motivated the design of Ceph, which treats files as sequences of objects to distribute such workloads across OSDs. This is in contrast to the basic NASD design, which can only distribute accesses to different *files* across OSDs. We have traces for the application with all 256 nodes accessing 1 file, 2 partitions of 128 nodes accessing 2 files, 4 partitions of 64 nodes accessing 4 files, and so on up to 256 partitions of each node accessing its own file. We performed our analysis using OSD pools of sizes 10, 10^2, 10^3, and 10^4. Table 2 reports the counts we obtained from the traces for three different partition sizes and OSD pool sizes, showing that, even if there are only 256 clients in a 10,000 disk system, there are a total of nearly 18,000 disk/node pairs. In a larger system with larger data sets and files, the number of pairs could be 3–4 orders of magnitude larger.

We computed the total time the application would spend on file operations by totaling each item times the quantity of that time:

$$\begin{array}{lll} & \#\text{files} & \times \ \text{cost/file} \\ + & \#\text{opens} & \times \ \text{cost/open} \\ + & \#\text{requests} & \times \ \text{cost/request} \\ + & \#\text{disk-node pairs} & \times \ \text{cost/disk-node pair} \\ + & \#\text{disk-file pairs} & \times \ \text{cost/disk-file pair} \end{array}$$

Our plots are based on the total cost. We define the *overhead* of a secure protocol as its total cost minus the total cost of the baseline given by Protocol 1.

Regardless of the number of partitions, the application performed 1,538 `open`s in total. The number of read and

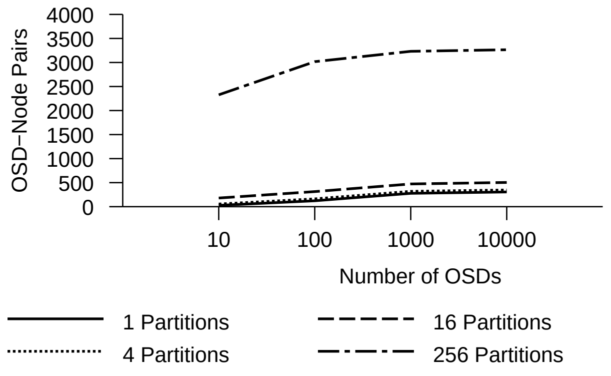

Figure 1: Disk-node pairs exhibited in analysis of the traces. Protocol 3 requires extra messages for each OSD-node pair, and Protocol 4 requires the client to perform additional public-key operations for each OSD-node pair. This graph shows that the total time spent on such operations across all nodes will grow sub-linearly with the number of OSDs.

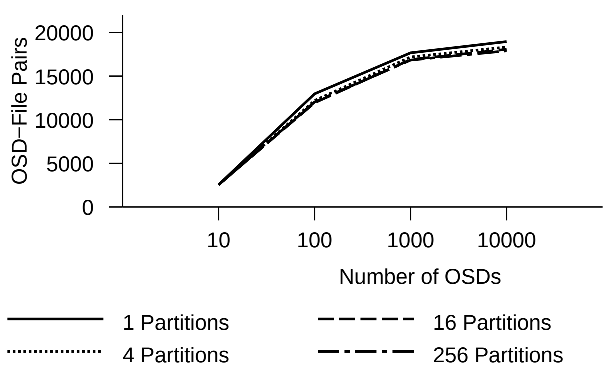

Figure 2: Disk-file pairs exhibited in analysis of the traces. For efficiency, the OSD may cache file capabilities they have already seen, and this graph shows the demand for cache will grow sub-linearly in the number of OSDs. Each new OSD will have its own cache, so cache space will grow linearly with the OSDs and outpace the need for cache.

write requests decreased slightly as the number of partitions increased. The number of files grew linearly with the number of partitions. Naturally, the number of disk-node pairings grew with the number of disks, since there were more disks for a node to pair with. However Figure 1 shows that this curve flattens quickly, perhaps because the application was only running on 256 processes, limiting its parallelism. Protocol 3 needs extra messages for each pairing of disk and node, and Protocol 4 requires public-key operations for each pairing. We can see that these costs grow sub-linearly with the number of OSDs. The number of disk-file pairs also grows with the number of disks and files, but Figure 2 shows this curve flattens as well, again perhaps due to the limited parallelism available in the small-scale trace we used. The OSD-side caches of verifications grow with the number of disk-file pairings; if we assume that each new OSD has the same amount of cache, then our cache space will grow linearly and outpace our requirement for cache which grows sub-linearly.

If the storage systems uses a network fabric such as the one used in the IBM BlueGene, which has a latency of $5\,\mu$s and a bandwidth of 350 Mb/s, latency will be very low, though bandwidth may not be as high as that available from other networks. For a system using such a network, Protocol 2 incurs almost as much overhead as Protocols 3 and 4, as Figure 3 shows. Using public key cryptography in this particular application would add less than 0.2% compared to using symmetric cryptography alone. Surprisingly, the more CPU-intensive cryptography and extra messages incur negligible additional cost while providing considerably greater security.

Protocol 4 trades additional public-key operations for additional messages. As Figure 4 shows, this tradeoff results in Protocol 3 gaining a slight advantage in a low latency network where the two additional messages are cheap; however, it quickly loses that advantage in higher latency networks.

Although the different protocols show nearly the same cost in this analysis, we found that even the symmetric key

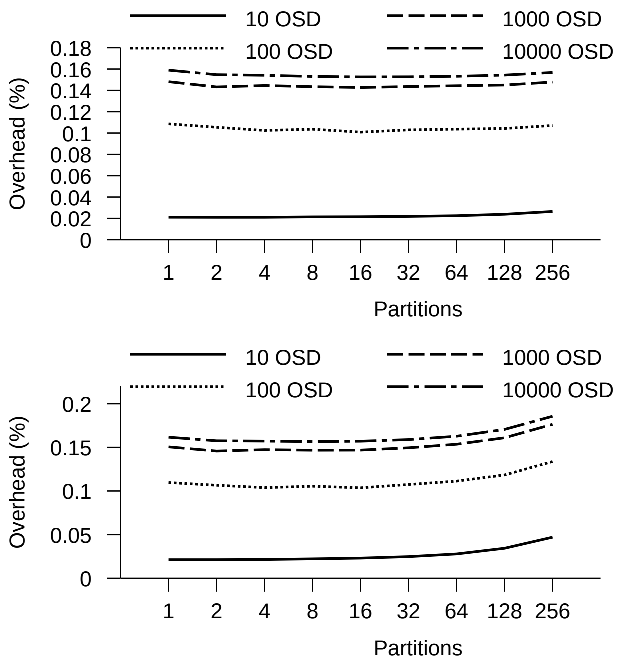

Figure 3: The overhead of Protocol 3 (top) and Protocol 4 (bottom), which use public-key cryptography, compared to the overhead of Protocol 2, which uses only symmetric key cryptography. To compute the overhead, we subtracted the time for Protocol 1; for example, the top graph shows $100 * \left(\frac{P3-P1}{P2-P1} - 1\right)$. We can have the additional security of public key cryptography without incurring significant additional costs, since the public key operations are a small part of the overall time. This is in a network with 350 Mb/s bandwidth and $5\,\mu$s latency.

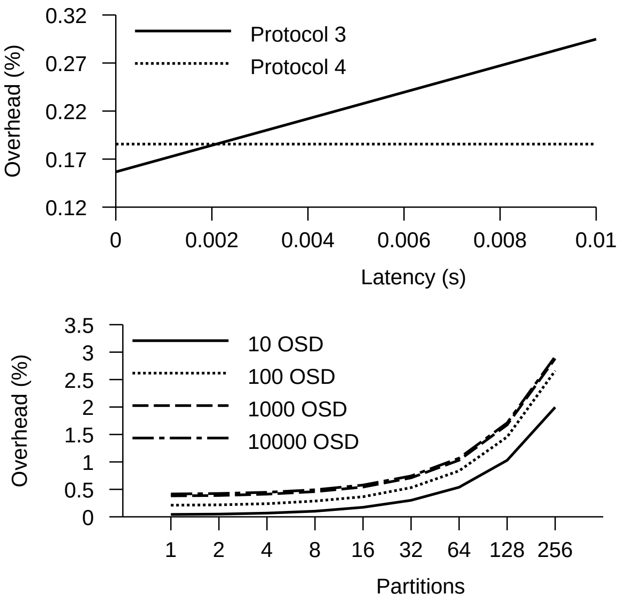

Figure 4: The overhead of Protocol 3 and Protocol 4 for varying network latencies (top), and the overhead of Protocol 3 compared to Protocol 2 in a high latency network (bottom). Protocol 3 has a slight advantage over Protocol 4 in low latency networks, but the client's second interaction with the MDS quickly overwhelms that advantage in a high latency network. In a 1 Gb/s Ethernet with a 0.2 s latency, the overhead of Protocol 3 compared to Protocol 2 rises sharply as more nodes enter the scientific computation.

protocol, Protocol 2, added 120% overhead compared to no security alone. This cost comes from the symmetric encryption of data, and can be reduced by only signing data and sending the data in the clear, or by using faster encryption or encryption hardware.

5. ANALYTICAL EVALUATION

Above we showed that the symmetric key protocol cost nearly as much as the public key protocols in a particular scientific application. However, we see from Table 2 that the application performs thousands of requests and only a few opens. Since Protocol 3 and Protocol 4 use public-key operations for each file, we must question how they compare to the symmetric protocol when used for different ratios of requests to opens. To answer this question, we turned to analytic evaluation of the protocols. While this evaluation approach uses a synthetic workload rather than a real workload, we believe that it is useful because it shows how our protocols perform on workloads with differing file request profiles.

We can easily vary the number of files, opens and requests in the cost computation given above. More effort is required to compute the number of disk-node and disk-file pairs. Suppose one node interacts with K objects that may or may not be on different OSDs from a pool of N OSD. How many distinct OSDs does the node interact with? To answer this, we define

$$\begin{aligned} T(m,k) &= 0, \text{ if } m=0 \text{ or } k=0 \\ T(m,k) &= \frac{m}{N}\left(1+T(m-1,k-1)\right) \\ &\quad + \left(1-\frac{m}{N}\right)T(m,k-1) \end{aligned}$$

$T(m,k)$ computes the number of distinct disks chosen, by making k independent and equally likely choices from a pool of N disks where m of them have not been picked in a previous choice. There are no distinct disks left to choose if $m = 0$. There is an m/N chance of choosing a new distinct disk, and an $1 - m/N$ chance of choosing a disk chosen previously. $T(N,K)$ will tell us the expected number of disk-node pairs if one client contacts disks for K different objects in a pool of N disks. $T(N,L)$ will tell us the expected number of disk-file pairs if a file has L objects stored on disks in a pool of size N.

Figure 5 shows the performance of one client opening one file and making varying numbers of requests to one object (which is on one disk). This figure compares the overhead of the public key protocols to the overhead of the symmetric protocol. The more secure protocols cost up to 14% more than the symmetric key protocol when we have only a few requests per file, but this high cost drops quickly. Figure 5 also shows one client opening one file and making one request to varying numbers of objects. Again, the more secure protocols cost up to 14% more than the symmetric key protocol, though the cost does not drop as sharply. The absolute times for a small number of requests will be imperceptible for interactive use. However, the costs may noticeably impact scripts or applications which frequently make only a few requests for each file open.

We designed the protocols with scalability in mind; in particular we expect all of them to handle large numbers of clients. Unfortunately, we have traces of the scientific application for only up to 256 nodes. Adding more clients adds more resources for client side operations, and we can add more OSD to handle load there, but the MDS require a high degree of coordination to maintain a consistent directory tree, so we cannot easily add MDS. Thus we believe the metadata servers will be the limiting factor in Ceph's ability to scale to tens of thousands of clients. To investigate scalability, we compute the time the MDS will spend on symmetric and public key operations to generate capabilities and tickets. Protocol 2 needs the MDS to create one symmetric ticket for each client-file pair; Protocol 3 needs one symmetric ticket for each client-disk pair, and one capability signed with a public key for each file (when we have a flash crowd). Protocol 4 needs one signed ticket for each client and one signed capability for each file.

Figure 6 shows the time the MDS spends generating tickets and capabilities when all clients open one common 1 GB file and each client opens one private 1 GB file; the files are spread across 1,000 OSD. For comparison, we include the "naïve" protocol which requires a symmetrically encrypted ticket for each client-object pair. We see that Protocol 2 requires the least effort from the MDS, however one must balance that consideration with it being the most risky of our secure protocols. Protocol 3 creates the most MDS load among our protocols, needing about 471 times more effort

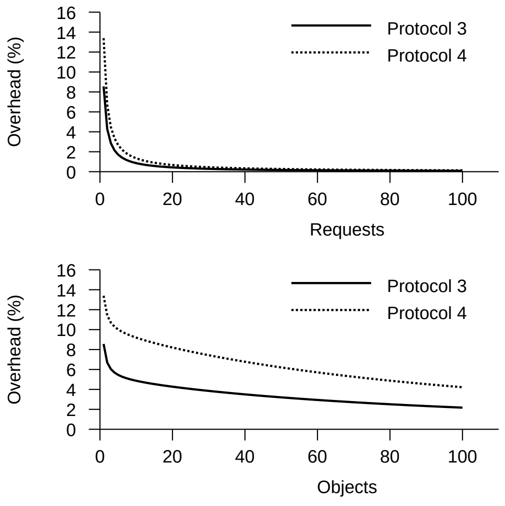

Figure 5: The overhead of Protocol 3 and Protocol 4 versus Protocol 2 for varying numbers of requests made to a single object (top), and for varying numbers of objects spread across multiple disks (bottom). Users which make a single request of a single object of one file will notice a 14% degradation over the more vulnerable protocol, but this penalty drops quickly as the user reads and writes more objects more often. Users which frequently access large parts of files can amortize the cost of obtaining the file capability over more requests.

than Protocol 2, though it's still not as demanding as the naïve protocol which needs about 1,000 times as much effort. Finally, the most secure option we have presented, Protocol 4 requires about 78 times as much work from the MDS.

6. FUTURE WORK

We have explored the trade-offs between different security protocols in this paper, but there are still several design options and issues that must be investigated to make Ceph and other petabyte-scale object-based storage systems secure.

In Protocol 4, the disk must encrypt the data for a read response using a different key for each client. Protocol 4 uses a key specific to the disk and client to authenticate both the sender and receiver of the request. However, in the reply it may be possible to use the key $K'_{C,D(H,i)} = \{\{K_0\}K^R_{D(H,i)}\}$, which is specific to only the disk. Since the MDS hands each client in a flash crowd the same K_0, the disk could cache the encrypted block so that it would be ready for subsequent read requests. We must investigate the security implications and performance effects of this optimization, which was inspired by similar work for NASD [6].

We expect that hardware support for IPSec will become common. Azagury, *et al.* proposed two-layer encryption using IPSec for NASD; however they assumed IPSec would provide keys and did not describe how their protocol integrates with ISAKMP [2]. We need to map our protocol to ISAKMP and experimentally measure the overhead with hardware support.

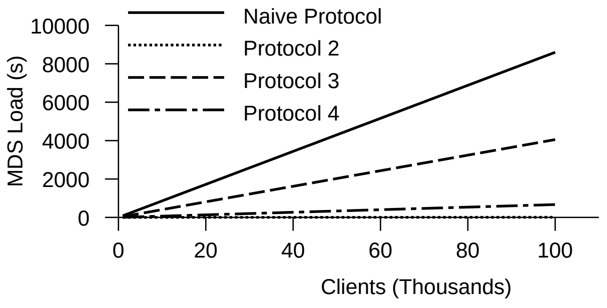

Figure 6: The time the MDS spends encrypting or signing tickets and capabilities. All clients open and access each object of one common 1 GB file, and each client opens and accesses each object of a private 1 GB file. The files are distributed throughout a pool of 1,000 OSDs. Protocol 4 requires 78 times more computation from the MDS than Protocol 2, though one should weigh that against its lower risk to subverted OSDs. Protocol 3 requires 471 times more work, and the naïve protocol requires 1,000 times more work.

In the event that an attacker does steal a disk, our protocols still leak the data on that disk since the data on the disk is stored in the clear. We are investigating encrypting data at the client and storing it encrypted on disk. This will provide confidentiality even for the data on a subverted disk, and it reduces the amount of encryption and decryption the disk must perform. However, end-to-end encryption of the data requires long-term keys that have the potential to create endless hassle for end users. We may use the user's public/private key pair to secure lock boxes as in SNAD [14] or techniques similar to those in Plutus [10], although in our system the MDS rather than the OSDs will maintain the lock boxes.

We are currently working on a formal statement of the security properties and a proof for the protocols. Also, we are working on a software implementation for experimental analysis.

7. CONCLUSIONS

We have described three protocols for securing the network traffic of a petabyte-scale object based distributed file system consisting of thousands of disks accessed by thousands of clients. This environment differs dramatically from conventional NASD and object-based environments in its scale: thousands of clients may access a single file simultaneously, and files may consist of 10^5–10^7 objects spread across thousands of object-based storage devices. In such an environment, our protocols perform much better than traditional NASD secure protocols.

Although the three protocols that we developed all outperform traditional NASD authentication protocols, we had thought that the use of public key cryptography would exact a high toll in performance, causing public-key based protocols to be significantly slower than protocols that used only symmetric key cryptography. However, by examining traces of a scientific application and analyzing performance on synthetic workloads, we found that protocols using public key cryptography did not significantly increase the overhead of using a secure protocol as compared to symmetric-key based protocols. Since public-key protocols offer better protection against attacks on the OSDs, we concluded that they were better suited for use in a petabyte-scale object-based storage system. We are continuing this work by investigating optimizations, using IPSec for secure channels, encrypting data on disk, and performing experimental analysis.

8. REFERENCES

[1] 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 *Proceedings of the Second USENIX Conference on File and Storage Technologies (FAST)*, pages 159–174, San Francisco, CA, 2003.

[2] 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 *IEEE Security in Storage Workshop*, pages 10–23, 2002.

[3] K. Fu, M. F. Kaashoek, and D. Mazières. Fast and secure distributed read-only file system. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 181–196, San Diego, CA, Oct. 2000.

[4] 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 *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 92–103, San Jose, CA, Oct. 1998.

[5] H. Gobioff, G. Gibson, and D. Tygar. Security for network attached storage devices. Technical Report TR CMU-CS-97-185, Carniege Mellon, Oct. 1997.

[6] H. Gobioff, D. Nagel, and G. Gibson. Embedded security for network attached storage. Technical Report TR CMU-CS-99-154, Carnegie-Mellon University, June 1999.

[7] E.-J. Goh, H. Shacham, N. Modadugu, and D. Boneh. SiRiUS: Securing remote untrusted storage. In *Proceedings of the 2003 Network and Distributed System Security Symposium*, pages 131–145. Internet Society, Feb. 2003.

[8] R. J. Honicky and E. L. Miller. Replication under scalable hashing: A family of algorithms for scalable decentralized data distribution. In *Proceedings of the 18th International Parallel & Distributed Processing Symposium (IPDPS 2004)*, Santa Fe, NM, Apr. 2004. IEEE.

[9] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. Wes. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, Feb. 1988.

[10] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus: scalable secure file sharing on untrusted storage. In *Proceedings of the Second USENIX Conference on File and Storage Technologies (FAST)*, pages 29–42, San Francisco, CA, Mar. 2003. USENIX.

[11] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, San Francisco, CA, Dec. 2004.

[12] D. Mazières, 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 '99)*, pages 124–139, Dec. 1999.

[13] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, Aug. 1984.

[14] E. L. Miller, D. D. E. Long, W. E. Freeman, and B. C. Reed. Strong security for network-attached storage. In *Proceedings of the 2002 Conference on File and Storage Technologies (FAST)*, pages 1–13, Monterey, CA, Jan. 2002.

[15] B. C. Neumann, J. G. Steiner, and J. I. Schiller. Kerberos: An authentication service for open network systems. In *Proceedings of the Winter 1988 USENIX Technical Conference*, pages 191–201, Dallas, TX, 1988.

[16] K. T. Pollack and S. A. Brandt. Efficient access control for distributed hierarchical file systems. In *Proceedings of the 22nd IEEE / 13th NASA Goddard Conference on Mass Storage Systems and Technologies*, Apr. 2005.

[17] B. C. Reed, E. G. Chron, R. C. Burns, and D. D. E. Long. Authenticating network-attached storage. *IEEE Micro*, 20(1):49–57, Jan. 2000.

[18] J. T. Regan and C. D. Jensen. Capability file names: Separating authorisation from user management in an internet file system. In *Proceedings of the Tenth USENIX Security Symposium*, pages 221–234. USENIX, Aug. 2001.

[19] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun network file system. In *Proceedings of the Summer 1985 USENIX Technical Conference*, pages 119–130, 1985.

[20] F. Wang, S. A. Brandt, E. L. Miller, and D. D. E. Long. OBFS: A file system for object-based storage devices. In *Proceedings of the 21st IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 283–300, College Park, MD, Apr. 2004. IEEE.

[21] S. A. Weil, K. T. Pollack, S. A. Brandt, and E. L. Miller. Dynamic metadata management for petabyte-scale file systems. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing (SC '04)*, Pittsburgh, PA, Nov. 2004. ACM.

[22] Q. Xin, E. L. Miller, T. J. Schwarz, D. D. E. Long, S. A. Brandt, and W. Litwin. Reliability mechanisms for very large storage systems. In *Proceedings of the 20th IEEE / 11th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 146–156, Apr. 2003.