Scalable Security for Large, High Performance Storage Systems

Andrew W. Leung
Storage Systems Research Center
University of California, Santa Cruz
aleung@cs.ucsc.edu

Ethan L. Miller
Storage Systems Research Center
University of California, Santa Cruz
elm@cs.ucsc.edu

ABSTRACT

New designs for petabyte-scale storage systems are now capable of transferring hundreds of gigabytes of data per second, but lack strong security. We propose a scalable and efficient protocol for security in high performance, object-based storage systems that reduces protocol overhead and eliminates bottlenecks, thus increasing performance without sacrificing security primitives. Our protocol enforces security using cryptographically secure capabilities, with three novel features that make them ideal for high performance workloads: a scheme for managing coarse grained capabilities, methods for describing client and file groups, and strict security control through capability lifetime extensions. By reducing the number of unique capabilities that must be generated, metadata server load is reduced. Combining and caching client verifications reduces client latencies and workload because metadata and data requests are more frequently serviced by cached capabilities. Strict access control is handled quickly and efficiently through short-lived capabilities and lifetime extensions.

We have implemented a prototype of our security protocol and evaluated its performance and scalability using a high performance file system workload. Our numbers demonstrate the ability of our protocol to drastically reduce client security latency to nearly zero. Additionally, our approach improves MDS performance considerably, serving over 99% of all file access requests with cached capabilities. OSD scalability is greatly improved; our solution requires 95 times fewer capability verifications than previous solutions.

Categories and Subject Descriptors

C.4.5 [**Performance of Systems**]: Performance attributes; D.4.3 [**Files Systems Management**]: Access methods; D.4.6 [**Security and Protection**]: Access controls

General Terms

security, performance

Keywords

scalability, capabilities, object-based storage

StorageSS '06, October 30, 2006, Alexandria, Virginia, USA.

1. INTRODUCTION

Recent architectures and designs of high performance commodity storage subsystems have aimed to improve scalability and performance. The object-based storage architecture [6] has facilitated large-scale, high-performance file systems through powerful storage abstractions and by decoupling data control and data access paths.

Current security protocols for object-based storage provide strong access control but are often unable to sustain high system performance in the face of demanding workloads. Our research group has developed an extremely scalable object-based storage system, Ceph [16], capable of managing petabytes of data under high performance workloads. As a result, traditional object-based security protocols act as a performance bottleneck and limit the scalability of such systems.

Object-based security protocols follow a capability model to manage access control [7]. Unforgeable capabilities are issued by a cluster of Metadata Servers (MDS) to clients who are requesting access to specific objects. The clients then present these capabilities to the Object Storage Devices (OSDs) when making data I/O requests. Finally, OSDs verify capability integrity and request permissions before returning data to the clients. Once a capability has been generated, acquired or verified it may be cached by the MDS, client or OSD, respectively. Future metadata or data requests that fall under the scope of a cached capability allow the MDS or OSD to bypass the generation and verification process, respectively.

Current security protocols issue fine grained capabilities which only express access rights for a single object and a single client. Large-scale storage systems, such as Ceph, have files which range over thousands of objects. In system such as these the MDS will have to generate thousands of capabilities per metadata request. High performance systems are also prone to hot files where certain files are commonly accessed by thousands of clients within milliseconds. These hot files force the MDS to generate thousands of unique capabilities for each object being accessed by each client in a short period of time, which causes an extreme performance penalty.

To ensure integrity OSDs must verify each non-cached capability when presented with a client data request. This impacts performance by introducing a computation cost and a latency to each new data request. Additionally, large numbers of capabilities have a negative effect on capability cache hit ratios. This is due to the constant high flux of capabilities in and out of the client, MDS and OSD caches.

Therefore, current security protocols result in a generally high overhead and extreme performance degradation under heavy workloads.

We present a new security protocol which provides secure access privileges while sustaining high system performance and low overhead even in the face of extremely demanding workloads. Our protocol employs powerful, coarse grained capabilities with an expressiveness equal to a collection of many finer grained capabilities. Our capabilities are valid for variable-sized groups of clients rather than individual clients. Additionally, our capabilities can range over a number of files, rather than a single object. To enhance the expressiveness of our capabilities we employ different methods of client and file grouping. These grouping methods reduce the need for the MDS and OSDs to generate and verify capabilities, respectively. This also aides in capability caching by reducing the flux of capabilities in the system.

Capability integrity is maintained with public key signatures rather than a keyed hash. A keyed hash requires a shared secret between the MDS and all, possibly tens or hundreds of thousands of, OSDs. This introduces a major security risk as well as complicates key management. Finally, we adopt a novel revocation strategy which allows us to quickly revoke access rights from clients without introducing complicated capability management techniques or impacting system performance.

We have implemented a prototype of our security protocol. We evaluated our prototype using a high performance file system workload. Our performance analysis demonstrates the ability to significantly reduce, to near zero, client latencies due to security. Additionally, our results show a minimal security overhead at the MDS with greater than 99% of all capabilities generations being eliminated. Our experiments shows the ability to achieve 95 times fewer OSD capability verifications than previous solutions.

The remainder of this paper is organized as follows. In Section 2 we discuss background and Section 3 discusses related work. In Section 4 we discuss coarse grained capabilities and present our security protocol in Section 5. Section 6 gives a description of our capability revocation strategy and Section 7 provides a performance analysis of our protocol. We discuss future work in Section 8 and summarize and conclude in Section 9.

2. BACKGROUND

Several object-based storage systems have already been released in the commercial market [4, 13, 11]. Our research group has developed a new object-based file system, Ceph [17], capable of scaling to peta- and exabytes of data while providing unparalleled performance and reliability. Ceph achieves scalability by removing file allocations tables and relying on intelligent object storage devices (OSDs) to distribute complex data accesses, update serialization, replication, reliability and recovery. A cluster of Metadata Servers (MDS) efficiently handles large and frequent metadata accesses. Large-scale systems, such as Ceph, may contain tens or hundreds of thousands of OSDs, each of which may contain several disk drives. Additionally, these systems may have tens or hundreds of thousands of clients.

Figure 1 demonstrates the architecture and security model for Ceph and other object-based storage systems. The security model assumes the MDS is a trusted reference monitor, OSDs are trusted principals and clients are untrusted principals. Commonly, the MDS is able to be physically secured and protected. Though due to the size of large-scale systems, such as Ceph, it is often infeasible to physically secure each OSD or client workstation. As a result, OSDs and clients are extremely vulnerable to attack by intruders.

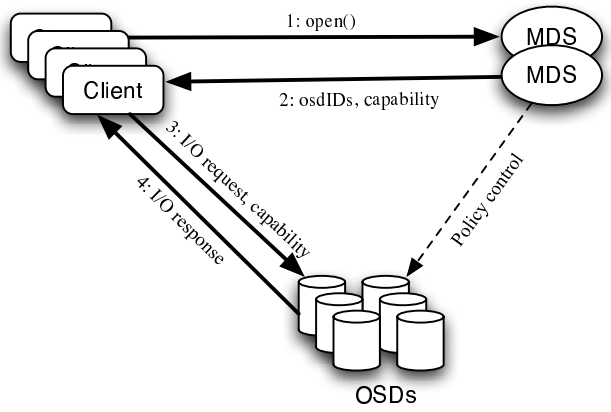

Figure 1: The Ceph architecture and security model. A client who wishes to access a file requests a capability from the MDS. The MDS returns the capability and locations of the objects in the file (osdIDs). The client presents the capability to an OSD along with an I/O request. The OSD verifies the capability and services the request.

Ceph is designed for large-scale scientific supercomputing infrastructures. These infrastructures commonly run parallel scientific applications which require extremely high-performance metadata and I/O support [15]. These applications have heavy I/O demands with data accesses of varying size. Access patterns are extremely bursty with frequent flash crowds and hot files. This can result in extreme cases where hundreds of thousands of clients are concurrently accessing the same small set of files within milliseconds of each other.

Due to the need for scalable load balancing, Ceph employs a unique data distribution algorithm, CRUSH (Controlled Replication Under Scalable Hashing) [17], which efficiently maps objects to OSDs in a decentralized, pseudo-random fashion. CRUSH reliably places and replicates objects based on physical composition and potential sources of failure, spreading replicas over many failure groups. Data distribution is controlled by a hierarchical *cluster map* which represents the available storage resources and describes device layout and associated weights. For reliability and load balancing, objects for a single file are commonly placed across thousands of OSDs using *placement rules* which define replication and placement policies. Additionally, CRUSH efficiently handles changes in the cluster map, such as new or failed devices, by remapping an optimal fraction of objects to new storage devices.

An important feature of CRUSH is that it is completely distributed, meaning any client, MDS or OSD can independently calculate the location of any object in the system, given the cluster map, placement rules and an input x which defines distinct storage targets. Therefore, when presented with a file handle, an OSD can easily calculate whether it contains any objects for that file handle.

3. RELATED WORK

Security often comes at the price of performance and scalability. Previous work has aimed at providing flexible security protocols which are often not designed with performance and scalability in mind, particularly on the level which we desire. Other solutions which have aimed at improving security performance often require sacrificing security primitives.

Many existing protocols use keyed hashes to enforce capability integrity [3, 5, 12, 1]. This requires a unique shared secret between the MDS and each OSD. As a result, capabilities may only be valid over a single OSD, since integrity can only be verified by a single shared key. This approach requires the MDS to generate many capabilities in systems in which files may span many OSDs. Therefore, these fine-grained, per-OSD capabilities presents a bottleneck in high performance systems which are heavy in metadata and I/O requests by requiring the MDS to produce an extremely large number of capabilities. Additionally, many protocols issue capabilities at the granularity of a single object [3, 5, 12]. These protocols force the MDS to generate a unique capability for each object in the file, regardless of whether the objects reside on the same OSD. This high flux of capabilities in and out of local MDS, OSD, and client caches reduces overall cache hit ratios which incurs additional computation time and latencies.

We adopt a solution similar to the solution developed by Olson and Miller [10], which uses public key signatures to enforce capability integrity. Though computing a public key signature is far slower than computing a keyed hash, it allows a single capability to span any number of OSDs as long as each OSD knows the public key of the MDS. Moreover, subverting any number of OSDs will not allow an intruder access to objects on any other OSD since the public key of the MDS cannot be used to forge capabilities. We will later show how we take further steps to reduce any overhead introduced by using public key signatures.

Some solutions employ fine grained capabilities which allow the system to bind security and cache coherency by having capabilities serve as both distributed locks and access tokens. Though this eliminates the need for additional synchronization mechanisms, we feel these two concepts are distinctly unique enough to warrant separation. Incorporating hard bound synchronization with more loosely bound access capabilities needlessly limits flexibility. Additionally, applications are ultimately responsible for managing coherency though this is not the case for access control. We feel an efficient solution would be to push synchronization management to the OSDs who have more substantial control over object access.

To improve performance and reduce MDS load, the SCARED protocol [12] allows clients to grant capabilities which are restricted subsets of the capabilities that they have already been granted. Though this alleviates some MDS capability requests and subsequent capability generations, it fails to enforce confinement in a system in which clients may not be trusted or may become compromised.

The protocol presented by Azagury, *et al.* [3] assumes the use of an authenticated secure channel, such as IPSec. Therefore, the protocol allows OSDs to use channel authentication rather than client authentication. Because the protocol does not do client authentication it suffices for the OSD to verify that whoever is on the other end of the channel has a valid capability and knows some special secret. This allows for easy and simple delegation of capabilities from one client to another. Though much like SCARED, the performance benefits of capability delegation are shrouded by resulting insecurities.

Gobioff [7] introduced the use of batched capabilities to reduce client latency by contacting the MDS less frequently and requesting several capabilities per contact. Clients batch capability requests into a single large request to the MDS. Though this requires clients to access the MDS less frequently each client must still wait until it has accumulated enough capability requests to contact the MDS and the MDS must still generate a unique capability for each request. Additionally, batching capability requests guarantees that the MDS will always have to handle a large number of capability requests in a short period of time, which negatively impacts MDS performance.

Gobioff also proposed indirect objects to allow a capability to name many objects. An indirect object is an ordered list of objects which are located on the same OSD. A capability grants access to a single indirect object which, in turn, grants access to all objects named in the indirect object's ordered list. This does allow multiple objects to be named in a single, small sized capability though it only allows a capability to name objects on the same OSD, meaning the MDS must still generate thousands of capabilities for a file which spans thousands of OSDs. Moreover, the MDS must write thousands of indirect object lists when files span thousands of OSDs. Using CRUSH, our protocol allows each OSD to deterministically identify all objects it contains for a file given the file handle.

Aguilera, *et al.* [1] introduced a capability based protocol for block level storage. They propose using capabilities that specify access over ranges of blocks (extents) or explicit non-contiguous blocks. Because these capabilities are self-describing they may result in exceedingly large capabilities when blocks are not located contiguously on disk. Additionally, these capabilities can only work over a single disk and thus require the MDS to generate many capabilities for files which span many disks.

Security in LWFS [9] also supports coarse grained capabilities. Their solution utilize *containers* of objects to include multiple objects into a single capability. Though rather than allow the MDS to dynamically associate objects with containers, each object is implicitly bound to a single container. All objects in a container must share the same access policies, which makes it difficult to modify the access rights of a single object and complicates dissociating an object from a particular container. Additionally, their capabilities do not explicitly name clients to which the capability applies meaning they are unable to enforce confinement in systems in which the network is not fully secured.

Singh, *et al.* [14] suggested using a trust framework to reduce security overhead and increase performance. Their protocol requires the MDS to monitor and record the correctness of each clients metadata accesses. This allows the MDS to assign a level of trustworthiness to each client. Clients are deemed trustworthy when they rarely try and access files using privileges they do not have. Once a client is deemed trustworthy the MDS informs the OSDs that they no longer need to verify capabilities from that client. Though this approach aides performance by allowing OSDs to skip the capability verification process for some clients, it introduces a major security risk. In order for a malicious client to access

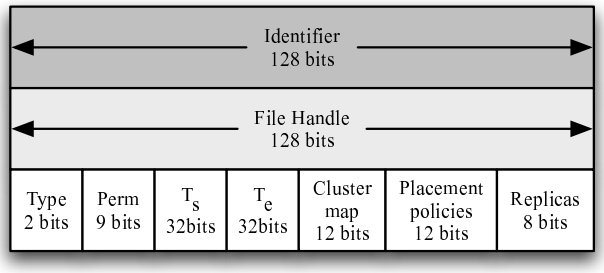

Figure 2: The structure of a capability. The capability is cryptographically secured with a public key signature by the MDS. The first field describes the user or group for whom the capability is valid. The second field describes the file handles to which the capability grants access. The *type* specifies the naming convention, T_s and T_e define the capabilities lifetime and *perm* specifies the privileges. *ClusterMap*, *PlacementPolicies*, and *Replicas* are all CRUSH specific fields.

a restricted object the client must simply 'behave' for long enough to gain trust from the MDS. Once it is deemed trustworthy it may access any object in the system regardless of access privileges. Additionally, this protocol only reduces overhead at the OSDs, allowing the MDS to still act as a bottleneck.

Olson and Miller [10] proposed a protocol which allows capabilities to extend over all objects in a file and across OSDs. Olson's protocol utilizes a placement algorithm, similar to CRUSH, which allows OSDs to deterministically identify if it contains objects for a given file handle. Additionally, this protocol uses public key signatures rather than keyed hashes to ensure integrity. Though this protocol is better suited for large-scale, high performance environments it still incurs a high overhead. The protocol makes little effort to reduce the number of expensive public key signatures and verifications performed at the MDS and OSDs, respectively. Capabilities are issued at a per-client granularity, therefore, hot files can have a high impact on system performance. Additionally, their protocol does not facilitate a capability revocation policy.

4. COARSE GRAINED CAPABILITIES

We present a protocol which employs capabilities which are general enough to grant multiple clients access privileges to multiple files using a single capability. Doing so reduces the number of capabilities the MDS must sign and generate, as well as, the number of capabilities the OSDs must verify. This reduces client request latencies and load on both the MDS and OSDs. Issuing fewer capabilities also reduces churn in client, MDS, and OSD caches, as well as, lessens network traffic.

High performance applications and workloads often have very bursty, hot file oriented access patterns. Additionally, file accesses are often highly correlated, resulting in clients accessing files in a common pattern. This allows us to encapsulate access privileges for many clients and files into a single capability rather than a capability for each client-file or client-object pair.

It is important to note, the MDS never issues a coarse capability which grants any client access privileges to a file which they should not have access to. Therefore, in a coarse grained capability, all clients in the capability have access privilege to all files in the capability according to the systems reference monitor.

Figure 2 shows the structure and contents of our coarse grained capability. Two 128 bit fields identify the clients and files the capability describes. This gives us the ability to either explicitly name individual clients or files or use more sophisticated naming techniques, such as hashes, ranges, or lists.

4.1 Managing Coarse Grained Capabilities

When the MDS is constructing a capability, simply including a list of clients or file handles, which may be long, in a capability can result in extremely large capabilities, which are inefficient for caching, passing on the network and encryption. Therefore, coarse grained capabilities contain group and file list identifiers rather than explicit client or file names. When presented with capabilities during I/O requests the OSDs need a way to translate these identifiers into lists of clients and/or files.

To remedy this, we use a method similar to indirect objects, described by Gobioff [7]. A file, stored in the system, will contain all identifiers and their associated client or file list. This *identifier to list map file* is stored as a normal file in the system, though it is signed by and may only be modified by the MDS. To save space locally the OSDs may want to maintain only a subset cache of all of the lists in the system.

When an OSD wishes to learn about clients or files associated with an identifier it may request the location of the map file from the MDS. The MDS must authenticate the OSD before returning the address of the file and a capability to access it. Once an OSD knows the location of the file, it may access lists for every identifier in the system. This method allows the OSD to query the list without having to involve the MDS.

In order to create identifiers for client and file lists we use fixed-size Merkle trees [8], also known as hash trees. Merkle trees provide flexibility for creating additional, and appending to, client and file lists. A Merkle tree is a tree of hashes in which the leaves are hashes of, in our case, client names or file handles. Each node further up the tree is a hash of its respective children. The root node of the tree is a fixed-size hash based on all other hash values in the tree.

Using Merkle trees it is easy to construct new, long lists of clients or files inexpensively. This will greatly reduce the cost of updating or adding to client groups or file lists. When a long list of clients or files needs to be included in a capability the MDS may simply join already existing Merkle trees to produce a new root hash. This fixed-size root hash is included in the capability. Then updating the identifier to list map involves simply joining several Merkle trees, which saves both significant space in the identifier to list map and MDS computation time.

When presented with a capability an OSD will contact the map file stored in the system which associates the root hash in the capability with a list of clients or files. This is useful because each branch of a Merkle tree represent another associated list. This allows the MDS and OSDs to easily cre-

ate, join or disconnect Merkle trees to form dynamic lists of client names or file handles without heavy re-computation.

Since the OSD may cache only a subset of the map file, the OSD will need to contact the map file remotely when presented with a capability containing a root hash it does not recognize. The OSD will be able to store far more lists in less space with Merkle trees. Rather than caching a long list for each new hash value the OSD sees, it can store new hash values as combinations of other hash values it has already stored. By being able to compactly store many lists locally, the OSD greatly increases its chances of an identifier getting a local cache hit.

Methods for defining optimal group and file list definitions will be largely dependent on the design and semantics of the system. As a result, we leave the decision of how to define groups and file lists to the system designers and administrators. In the following section we will present some definitions we feel are useful for achieving descriptive coarse grained capabilities.

4.2 Defining Groups

We propose two approaches for describing groups of clients in a coarse grained capability. The first approach uses a relatively static group definition based on UNIX user groups. UNIX groups provide a descriptive way to define a large, related set of clients. Also, UNIX groups are relatively static, requiring few changes once group identities are known.

In UNIX, and therefore in POSIX, all files have permissions for three categories: *user*, *groups* and *global*. If a file is accessed by any client named in *groups* then the MDS may cache and return a single capability granting permissions to all clients named in *groups*. Because all clients in *groups* have the same access privileges to a file, a single capability will suffice for the entire group. Any subsequent requests for that file from any other clients in *groups* will result in the cached capability being returned.

If any clients not named in *user* or *groups* requests access to the file the MDS may cache and return a capability granting permissions to all clients as specified by the *global* permissions. Therefore, the MDS will only have to generate at most three unique capabilities per file, pending any file permission changes.

The second approach is to use dynamic groups which are created on-the-fly. This is based on the notion that files are accessed in a bursty, hot file oriented fashion and a single request for a file is often indicative of future requests. These groups describe, exclusively, the clients who request access to a file.

As the MDS generates and signs a capability for a file, it buffers, rather than services, all similar requests for that file. Once the MDS has returned the capability, it gathers all buffered requests for a file and includes those clients into a single capability. While the MDS is generating this new capability it can likewise buffer any further requests received for the file. Therefore, the MDS is able to batch all client requests received during the lengthy capability generation and signature process into a single capability, greatly reducing the number of capabilities which need to be generated. Another important feature is that it does not introduce any additional latencies since batching is done while the MDS is busy generating and signing capabilities.

A key tradeoff between the two approaches is that UNIX groups require a UNIX or POSIX compliant system. This limits the amount of heterogeneity that can exist amongst nodes in the system. Also, UNIX groups cannot maintain a notion of fine-grained access control. Another tradeoff is on-the-fly groups hinge on the notion that files are accessed in a bursty pattern. If files are not accessed in such a way this approach is often unable to encapsulate many clients in a group. Also, frequently changing groups require the OSDs to spend more time inquiring about group members.

4.3 Defining Files

We propose a method for including a list of files in a capability based on a file's likelihood to be accessed in the near future. To do this we employ a very accurate prediction algorithm, Recent Popularity [2].

The Recent Popularity algorithm has strong stability benefits and quickly adapts to changes in accesses. Recent Popularity stores the k last observed successors to a file. It selects a possible prediction for the most popular file and if that selection occurs at least j times then Recent Popularity offers that selection as a successor prediction.

Accuracy is important for establishing long chains of successor files in a capability. For example, suppose it is the case that accessing file A usually results in an access to file B, which results in an access to file C and then an access to file D. Then C and D are along the causal path of likely successors of A and should be included in the capability as such, in addition to B.

5. A HIGH PERFORMANCE SECURITY PROTOCOL

In this section, we formally introduce our scalable object-based security protocol.

Notation

We will write $A \rightarrow B : M$ to signify a message, M going from principal A to principal B. Unless the channel between A and B is secured, an attacker may freely eavesdrop the channel, capture packets, modify packets and replay packets. When principal A receives a message M, A is unable to make any assumptions about the freshness of M.

To denote public and private keys for principal A, we will use K_A^U and K_A^R, respectively. To denote a shared secret key between principals A and B, we will use K_{AB}. We will write encryption of message M with A's public key as $\{M\}K_A^U$. Encryption of message M with A's public key results in M being readable only to A. The notation $\{M\}K_{AB}$ encrypts message M with symmetric key K_{AB} which makes M unreadable to anyone besides A or B, as well as, authenticates either principal A or B.

The notation $\langle M\rangle K_A^R$ denotes message M being signed by the private key of A. The signature can be verified with A's public key, K_A^U. To denote user U making a request through client workstation C we simply use C.

Assumptions

We assume encryption and signature algorithms are sufficiently hard to break. Additionally, we assume private keys will be kept secret by their respective principals and that public keys are well known to all principals in the system.

We assume the MDS properly serves metadata to clients. Additionally, we assume the MDS acts as a trusted reference monitor for granting access control through capabilities. We

also assume the OSDs are trusted principals and properly serve data to clients. We assume the clients are untrusted principals and that both the clients and OSDs are vulnerable to attack.

Protocol

Message exchanges:

$$C \rightarrow M \quad : \quad C, \text{request ticket} \tag{0.1}$$

$$M \rightarrow C \quad : \quad \{\mathcal{T}\}K_{CM} \tag{0.2}$$

$$C \rightarrow M \quad : \quad C, \{\text{open}(path, mode)\}K_{CM} \tag{0.3}$$

$$M \rightarrow C \quad : \quad \{\mathcal{C}\}K_{CM} \tag{0.4}$$

$$C \rightarrow D(H,i) \quad : \quad \begin{array}{l}\{K_{C,D(H,i)}\}K^U_{D(H,i)}, \\ \{\mathcal{T}, \mathcal{C}, \text{read}(bno, i, H), T_1\}K_{C,D(H,i)}\end{array} \tag{0.5}$$

$$D(H,i) \rightarrow C \quad : \quad \{T_1, data\}K_{C,D(H,i)} \tag{0.6}$$

where

$$\begin{aligned} \mathcal{T} &= \langle C, K^U_C, K_0, T'_s, T'_e\rangle K^R_M \\ \mathcal{C} &= \langle G, L, type, perm, T_s, T_e\rangle K^R_M \\ G &= hash(S_1, S_2) \\ L &= hash(S_3, S_4) \\ K_{C,D(H,i)} &= \{\{K_0\}K^U_{D(H,i)}\}K^R_C \end{aligned}$$

This protocol employs a secure channel between the client and MDS using shared secret, K_{CM}. In message 1, the client first requests a ticket which is used to authenticate the client to each OSD. The ticket contains the clients id, C, the clients public key, K^U_C, an initialization key, K_0, and a ticket lifetime, T'_s to T'_e. The ticket is signed by the MDS and returned in message 2.

Though generating and signing the ticket may be expensive, the MDS need only generate a ticket once for each client in the system. Assuming tickets have a relatively long lifetime, the ticket will be refreshed infrequently and message 1 and message 2 will not often occur.

In message 3, the client requests a capability from the MDS for file *path*. The MDS identifies the appropriate clients and files to include in the capability using the methods previously discussed. The MDS then generates the root hashes G and L which are the group and file identifiers, respectively. Both G and L are hashes of the two Merkle subtrees S_1 and S_2 and S_3 and S_4, respectively. These subtrees are constructs inherent to building Merkle trees and may or may not already be known to the OSD. The *type* field of the capability defines how group and file list identifiers are to be interpreted and therefore its semantic meaning is relative to the specific system. The *perm* field describes the permissions granted by the capability. The capabilities lifetime is T_s to T_e. The capability is signed by the MDS and returned in message 4.

CRUSH yields the identifier of the OSD to contact for the i^{th} object of file H, represented by $D(H,i)$. Using the initialization key, the OSD's public key and the clients private key, the client is able to generate a session key for secure communication with each OSD. In message 5, the client sends the session key, $K_{C,D(H,i)}$, to the OSD encrypted with the OSD's public key. Once a secure session is established with the OSD, the first part of message 5 will not be necessary. Because the session key is derived from information contained in the ticket and unchanging public and private keys, the secure channel between the client may be used until the client is issued a new ticket. Therefore, though generating a session key and establishing a secure channel may be expensive, it will happen no more frequently than ticket renewal, which is seldom.

In the second part of message 5 the client presents the OSD with the ticket, the capability, the data request and a timestamp to prevent replay of the message. The data request specifies the block *bno* of the object i of file H. This is all encrypted using the per-OSD session key.

If the capability, in message 5, names a client then the OSD verifies that the client name matches the ticket. If the capability names a group identifier then the OSD verifies that the client named in the ticket exists in the group associated with the identifier. The OSD does so by first checking for a list associated with the group identifier in it's local cache. If no such list exists the OSD must consult the identifier to list map. Because this list is simply a file in the system the OSD must contact the MDS for access to the file in a manner very similar to the protocol specified above. For this reason, we leave its description out of the message exchanges above. Once the OSD has verified that the client in the ticket is the client in the capability or is in the group associated with the identifier it may begin resolving the capability's file handle or file identifier.

Matching file identifiers to lists is very similar to the group matching process just described, thus we will omit its description. Once the OSD has verified the capability names the file handle specified in the data request it resolves the file handle to a series of object ids using CRUSH. The OSD then returns data for object i in message 6. The returned data also includes a timestamp and is encrypted with the session key.

5.1 Security Analysis

Here we asses the risks certain attacks pose on our protocol and discuss any security implications coarse grained capabilities may introduce.

Risks

If an attacker were to subvert a client workstation, then the attacker would obtain K_{CM} and K^R_A. This allows the attacker to impersonate the user, U, associated with C, to the MDS and OSDs. The attacker would be able to gain access to all objects that U could access. Unless K_{CM} is occasionally refreshed to preserve perfect forward secrecy, an attacker would be able to view all previous messages between the client and the MDS, as well as, all previous messages between the client and OSDs.

If an attacker were to subvert an OSD, then the attacker would obtain K^R_D for that OSD. Additionally, an attacker would obtain $K_{C,D(H,i)}$ for all clients who have recently contacted the OSD. The attacker would then be able to impersonate the OSD to the MDS and any client. Additionally, the attacker would have access to all previous messages between all clients and that OSD. If the attacker fully subverts the OSD, rather than just compromising secret or private keys, the attacker may be able to view, modify or delete any objects located on that OSD.

Security of Coarse Grained Capabilities

Here we show, informally through contradiction, that coarse grained capabilities preserve fundamental security primitives.

Assume it is possible for the protocol described above to grant an attacker access to a restricted file through coarse grained capabilities which yield access to multiple files. An

attacker may exploit these capabilities by obtaining a capability in which it is named and trying to use it to access restricted files. This amounts to the attacker choosing a file handle and querying an OSD, using the obtained capability, to see if the capability grants access to that file. An attacker will gain access to the file if and only if the OSD verifies that the capability indeed permits access to the file. Because capabilities are unforgeable, the MDS must have generated the capability. Therefore, the previous attack is equivalent to the attacker querying the MDS for access to the chosen file in a system which names a single file per capability. In order for the attacker to gain access to a restricted file in the second system the MDS must grant the attacker a capability which provides access to a restricted file. Because the MDS is assumed to be a trusted reference monitor, with an ACL at the granularity of files, the MDS will never grant capabilities providing access to restricted files. Therefore, the MDS will never issue a coarse grained capability which grants access to a restricted file.

Again, assume it is possible for the protocol described above to grant an attacker access to a restricted file through coarse grained capabilities which provide multiple clients access to a file. An attacker can exploit these coarse grained capabilities by obtaining a capability in which it is named and delegating the capability to other clients whom are accomplices. These accomplices may attempt to access the file by querying an OSD for access to the file. Again, capabilities are unforgeable and must be generated by the MDS. An accomplice will gain access to the file if and only if the OSD verifies that the capability indeed permits the accomplice access to the file. Therefore, the previous attack is equivalent to the accomplice querying the MDS for a capability to access the file in a system where capabilities describe access for only a single client. In order for the accomplice to access the file in the second system the MDS must issue a capability that grants the accomplice access to the file. Because the MDS is a trusted reference monitor, with an ACL at the granularity of clients, the MDS will never grant capabilities providing unauthorized clients access to a restricted file. Therefore, the MDS will never issue a coarse grained capability which grants an unauthorized client access to a restricted file.

6. CAPABILITY REVOCATION

It is very difficult to enforce any strict methods of capability revocation when capabilities may be distributed and shared amongst thousands of clients and OSDs. If access permissions for a file change it may affect a capability owned by many clients and cached at many OSDs. The same applies if a client has its access rights modified. As a result, we take a novel approach to capability revocation and simply use short-lived capabilities. This limits the possible window of vulnerability to the lifetime of the capability.

As a corollary, capabilities will expire quicker and clients will need to request additional capabilities, burdening both the MDS and OSDs. To resolve this we implement capability lifetime extensions. An extension is a document, generated and signed by the MDS, stating a new, extended expiration time for a capability.

When a client notices a capability is nearing its expiration the client may request a lifetime extension from the MDS. Assuming the client is still authorized for the capability, the MDS will generate, cache, and return this extension to the client. The client then presents this extension to the OSDs, who verify and cache it. Capabilities which are held by many clients will cause the MDS to receive multiple extension requests for the same capability. By caching capability extensions the MDS only needs to generate a single extension for all of the clients who hold a capability.

The MDS can also batch extension requests for different capabilities and return clients an extension which extends the lifetime of many capabilities. This is useful when the MDS receives many extension requests, for different capabilities, in a short period of time. Because extensions simply name capabilities and a new expiration time, there is no reason the MDS cannot name multiple capabilities in a single extension. This allows the MDS to reduce the number of lifetime extensions it needs to generate and sign, as well as, the number of extension verifications that the OSDs need to perform.

7. PERFORMANCE EVALUATION

We evaluate three different criteria which demonstrate the effectiveness of our protocol to efficiently handle large-scale systems and high performance workloads. First, we analyze how effective our protocol is at reducing client latency. Second, we examine the ability of our protocol to increase MDS scalability and reduce requests. Finally, we gauge how our protocol aides OSD capability verification.

We developed a prototype implementation of our protocol. We used UNIX groups as the basis for group establishment and used Recent Popularity to predict file successors. We chose UNIX groups because our benchmark was a POSIX compliant UNIX file system trace and thus UNIX groups reflect real file system semantics. We selected Recent Popularity because of its ability to produce extremely accurate predictions. We used three and six as the Recent Popularity parameters j and k, respectively.

We implemented versions of our protocol which use only groups without file prediction and only file prediction without groups. We also implemented a protocol which does not use groups or file prediction which will serve as our baseline comparison. We also developed a protocol which issues capabilities at the granularity of a single object and individual clients for additional comparison. For this we used a simple and reserved algorithm for mapping files to objects resulting in files containing a relatively limited number of objects.

A high performance object-based, cluster file system trace from Lawrence Livermore National Lab was used as our workload benchmark. The workload consisted of 256 clients each running the same high performance simulation application. Each client issues approximately 90 MDS requests and more than 1,800 I/O requests.

Our experiments were conducted using three separate machines. The client and OSD machines were both Dell OptiPlex SX270s with 2.80 GHz Intel Pentium 4 CPUs. The MDS was a Dell OptiPlex SX280 with a 3.2 GHz Intel Pentium 4. All three machines were connected through a high speed LAN, and each had one gigabyte of main memory and ran Fedora Core 3 Linux, version 2.6.12-1.

7.1 Latency Analysis

We ran the high performance benchmark on our prototype and measured the latency of each client `open()` request sent to the MDS. We compared this to the same workload run over our protocol when using either group or file prediction

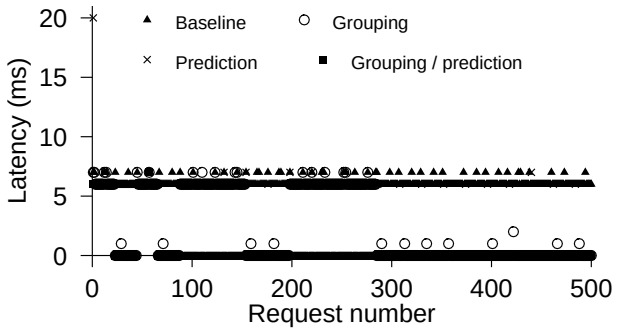

Figure 3: The latency for the first 500 `open()` requests made to the MDS for protocols with groups and file prediction, just groups, just file prediction and no groups or file prediction.

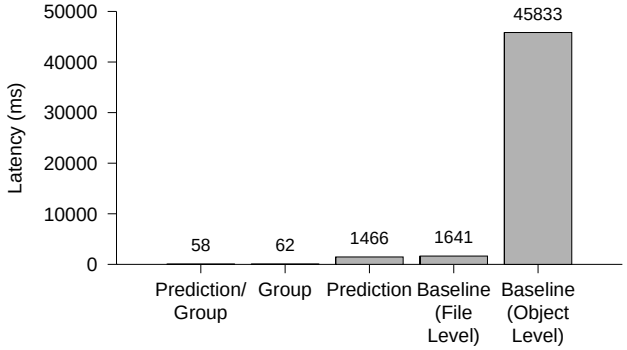

Figure 4: The aggregate client latency experienced for a hot file. Each client issued slightly more than 7,500 `open()` requests, except for the object level protocol which issued more than 110,000 requests.

capabilities only, as well as, a protocol which issues capabilities at the granularity of single files and individual clients. This identifies the role each part of the protocol plays in achieving scalability. We also tested a protocol at the granularity of objects however this results in a much larger volume of total requests and does not make for a good direct comparison. A client side capability cache was utilized for each client in the experiment.

Figure 3 shows the latency breakdown for client `open()` requests to the MDS. Latencies are generally either zero or approximately six to seven milliseconds. Zero latency requests generally translates to cache hits at the MDS. Requests with greater latencies are generally misses which force the MDS to generate a capability and do a lengthy signature operation per miss.

Our protocol, using both groups and file prediction, almost immediately drops to zero latency per request. This is because our prediction algorithm is quickly able to accurately predict future file accesses and include them in the capability. As a result, all requests from a client, after the first, will result in the request being serviced by the MDS cache. With UNIX groups, clients only experience this single latency when they happen not to share a UNIX group with any client who has previously sent a request to the MDS.

The group-only and file prediction-only protocols also perform extremely well, though not as well as when they were used in conjunction. In the group only protocol, clients only experience latencies when they are the first member of any of their UNIX groups to make a MDS request. Though these clients will experience a latency for the duration of their requests, all subsequent clients in the same group will have zero latency requests.

Our protocol, using both groups and file prediction, almost immediately drops to zero latency per request. This is because our prediction algorithm is quickly able to accurately predict files and include them in the capability. As a result, all requests from a client, after the first, will result in the request being serviced by the MDS cache. With UNIX groups, clients only experience this single latency when they happen not to share a UNIX group with any client who has previously sent a request to the MDS.

The group only and file prediction only protocols also perform extremely well, though not as good as when they were used in conjunction. In the group only protocol, clients only experience latencies when they are the first member of any of their UNIX groups to make a MDS request. Though these clients will experience a latency for the duration of their requests, all subsequent clients in the same group will have zero latency requests.

In the prediction only protocol, clients experience a latency on every MDS request until the MDS is able to confidently make file access predictions following the requests of j clients. Afterwards, every client's first MDS request results in a latency while all subsequent requests do not.

The baseline protocol without groups or file prediction performs radically worse than the previous protocols. Using the baseline protocol, every client request has a latency. This is because all requests that could be serviced by the MDS cache are instead serviced by the client's local cache. As a result, all requests that cannot be fulfilled by the client's local cache also cannot be serviced by a capability in the MDS cache.

We compared our protocol to a traditional protocol at the object granularity. This data is not reflected in our numbers because it requires almost 80,000 additional MDS requests making it difficult to compare on the same scale. Though because this protocol utilizes keyed hashes rather than digital signatures, latencies are generally only several hundred microseconds rather than five or six milliseconds. By comparison our protocols aggregate client latency was 2,575 ms on average while aggregate latency for the traditional object protocol was 822,476 ms.

Figure 4 shows the aggregate client latency to service a hot file scenario that consisted of more than 7,500 `open()` requests for a single file from 256 clients. Our protocol required a minuscule 58 ms to service all client requests. This means our protocol produced client latencies which were on average approximately 7 μs. The group only protocol also performed extremely well, servicing all of the clients in 62 ms.

The file prediction only protocol did not perform nearly as well, requiring 1,466 ms to service all clients. This is due to the fact that the hot file scenario in our experiment only issued requests for a single file, thereby negating any positive effects of a file prediction protocol. The baseline

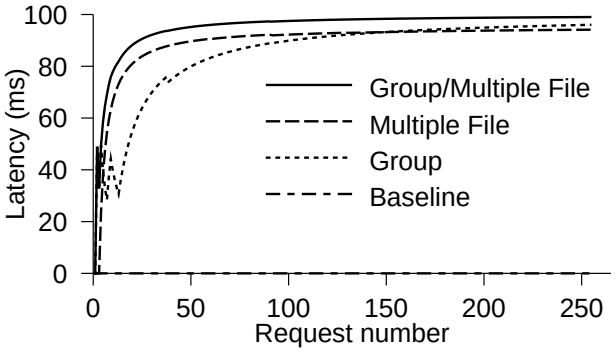

Figure 5: The percentage of `open()` requests received by the MDS which were serviced by capabilities in the MDS cache.

protocol without groups or file prediction performed about the same, taking 1,641 ms to service all clients.

When the object level protocol was tested it performed much worse, taking 45,831 ms to service all of the client requests. Though on average client requests only took 400 μs to service, the sheer number of requests resulted in an extremely high aggregate latency.

7.2 MDS Scalability and Load Reduction

Figure 5 shows the MDS capability cache hit percentages for all requests issued by the workload. Our protocol performed exceptionally well, having serviced 48% of the capabilities from the cache after only 1% of the workload has completed. By the time 7% of the workload has completed 90% of the MDS requests have been serviced by the cache. In total, of all of the approximately 5,600 requests received, only 55 were not serviced by capabilities in the MDS cache.

The group only and prediction only protocols also both performed extremely well, though neither protocol climbs in cache hit percentage as fast as our combined protocol. Each serves approximately 95% of all requests from the MDS capability cache. The file prediction algorithm has served 75% of all requests from the capability cache with only 5% of the total workload run. The group protocol reaches the same mark after 16% of the workload has completed.

The baseline protocol, which does not employ groups or file prediction performs poorly, not producing a single cache hit at the MDS. This is due to the need for exclusiveness between the MDS cache and a client's cache.

Figure 6 demonstrates the effectiveness of the client's cache at reducing MDS load for object and file level protocols. The client's cache is able to absorb about 75% of all of the client's requests, for both object and file level protocols, under our benchmark.

Though the percentage of requests served by the client's cache is the same across the two protocols, the difference in the absolute number of requests sent to the MDS is quite daunting. The file level protocol issues about 5,600 requests to the MDS while the object level protocol issues about 109,000. Therefore, despite the fact that a keyed hash integrity computation is faster than a public key signature, far more keyed hash computations are being performed which rapidly negates the performance difference.

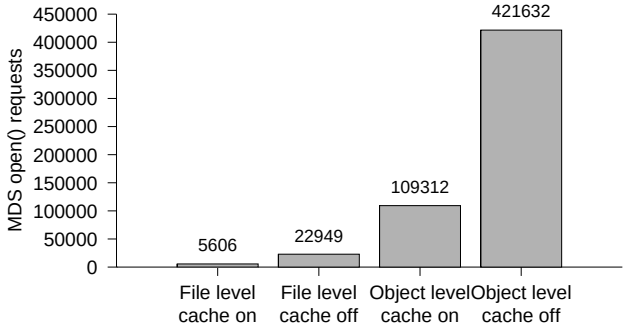

Figure 6: The effectiveness of the client cache and file level protocols at reducing the requests sent to the MDS.

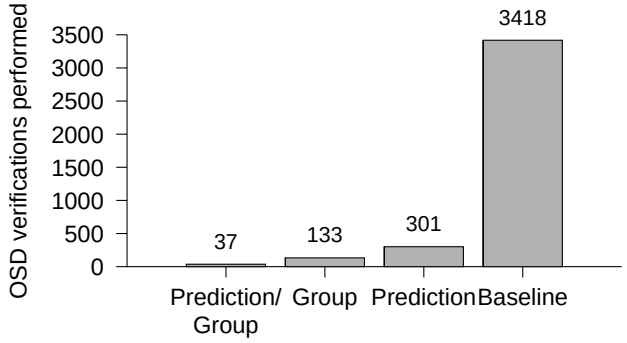

Figure 7: The aggregate number of verifications done by the OSD for more than half a million I/Os.

7.3 OSD Scalability

We ran the high performance benchmark on our prototype and measured the number of capability verifications the OSD was forced to do. We compared this to the same workload using our baseline protocol without group or file prediction capabilities, as well as, group only and prediction only protocols.

Figure 7 shows the aggregate number of verifications done by the OSD. Our protocol requires only 37 OSD verifications for over a half a million I/Os. However, this is greatly due to the fact that the workload issued I/Os to a very small set of files relative to the total number of I/Os. Nonetheless, our protocol drastically out performs the baseline protocol without groups or prediction.

The benchmark produced 133 verifications and 301 verifications with the group only and prediction only protocols, respectively. The baseline protocol required 3,418 OSD verifications. This is nearly 95% more capability verifications than is required with our protocol.

8. FUTURE WORK

The protocol we developed aims to reduce the time spent generating, signing and verifying capabilities. Unfortunately, some of our solutions incur both a space and time overhead. For example, a successor lookup table must contain $n + 3$ rows, where n is the Recent Popularity parameter for a reasonably sized subset of all files in the system. In a large-scale system this table has the potential to become exceedingly

large. This can waste MDS memory and make the table difficult to search. Though we believe this approach is sophisticated in its ability to generate accurate successor chains, we would like to know how it fares against other approaches.

Also, protocols which use fine-grained capabilities are also able to use capabilities as a way to enforce cache coherency. Our coarse grained capabilities will not be able to make such an enforcement. Though Ceph uses object-range locking to maintain cache coherency, we must factor this cost into the total cost of our coarse-grained capability protocol. We would like to quantify the costs of using an additional object-range locking scheme in addition to our security protocol.

We feel our protocol is well suited for some POSIX extensions currently being examined, particularly group opens, shared file descriptors, and group locking. We would like to examine our protocol in these contexts, both to understand our own protocols performance and to provide insight into the value of these extensions. We believe this will aide in both the development of our protocol as well as the propagation of these extensions into everyday use.

9. CONCLUSIONS

We presented a new scalable security protocol for large, high performance storage systems. In contrast to previous work, our protocol can handle both extremely large systems and high demand workloads. By extending the granularity of capabilities we are able to make them far more expressive. Through groups and prediction we take advantage of this expressiveness to reduce many of the MDS and OSD capability generation and verification scenarios. This in turn, reduces both computation times and latencies. To enforce revocation of access privileges we utilized short-lived capabilities and allowed for efficient capability lifetime extension.

We implemented a prototype of our protocol and evaluated its performance and scalability. We found we were able to quickly reduce client latencies for requests to near zero. The MDS managed to service more than 99% of all requests using capabilities from its cache. Our prototype required 95 times fewer OSD capability verifications than previous solutions.

Hence, we believe our protocol is a practical and efficient way to achieve security in large, high performance storage systems. Furthermore, we feel our protocol offers performance benefits to other smaller, block-based systems as well, liberating these systems from many of their own security protocol performance bottlenecks.

Acknowledgments

We would like to thank our Lawrence Livermore National Laboratory, Los Alamos National Laboratory, and Sandia National Laboratory both for sponsoring this work and for providing valuable mentoring. We would also like to thank Martín Abadi and all of the members of the Storage Systems Research Center, whose support and advice helped guide us through this work.

10. REFERENCES

[1] Aguilera, M. K., Ji, M., Lillibridge, M., MacCormick, J., Oertli, E., Andersen, D., Burrows, M., Mann, T., and Thekkath, C. A. Block-level security for network-attached disks. In *Proceedings of the Second USENIX Conference on File and Storage Technologies (FAST)* (San Francisco, CA, 2003), pp. 159–174.

[2] Amer, A., Long, D. D. E., Pâris, J.-F., and Burns, R. C. File access prediction with adjustable accuracy. In *Proceedings of the International Performance Conference on Computers and Communication (IPCCC '02)* (Phoenix, Apr. 2002), IEEE.

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

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

[5] Factor, M., Nagle, D., Naor, D., Riedel, E., and Satran, J. The OSD security protocol. In *Proceedings of the 3rd International IEEE Security in Storage Workshop* (2005), pp. 29–39.

[6] Gibson, G. A., Nagle, D. F., Amiri, K., Butler, J., Chang, F. W., Gobioff, H., Hardin, C., Riedel, E., Rochberg, D., and Zelenka, J. A cost-effective, high-bandwidth storage architecture. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (San Jose, CA, Oct. 1998), pp. 92–103.

[7] Gobioff, H. *Security for a High Performance Commodity Storage Subsystem*. PhD thesis, Carnegie Mellon University, July 1999. Also available as Technical Report CMU-CS-99-160.

[8] Merkle, R. C. *Secrecy, authentication, and public key systems*. PhD thesis, Stanford University, 1979.

[9] Oldfield, R. A., Maccabe, A. B., Arunagiri, S., Kordenbrock, T., Riesen, R., Ward, L., and Widener, P. Lightweight I/O for scientific applications. Tech. rep., Sandia National Laboratories, SAND2006-3057, May 2006.

[10] Olson, C. A., and Miller, E. L. Secure capabilities for a petabyte-scale object-based distributed file system. In *Proceedings of the 2005 ACM Workshop on Storage Security and Survivability* (Fairfax, VA, Nov. 2005).

[11] Panasas. http://www.panasas.com.

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

[13] Schwan, P. Lustre: Building a file system for 1000-node clusters. In *Proceedings of the 2003 Linux Symposium* (July 2003).

[14] Singh, A., Gopisetty, S., Duyanovich, L., Voruganti, K., Pease, D., and Liu, L. Security vs performance: Tradeoffs using a trust framework. In *Proceedings of the 22nd IEEE / 13th NASA Goddard Conference on Mass Storage Systems and Technologies* (2005).

[15] Wang, F., Xin, Q., Hong, B., Brandt, S. A., Miller, E. L., Long, D. D. E., and McLarty, T. T. File system workload analysis for large scale scientific computing applications. In *Proceedings of the 21st IEEE / 12th NASA Goddard Conference on Mass*

Storage Systems and Technologies (College Park, MD, Apr. 2004), pp. 139–152.

[16] Weil, S. A., Brandt, S. A., Miller, E. L., Long, D. D. E., and Maltzahn, C. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)* (Seattle, WA, Nov. 2006).

[17] Weil, S. A., Brandt, S. A., Miller, E. L., and Maltzahn, C. CRUSH: Controlled, scalable, decentralized placement of replicated data. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (SC '06)* (Tampa, FL, Nov. 2006), ACM.