Scalable Security for Petascale Parallel File Systems

Andrew W. Leung Ethan L. Miller Stephanie Jones
Storage Systems Research Center, University of California, Santa Cruz, CA 95064, USA
{aleung,elm,snjones}@cs.ucsc.edu

ABSTRACT

Petascale, high-performance file systems often hold sensitive data and thus require security, but authentication and authorization can dramatically reduce performance. Existing security solutions perform poorly in these environments because they cannot scale with the number of nodes, highly distributed data, and demanding workloads. To address these issues, we developed Maat, a security protocol designed to provide strong, scalable security to these systems. Maat introduces three new techniques. *Extended capabilities* limit the number of capabilities needed by allowing a capability to authorize I/O for any number of client-file pairs. *Automatic Revocation* uses short capability lifetimes to allow capability expiration to act as global revocation, while supporting non-revoked capability renewal. *Secure Delegation* allows clients to securely act on behalf of a group to open files and distribute access, facilitating secure joint computations. Experiments on the Maat prototype in the Ceph petascale file system show an overhead as little as 6-7%.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Access controls; D.4.3. [**File Systems Management**]: Distributed file systems

General Terms

performance, security

Keywords

secure object-based storage, capabilities, high-performance computing, scalability

1. INTRODUCTION

The demand from science, research, and business for large, high-performance storage has risen in recent years. High performance computing (HPC) scientific applications such as physical and chemical simulations have demanding I/O patterns, and their large data sets require terabytes to petabytes of storage. Businesses such as Google and Yahoo! also have HPC applications, such as MapReduce [7], that require giant web indices or image archives, placing a heavy load on storage infrastructure. Securing such large-scale HPC storage systems is an important challenge because of the large number of users and potentially sensitive data stored on them. For example, scientific research data stored on large-scale storage systems can include highly classified data; even unclassified data, such as simulations of drug effectiveness or geologic survey analysis data for oil drilling, can be worth millions of dollars. However, existing large-scale storage systems largely ignore security, using (at most) advisory security techniques to restrict access to data. Unfortunately, this approach has led to unprivileged access to data and subsequent litigation and government investigations.

To secure I/O in petascale parallel file systems, it is insufficient to use traditional distributed access control techniques that are designed for smaller systems with general or random workloads. Petascale file systems may service tens of thousands of clients and storage devices and must support I/O patterns that are highly parallel and very bursty [31]. These two factors increase the cost of traditional security techniques, which are often based on pairwise associations between clients and storage devices, and can result in weak security. There have been numerous efforts to secure distributed storage, but most were not designed for the size and demand of large HPC systems [1, 3, 8, 12, 20, 25]. These solutions were designed for general workloads with a modest number of relatively small files and requests, and most were intended for systems with a limited number of clients and storage devices. When incorporated into large parallel file systems, current approaches either degrade performance or rely upon weaker security mechanisms.

Existing protocols perform poorly in large parallel file systems because they do not scale well—the number of security operations is strongly tied to the number of devices and requests. For example, existing approaches may issue a capability for every block or object being accessed (*i. e.*, a capability authorizes a single block or object I/O) [12]. While this approach works well in smaller systems, it does not scale to petabyte-scale systems, in which files containing terabytes of data are striped across thousands of devices and accessed by thousands of clients. A single access to a 1 TB file striped into 1 MB objects would need a million capabilities; it is impractical to return that many capabilities when a file is opened. Worse, *each* client accessing the file would require these capabilities; while each client might use

This paper will be published in the *Proceedings of SC07*, Reno, Nevada, November 11–16, 2007.

a different subset, capabilities would have to be sent to the proper locations in advance because capability sharing by clients is difficult or non-existent in current systems.

To address these shortcomings, we designed and implemented Maat, a strong security protocol designed to scale to petabyte-scale parallel file systems. In developing Maat, we reconsidered traditional I/O security techniques with the goal of allowing security to scale to large systems with very demanding workloads. This paper describes the mechanisms that Maat uses to provide scalable I/O security for: preventing unauthorized data access, revoking user access privileges, and safeguarding against common security threats such as spoofing, replay, and man-in-the-middle attacks.

Maat introduces three scalable security techniques. First, access control is enforced through *extended capabilities*, an extension of traditional capability tokens, that can authorize I/O for any number of clients to any number of files. For example, a single extended capability may authorize a read or write operation for a hundred clients to any block in each of a hundred files. By authorizing access at the granularity of files and aggregating many authorizations into a single capability, Maat is able to greatly reduce the number of capabilities needed. Second, *automatic revocation* makes it possible to revoke a client's access privileges without the need to explicitly contact any clients or storage devices by giving capabilities short lifetimes. As a result, revoking a capability can be done by allowing the capability to expire—no explicit notification to storage devices is required. Continued use of valid capabilities is handled by a renewal protocol that extends the lifetimes of batches of existing capabilities with minimal overhead. This paradigm shifts the cost of revocation to renewal, where it can be handled in a more scalable fashion. Third, *secure delegation* allows for scalable cooperative computation and I/O, a common feature of HPC workloads. A single client generates a temporary asymmetric key pair and opens a file on behalf of the public portion of the key pair. The private portion of the key pair is distributed to other clients who use it to access the file without having to receive any additional authorization. The use of a temporary key pair shifts security from an insecure opaque capability to the possession of a secure private key. Maat's secure delegation provides an efficient and secure implementation of the proposed POSIX HPC I/O extensions `openg()` and `openfh()` [35].

We implemented Maat in Ceph [33], a petabyte-scale, high-performance distributed file system. Experiments both with and without security show that Maat is able to achieve strong security on Ceph while incurring less than 7% overhead for high performance workloads. Additionally, Maat has little impact on latency and throughput, allowing secured Ceph to achieve nearly the same performance as insecure Ceph operation.

2. BACKGROUND

There are a number of parallel file systems that have been developed recently and are in use today [4, 10, 11, 21, 27, 29, 33]. Most of these file systems consist of three main components: the client, a metadata server cluster (MDS), and a cluster of storage devices, such as network-attached disks or object storage devices (OSD). A key concept behind this design is the decoupling of metadata and data paths. Clients communicate all namespace operations, such as `open()`, to the MDS and all file I/O operations, such as `read()` and `write()`, to the storage devices.

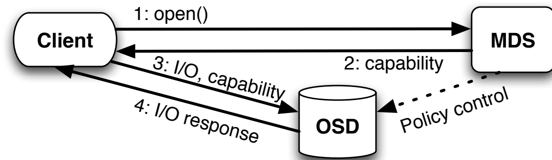


Figure 1: Parallel file system architecture and security flow. Clients request capabilities from the MDS and use them to request I/O from storage devices.

A result of this design is that storage devices have no implicit knowledge of access privileges or authorizations because this information is stored at the MDS. Thus, the MDS must communicate authorizations to storage devices via capabilities—communicable tokens of authority [18]. Before any storage device can authorize an I/O, a client must receive a capability authorizing the I/O from the MDS and present it to the storage device with the I/O request. The MDS cryptographically hardens the capability with a digital signature or HMAC to guarantee to the storage device that the capability was not forged or altered. Figure 1 demonstrates the architecture and security flow in most parallel file systems. The trust model assumes the MDS is a trusted oracle and reference monitor. The storage devices are trusted to store data and only perform I/O for authorized requests. No implicit trust is placed on clients.

2.1 What's Special About Petascale File Systems for HPC?

Petabyte-scale distributed file systems used for high performance computing (HPC) are quite different from the somewhat smaller file systems for which most security systems were developed. Petascale file systems are a much more challenging environment to secure for the following reasons:
Data is large and highly distributed. Files in large-scale file systems are often extremely large, containing gigabytes or terabytes of data, and can be striped across thousands of devices [31]. Previous security protocols distribute capabilities at the granularity of a block or object, requiring the generation of thousands or even millions of capabilities. Some security systems have tried to alleviate this by issuing capabilities that grant access to all file data on a device. Though this helps, thousands of capabilities must still be used to access a large file which is striped over thousands of devices, thus creating high load for servers generating capabilities, latency for clients opening files, high load for data servers verifying capabilities, and latency for clients performing I/O.
Many clients and storage devices. Large-scale HPC systems may have tens of thousands of clients and storage devices, increasing the cost of many security operations. For example, changing file permissions becomes extremely expensive when thousands of devices must be contacted to revoke a capability. Doing so quickly and reliably is often impractical in large-scale systems.
Demanding I/O and access patterns. Parallel file systems are designed to support HPC workloads with demanding access patterns that create worst-case performance scenarios for existing security solutions. File access and I/O are both extremely bursty and highly parallel [31], creating workloads in which thousands of clients accessing a single

file within seconds is common.

Added threat environment. Physical security for tens of thousands of clients and storage devices may not be feasible. Additionally, it is unlikely that network security can be strictly enforced across the whole network, invalidating assumptions of secure communication in previous solutions. For example, a system in which the simple possession of a capability is sufficient to authorize I/O can fail on insecure networks where attackers can easily eavesdrop the network and obtain capabilities.

2.2 Existing Parallel and Distributed File System Security

Parallel and distributed file systems feature disks or OSDs connected directly to the network. Because of this model's inherent vulnerabilities, there has been a great deal of research in providing secure I/O in these systems. Though many of these solutions have been successful in smaller systems, most do not perform well in the large-scale, demanding environment for which Maat was designed.

Granting a capability at the granularity of a block or an object is often too expensive, even for smaller systems. As a result, prior approaches have authorized access rights for multiple blocks or objects with a single capability. NASD [12], the T10 OSD protocol [32], LWFS [23], and SnapDragon [1] all allow a capability to authorize I/O to a group of objects that reside on the same storage device. Restricting access to a single device does not significantly decrease the number of capabilities required when files are striped across thousands of devices. Additionally, these grouping strategies often require manual specification or are dependent on parameters such as on-disk layout, further limiting capabilities from authorizing I/O to many objects.

In many systems [1, 8, 12, 25, 37], an HMAC is used to provide a guarantee that a capability was generated by the MDS and has not been tampered with. An HMAC requires a shared key between the MDS and the storage devices that recognize the capability. While HMACs are simple to generate (they do not require public-key encryption), they are too insecure to be used in large-scale systems. If an attacker compromises a single storage device, the attacker gains the key shared between the MDS and any storage devices that share the same HMAC key, allowing the attacker to impersonate the MDS, the system's trusted oracle and reference monitor, to any of those devices. To alleviate this insecurity, keys are often shared only between the MDS and a single disk, eliminating the threat of impersonation. This approach restricts a capability to authorizing I/O on only a single device, since only a single device can verify the capability. This prevents a capability from authorizing I/O to large groups of blocks or objects which may reside on multiple devices. SNAD [20], Plutus [16], Olson and Miller [24], and Leung and Miller [17] all use public key cryptography to secure access to files. SNAD and Plutus use public key cryptography to make write operations externally verifiable, while Olson and Miller use it to ensure a capability's integrity. Using public key cryptography for capability integrity allows a capability to span any number of storage devices because each device must simply know the MDS's public key. This also adds security because a subverted device will not be able to spoof any other device. The downside is public key cryptography is orders of magnitude slower than shared key cryptography, potentially introducing high performance penalties in the demanding HPC environments in which Maat operates.

Previous work has tried to avoid some of these performance issues by relaxing security constraints or relying on an existing security infrastructure. Azagury, *et al.* [3] assumes the existence of a network-level security infrastructure, such as IPSec, allowing them to authenticate secure channels rather than clients. LWFS [23] also assumes a secure transport layer allowing it to ignore potential replay and eavesdropping of capabilities. However, most real-world systems do not employ strong network or transport security. This is due to the high overhead of encrypting and decrypting all traffic and the difficulty of enacting a key infrastructure across the entire network, thus limiting the scope of these solutions. Singh, *et al.* [30] employ a trust framework based on a client's past trustworthiness, requiring the MDS to monitor correctness of client accesses. Clients who rarely make incorrect accesses are deemed trustworthy, and no effort is made to ensure that their accesses are valid. While this framework does improve performance for trustworthy clients, an obvious attack would be to gain trust through a series of correct I/O requests and subsequently misbehave.

Access to a file is not secure unless that privilege can be expunged. As file data becomes larger and the number of devices increase, thousands of devices may need to be contacted to revoke a single user's access or change a file's permissions. Explicitly contacting every storage device that contains data for a file is not scalable. Moreover, it is difficult to guarantee that the messages are received at their destinations, meaning that the system "fails unsafe": the default behavior is to *allow* access, with revocation denying subsequent accesses. Systems such as NASD [12] and SCARED [25] use object version numbers for revocation. Capabilities in these systems authorize I/O to a specific object version; thus, changing the object version acts to invalidate all capabilities for that object. With this method, revoking access to an entire file requires incrementing the version number of every object in the file, which may be millions of objects. SnapDragon [1] uses a similar approach, though capabilities rather than objects are versioned. Other systems use similar methods with revocation lists [26], backpointers [23], and key re-distribution [13], all of which require explicit messaging to all of the storage devices.

To mitigate the cost of explicitly contacting all storage devices which may hold a specific capability, Cepheus [9], SNAD [20], and Plutus [16] suggest the use of lazy revocation. When permissions change, access is revoked on the first write operation rather than immediately revoking access. This approach allows the immediate cost of a permission change to be deferred until the first write request, but it also allows a revoked user to continue to read the file data until it is overwritten.

3. Maat DESIGN AND IMPLEMENTATION

Maat provides strong, scalable access control using extended capabilities and automatic revocation and supports cooperative computation with secure delegation. This section discusses the concepts behind these techniques and how Maat addresses them.

3.1 Design Assumptions and Notation

Maat has been implemented in the Ceph petascale, high performance, distributed file system [33], allowing Maat to

make some simplifying assumptions. First, Maat assumes that all storage devices are object storage devices, and are thus intelligent devices with a CPU, network interface, local cache, and a number of underlying disks [32]. Maat also assume that each OSD can associate local object IDs with global file identifiers: using CRUSH [34], an OSD can map a global file identifier to the object IDs and locations of the objects that contain the file data. Thus, Maat can issue capabilities that identify *files*, which OSDs can later associate with object IDs in I/O requests. We also assume that clients act as proxies for users. More specifically, each client acts on behalf of a number of users, each of whom can be uniquely identified. Throughout, whenever we refer to a client, we are referring to a user acting through a client proxy. Finally, we assume a secure synchronized clock protocol to keep time relatively synchronized across nodes. Although these assumptions are slightly restrictive, they hold true in many parallel file systems. Additionally, making several concessions to the current Maat design will make it possible to port Maat to systems which do not meet these assumptions.

Securing a petascale storage system requires a protocol with many messages; to ensure that the content of the messages is clear, we will use a standard notation to describe the messages throughout this section. The notation $A \rightarrow B : M$ denotes a message, M, sent from principal A to principal B. The public and private keys of principal A are denoted as K_A^U and K_A^R, respectively. To denote a shared secret key between principals A and B, we use K_{AB}. The encryption of message M with A's public key, K_A^U will be written as $\{M\}K_A^U$; This makes M unreadable to anyone who does not possess A's private key. The notation $\{M\}K_{AB}$ denotes the encryption of M with shared key K_{AB}; again, this makes M unreadable to anyone who does not posses K_{AB}. $\langle M \rangle K_A^R$ denotes a message M signed with principal A's private key, allowing any principal with access to A's public key to verify that A "vouched for" the content of M. The hash of message M is denoted using $hash\langle M \rangle$. An HMAC uses a similar notation, $hash\langle M, K_{AB} \rangle$, where a shared key K_{AB} is hashed in addition to M, allowing any principal that knows K_{AB} to verify that the message source also knows the secret key and that M was not modified in transport. Finally, the letters C, M, and D will be used to represent a client, MDS, and OSD, respectively.

3.2 Authentication

Authentication in Maat requires each principal to have a public/private key pair. We assume that all principals know the authenticated public key of every MDS and OSD. Before entering the system, each client creates a public/private key pair, K_C^R and K_C^U, and a shared key, K_{CM}, and shares the public and shared keys with the MDS. When a client "logs into" the system, it receives a signed ticket $\mathcal{T}$ that verifies the authenticity of the client's public key using an approach similar to that of authentication server tickets used in Kerberos [22]. The ticket, shown in Figure 2(b), also contains an initialization vector and an expiration time.

Once a client has received a ticket, it negotiates a unique shared key K_{CD} with each OSD, as shown in Figure 2(a). The ticket's initialization vector, the OSD's public key, and random data are hashed to generate the shared key. Maat uses shared keys rather than public/private keys because of the dramatic performance benefits, though, unlike a shared key between the MDS and some number of OSDs, subverting a client-OSD shared key does not allow any additional principals to be spoofed. Clients securely distribute keys to each OSD, who then verify correct receipt by responding to a nonce challenge with a second nonce $nonce'$. The protocol in Figure 2(a) is done infrequently—no further negotiations between clients and OSDs need be done until the ticket is refreshed and the initialization vector is changed.

$$C \rightarrow M : request_ticket, T_s, hash\langle request, T_s, K_{CM} \rangle$$
$$M \rightarrow C : \mathcal{T}$$
$$C \rightarrow D : \{\langle K_{CD}, T_s, nonce \rangle K_C^R\} K_D^U, \mathcal{T}$$
$$D \rightarrow C : nonce', hash\langle nonce', K_{CD} \rangle$$

(a) Messages sent to establish a shared key between a client C and an OSD D. In Message 3, an OSD extracts the client-disk shared key K_{CD} by decrypting it using its private key and authenticating the message source using the public key in $\mathcal{T}$. An OSD confirms correct receipt by responding with a nonce challenge $nonce'$.

$$\mathcal{T} = \langle ID_U, K_C^U, IV, T_s, T_e \rangle K_M^R$$
$$K_{CD} = hash\langle IV, K_D^U, random_data \rangle$$

(b) Definitions for a ticket $\mathcal{T}$ and client-disk shared key K_{CD}. $\mathcal{T}$ contains the user's ID (ID_U), public key (K_C^U), initialization vector (IV), and the ticket's lifetime. K_{CD} is computed by hashing IV with the OSD's public key and random data.

Figure 2: Protocol to negotiate a shared client-disk key in Maat.

Though tickets are refreshed infrequently, refreshing a ticket requires the client to migrate all shared keys to use the new initialization vector. To improve the performance of the resulting key re-negotiations, the MDS provides the client with the new initialization vector *prior* to actually refreshing the ticket, allowing the client to renegotiate shared keys during slack time rather than re-negotiating all keys at once. When the ticket is "formally" refreshed, the client will have migrated most, if not all, of its shared keys to the new initialization vector, so few keys would need to be negotiated at ticket refresh.

3.3 Extended Capabilities

Access control is the primary contributor to security overhead because the number of capabilities and their resulting cryptographic overhead tends to scale up as systems or workloads become larger. To reduce capability overhead, Maat introduces the notion of an *extended capability*: a capability able to authorize I/O for any number of clients to any number of files. Extended capabilities are conceptually equivalent to the I/O authorizations of many traditional capabilities aggregated into a single data structure. For example, a traditional capability may state "user a has read access to object y", while an extended capability may say "users a, b, and c have read access to files x, y and z." By combining permissions, the MDS can generate fewer capabilities because a single extended capability can replace multiple traditional capabilities. This change also means that OSDs need to verify fewer capabilities because more I/O requests can be done using a single, previously verified, capability. It should be noted that, while an extended capability can authorize many I/Os, access control is still at the granularity of a single file; individual files may be included or excluded from a single extended capability. This is in contrast to

$$C \rightarrow M : \text{open}(path, mode), T_s, hash\langle \text{open}(path, mode), T_s, K_{CM}\rangle$$
$$M \rightarrow C : \mathcal{C}, hash\langle \mathcal{C}, K_{CM}\rangle$$
$$C \rightarrow D : \mathcal{C}, \text{read}(oid), T_s, hash\langle \text{read}(oid), T_s, K_{CD}\rangle$$
$$D \rightarrow C : \{data, T_s\}K_{CD}$$

(a) Protocol to open and read a file in Maat. Each message has an HMAC allowing its source and contents to be verified. File data may be encrypted in transit to prevent eavesdropping.

$$D \rightarrow C : \text{update}(RH), T_s, hash\langle \text{update}(RH), T_s, K_{CD}\rangle$$
$$C \rightarrow M : \text{update}(RH), T_s, hash\langle \text{update}(RH), T_s, K_{CM}\rangle$$
$$M \rightarrow C : \mathcal{H}$$
$$C \rightarrow D : \mathcal{H}$$

(b) Protocol to retrieve a Merkle tree in Maat. A client forwards update requests to the MDS when it does not have $\mathcal{H}$ cached locally; thus Messages 2 and 3 only occur when the client does not have the tree cached.

$$\mathcal{C} = \langle U, I, ID_C, mode, T_s, T_e\rangle K_M^R$$
$$\mathcal{H} = \langle RH, tree\rangle K_M^R$$

(c) Contents of a capability $\mathcal{C}$ and a signed Merkle tree $\mathcal{H}$. In $\mathcal{C}$, U and I are the root hashes of authorized users and files, respectively, and ID_C is a unique capability identifier. $\mathcal{H}$ contains the Merkle tree associated with root hash RH.

Figure 3: Protocols using extended capabilities.

coarse-grained access control models [15] in which access is granted at the granularity of a set of files. The MDS enforces the rule that all authorizations in a extended capability are legal according to the MDS's access control matrix.

Extended capabilities do not alter the I/O security model presented in Section 2; the primary change is that the MDS may intelligently insert additional I/O authorizations when it generates a capability, as discussed in Section 3.3.3. The protocol used by the client to open and read a file is shown in Figure 3(a). After generating a capability, the MDS caches it, bypassing the expensive capability generation process for subsequent `open()` requests from clients that the MDS has already pre-authorized; this activity on the MDS is transparent to the client. Similarly, when an OSD verifies an extended capability in response to an I/O request, it caches the results of the signature verification, allowing the OSD to bypass signature verifications for I/O requests which use previously verified capabilities [36]. By authorizing many I/Os in a single capability, extended capabilities increase the number of cache hits at the MDS and allow the OSDs to bypass more capability verifications.

3.3.1 Securing Extended Capabilities

To ensure that capabilities cannot be forged or altered, Maat secures them using public-key cryptography. Each capability is signed by the MDS's private key, allowing anyone who knows the MDS's public key to verify its integrity and authenticity. Public key cryptography is used, rather than shared key cryptography, for three reasons: convenience, security, and affordability. Any OSD can verify a capability, conveniently allowing a capability to authorize I/O for a file that may reside on thousands of different OSDs. For this to be possible with shared keys, the MDS must share a common key with all OSDs. If this key were to become compromised, which may not be uncommon in very large systems, the attacker can spoof the MDS and any OSD. With public key cryptography, however, obtaining the MDS's public key does not allow an attacker to forge signatures. Though Maat pays a cost for the convenience and security of public key cryptography, this cost for a small number of cryptographic operations is amortized across thousands of I/O requests by caching the results of both capability generation at the MDS and verification at the OSDs. Thus, Maat can afford to use public key cryptography even though it is orders of magnitude slower than shared key cryptography, because it dramatically reduces the number of capability generations and verifications.

Ensuring integrity is not sufficient, however—extended capabilities must also ensure that simply obtaining a capability, *e. g.* via eavesdropping an unencrypted network, does not allow unprivileged data access. Such breaches can occur in "pure capability systems," in which simple possession of a capability is sufficient to grant access. Maat confines I/O authorization by *explicitly* naming all authorized users and file identifiers in the capability. When an OSD verifies a capability, it checks that the authenticated user making the request is named in the capability and that the file identifier in the capability maps to the object ID being read or written, thus preventing an attacker from using a capability to perform any unauthorized operations.

3.3.2 Making Extended Capabilities Fixed Size

Extended capabilities can become very large when they must explicitly state all authorized users and files. Since large capabilities can consume lots of cache space and are inefficient for frequent network transmission, Maat creates small, fixed size capabilities using Merkle hash trees [19]. Capabilities identify authorized users and files via the root hash of a Merkle tree constructed from the user IDs or file identifiers, with the root hash acting as a unique, fixed size identifier for all elements in the tree. Merkle trees are used because each inner node of the tree is itself the root hash of a sub-tree corresponding to a subset of the original tree. This allows easy composition and decomposition of new Merkle trees, as illustrated in Figure 4.

A result of using root hashes in capabilities is that when an OSD first receives a capability, it does not know which users or files are authorized by the capability because it does not know the user IDs or file identifiers associated with a root hash. Maat uses an update protocol, shown in Figure 3(b) to allow an OSD to query the requesting client for the Merkle tree associated with a root hash. Clients retrieve and cache the signed[1] Merkle tree from the MDS. By caching the tree, clients ensure that all subsequent update requests (*i. e.*, another OSD verifying the capability) do not burden the MDS. Once the OSD has cached the tree, it ensures that any future uses of the root hash do not require an update. Additionally, messages 2 and 3 in Figure 3(b) only occur when a client does not have the Merkle tree, $\mathcal{H}$, associated with the root hash, RH. Thus, a capability which authorizes

[1]The Merkle tree signature has subsequently been removed from the protocol, as it is not necessary. Each root hash represents a unique Merkle tree, therefore, a tree cannot have the correct root hash unless it is the correct tree. This makes forgery pointless and tampering obvious. The signature has been included here because it is reflected in our benchmarking results and removing it would lead to inconsistencies.

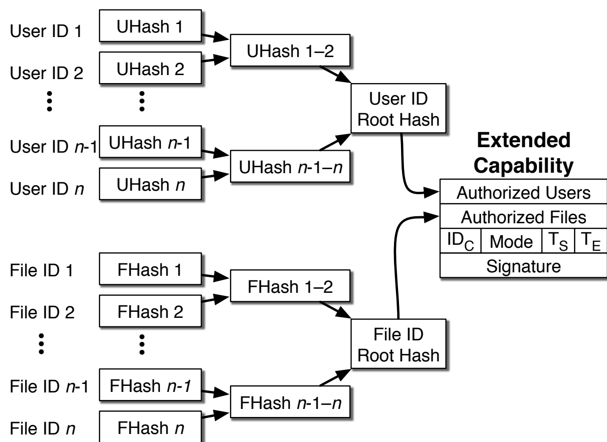


Figure 4: Extended capabilities identify authorized users and files using root hashes. This allow capabilities to be fixed size, no matter how many I/Os are authorized.

I/O for N users to M files located across J OSDs requires a maximum of J OSD update requests for the $N \times M$ I/Os authorized by the capability.

3.3.3 Grouping I/O Authorizations

The technique by which I/O authorizations are grouped into an extended capability directly affects how effectively Maat can reduce capability generation and how well OSD capability verification performs, *i. e.*, the number of root hash updates that must be performed. The four factors that impact grouping effectiveness are the number of authorizations per capability, the frequency with which groups change, the overhead of the grouping scheme, and compatibility in heterogeneous environments. Increasing the number of I/O authorizations per capability causes fewer capabilities to be generated. The frequency with which groups change affects how frequently OSDs must resolve new, unknown root hashes, impacting capability verification performance. The overhead of the grouping strategy determines the time required to actually construct the group. Compatibility defines how well the grouping strategy works in heterogeneous environments. Ultimately, the optimal grouping strategy varies from system to system and depends on workload and system size, among other things. We briefly discuss several grouping approaches that we believe work well for large-scale systems and HPC workloads.

Access Prediction. HPC applications often have repeated access patterns, such as reading data files on boot and writing log files on close. Future file or user accesses can be predicted based on past behavior and pro-actively included in a capability. Clients can provide predicted future file accesses to the MDS with `open()` requests, similar to the prediction strategy discussed by Gobioff [12]. In addition, the MDS can predict future users' accesses. Prediction has the upside of potentially grouping many authorizations into a single capability, but can cause frequent group changes if predictions are limited or incorrect. Also, prediction incurs a space and time penalty to store past behaviors and calculate predictions.

UNIX **Permissions.** Using UNIX access control (*i. e.*, owner, group, world permissions) in a way similar to sharing in Cepheus [9] has the major advantage of requiring a maximum of three capabilities for any single file. Only three capabilities are needed because anyone accessing the file must use either the user, group, or world permissions. Additionally, UNIX groups do not change frequently. The major downside is its incompatibility with other access control semantics, such as those of Windows. In large-scale systems various access control semantics may be needed for different domains, therefore, not all files may be controlled by UNIX access semantics. Unless the MDS is sophisticated enough or other steps are taken [14], multiple access control semantics cannot be enforced.

Temporal Patterns. HPC workload access patterns have strong temporal relationships, bursty accesses, and flash crowds [31]. Batching groups of capability requests at the client or MDS can alleviate congestion caused by these patterns, similar to the batching approach used in the original NASD design [12]. Temporal request batching can absorb bursty requests to reduce overall load, but can also increase client latencies. Also, batching will only produce large groups if temporal relationships are strong.

User or Application Knowledge. Users or applications may know of an efficient grouping strategy, and may thus manually define their own groupings, similar to T10's set attribute (SET ATTR) functionality [32]. Manual definitions may be accurate but are also tedious, making it unlikely group definitions will be rigorous or maintained.

Other Relationships. Other, simpler relationships may also prove effective. For example, including all files in the current working directory may serve as a quick and accurate method of file access prediction. Additionally, combining grouping strategies can improve the number of authorizations per capability.

3.4 Automatic Revocation

Revoking a capability by explicitly contacting storage devices in a petascale storage system is very difficult for several reasons. First, file data is striped across many storage devices, meaning many devices must be contacted if a capability is to be revoked. Second, extended capabilities may exacerbate the problem by authorizing I/O to many files, increasing the number of devices which hold the capability. Third, each storage device must remember which capabilities were revoked, so as to not allow reuse later. To achieve better scalability, Maat uses *automatic revocation*, which allows global capability revocation without the need to explicitly contact any storage devices. In this section we discuss the techniques that make this possible.

Maat requires that each capability have a short lifetime; our current prototype uses five minute lifetimes. This allows Maat to shift the revocation paradigm from explicitly contacting devices to simply allowing a capability to expire. When an OSD verifies an I/O request, it checks to see if the capability's lifetime is valid; if not, it does not authorize the I/O. When a file's permissions change (*e. g.*, using `chmod()`), after a maximum of five minutes, all capabilities permitting the now-invalid access have expired. By immediately applying the permission changes to its local access control matrix, the MDS ensures that it will no longer issue capabilities that authorize the revoked access. As an additional benefit, revoking many accesses requires no greater effort than revoking a single access. For example, revoking all access privileges of a user simply requires the MDS to no

$C \rightarrow M$: renewal$(P), T_s, hash\langle$renewal$(P), T_s, K_{CM}\rangle$
$M \rightarrow C : \mathcal{R}, hash\langle\mathcal{R}, K_{CM}\rangle$
$C \rightarrow D : \mathcal{C}, \mathcal{R}$, read$(oid), T_s, hash\langle$read$(oid), T_s, K_{CD}\rangle$

(a) Protocol to renew capabilities. Clients renew all capabilities in use, P, and present renewal tokens, $\mathcal{R}$, to OSDs with I/O requests. The OSD uses the token to extend the lifetime of the capabilities it names to a valid time.

$$\mathcal{R} = \langle P + O, T_e\rangle K_M^R$$

(b) Contents of a renewal token. The token renews all valid requested capabilities, P, and all outstanding capabilities for all users, O in a single renewal token. All renewed capabilities are valid until time T_e.

Figure 5: Capability renewal protocol.

longer issue capabilities for that user. After a maximum of five minutes, all of the user's capabilities will have expired and the user will no longer be able to access data. This allows expiration to act as a global revocation, no matter how large the system or where capabilities are located. While there is a five minute window of vulnerability, techniques similar to this have previously been used in formal proof of network-attached storage security: the MDS does not apply a permission change until all capabilities authorizing the modified access have expired, ensuring that permission state on the MDS and OSDs are consistent [5].

While the goal of capability expiration is to allow scalable revocation, it must also ensure that valid capabilities can continue to be used. To do this, Maat uses a fast capability renewal protocol, as shown in Figure 5. Clients periodically submit requests to extend the lifetimes of capabilities that they wish to continue using. In response, the MDS generates, caches, and returns *renewal tokens* that extend the lifetime of a set of capabilities for the same length as the original period—five minutes in the current implementation. When making an I/O request, clients present renewal tokens along with the expired capability to an OSD. The OSD can subsequently verify that the token indeed extends the lifetime of the capability to a valid time and cache the result of the verification.

The core concept behind scalable renewal is that the MDS can generate a single renewal token which renews a large number of valid capabilities by generating a renewal token for *all* valid outstanding capabilities. In this way, a small number of renewal tokens can extend the lifetimes of all valid capabilities still in use, ensuring that the MDS spends a limited amount of time serving renewal requests. In essence, this shifts the cost of revocation from revoking invalid capabilities to renewing valid ones, which can be done in a much more scalable fashion. As Figure 5 shows, the renewal token $\mathcal{R}$ extends the lifetime of all requested capabilities P and all outstanding capabilities O.

It may be the case, albeit rarely, that file or user access must be immediately revoked. In the example of a compromised client workstation, eventual revocation (*i. e.*, waiting for capabilities to expire) is insufficient. For these scenarios, Maat uses an immediate revocation scheme. While explicitly contacting a large number of devices cannot be avoided, Maat uses short capability lifetimes to eliminate the need for OSDs to persistently remember all revoked capabilities. Once an OSD has been notified of a capability revocation, it only needs to remember the revocation for a maximum of five minutes, since after that time the capability becomes invalid. This allows Maat to support immediate revocation without the need to require OSDs to remember past capabilities or rotate capability IDs.

3.5 Secure Delegation

Large-scale high performance computing often involves thousands of compute nodes collaborating on a common job, thus requiring clients, even those without file access privilege, to perform I/Os to and from shared files. Conceptually, facilitating these operations seems easy: a client simply opens a file and distributes a capability to other clients participating in the computation. Unfortunately, this gives rise to several issues. First, a capability that authorizes I/O for anyone who holds it is dangerous since anyone who obtains it, including an attacker, can use it to access data. On the other hand, the capability must be general enough to be distributed to any client participating in the computation. Second, delegated access must be temporary and limited because it provides privileged access to unprivileged users, though it should last as long as the computation. Third, while group file opening and delegation must be secure, they must also be fast, and ideally much faster than each client individually opening the file.

The POSIX HPC I/O extensions `openg()` and `openfh()` were designed to provide support for collaborative computation by allowing collective file opens and access delegation [35]. The `openg()` operation takes a path and mode and returns a file handle that can be transferred to cooperating clients and subsequently converted to a file descriptor with `openfh()`. The proposed semantics of these operations require that the file handle, and therefore capability, be transferable to any client. We describe how Maat uses secure delegation to support secure cooperative computation, address the concerns above, and support these HPC I/O extensions.

Group opens and delegation in Maat are implemented using temporary asymmetric computation keys. At the start of a large compute job, a single client (the "lead") initiates a joint computation and generates an asymmetric key pair that will last the duration of the computation. For each file, the "lead" client calls `openg()`, passing the computation public key along with the usual `open()` arguments, as shown in Figure 6(a); this is the only `open()` call sent to the MDS for this file. The MDS returns a file handle that includes a capability to access the file and a token stating the lifetime of the computation public key. The capability includes the hash of the computation public key in place of a root hash of user IDs, thus associating the capability with an asymmetric key pair, rather than a set of users and allowing an OSD to authorize I/O for anyone who can prove possession of the computation private key. In Figure 6, $\mathcal{F}$ and $\mathcal{L}$ are the file handle and public key lifetime token, respectively.

Once `openg()` returns, the "lead" client is free to pass the file handle, computation key pair, and signed key lifetime token to any clients who are participating in the computation. To prevent eavesdropping of the computation private key, the key is encrypted with the receiving client's public key. Delegation can be optimized by the source client precomputing the encrypted private key for each receiving client. To perform I/O, clients convert the file handle to a local file descriptor using `openfh()`. Each client proves possession of the computation private key by submitting a proof token with I/O requests. The proof token is computed by signing

$$C \rightarrow M : \text{openg}(path, mode), K^U_{Comp}, T_s,$$
$$hash\langle \text{openg}(path, mode), K^U_{Comp}, T_s, K_{CM}\rangle$$
$$M \rightarrow C : \mathcal{F}, \mathcal{L}, hash\langle \mathcal{F}, \mathcal{L}, K_{CM}\rangle$$

(a) Protocol to open a file on behalf of a group. The "lead" client submits the computation public key with `openg()` requests. The MDS returns a file handle, $\mathcal{F}$, and a public key lifetime token, $\mathcal{L}$. $\mathcal{F}$ contains the capability needed to access the file.

$$C \rightarrow C' : \{K^R_{Comp}\}K^U_{C'}, \mathcal{F}, \mathcal{L}$$
$$C \rightarrow D : \text{read}(oid), \mathcal{C}, \mathcal{P}, T_s, hash\langle \text{read}(oid), T_s, K_{CD}\rangle$$
$$D \rightarrow C : \text{update}(K^U_{Comp}), T_s,$$
$$hash\langle \text{update}(K^U_{Comp}), T_s, K_{CD}\rangle$$
$$C \rightarrow D : \mathcal{L}$$

(b) Messages to securely delegate file access privilege. The encrypted computation private key, file handle, and public key lifetime token are distributed to clients participating in the computation. Each client computes a token $\mathcal{P}$ that proves possession of the private computation key and present it with I/O requests. Messages 3 and 4 only occur if the OSD has not previously seen the computation public key.

$$\mathcal{F} = \langle path, mode, flags, \mathcal{C}, K^U_{Comp}, T_s, T_e\rangle K^R_M$$
$$\mathcal{L} = \langle K^U_{Comp}, T_s, T_e\rangle K^R_M$$
$$\mathcal{P} = \{\langle hash\langle K^U_{Comp}\rangle\rangle K^R_{Comp}\}K_{CD}$$

(c) Definitions for a file handle $\mathcal{F}$, public key lifetime token $\mathcal{L}$, and private key proof token $\mathcal{P}$. $\mathcal{F}$ contains a capability to access the file and data for `openfh()` to produce a local file descriptor. $\mathcal{L}$ authenticates the public key and its expiration time. $\mathcal{P}$ proves possession of the computation private key.

Figure 6: Secure group open and delegation.

the hash of the computation public key with the computation private key and encrypting it with the OSD shared key. The signed hash proves possession of the computation private key, while the encryption authenticates the client. If an OSD has not previously seen the computation public key, a client must also pass the signed key lifetime.

The computation key pair is the root of security in Maat's delegation protocol. Maat requires the key pair to exhibit three properties to make it efficient and secure—it must be temporary, renewable, and revocable. The key pair must be temporary because it allows clients who may not otherwise have access rights, the rights to access privileged files. However, since cooperative computation is often long-lived, this privilege should last as long as the computation, requiring the key pair to be renewable. Also, the key pair must be immediately revocable since clients without normal file access privileges may be participating in the computation.

When the MDS generates the computation public key lifetime token, $\mathcal{L}$, it gives the key a short lifetime (five minutes in the current implementation), making the key temporary. Since cooperative I/O relies on an OSD validating both a capability and a private key proof token, renewal and immediate revocation can be achieved by renewing or revoking the public key lifetime token. Maat implements protocols very similar to capability renewal and revocation that allow a public key lifetime token to be renewed or revoked. The only significant difference is that only the client who initiated the joint computation can renew the public key lifetime. This prevents a client who was delegated access to a file from continually renewing the computation public key and having persistent access to the file.

3.6 Implementation Details

We have implemented Maat in the Ceph petascale, high-performance distributed file system. All cryptographic operations were implemented using the Crypto++ library [6]. Maat has support for various cryptographic algorithms; the current implementation uses 1023-bit ESIGN for public/private key operations, 128-bit AES for shared key operations, and SHA-1 for one-way hash functions. All were chosen for their high performance.

The current Maat implementation supports several of the authorization grouping strategies previously mentioned: prediction, UNIX groups, and temporal batching. UNIX groups are implemented by using Merkle trees with group IDs pointing to the root hash value associated with the group's Merkle tree. Prediction uses the Recent Popularity algorithm [2], which predicts a successor s if s occurs at least j times in the k previous observations and makes no prediction otherwise. We chose Recent Popularity because adjusting the j and k values allows us to adjust the accuracy of our prediction; we chose $j = 4$ and $k = 6$. Additionally, only making predictions with confidence avoids the penalty of creating groups which are likely incorrect; it is important to note, however, that the MDS never creates capabilities that are not permitted by its current access control matrix. Temporal batching is designed to handle flash crowd-like workloads. The MDS begins batching `open()` requests for a file if four requests for the same file are received within 20 ms of each other. Additionally, the MDS begins batching requests from a user if more than four requests from the same user for different files are received within 20 ms of each other. Batching is currently set for one second, though a smaller value is probably more desirable for most workloads. Once a group of requests is batched, a capability is generated to authorize all requests in the batch. Whenever Maat cannot group multiple authorizations into a capability, it issues capabilities which authorize a single user to access a single file by explicitly naming them in the capability and eschewing the use of Merkle trees.

Capability expiration is currently configured for five minutes; after four minutes, each client renews capabilities for all open files. By renewing after four minutes, clients are pro-active about not allowing capabilities in use to expire. Also, by renewing all open capabilities, the clients ensure that no currently used capability will expire while in use. In addition to the requested capabilities, the MDS renews capabilities for all currently opened files, for all users.

4. PERFORMANCE EVALUATION

We evaluated Maat to assess the overhead and scalability of securing a petabyte-scale file system using "insecure" Ceph as a baseline. We evaluated I/O performance using a microbenchmark run with two Maat authorization grouping strategies, UNIX groups and prediction, and compared them both to secure I/O without grouping and to baseline Ceph. This experiment highlighted the benefits of extended capabilities, and explored the pros and cons of different authorization grouping strategies. We also evaluated the performance of batch-based authorization grouping during flash crowds and the overhead incurred for capability renewal.

Operation	open()	write()	read()
Baseline	41	329	45
Maat	597	619	291
Maat (Merkle)	615	1534	1301
Maat (Cache Hit)	44	333	48

Table 1: A comparison of `open()`, `write`, and `read()` operations in microseconds. The drastic increase in overhead when a capability must be generated or verified demonstrates the critical need for Maat to maximize capability cache hits.

We evaluated Maat's performance under an HPC workload by running the IOR2 benchmark [28], an HPC parallel file system benchmark. Finally, we used an analytical model to compare Maat and other security schemes in a petascale environment.

4.1 Experimental Setup

Our experiment test bed consisted of an 18 node Linux cluster in which each node was a PC with a 2.8 GHz Pentium 4 processor with 3.1 GB of RAM connected to a local SCSI disk and running Red Hat Enterprise Linux 4 (kernel version 2.6.9). The nodes were networked via gigabit Ethernet through an Extreme Networks switch. The cluster was partitioned into 1 MDS, 10 OSDs, and 7 client nodes, each of which was able to run up to 20 client processes concurrently without performance degradation. Our benchmarks were done without on-wire encryption because encryption tends hide the sources of overhead by adding encryption costs to each I/O operation. Additionally, all experiments were conducted with the local client data cache disabled because the Ceph client write back policy resulted in high variability between runs.

4.2 Overhead from Capability Operations

To determine the performance impact of introducing security via capabilities and quantify the potential savings from caching capability generations and verifications, we timed `open()` operations at the MDS and, `write()` and `read()` operations at an OSD under several scenarios and compared these operations to the times required in baseline Ceph, with the results summarized in Table 1. Open requests that require capability generation run 14 times slower than baseline Ceph, 90% of which can be attributed to the signature required for capabilities. Because the operation was timed at the MDS, this difference is greater than it would be if timed at a client, since network round trip time would be accounted for. Handling an open with a cached capability performs significantly better, incurring a penalty of $3\,\mu s$ (7%) due to lookups. When verifying a capability, write and read operations perform 2 and 6 times slower than baseline Ceph, respectively. When Merkle trees are used, these overheads increase by a factor of 5 for writes and 29 for reads because the OSD has not already cached the Merkle tree below the root hash. In our experiment, the client did not have the tree cached either, requiring additional messages from the OSD to the client and from the client to the MDS. However, when an OSD has previously verified a capability, I/O performance is on par with baseline Ceph performance, with performance penalties under 7% for strong security. These numbers demonstrate the need to reduce capability generations and verifications via caching.

4.3 Microbenchmark Performance

To evaluate Maat's performance and scalability under various system sizes, we ran several experiments with a microbenchmark in which each client writes to 6 shared files and 4 non-shared files with 5 MB being written to each in 128 KB byte chunks. Each run began with a fresh file system and a cold capability cache. We pre-configured UNIX groups such that every 10 clients shared a group. Using this setup, we evaluated how extended capabilities performed under various authorization grouping strategies, varying the number of clients from 10 to 140.

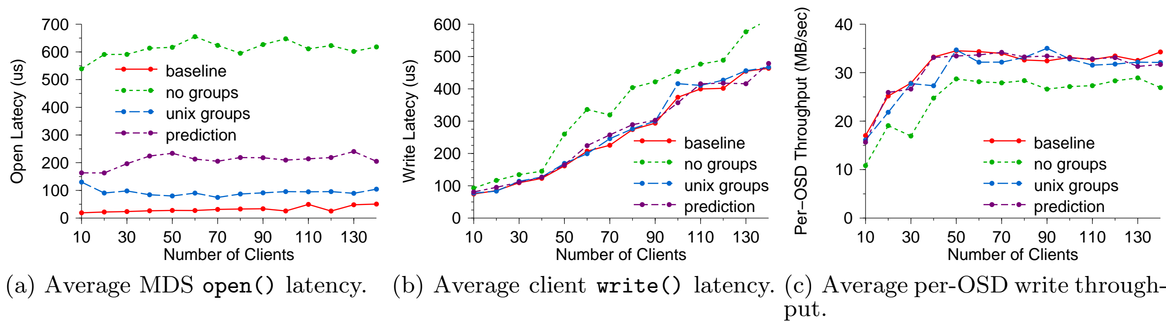
`open()` Latency. Figure 7(a) charts the average latency for an MDS `open()` request, showing that the use of extended capabilities to group authorizations results in a major performance improvement. Both UNIX and prediction grouping perform over 3 times better than no grouping, and UNIX grouping approaches baseline Ceph performance. UNIX groups perform better than prediction because prediction does not group as many authorizations per capability. For each client, prediction generates a capability that authorizes access to all shared files and capabilities for all four non-shared files. In contrast, UNIX grouping generates one capability per file, but that capability authorizes write privileges for all ten users in the group. While both approaches performed well, we believe combining both grouping strategies would produce better results as even fewer capabilities would need to be generated.

`write()` Latency. Figure 7(b) shows the average latency for client `write()` operations. UNIX and prediction grouping add a negligible overhead to baseline Ceph performance, but not using authorization grouping incurs a higher overhead. While Maat's low overhead contributes to the low latency, other factors do as well. First, the round trip network latency for client requests masks differences between baseline Ceph and Maat by adding a relatively constant overhead for all writes. Second, each client writes 5 MB in 128 KB chunks, so at most one (the first) of the 40 writes per file will miss the capability verification cache at any given OSD, regardless of the grouping strategy. Grouping authorizations decreases the number of *initial* writes that will miss the cache, accounting for the discrepancy between the different strategies.

`write()` Throughput. Figure 7(c) shows that, as before, UNIX and prediction grouping do not noticeably decrease per-OSD write throughput compared to the baseline, while not grouping authorizations lowers throughput. The UNIX and prediction grouping decrease total throughput 3.8 and 1.3%, respectively. Total throughput is decreased 20% without grouping.

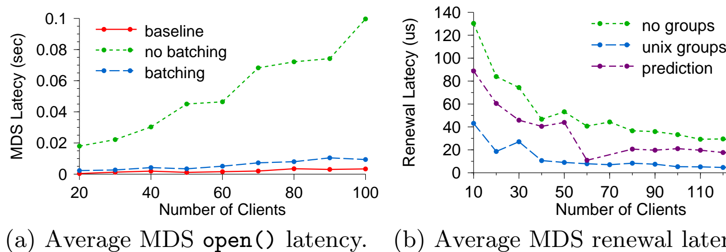
4.4 Scalability

To further explore Maat's scalability, we conducted two experiments first using flash crowds then capability renewal. Our flash crowd experiment consisted of each client issuing an `open()` request for the same file. We varied the number of clients from 20 to 100. Figure 8(a) shows the results of our flash crowd experiment. Batching is able to keep open latency low, very close to baseline Ceph. Without using batching, open latency quickly increases to over 30 times that of baseline Ceph. This difference is easily attributed to the disparity in the number of capabilities generated. With-



(a) Average MDS `open()` latency. (b) Average client `write()` latency. (c) Average per-OSD write throughput.

Figure 7: Results of a mixed workload microbenchmark.



(a) Average MDS `open()` latency. (b) Average MDS renewal latency.

Figure 8: An analysis of Maat's scalability for flash crowds and capability renewal.

out batching, the MDS generates a capability per request, while batching requires only a single capability per batch.

For our capability renewal experiment, we adjusted Maat's renewal period to have clients request renewals every 15 seconds while clients wrote a series of non-shared, 32 MB files for 50 seconds. The workload allowed each client to make three renewal requests before finishing. Figure 8(b) shows that the average MDS latency for renewal requests declines as the number of clients increases. This behavior results from the number of renewal token generations staying constant even as the number of renewal requests increases.

4.5 IOR2 Benchmark

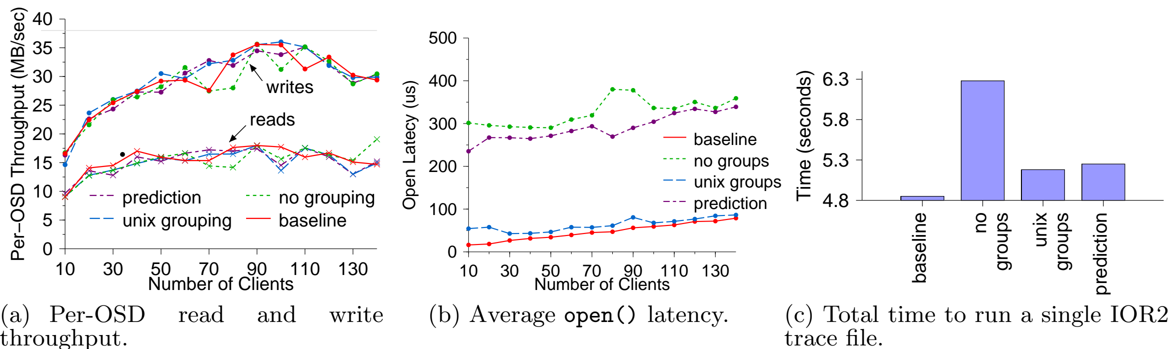
To gauge Maat's performance under a real HPC workload, we used a modified version of the IOR2 benchmark [28], a parallel file system benchmark designed for large-scale systems. The benchmark is broken down into 512 trace files collected from 512 separate processes, each of which writes a series of non-shared output files and reads them back. A limited number of shared files are also written.

For our experiments, each client ran two trace files randomly selected from the 512 possible traces, with the number of clients varying from 10 to 140. Because each experiment began with a fresh file system, we modified each trace file to issue a `write()` and `lseek()` prior to any read to ensure reads were not issued to an unwritten offset. Each trace issues about 520 reads, so our modifications added another 520 writes, resulting in each trace issuing about 1,030 total writes. The majority of I/O is done in 64 KB chunks and approximately 25 files are opened in each trace file. Again, we configured UNIX groups so that every 10 clients shared a group. We ran all 512 trace files though our Recent Popularity predictor to calculate predictions.

The results in Figure 9(a) show that per-OSD read and write throughput were comparable across all four configurations. The minimal effect on throughput is slightly different than what we expected, but can be attributed to the I/O-centric nature of the benchmark. With our modifications, over 125 MB are written per trace file, which, at two processes per client, results in 250 MB being written per client. As a result, the cost of verifications is amortized across the large number of I/O requests. Average `open()` latency, in Figure 9(b), shows that UNIX grouping produces results comparable to baseline Ceph, while prediction latencies are generally over 100 μs slower. This discrepancy is due to Recent Popularity producing relatively few predictions; only a few of the IOR2 access patterns were sufficiently common for Recent Popularity to have confidence to make a prediction. Finally, Figure 9(c) shows the total time required to run a single IOR2 trace file. Without any authorization grouping, a 22% overhead is incurred over baseline Ceph, but UNIX and prediction grouping reduce the overhead significantly to 6% and 7%, respectively.

4.6 Petascale Analysis

Based on our experiments timing Maat's performance in a real environment, we developed an analytical model to show how Maat would perform in petascale systems and how it compares to other security schemes. Total MDS capability generation overhead, $M(S)$, can be calculated by $M(S) = (((objects/bytes) \times S)/G) \times C$, where S is the size, in bytes, of the file being opened, G is the grouping factor (the number of authorizations per capability), and C is the cryptographic cost associated with each capability generation. We assume that each object is 2^{20} bytes (1 MB) because Ceph objects default to this size. For extended capabilities, we fix the grouping factor to $10 \times (objects/bytes) \times S$, resulting in 10 file authorizations per capability. Whole file capabilities have one authorization per capability, for a grouping factor of $(objects/bytes) \times S$, and per-object capabilities have



(a) Per-OSD read and write throughput.

(b) Average `open()` latency.

(c) Total time to run a single IOR2 trace file.

Figure 9: Results of the IOR2 benchmark.

a grouping factor of 1. Our experiments fixed the values of C at $3\,\mu s$ for an HMAC and $350\,\mu s$ for an ESIGN signature, resulting in the following overheads for different capability schemes:

$M(S) = 35$	extended capability
$M(S) = 350$	whole file capability
$M(S) = (3 \times S)/2^{20}$	per-object capability

The cost of opening a file is constant for extended capabilities (assuming a fixed number of authorizations) and whole file capabilities because they do not depend on file size. Per-object capability overhead, however, is a function of file size. Using the timings above, extended capabilities incur a lower overhead than per-object capabilities when opening any file larger than 12 MB. Thus, in a petascale environment where clients issue 10^6 `open()` requests for files that each contain a gigabyte of data, extended capabilities would result in a $35 \times 10^6\,\mu s$, or 35 second overhead at the MDS. Whole file capabilities would incur an overhead of 350 seconds—ten times greater. However, per-object capabilities would require $3 \times (2^{30}/2^{20}) \times 10^6\,\mu s = 3072$ seconds, nearly 100 times slower than extended capabilities.

5. FUTURE WORK

The current Maat design works very well at providing security for petabyte-scale storage systems; however, some issues remain. First, Maat's authorization grouping strategy has a dramatic impact on the performance improvements gained by extended capabilities. While we presented several approaches for how to group authorizations, our list is by no means exhaustive. Exploring other grouping strategies can identify those that work best for specific workloads.

A second issue is that Maat was designed to provide scalable I/O security for petascale, high-performance storage. While it provides an authentication and authorization framework, it does not provide on-disk security. For many systems, access control is not sufficient; rather, such file systems want to keep all file contents encrypted on the network-attached devices [1, 16, 20] because on-disk security prevents an attacker from obtaining data even when the attacker has physical possession of the device. Additionally, on-disk security further limits the amount of trust that need be placed on storage devices. We are currently exploring scalable techniques for encrypting data on disk in petascale storage systems.

6. CONCLUSIONS

This paper described Maat, a scalable method for securing petabyte-scale parallel file systems by using three novel techniques for achieving scalability: extended capabilities, automatic revocation, and secure delegation. By limiting the number of cryptographic operations while still providing strong security, Maat can scale to handle file systems with thousands of clients accessing files striped across thousands of network-attached storage devices. Maat accomplishes this goal by using capabilities that can authorize I/O for any number of clients to any number of files, revocation which does not require explicit messaging to any devices, and a secure method for access delegation.

We evaluated a prototype implementation of Maat in the Ceph petascale distributed file system, focusing on Maat's scalability. Our scalability experiments show that, as system size increases, Maat has a minimal impact on latency and throughput for high-performance computing workloads. More concretely, Maat is able to add strong security while incurring as little as 6–7% overhead on an I/O-intensive HPC benchmark. With strong security available for scalable storage for so little overhead, there is no longer any reason to exclude secure file system authentication and authorization from petabyte-scale high performance storage.

Acknowledgments

This work was supported in part by the Department of Energy under award DE-FC02-06ER25768, the National Science Foundation under award CCF-0621463, the Institute for Scalable Scientific Data Management, and by the industrial sponsors of the Storage Systems Research Center, including Agami Systems, Data Domain, DigiSense, Hewlett Packard Laboratories, LSI Logic, Network Appliance, Seagate Technology, Symantec, and Yahoo. We thank Martín Abadi, Sage Weil, and the other members of the SSRC, whose advice helped guide this research. Finally, we thank our shepherd Paul Stodghill and the anonymous reviewers for their insightful feedback.

7. 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 *Proc. of FAST '03*, 2003.

[2] A. Amer, D. D. E. Long, J.-F. Pâris, and R. C. Burns. File access prediction with adjustable accuracy. In

Proceedings of the International Performance Conference on Computers and Communication (IPCCC '02), Phoenix, Apr. 2002.

[3] 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*, 2002.

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

[5] A. Chaudhuri and M. Abadi. Formal analysis of dynamic, distributed file-system access controls. In *Proc. of Int'l Conf. on Formal Techniques for Networked and Distributed Systems*, Sep. 2006.

[6] W. Dai. Crypto++ version 5.4. http://www.cryptopp.com, 2006.

[7] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proc. OSDI '04*, Dec. 2004.

[8] M. Factor, D. Nagle, D. Naor, E. Riedel, and J. Satran. The OSD security protocol. In *Proc. 3rd IEEE Security in Storage Workshop*, 2005.

[9] K. Fu. Group sharing and random acces in cryptographic storage file systems. Master's thesis, MIT, June 1999.

[10] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *Proc. SOSP '03*, Oct. 2003.

[11] G. A. Gibson, *et al.* A cost-effective, high-bandwidth storage architecture. In *Proc. 8th ASPLOS*, Oct. 1998.

[12] H. Gobioff. *Security for a High Performance Commodity Storage Subsystem*. PhD thesis, Carnegie Mellon University, July 1999.

[13] E.-J. Goh, H. Shacham, N. Modadugu, and D. Boneh. SiRiUS: Securing remote untrusted storage. In *Proc. of the 2003 Network and Distributed System Security Symposium*, Feb. 2003.

[14] D. Hitz, B. Allison, A. Borr, R. Hawley, and M. Muhlestein. Merging NT and UNIX Filesystem Permissions. In *Proc. of the USENIX Windows NT Symposium*, Aug. 1998.

[15] 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 Trans. on Computer Systems*, 6(1):51–81, Feb. 1988.

[16] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus: scalable secure file sharing on untrusted storage. In *Proc. of FAST '03*, Mar. 2003.

[17] A. Leung and E. L. Miller. Scalable security for large, high performance storage systems. In *Proc. 2nd Workshop on Storage Security and Survivability*, 2006.

[18] H. M. Levy. *Capability-Based Computer Systems*. Butterworth-Heinemann, Newton, MA, USA, 1984.

[19] R. C. Merkle. A digital signature based on a conventional encryption function. In *Advances in Cryptology - Crypto '87*, pages 369–378, 1987.

[20] E. L. Miller, D. D. E. Long, W. E. Freeman, and B. C. Reed. Strong security for network-attached storage. In *Proc. of FAST '02*, Jan. 2002.

[21] D. Nagle, D. Serenyi, and A. Matthews. The Panasas ActiveScale storage cluster—delivering scalable high bandwidth storage. In *Proc. of SC04*, Nov. 2004.

[22] B. C. Neumann, J. G. Steiner, and J. I. Schiller. Kerberos: An authentication service for open network systems. In *Proc. Winter USENIX Conference*, 1988.

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

[24] C. A. Olson and E. L. Miller. Secure capabilities for a petabyte-scale object-based distributed file system. In *Proc. of the 1st ACM Workshop on Storage Security and Survivability*, Nov. 2005.

[25] B. C. Reed, E. G. Chron, R. C. Burns, and D. D. E. Long. Authenticating network-attached storage. In *Proc. of Hot Interconnects VII*, Aug. 1999.

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

[27] O. Rodeh and A. Teperman. zFS: a scalable distributed file system using object disks. In *Proc. Mass Storage Systems and Technologies Conf.*, 2003.

[28] Scalable I/O Project. http://www.llnl.gov/icc/lc/siop/, 2006.

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

[30] A. Singh, S. Gopisetty, L. Duyanovich, K. Voruganti, D. Pease, and L. Liu. Security vs performance: Tradeoffs using a trust framework. In *Proc. Conf. on Mass Storage Systems and Technologies*, 2005.

[31] F. Wang, *et al.* File system workload analysis for large scale scientific computing applications. In *Proc. Conference on Mass Storage Systems and Technologies*, Apr. 2004.

[32] R. O. Weber. Information technology—SCSI object-based storage device commands (OSD). Technical Council Proposal Document T10/1355-D, Technical Committee T10, Aug. 2002.

[33] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proc. of OSDI '06*, 2006.

[34] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn. CRUSH: Controlled, scalable, decentralized placement of replicated data. In *Proc. of SC06*, Nov. 2006.

[35] B. Welch. POSIX IO extensions for HPC. In *Proc. of FAST '05*, Dec. 2005.

[36] E. Wobber, M. Abadi, A. Birrell, and B. Lampson. Access control subsystem and method for distributed computer system using locally cached authentication credentials. U. S. Patent 5,235,642, Aug. 1993.

[37] Y. Zhu and Y. Hu. Snare: A strong security scheme for network-attached storage. In *Proc. of the 22nd Symp. on Reliable Distributed Systems*, 2003.