Horus: Fine-Grained Encryption-Based Security for High Performance Petascale Storage[*]

Ranjana Rajendran
ranjana@soe.ucsc.edu

Ethan L. Miller
elm@cs.ucsc.edu

Darrell D. E. Long
darrell@cs.ucsc.edu

Storage Systems Research Center & Computer Science Department
University of California, Santa Cruz
Santa Cruz, CA 95064

ABSTRACT

Data used in high-performance computing (HPC) applications is often sensitive, necessitating protection against both physical compromise of the storage media and "rogue" computation nodes. Existing approaches to security may require trusting storage nodes and are vulnerable to a single computation node gathering keys that can unlock *all* of the data used in the entire computation. Our approach, Horus, encrypts petabyte-scale files using a keyed hash tree to generate different keys for each region of the file, supporting much finer-grained security. A client can only access a file region for which it has a key, and the tree structure allows keys to be generated for large and small regions as needed. Horus can be integrated into a file system or layered between applications and existing file systems, simplifying deployment. Keys can be distributed in several ways, including the use of a small stateless key cluster that strongly limits the size of the system that must be secured against attack. The system poses no added demand on the metadata cluster or the storage devices, and little added demand on the clients beyond the unavoidable need to encrypt and decrypt data, making it highly suitable for protecting data in HPC systems.

Categories and Subject Descriptors

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

General Terms

Design, Security

1. INTRODUCTION

Many high performance computing (HPC) installations require the storage and manipulation of terabyte- and petabyte-scale data sets, such as geographic and satellite data, molecular dynamics simulation, nuclear reaction modeling, and structural simulation. Often, these files contain sensitive data, such as simulations of nuclear explosions, molecular models of drug interactions, and maps that include sensitive areas. Since HPC systems typically have hundreds to thousands of users, many of whom require root access to client nodes, approaches that rely on discretionary security are not sufficient. However, providing confidentiality for this data is very important, both for business and national security reasons.

Since HPC data must be protected against users with system-level access to client nodes as well as against malicious storage devices, it is clearly necessary to encrypt and decrypt data at the client and carefully control key distribution, limiting the ability of an intruder or malicious storage device to read protected data. Existing HPC file systems do not meet this challenge, providing (at best) access control via capabilities or other mechanisms [13]. The use of a single key to encrypt an entire file likewise fails because compromise of a *single* node, an event that might go unnoticed in a system with tens of thousands of nodes, would result in compromise of the *entire* file. However, while existing approaches that generate a key per block (or object) making up the file can limit this compromise, they result in much higher overhead for both key distribution and key storage.

Our approach, Horus, uses a tree of keyed hashes to generate keys for different block ranges of an HPC file, allowing a client with a range key to easily generate an encryption key for any blocks "within" the range in the tree. Systems without the key for a particular block (*e. g.*, disks and clients without an appropriate range key) cannot decrypt the block; thus, Horus provides strong confidentiality for file data. Each file only requires storage for a small fixed number of root keys at the base of the keyed hash tree, and the ability to utilize range keys reduces the number of keys that must be distributed. While a client may trivially derive a block key for a block within the range from a file range key, it *cannot* derive the key for a block *outside* the range from the range key, preserving the ability to restrict access at a fine granularity. The root keys may be protected using a lockbox [11, 16], and may be distributed by the metadata server; however, the use of a separate key distribution cluster can improve security and scalability. Horus can also be implemented as an encryption library on the clients, simplifying deployment on existing file systems.

2. BACKGROUND

Most large parallel file systems [4, 19, 20, 22, 23] decouple data control and data access paths, as shown in Figure 1, allowing management and security decisions to be centralized in a small meta-

[*]This research was supported in part by the National Science Foundation under awards IIP-0934401 and CCF-0937938, the Department of Energy under Award number DE-FC02-10ER26017/DE-SC0005417, and by the Department of Energy's Petascale Data Storage Institute under award DE-FC02-06ER25768. We also thank the industrial sponsors of the Storage Systems Research Center and the Center for Research in Intelligent Storage for their generous support.

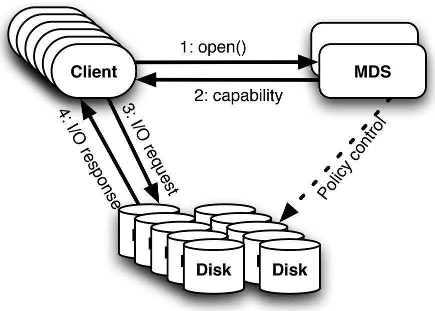

Figure 1: Clients receive location information from the metadata server (MDS). Access control can be implemented by having the MDS provide capabilities that are verified by the disk when the client requests I/O [13].

data cluster. However, few, if any, HPC file systems implement any security beyond the use of POSIX-style permissions to restrict access to files. Moreover, existing systems that rely upon the MDS to provide security via capabilities must place complete trust in the MDS cluster, providing a clear target for an attacker.

Recently, there have been several attempts to provide greater security for HPC file systems. Maat [13] provides scalable authorization and authentication, albeit not data confidentiality, for the Ceph file system [22]. Ceph breaks files up into *objects*, each of which is stored and accessed individually; the Maat protocol allows the restriction of access to individual objects on particular disks. However, permissions in Maat are granted at the file level, and Maat was explicitly designed to *limit* the number of unique capabilities, making it poorly suited for fine-grained access control. Moreover, Maat only handles authorization—files handled by Maat are stored unencrypted. Thus, under Maat, a single malicious client can cache authorizations for an entire file, and someone who steals a disk from this system can read data stored on the disk. There have also been attempts to integrate stronger security into map-reduce frameworks [9], but they are primarily limited to network authentication protocols and encrypting data transfers, and do not support storing encrypted data on disk. Airavat [10] further confines computations on data in the map-reduce framework, but still operates on data stored in the clear on storage nodes.

Actual encryption of portions of HPC files was first proposed by Banachowski, *et al.* [3]. However, the approach they proposed requires one entry in an s-node (security node) for each region to be separately secured; this approach requires too much storage and key distribution overhead for a file in which each 4 KB block must be encrypted by a separate key with a separate set of permissions. Moreover, their approach restricts access by user; we want to dynamically restrict access by client as well.

While strong security is rare in HPC file systems, there are many approaches to providing strong security for smaller-scale file systems. Cepheus [8], SNAD [16], and Plutus [11] all facilitate encryption at the client in a networked file system, preventing the compromise of a disk from leaking confidential data. All use *lockboxes* to secure a symmetric data encryption key common to all users by encrypting the symmetric key with each user's key, and storing one lockbox per user. The symmetric encryption key can be generated per-file or per-block; per-file keys are more efficient, but allow a malicious client to read an entire file with just one key. Aguilera, *et al.* [1] introduced a capability-based approach that allows access to be granted to ranges rather than simple blocks; as with the approach proposed by Banachowski, *et al.* [3], this approach does not scale well to fine-grained protection of terabyte-scale files because of the overhead of key storage and capability distribution.

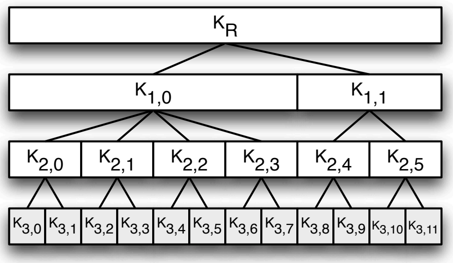

Figure 2: Basic keyed hash tree design. Keys at lower levels of the tree control smaller regions of the file. The leaf nodes (shaded in the figure) are the keys for individual file blocks. While all regions at a given level are nominally the same size, the region with $K_{1,1}$ is truncated because it is at the end of the file.

Horus heavily leverages hash trees, which have long been used for authentication and integrity checking. Merkle first proposed an authentication method based on hash trees [15]. Fiat and Noar [7] used a one-way function in a top-down fashion in their group keying method for the purpose of reducing the storage requirements for information theoretic key management. Chan and Chan [5] describe a key tree based method to negotiate subgroup keys. They also describe a contributory key agreement protocol based on Diffie-Hellman key exchange [6] and a computational number theoretic approach based on the Chinese Remainder theorem. Even though these methods are targeted at reducing the number of keys in a multicast group environment, they still assume too many keys to be stored at each node and too many key negotiation messages between nodes for a petascale storage system consisting of millions of file blocks stored on thousands of disks. Instead, we need a scalable method that allows a client to generate all the required keys from a single key through local computations without the need for contributory data from other nodes.

3. SYSTEM DESIGN

Horus uses a tree of keyed hashes to derive unique encryption keys for individual blocks from a per-file root key K_R, which is the only key that need be stored. The "block" size at each level of the keyed hash tree (KHT) is a fixed size, with block size decreasing at the lower levels of the tree. The keys at the leaf nodes of the KHT are each used to encrypt individual data blocks; keys higher in the tree, from which the lower-level keys can be derived, are used to control access to larger regions of the file. A schematic view of a sample KHT is shown in Figure 2.

While K_R must be stored by the file system, the key for region y at level $x \geq 1$ is calculated by $K_{x,y} = KH(K_{parent}, x||y)$, where K_{parent} is the key for $K_{x,y}$'s parent region, $KH(K,M)$ is a keyed hash function of any text M with key K, and $||$ is the concatenation operator. Using the message $x||y$ to generate each key ensures that each $K_{x,y}$ generated is unique, and using K_{parent} as the key ensures that anyone with the key for the parent region can generate keys for regions or blocks below that point in the tree. For example, for $K_{2,1}$, $K_{parent} = K_{1,0}$, so $K_{2,1} = KH(K_{1,0}, 2||1)$. This calculation can be applied recursively to derive a leaf key from any key above it in the tree, as shown in Figure 3. It is important to note that, given $K_{x,y}$, it must be impossible to derive either K_{parent} or $K_{x,y'}$, for any $y' \neq y$. Any implementation of keyed hash that satis-

```
**Require:** $0 \le start < end < d$
  **for** $x = start + 1$ **to** $end$ **do**
    $k \leftarrow \text{keyed\_hash}(k, x || \lfloor b/B_x \rfloor)$
  **end for**
  **return** $k$
```

Figure 3: Algorithm to calculate a range encryption key. The caller provides the starting region key k, the byte offset b, the *start* and *end* levels, and block sizes $B_0, \ldots, B_{d-1}$, where d is the number of levels in the tree, including the root key (at level 0) and leaf nodes (at level $d-1$). If the end level is $d-1$, the resulting key may be used for data encryption.

fies this condition can be used, including the keyed hash used for Hashed Message Authentication Codes (HMACs) [12]. Since Horus keys have constrained formats, a simpler hash function such as $KH(K,M) = SHA1(K||M)$ (folding the 160 bit SHA1 value into 128 bits) should suffice; the specific keyed hash function can be easily changed if weaknesses are found in a particular algorithm.

In Horus, a client node can be given the keys for only the ranges that it needs to access, preventing it from decrypting any parts of the file outside its allowed region. For example, a client that needs access to blocks 1–3 of the file in Figure 2 would be given $K_{3,1}$ and $K_{2,1}$; the latter key would allow the client to derive $K_{3,2}$ and $K_{3,3}$. If a different client must access the first 8 blocks of the file, it only needs to obtain $K_{1,0}$, from which it can derive $K_{3,0}, \ldots, K_{3,7}$.

The KHT can be adjusted to handle any level of granularity that is a multiple of the size of a single encryption block (typically 16 bytes), even permitting branching factors that are not powers of 2. However, there is a tradeoff between having a smaller "branching factor"—a small number of children for each node—and having a more compact tree with fewer levels. While fewer children per internal node provides more options for reducing the number of keys transmitted, it also results in longer key calculation times because there are more internal nodes between a level of a given size and the root. For example, in Figure 2, where the granularity of encryption is a key for every 4 KB block of data, level 1 of the tree only covers 32 KB; a node that wanted access to a 1 MB region would have to obtain 32 keys. For a terabyte-sized file, level 1 would contain 32 million keys, leaving a large key distribution problem. Instead, we expect that petabyte-scale file systems will have regions of 16 MB–1 GB at level 1 of the KHT, with branching factors of 4–8 for successively lower levels of the tree. Each node in the tree can have as many children as the block size at its level divided by the block size at the next level. With a level 1 block size of 1 GB and a reduction of 8× for each successive level, a KHT would require 5 additional levels between level 1 and the 4 KB leaf nodes. Since keyed hash calculations on modern processors can be done in microseconds, and the resulting keys will be used to access 4 KB blocks that must be encrypted or decrypted, we believe that the small overhead for keyed hash calculation is reasonable. Also, Horus need not use the same block sizes or tree depth for all files; instead, it can maintain a separate block size list and tree depth for each file, or (for storage efficiency) have files select from a set of predetermined depths and block size lists.

Since files in petascale HPC file systems often expand dynamically, with multiple clients writing to locations past the original end of the file, Horus can generate a key for any region given the root key for the file. This property allows Horus to pre-calculate keys for regions that haven't yet been written, and also ensures that any two nodes can arrive at the same value for a block key without exchanging information beyond the root key. Thus, Horus can scale

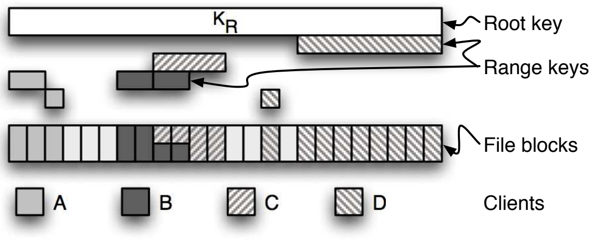

Figure 4: Sample set of keys for ranges of blocks in a file. Clients B and C can both access blocks 8 and 9.

to thousands of clients because there is little dependency, and hence little need for synchronization, to generate block encryption keys.

An example scenario for our approach to partial file encryption is shown in Figure 4. Client *A* accesses blocks 0–2, *B* accesses blocks 6–9, *C* accesses blocks 8–11, and *D* accesses block 14 and blocks 16–23. In this scenario, *A*, *B*, and *D* each receive two keys (albeit at different levels of the KHT), and *C* receives a single key. Note that, though *B* and *C* both can access blocks 8 and 9, they receive keys from different levels of the KHT. However, the block keys derived from the different keys are the same; in fact, *B*'s key for the 8–9 range can be derived from *C*'s key for the range 8–11.

4. DESIGN ISSUES

While the basic design of the KHT approach for encryption of block ranges enables the use of thousands of keys to encrypt different regions of each file while requiring minimal storage requirement for a small fixed number of root keys per file and reducing network bandwidth for key distribution, it still requires that the root keys be stored and distributed securely. In addition, key revocation is a potential issue. Since it may be impractical to rewrite file systems to include Horus, we also describe techniques for implementing Horus as a library layer on top of existing file systems or other storage mechanisms such as HDF5 [21].

4.1 Protecting file root keys

File root keys in Horus must be protected carefully, since disclosure of a file root key compromises the entire file. Thus, Horus uses lockboxes [11, 16] to secure file root keys. Lockboxes can use either symmetric key encryption or public key encryption to provide both confidentiality and integrity for the file root key; the usual tradeoff of speed (symmetric) against flexibility (public key) applies. Since the lockboxes are themselves encrypted using a key unknown to the metadata servers and the rest of the file system, they may be stored in the file system, either as metadata or as files. If Horus is implemented as a client library (Section 4.4), file key lockboxes could be stored in one of three ways: separate files, file system-based metadata such as extended attributes (a POSIX standard), or metadata in a format such as HDF5.

4.2 Distributing block range keys

It may be unwise to trust the MDS to securely distribute block range keys, since the MDS must maintain file system state and is a likely target for attackers. Instead, range keys for Horus can be calculated and distributed by an independent key distribution cluster (KDC), implemented as a set of processes that can be run on one or more stateless nodes. The KDC must have access to the lockboxes for the access-controlled files and keys to decrypt the lockboxes, and must also have a mechanism for determining which clients need access to which parts of which files. The first two requirements are straightforward: the KDC can obtain lockboxes from the

MDS or from files in the file system, or it can have them supplied by clients with key requests. Lockbox keys are then supplied by the user. The third requirement is more challenging, and can be done either by a model of the computation run on the KDC or by explicit request from clients for the range keys that they need. These two options map well to approaches used by HPC computing: the first corresponds to a computation for which work is pre-assigned to nodes, and the second approach could be implemented by integrating key distribution with whatever (typically centralized) system is used to assign work to client nodes. Once the KDC has generated the necessary range keys, it must supply them to clients. There are many systems for securely providing keys to clients; Maat [13] does so in an HPC environment, but any scheme that encrypts communications (including SSL) will suffice.

Since key generation is fast (a few microseconds per keyed hash), a single KDC node can generate tens of thousands of range keys per second; the performance-limiting factor for the KDC is more likely to be network communication or access control decisions. If the KDC becomes a bottleneck, it can be scaled out by adding more nodes; the only information that must be shared in the KDC is lockboxes, lockbox keys and access control policy. Only access control policy requires synchronization, and it only requires synchronization if individual KDC nodes can't each arrive at the same decision by independently running (potentially the same) code.

4.3 Key revocation

Encryption keys can be revoked *only* at the granularity of an entire file by modifying the root key for the file and then reading, re-encrypting, and rewriting all of the data. It is possible to support multiple versions of the root key, with different parts of the file using different root keys. However, doing so would require that the system maintain a copy of all of the root keys (or provide a mechanism for deriving them, as in Plutus [11]), and also provide a way for determining *which* root key to use for which regions. If revocation is infrequent, a range key could include a key derived from each version of the root key; the client could then either use hints stored with the file or try each version in turn to see which one is valid for the given range.

4.4 Using Horus as an encryption layer

While Horus provides strong protection for petascale storage, it may be impractical to integrate it into existing file systems. In addition, users of file formats such as HDF5 [21] may wish to apply Horus to some (or all) of the data sets, rather than applying it on a file block basis. Fortunately, Horus can easily be implemented as an encryption layer, since it only requires a mechanism for distributing keys to clients and support for encryption on the client. A KDC may be implemented separately from the rest of the file and computation system, as described above, and it is straightforward to support encryption in user-level libraries. If Horus is layered on top of a file format library such as HDF5, Horus can even use the logical data set offsets rather than offsets in the actual file, potentially making ranges more contiguous and reducing the number of range keys that must be distributed to clients.

5. SECURITY ANALYSIS

Using a keyed hash tree to generate the encryption keys for each block of a file with millions or even billions of blocks greatly improves the security of HPC file systems. This section discusses the ability of Horus to resist a range of attacks [18], and shows that it can be used to thwart many attacks on data confidentiality and integrity.

Data in Horus is never stored in the clear on a disk. Thus, a subverted disk cannot yield data in the clear. The only way that a subverted disk can give up cleartext data is with the necessary key to decrypt it, and the disk has no way to gain such a key.

In many systems, the metadata server (MDS) must be trusted with the confidentiality of data stored in the file system; however, Horus removes this need and prevents the MDS from being able to expose data stored in a file system it manages by using lockboxes. Unless the MDS gains the key needed to open a lockbox, it cannot obtain a file's root key and is thus unable to generate any of the block keys.

While an MDS cannot expose data, however, it *can* execute a denial of service attack by refusing to provide a requested lockbox. Of course, the MDS could also refuse to provide location information for a file; in general, there is little that can be done to prevent a compromised MDS from denying access to files in *any* file system.

While the MDS and the disk cannot decrypt data stored on disk, clients *must* be able to do so in order to use the data. Thus, at least *some* clients need to be able to read and write data. The goal, then, is to ensure that clients can only access data they need, without allowing them to access other data in a file.

As noted above, each client is only given the key(s) that allow it to generate the block keys for data to which it has access. Because a client cannot derive the parent key for any keys it has, and cannot "extend" a key to neighboring ranges at the same level, it cannot "escape" out of the ranges to which it has keys. This is particularly effective in large-scale HPC clusters with thousands of nodes, since there may be tens of thousands of nodes, each of which may only need access to 0.1% of the entire file. If a node in such a system is compromised, the intruder only gets a small fraction of the data; while revealing any data is harmful, 0.1% is far less dangerous than an entire file.

For read access to a disk, Horus eliminates the need for an authentication mechanism because a client who reads the data cannot decrypt the data without having the corresponding key. For write access, however, an authentication mechanism such as that in Maat [13] has to be in place to avoid unauthorized clients from writing garbage into the storage disks. Mechanisms such as Quota [17] can also be adopted to prevent a client from writing to more storage space at a storage device than its allocated quota.

While a disk cannot decrypt data stored on it, a subverted disk could provide fabricated data to clients. There are several techniques for preventing this, including the use of cryptographic hashes encrypted with the block key along with the data [16] and public-key signature of the data blocks [11]. The latter approach is more secure, and may leverage techniques from the Plutus file system [11] to ensure that data is only written by authorized writers. However, writers in Plutus must generate a new block key when a block is written, and must then sign the root of the tree that contains that block. Since our approach uses only a single root key, the Plutus approach cannot verify that individual block keys are valid. However, it can be used by a writer to sign the hash for an individual block, though this may be too slow for use in an HPC system.

Simply storing a cryptographic hash of a block encrypted along with the block data is sufficient to ensure that only authorized clients have modified the data, though it cannot distinguish between readers and writers. It is also vulnerable to attacks in which a disk provides old versions of a data block; while systems such as SUNDR [14] guard against this, the high overhead of such systems, particularly for petabytes of data, make such a high level of security impractical.

The KDC distributes range keys to individual clients, allowing them access only to the portions of the file permitted by the KDC.

Since the KDC is stateless, it is more secure than having the MDS manage key distribution [2] because it can be totally wiped between computations. Key distribution must also be done securely; however, there are many schemes such as Maat [13] that can securely distribute keys to many clients. Moreover, if a small number of messages are compromised, yielding a few range keys, only a small amount of data is leaked.

As with other encrypted file systems, data in Horus is vulnerable to threats such as brute-force attacks on ciphertext and weaknesses in encryption algorithms. Horus is also vulnerable to attacks that rely upon stolen keys; again, these attacks are common to all encrypted storage systems.

6. CONCLUSIONS

Large-scale file systems for high-performance computing must store very large files with sensitive data that are shared by thousands of compute nodes in environments that are increasingly subject to attack. To address this problem, we developed Horus, which encrypts data to prevent compromised storage devices and malicious metadata servers from accessing *any* file data in the clear, and also limits data leakage from a compromised client to *only* the parts of a file that the client is authorized to access. Horus provides this security using a keyed-hash tree, allowing it to efficiently provide file range keys to individual clients while limiting the overall key storage overhead and using minimal CPU time in key generation. The Horus mechanism can be used both to confine the accesses made by individual clients, and to hide sensitive parts of a large file from public view.

Since computation of block keys depends only on a file's root key and per-access information, computation of range keys and block encryption keys is highly parallelizable, making it ideal for use in HPC systems. Horus only requires a set of processes to distribute keys to clients to encrypt and decrypt data; thus, Horus can not only be integrated into the file system, but can also run as a client library operating either on file blocks or on higher-level constructs such as HDF5 data sets. Because Horus can run as a client library or integrated into the file system while providing significantly stronger confidentiality guarantees than currently-available approaches to HPC storage, it is ideally suited for use in increasing the security of data stored on very large-scale storage systems.

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 *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST '03)*, pages 159–174, Mar. 2003.

[2] R. Anane, S. Dhillon, and B. Bordbar. Stateless Data Concealment for Distributed Systems. *Journal of Computer and System Sciences*, 74(2):243–254, Mar. 2008.

[3] S. A. Banachowski, Z. N. J. Peterson, E. L. Miller, and S. A. Brandt. Intra-file security for a distributed file system. In *Proceedings of the 19th IEEE Symposium on Mass Storage Systems and Technologies*, pages 153–163, Apr. 2002.

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

[5] K.-C. Chan and S.-H. G. Chan. Key management approaches to offer data confidentiality for secure multicast. *IEEE Network*, 17(5):30–39, Sept. 2003.

[6] W. Diffie and M. E. Hellman. New directions in cryptography. *ACM Transactions on Internet Technology*, IT-22(6):644–654, Nov. 1976.

[7] A. Fiat and M. Naor. Broadcast encryption. In *Proceedings of CRYPTO '93*, pages 480–491, 1993.

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

[9] HDFS users guide. `http://hadoop.apache.org/common/docs/current/hdfs_user_guide.html`, Aug. 2011. Version 0.20.204.0.

[10] R. Indrajit, S. T. V. Setty, A. Kilzer, V. Schmatikov, and E. Witchel. Airavat: Security and privacy for MapReduce. In *Proceedings of the 7th Symposium on Networked Systems Design and Implementation (NSDI '10)*, Apr. 2010.

[11] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus: scalable secure file sharing on untrusted storage. In *Proc. of the 2nd USENIX Conference on File and Storage Technologies (FAST '03)*, pages 29–42, Mar. 2003.

[12] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-hashing for message authentication, Feb. 1997.

[13] A. W. Leung, E. L. Miller, and S. Jones. Scalable security for petascale parallel file systems. In *Proceedings of SC07*, Nov. 2007.

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

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

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

[17] K. T. Pollack, D. D. E. Long, R. Golding, R. Becker-Szendy, and B. C. Reed. Quota enforcement for high-performance distributed storage systems. In *Proceedings of the 24th IEEE Conference on Mass Storage Systems and Technologies*, pages 72–84, Sept. 2007.

[18] E. Riedel, M. Kallahalla, and R. Swaminathan. A framework for evaluating storage system security. In *Proceedings of the Conference on File and Storage Technologies (FAST)*, 2002.

[19] F. Schmuck and R. Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the Conference on File and Storage Technologies (FAST)*, pages 231–244, Jan. 2002.

[20] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop distributed file system. In *Proceedings of the 26th IEEE Conference on Mass Storage Systems and Technologies*, May 2010.

[21] The HDF Group. HDF5 user's guide. `http://www.hdfgroup.org/HDF5/doc/PSandPDF/HDF5_UG_r187.pdf`, May 2011. Release 1.8.7.

[22] 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 the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, Nov. 2006.

[23] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou. Scalable performance of the Panasas parallel file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST)*, pages 17–33, Feb. 2008.