Horus: Fine-Grained Encryption-Based Security for Large-Scale Storage

Yan Li[1], Nakul Sanjay Dhotre[1], Yasuhiro Ohara[1],
Thomas M. Kroeger[2], Ethan L. Miller[1], Darrell D. E. Long[1]
[1]*Storage Systems Research Center, University of California, Santa Cruz, CA 95064, USA*
[2]*Sandia National Laboratories, Livermore, CA 94550, USA*

Abstract

With the growing use of large-scale distributed systems, the likelihood that at least one node is compromised is increasing. Large-scale systems that process sensitive data such as geographic data with defense implications, drug modeling, nuclear explosion modeling, and private genomic data would benefit greatly from strong security for their storage. Nevertheless, many high performance computing (HPC), cloud, or secure content delivery network (SCDN) systems that handle such data still store them unencrypted or use simple encryption schemes, relying heavily on physical isolation to ensure confidentiality, providing little protection against compromised computers or malicious insiders. Moreover, current encryption solutions cannot efficiently provide fine-grained encryption for large datasets.

Our approach, Horus, encrypts large datasets using keyed hash trees (KHTs) to generate different keys for each region of the dataset, providing fine-grained security: the key for one region cannot be used to access another region. Horus also reduces key management and distribution overhead while providing end-to-end data encryption and reducing the need to trust system operators or cloud service providers. Horus requires little modification to existing systems and user applications. Performance evaluation shows that our prototype's key distribution is highly scalable and robust: a single key server can provide 140,000 keys per second, theoretically enough to sustain more than 100 GB/s I/O throughput, and multiple key servers can efficiently operate in parallel to support load balancing and reliability.

1 Introduction

High performance computing (HPC) systems are used in many critical computation tasks, such as simulation of new energy sources as used by the Department of Energy [12], finite element analysis for structural simulation, and new drug development. However, much of this information must remain confidential due to either national security or business concerns.

Many of today's HPC systems rely on the perimeter defense model: strong physical defense is deployed around a system's perimeter, but the security within the defense perimeter is relatively loose. Per our discussion with U.S. Department of Energy researchers, this "hard exterior, soft interior" model is failing[1]. The lack of security in HPC systems is becoming an issue because individual systems are very expensive, and thus must be shared between multiple projects of different classification levels. Increasingly, there are also concerns over information leakage from malicious insiders.

Unfortunately, current data encryption technologies don't work well with petascale datasets: they are either coarse-grained or have high key management overhead, and they fail to provide security in the face of a single component compromise or untrusted operator.

To address these problems, we developed Horus[2], a novel approach to data encryption for large-scale systems that expands on the preliminary ideas proposed by Rajendran et al. [28]. In this paper, we present a full design for Horus, describe an implementation and its evaluation, and give a thorough performance discussion. Horus provides fine-grained encryption-based security for very large datasets with low key distribution overhead by leveraging keyed hash trees (KHTs) to generate different keys for each region of the file. A client can only access a file region for which it has a key, and cannot use its keys to access other regions. The tree structure allows keys to be generated for large and small regions as needed. Each client only has the minimal set of range keys it needed to finish its work, limiting data leakage when multiple jobs are running on the same HPC nodes. Furthermore, Horus confines the server-side key calculation function to a key distribution cluster (KDC) that is independent from both the persistent file store and the nodes doing file I/O. In

[1]Private communication.
[2]Horus is the god of protection in ancient Egyptian religion.

this arrangement, the root key for a file's KHT is stored encrypted in the file system, and only the KDC sees the root key; the computation nodes only receive the keys they require. Thus, a single compromised node can only reveal data it can access; data from the rest of the file remains safe. Deployment of Horus is straightforward because only small changes are needed to existing systems: Horus can be integrated into an existing file system or used as an encryption layer in existing middleware systems.

This paper presents first describes the design of Horus. We then discuss the implementation of our prototype system. Our evaluation of the prototype shows high performance and scalability: one key distribution server was able to provide 140,000 keys per second, sufficient to sustain more than 100 GB/s I/O throughput. In the BTIO Class C benchmark, Horus write performed much better than the traditional fine-grained encryption mode and was only 6.9% slower than the theoretical AES encryption speed upper bound.

2 Background

A typical HPC architecture [14, 8] includes a large number of compute nodes, a relatively small number of special purpose servers, e.g., metadata servers (MDS) and I/O nodes, and a separate dedicated storage system, as shown in Figure 1. Large parallel file systems such as Lustre [7], PFS [36], Ceph [35], Hadoop [32], and GPFS [30] decouple data control and data access paths, allowing management and security decisions to be centralized in a small metadata 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 or the I/O node to provide security via capabilities must place complete trust in those nodes.

The low level of security in HPC systems raises the following issues:

1. HPC systems are often shared among applications because they are very expensive to build and run. Most HPC compute nodes are running processes as the root user, making stealing information from other tasks or previous tasks from the same node very easy.

2. Without encryption, all storage media must be carefully tagged, guarded, and tracked; losing media can mean data leakage. The management cost can be very high for large datasets. For HPC applications running in the cloud, tracking all storage media is impossible.

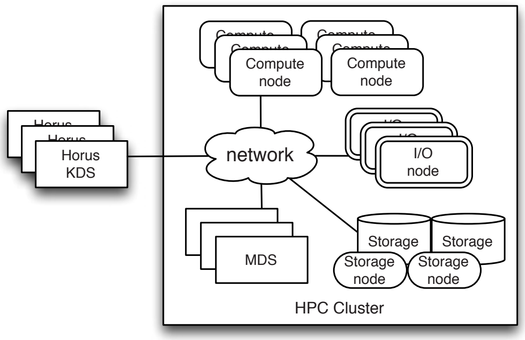


Figure 1: A simplified HPC model. There are many compute nodes with separate dedicated storage connected via a network. The system typically includes a number of special nodes, such as I/O nodes and/or MDS. The Horus key distribution servers are shown as independent nodes outside the cluster.

3. Encryption can be used to secure data on disk, but key management for petascale datasets is a huge burden. If the data owner needs fine-grained security and encrypts a dataset of 1 TB using a unique key for each 4 KB data block, there will be more than 268 million keys to store, track, and distribute. If each key is 256 bits, those keys will take 8 GB to store, and will create a performance disaster during key distribution. We call this distributed-all-block-key method the naïve key distribution mode.

4. The "trust barrier": software, system operators, or cloud service providers need to be fully trusted. A compromised computer or a malicious insider can cause a great deal of damage, as exemplified by the leaked documents on WikiLeaks [6] and Google Gmail snooping scandal [23].

5. The rigid perimeter defense model limits the advancement of HPC and cloud computing technologies. Huge future computing tasks may need a cross-site, planetwise supercomputer that spans several data centers or institutions; the defense perimeter will be so huge that defending it will become very expensive or even impossible.

6. Prior studies have shown that adding local storage to compute nodes can improve an HPC system's performance by alleviating the checkpoint-restart overhead [14, 8]. However, concerns over exposing locally-stored sensitive data to other processes on the same compute node have hindered the adoption of this optimization technique, in part because of the insecurity of per-node local storage.

The last issue is looming larger, as HPC architectures look to node-local storage both for checkpointing and overall integration of storage into the computing framework [10]. While this approach has the potential to greatly improve performance, it also poses a significant security risk, as data from prior computations may still be stored on a node's local storage, exposed to attacks by the currently-running computation.

Recently, there have been several attempts to provide greater security for HPC file systems. Maat [21] provides scalable authorization and authentication, albeit not data confidentiality, for the Ceph file system. 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 a malicious insider can obtain data from stolen storage media since it is stored in the clear. There have also been attempts to integrate stronger security into MapReduce frameworks [17], but they are primarily limited to network authentication protocols and encrypting data transfers, and do not support storing encrypted data on disk. Airavat [18] further confines computations on data in the MapReduce framework, but still operates on data stored in the clear on storage nodes.

Actual encryption of portions of HPC files was proposed by Banachowski, et al. [5]. 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.

Cloud service providers are also providing primitive encryption for HPC applications, such as Amazon's server-side [2] and client-side encryption [3]. However, with server-side encryption, the data owner must fully trust the service provider, and all data blocks can be decrypted by one master key, which provides a single point of compromise. With client-side encryption, data owners must do their own key management, which can be a huge burden as described above.

While strong security is rare in HPC file systems, there are many approaches to providing strong security for smaller-scale file systems. Systems such as Cepheus [16], SNAD [25], and Plutus [19] 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, but it does not scale well to fine-grained protection of terabyte-scale files because of the overhead of key storage and capability distribution.

Rajendran et al. [28] described an initial design for Horus, which uses a keyed hash tree to generate keys for different block ranges of a large file. When a cryptographically strong hash function is used with the KHT, the properties of a KHT guarantee that deriving lower level keys *within* the range covered by a higher level key is easy, but it's mathematically "difficult" to derive any key for a region *outside* that range. Hash trees have long been used for authentication [24] and integrity checking [22, 33]. There are methods using hash trees to reduce the storage requirements for key management, as described by Fiat and Noar [15] and Chan and Chan [11], who also describe a contributory key agreement protocol based on Diffie-Hellman key exchange [13] 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 employs a simple client-server model, where the client requests the needed keys from a key distribution server (KDS) and uses the keys returned by the KDS to encrypt/decrypt each block of the data. Given the *root key* and the access location for the file, anyone can calculate the *leaf key* that is actually used to encrypt/decrypt the file. Theoretically, the KDS can be located at any part of the current HPC architecture (i.e., I/O node, storage node, MDS, and even one of the compute nodes in Figure 1). Alternatively, it can be placed independently, or can be clustered as a key distribution cluster (KDC). The client function can be implemented as a part of the file system, a part of a system library such as HDF5 [34], an independent dynamically linked library, or a software

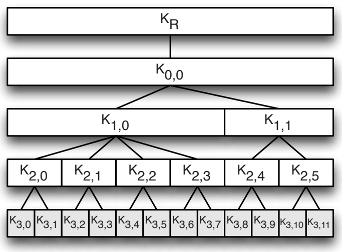

K_R
$K_{0,0}$
$K_{1,0}$
$K_{1,1}$
$K_{2,0}$ $K_{2,1}$ $K_{2,2}$ $K_{2,3}$ $K_{2,4}$ $K_{2,5}$
$K_{3,0}$ $K_{3,1}$ $K_{3,2}$ $K_{3,3}$ $K_{3,4}$ $K_{3,5}$ $K_{3,6}$ $K_{3,7}$ $K_{3,8}$ $K_{3,9}$ $K_{3,10}$ $K_{3,11}$

Figure 2: Basic keyed hash tree design [28]. Keys at lower levels of the tree control smaller regions of the file. The leaf nodes (shaded) 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.

```
Require: $0 \leq start < end < d$
1: for $x \leftarrow start + 1$ to $end$ do
2:     $y \leftarrow \lfloor b/bs_x \rfloor$
3:     $k \leftarrow$ keyed_hash$(k, x||y)$
4: end for
5: 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 $bs_0, \ldots, bs_{d-1}$, where d is the number of levels in the tree, including the root key and leaf nodes (at level $d-1$). If the end level is $d-1$, the resulting key may be used for data encryption.

module incorporated in each application software, either on the compute node or on the I/O node.

For SCDN, encrypted data and KHT root keys, which are secured in lockboxes, can be stored in a cloud storage service, while the KDC is operated separately by the data owner. Clients who need to access the data can query the KDC to get the proper range keys. This approach reduces the burden of trust on the storage service provider.

For applications that run on nodes in the cloud, the owner still must trust the service provider, who, theoretically, can commandeer all of the client nodes to collect all the range keys. However, the need to trust *every* administrator of the provider is lifted; in practice, it's generally more secure to trust Amazon or Google as a company than to trust individual Amazon or Google employees. Moreover, these properties enable efficient and simple security designs, with a higher barrier to intrusions.

3.1 Key calculation

Horus uses KHTs to derive unique encryption keys for individual blocks from a per-file root key K_R; the root key is the only key that need be permanently stored. The "block" size at each level of the KHT is a fixed size, with block size decreasing at lower levels of the tree. The keys at the leaf nodes of the KHT, called the *leaf keys*, 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, the key for region y at level $x \geq 0$ 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 bit-string 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 in Figure 2, K_{parent} for $K_{2,1}$ is $K_{1,0}$, so $K_{2,1} = KH(K_{1,0}, 2||1)$. To derive a leaf key from any key above it in the tree, this calculation can be applied recursively, 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 satisfies this condition can be used; the keyed hash function can be easily changed if weaknesses are found in a particular algorithm or if a particular algorithm has computational advantages.

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 $K_{3,0}, \ldots, K_{3,7}$. Both $K_{2,1}$ and $K_{1,0}$ can be used to generate the *same* value for $K_{3,2}$.

The KHT can be adjusted to handle any level of granularity that is a multiple of the size of a single encryption block, even permitting branching factors that are not powers of 2. However, there is a tradeoff between having a smaller "branching factor"—the number of children of a node—and having a shorter tree with fewer levels. While fewer children per node provides finer granularity for aggregating the keys in upper nodes, it also results in longer key calculation times because the tree height of the KHT has to become deeper to support the same range for a given internal tree node, resulting in more hash calculations. For example, in Figure 2, where the minimum granularity of encryption is a 4 KB block, level 0 of the tree only covers 64 KB; a node that wanted access to a 1 MB region would have to obtain 16 keys. For a terabyte-sized file, level 0 would contain 16 million keys. Instead, we expect that petabyte-scale file systems will have regions of 16 MB–1 GB at level 0 of the KHT, with branching factors of 4–8 for successively lower levels of the tree. The block size at the next lower level (child level) can be derived by dividing the block size of the level by the branch factor. With a level 0 block size of 1 GB and a reduction of 8 for each successive level, a KHT would require 5 additional levels between level 0 and the 4 KB leaf nodes. Since a keyed hash calculation on modern processors can be done in less than a microsecond, the small overhead for keyed hash calculation is reasonable, as we show in Section 5.

However, Horus need not use the same block size or tree depth for all files; instead, it can maintain a separate block size list and tree depth for each file. Assuming block sizes are always powers of two, the base block size (2^B) and branching factor (2^{f_i}) at each level of a KHT of depth d can be represented as $\langle B, f_0, \dots, f_{d-1}\rangle$, where $0 \leq B \leq 255$ and $0 \leq f_i - 1 \leq 15$, requiring just $1+(d-1)/2$ bytes to describe the KHT.

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. Horus can thus pre-calculate keys for regions that haven't yet been written, and ensure that any two nodes can arrive at the same value for a block key without exchanging information beyond the root key, enabling the calculation of encryption keys for each block independently and simultaneously using the same root key.

The idea of using a KHT to encrypt a large file [28] can be generalized to datasets in other forms. Horus leaf keys can be used to encrypt the smallest data unit, which can be an object, blocks of a large object or file system, a trunk, or one database record. For example, in our prototype implementation, Horus works on block level and encrypts each file block by using a unique leaf key; for systems that store data in the Hadoop Distributed File System (HDFS) [17], Horus can use the leaf keys to encrypt data trunks of 64 MB; for customers' credit card information stored in a database, Horus can encrypt each record by using a unique leaf key. Thus, Horus can work with a wide range of datasets regardless of their data layout and structure.

3.2 Key exchange protocol

We have designed our key exchange protocol so that Horus can be used in distributed parallel file systems that decouple the data control and the data access path, such as Ceph [35], Lustre [31], and PVFS2 [9]. In this way we can ensure that Horus can perform and scale well in parallel file systems as well as in file systems with simpler models. We denote the entity that manages files as MDS and the entity that stores the data as ST.

$$A \rightarrow \text{MDS} \quad : \quad f, E_{KU_{\text{KDS}}}(K_R), E_{KU_A}(K_R), BS, P_0, P_1, \dots \tag{1}$$

$$C_i \rightarrow \text{KDS} \quad : \quad f \tag{2}$$

$$\text{KDS} \rightarrow \text{MDS} \quad : \quad f, C_i \tag{3}$$

$$\text{MDS} \rightarrow \text{KDS} \quad : \quad f, E_{KU_{\text{KDS}}}(K_R), BS, P_k \tag{4}$$

$$\text{KDS} \quad : \quad K_R \Leftarrow E_{KR_{\text{KDS}}}[E_{KU_{\text{KDS}}}(K_R)] \tag{5}$$

$$\text{KDS} \quad : \quad \mathscr{C}_{C_i,f,x_i,y_i} \Leftarrow E_{KU_{C_i}}[HKC(K_R, BS, x_i, y_i)] \tag{6}$$

$$\text{KDS} \rightarrow C_i \quad : \quad f, \mathscr{C}_{C_i,f,x_i,y_i} \tag{7}$$

$$C_i \quad : \quad K_{f,x_i,y_i} \Leftarrow E_{KR_{C_i}}(\mathscr{C}_{C_i,f,x_i,y_i}) \tag{8}$$

$$C_i \rightarrow \text{MDS} \quad : \quad f \tag{9}$$

$$\text{MDS} \rightarrow C_i \quad : \quad f, BS \tag{10}$$

$$C_i \quad : \quad K_{d-1,*} \Leftarrow HKC(K_{f,x_i,y_i}, BS, d-1, *) \tag{11}$$

$$C_i \rightarrow \text{ST} \quad : \quad \mathit{data\_request} \tag{12}$$

$$\text{ST} \rightarrow C_i \quad : \quad \mathit{encrypted\_data} \tag{13}$$

$$C_i \quad : \quad E^{-1}_{K_{f,d-1,*}}(\mathit{encrypted\_data}) \tag{14}$$

Notation	Description
A	a user application.
C_i	i-th compute node.
f	the file identity (e.g., file path).
$E_k(D)$	encrypted D by using key k.
E_k^{-1}	decryption using key k.
HKC	the Horus key calculation operation.
MDS	a metadata server.
ST	the storage of the file.
KDS	a key distribution server.
(KU_A, KR_A)	the public/private key pair of A.
$(KU_{\text{KDS}}, KR_{\text{KDS}})$	the public/private key pair of KDS.
d	the length (depth) of KHT.
$BS = \langle bs_0, ..., bs_{d-1}\rangle$	the block sizes for each KHT level.
$P = (C_i, x_i, y_i)$	an access control (permission) entry allowing C_i access to f over the range $\langle x_i, y_i\rangle$.
K_R	the root key.
K_{f,x_i,y_i}	the range key for f at position x_i, y_i
$\mathscr{C}_{C_i,f,x_i,y_i}$	range key armored for C_i
$K_{f,d-1,*}$	the leaf keys.

Figure 4: An example sequence of the key exchange protocol. As an optimization, values that have already been transmitted need not be resent. For example, the KDS need not retrieve the armored K_R on every request.

We explain the protocol sequence in Figure 4. The root key for a file K_R is normally stored using a lockbox [25, 19] protected by a user's public key KU_A. Rather than giving KDS the user's private key, however, applications are more likely to create a new K_R when a file f is created, and use the KDS's public key to protect the root key; thus, they would store K_R protected by both KU_{KDS} and KU_A. The application must then prepare the Horus configuration $\langle E_{KU_{\mathrm{KDS}}}(K_R), E_{KU_A}(K_R), BS, P_0, P_1, \ldots\rangle$ for file f. BS describes the KHT configuration for this file, and $\langle P_0, P_1, \ldots\rangle$ is a set of permissions listing the allowable accesses to the file from a given process or node. The application stores this Horus configuration in a lockbox in the MDS ((1) in Figure 4). This way, though the MDS stores the root key K_R, only the KDS (and the user) can access the data in the clear. The KDS can access the lockbox and can obtain the clear K_R by decrypting with its own private key (KR_{KDS}).

When a compute node C_i needs to access the file f, it requests from KDS the key for a range to which it has access (2). First, the KDS obtains the Horus configuration $\langle E_{KU_{\mathrm{KDS}}}(K_R), BS, P_0, P_1, \ldots\rangle$ for the file f from the MDS (3), (4). Next, the KDS tests if C_i is allowed to access the range by consulting the list of permissions $\langle P_0, P_1, \ldots\rangle$ for C_i. If the client C_i is allowed, the key K_{f,x_i,y_i} is calculated by the KDS, using K_R and the KHT configuration BS (5), (6). The KDS returns the key K_{f,x_i,y_i} encrypted by the public key of C_i ($\mathscr{C}_{C_i,f,x_i,y_i} = E_{KU_{C_i}}(K_{f,x_i,y_i})$) to C_i (7), so that only an authorized C_i that has the private key KR_{C_i} can decrypt the range key K_{f,x_i,y_i} (8), and then C_i can calculate the rest of the KHT tree down to the leaf keys (9)–(11). Alternatively, a per-node symmetric key known to the KDS and C_i can be used in place of the public/private key pair, similar to what is done in Kerberos [26].

The major advantage of this protocol is that the KDS is essentially stateless, and hence highly parallelizable and securable. The MDS has persistent copies of keys, but lacks the private keys to decrypt them. Also notice that the protocol is simple; most of the transactions are used to obtain the Horus configuration from the MDS and can be further simplifiied.

The communication between A, C, KDS, MDS, and ST must only be carried out after successful authentication. We do not provide authentication methods in this paper, relying instead on existing techniques. However, we anticipate that it might be possible to apply a stronger independent authentication mechanism, since the KDS can run independently from the other systems.

3.3 Protecting file root keys

The KHT root key of a file must be stored in the storage system, because we want the KDS to be stateless. File root keys in Horus must be protected carefully—disclosure of a file root key compromises the entire file—so, Horus uses lockboxes [25, 19] 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 trade-off of speed (symmetric) against flexibility (public key) applies. Since the lockboxes are themselves encrypted using a key unknown to either the MDS, the storage or the rest of the file system, the lockboxes may be stored in the file system, either as metadata or as files. If Horus is implemented as a client library as discussed in the following section, 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.

3.4 KDC design

Since the key management function in Horus is decoupled from other file system related metadata management functions, the function can be implemented as a set of processes running on one or more stateless KDS nodes, constructing an independent key distribution cluster (KDC). If the KDC becomes a bottleneck, it can be scaled out by adding more nodes.

In order to manage the encryption keys securely, the KDC must have access to the lockboxes 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 requirements for access to the lockboxes and the lockbox keys are straightforward: the KDC can obtain lockboxes from the MDS (as in the previous section), from files in the file system, or it can have them supplied by clients with key requests. Lockbox keys are then supplied by the creator or owner of the files.

Determining the access control permission for clients is similar and can be done either by a pre-determined policy model that is decided by a deterministic computation on the KDC, or by a configuration based model where explicit range requests from clients are tested against the settings retrieved from the MDS (protected in a lockbox).

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 are unable to arrive at the same decision by independently running (potentially the same) code.

3.5 Client implementation

Ideally, the Horus client functions should be implemented in the client file system layer for performance and supporting existing applications transparently. In the real world, changing file system code may be difficult or expensive, therefore makeshift file system overlays, such as FUSE, may be used instead. When changing file system is infeasible, the functions can be integrated into applications directly.

In addition, users of file formats such as HDF5 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. If Horus is layered on top of a file format library such as HDF5, Horus can use 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.

3.6 Key revocation

Keys may need to be revoked for reasons like security breaches or revoking user accounts, and the data that was encrypted by using those old keys needs to be re-encrypted by using a new key. With Horus, those old keys can be range keys from a KHT. We don't need to revoke the whole KHT, instead we can allocate a new KHT for those ranges and keep a mark in the metadata, designating ranges of the data that should be accessed by using the new KHT. Many versions of KHTs can coexist. For systems where key revocation is infrequent, this design incurs little overhead and is simple to implement. But if the key revocation happens very often, the system may end up with too many KHTs. For such systems, extra rounds of garbage collection and re-encryption can be carried out periodically to bring the system back to full performance. A full discussion of key revocation is beyond the scope of this paper and will be addressed in future work.

4 Security Analysis

There are many possible attacks against sensitive data in large-scale storage systems; in this section, we show how Horus can help to thwart them. For those attacks that the current design of Horus can't stop, we outline possible ways to expand the design in the future.

Data in Horus is never stored in plaintext on a disk. Thus, even when a disk is subverted, it is impossible to obtain cleartext data from it. Without the necessary file root key to decrypt it, the data is unreadable, and there is no way to gain such a key from the disk other than unlocking the lockbox holding the key, which would require the appropriate private key.

In many systems, the metadata server must be trusted with the confidentiality of data stored in the file system. 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 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 keys that allow it to generate the block keys for data to which it has access. A client cannot derive the parent key for any keys it has and cannot "extend" a key to neighboring ranges at the same level, so it cannot "escape" out of the ranges for which it has keys. As long as a client is not given a private key to decrypt a lockbox and obtain a file's root key, it can only access the blocks for which it has been given range keys. This approach is particularly effective in large-scale HPC clusters with 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 typically far less dangerous than an entire file.

For read access to a disk, Horus reduces 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, Horus alone cannot prevent unauthorized clients from writing garbage into the storage disks. Actually preventing writes to files is outside the scope of Horus, though it could done by an authentication mechanism such as that in Maat [21]. Quotas [27] could also be adopted to prevent a client from writing to more storage space at a storage device than its allocated quota.

Although stored data is protected from disclosure, a subverted disk could fabricate data to provide to clients. There are several techniques for preventing this, including the use of cryptographic hashes encrypted with the block key along with the data [25] and public-key signature of the data blocks [19]. The latter approach is more secure, and may leverage techniques from the Plutus file system [19] 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 the plaintext in a block along with the encrypted data itself 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 [22] guard against this, the high overhead of such systems, particularly for petabytes of data, make such a high level of security impractical for HPC.

Horus's ability to prevent disclosure to a subverted disk is particularly useful as non-volatile memory such as SSD is added to compute nodes [10, 14, 8]. Data stored locally can *only* be read with the appropriate range key; a hostile user cannot subvert node *X* to gain a key and subsequently use that key to read checkpoint data from the other nodes in the system. Thus, the safety of data stored on a local node depends only on preventing the leakage of keys on that node, not on the safety of other nodes. By securing data against local attacks on nearby nodes, Horus makes it safe to use local storage for checkpoint data and higher performance storage, even on systems where local processes have unfettered access to the entire node.

As in 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. However, Horus is more resistant to stolen keys than most file systems. Since the KDC is stateless, it can be restarted between computations or even run on different nodes, reducing the likelihood of key leakage and making it more secure than having the MDS manage key distribution [4]. Key distribution must also be done securely; however, there are many schemes such as Maat [21] and Kerberos [26] 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.

5 Implementation and evaluation

We have implemented a prototype of Horus. The KDS is implemented as a multithreaded server process. The client is a portable library that intercepts POSIX I/O system calls. The Horus configuration is stored in file's extended attributes. The key exchange protocol is implemented using UDP. We used OpenSSL's SHA1 hash function as the keyed hash function. In our implementation the block size is fixed at 4 KB.

The following benchmarks were conducted on machines with the following setup: Intel(R) Xeon(R) CPU E5620 4-core at 2.40 GHz, RAM 24 GB, Seagate(R) Constellation.2 SATA hard drive, running Fedora 16 Linux in x86-64 mode using the Ext4 file system.

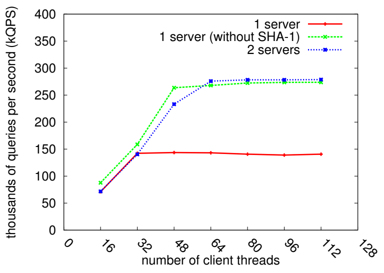


Figure 5: Performance evaluation of the KDS. A number of client threads requested leaf keys from one and two KDSes. It shows that one KDS sustains about 150 kQPS. The KHT settings were branch = 4 and depth = 11. After about 275 kQPS network became the bottleneck.

5.1 Simple leaf key request

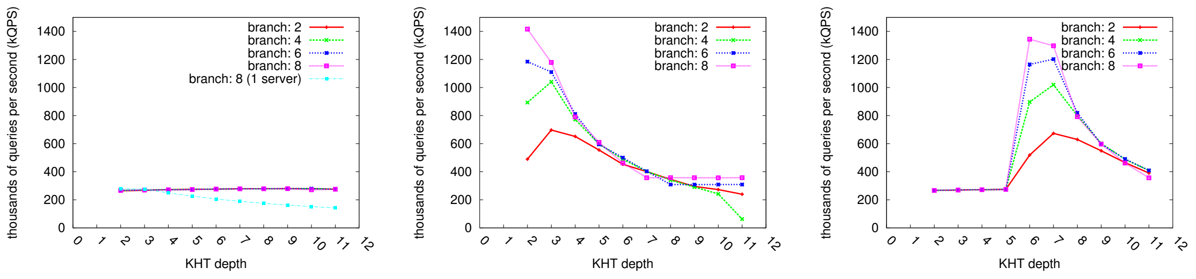
We first tested the raw performance of the KDS. We simulated multiple clients using multithreading from a single client node. The client threads request leaf keys from one or two KDSes.

Figure 5 shows the performance of requesting all 1,048,576 leaf keys necessary to access to the first 4 GB of a file, between a different number of client threads and KDSes. The figure shows that a single KDS can sustain around 140 thousand queries per second (kQPS) from 32 client threads. It should be noted that in this benchmark we wanted to saturate the KDS so the client did nothing other than sending key requests; real-world clients would request I/O in addition to requesting keys. With two KDSes, we achieved roughly 278 kQPS We also ran the benchmark with SHA1 calculation disabled in the KDS in order to measure the performance of pure network operations, and it showed that 278 kQPS is also roughly the peak that our test client can sustain. In all, it shows that one KDS can support about 140 kQPS. In real-world I/O, where we expect average block sizes to be around 1 MB, one KDS can theoretically sustain more than 100 GB/s I/O throughput. Our experiment also showed that the QPS performance increases nearly linearly with more KDSes, showing the high scalability of our Horus design.

5.2 KHT branch and depth

Figure 6 shows the performance on various KHT configurations, using a single client and two KDSes (unless otherwise noted). In this experiment, the client requests all the leaf keys of a 4 GB file.

In requesting a key from a KDS, the client has choices in the level of the key to request. If the lowest level key (leaf key) is requested, the KDS will calculate the KHT all the way down to the leaf key, and send the calculated



(a) Requesting the leaf keys (level=depth) (b) Requesting the highest level keys (level=0) (c) Requesting the keys (level=4)

Figure 6: Performance evaluation on the number of keys calculated per seconds depending on the number of branch and depth in the KHT. The number of client threads is 64, and the number of server is 2.

Table 1: KHT configuration and number of keys to be requested and calculated. Number of clients threads = 64.

Req. level	Branch	Depth	# requested	# requested/thread	# calculated/thread	# calculated total
leaf	*	*	1,048,576	16,384	16,384	1,048,576
0	2	2	524,288	8,192	16,384	1,048,576
0	2	3	262,144	4,096	16,384	1,048,576
0	2	6	32,768	512	16,384	1,048,576
0	2	11	1,024	16	16,384	1,048,576
0	4	2	262,144	4,096	16,384	1,048,576
0	4	11	1	1	1,048,576	1,048,576
0	6	11	4	1	279,936	1,119,744
0	8	2	131,072	2,048	16,384	1,048,576
0	8	11	4	1	262,144	1,048,576
4	8	6	131,072	2,048	16,384	1,048,576

key back to the client. The client just receives the leaf key. If the highest level (level = 0) key is requested, it means that the client is requesting a large portion of a file using just one key. The client will receive only a small number of keys, and must calculate for himself the KHT down to the leaf keys in order to actually encrypt/decrypt the file. Figure 6 shows the requests in (a) the lowest level, (b) the highest level, and (c) a middle level (4). 64 threads were used in both the KDS and the client. The number of keys to be requested and calculated were equally split among the client threads.

In the lowest level requests (Figure 6(a)), the performance does not change with the KHT depth or the number of branches, staying constant around 275 kQPS. This is strong performance, compared to common DNS server implementations, which start to drop requests when the requests exceed 50 kQPS [38]. When compared to the single KDS version with branch = 8 at the bottom in the figure, whose performance degrades slightly depending on the KHT depth, it appears that the SHA1 calculation in the KDS is the performance bottleneck in the case of a single KDS (supporting the discussion in the previous section), and that adding another KDS resolved the bottleneck, justifying the cluster design in Section 3.4.

In the highest level requests (Figure 6(b)), the server supplies the fewest keys, while the clients do the most calculation. Since there are many more clients than servers, this configuration supports the highest I/O performance by allowing the largest amount of I/O per server-supplied key.

This evaluation helps guide the selection of a Horus KHT configuration for different applications. The flexible configuration of Horus can balance the computation resources among the KDSes, the client, and the network, resulting in different performance. The combination of KHT branch, KHT depth, and the requesting level determines the number of keys calculated on the server, the number of keys exchanged in the network, and the number of keys calculated in the client. Table 1 summarizes the relationship between a KHT configuration and the number of keys. When the number of keys to be exchanged (# requested) is large, we put the key calculation burden on the KDS, and more network bandwidth is consumed. In contrast, if the number of keys to be exchanged in the network is small, a large number of keys must be calculated in the client locally, loading the client CPU.

Requesting keys at the leaf level puts all of the burden of key calculation on the KDS, resulting in relatively lower performance (Figure 6 (a)). If the highest level keys are used, the number of requested keys decreases as the branch and the depth of KHT grow, as shown in Table 1. The same number of keys does not necessarily mean the same performance, e.g., request level = 0, branch = 2, depth = 3, and branch = 4, depth = 2, both in Table 1 and Figure 6 (b). When the number of key exchanges is too small to be equally split among the client threads, the performance drops because all of the processing tasks to exchange and to calculate the keys will be done in a single client thread, such as shown in the case of request level = 0, branch = 4, depth = 11.

The branch and the depth determines the shape of the KHT, and in turn determines the granularity of security control. For example, requesting the highest 4 keys in the KHT configuration of branch = 6 and depth = 11 will result in the exposure of a slightly larger number of keys (1,119,744) than required (1,048,576), obtaining efficiency in network bandwidth by sacrificing a bit of security granularity.

Although there are many possible performance bottlenecks in each of the systems between the KDS and the client, the flexible configuration of Horus can balance the computation resources to avoid most of them. For example at request level = 0, branch = 8, and depth = 2, as shown in Table 1, the number of keys that will be exchanged in the network is 131,072, the number of keys exchanged per each client thread is 2,048, the number of keys that must be calculated in each of the client thread is 16,384, and 1,048,576 keys were calculated for the 4 GB file. With this KHT configuration, Horus can attain 1,416 kQPS in performance. We see similar performance in configurations that require a similar number of keys to process, such as requesting level = 4, branch = 8, and depth = 6 (See in Figure 6(c), and also the entry in Table 1).

The spike in Figure 6(c) can be controlled by changing the requesting level; it will shift left or right when the request level changes. Notice that requesting level-4 keys in a KHT with depth less than 5 is equivalent to requesting the leaf keys.

5.3 Read/write performance

We evaluate the performance of the sequential read and write operations. We used the local ext4 file system for the emulation of a high performance distributed file system in this section, in order to not depend on the performance of a certain file system implementation.

Figure 7(a) illustrates the read performance of Horus. The read operation with AES-XTS decryption performed around 165 MB/s, as shown in the line labeled "ext4+AES read". In the best case of Horus labeled "Horus read", when only one aggregated range key was requested to cover the whole file, the performance was around 115 MB/s (for the 4 GB file). The overhead was mainly caused by the AES operation.

The lowest line in the figure labeled "Horus read w/o Aggr" shows the performance of the naïve mode, where the client requests one key and waits for the response for each block before reading, results at merely 12 MB/s. Comparatively, "Horus read" has a 533% performance improvement. The poor performance of the naïve mode can be improved by prefetching. When prefetching 64 keys in advance (labeled "Horus read w/o Aggr (prefetch)"), the performance was around 64 MB/s, still only half of the performance of Horus read. This shows the efficiency of Horus over current encryption solutions.

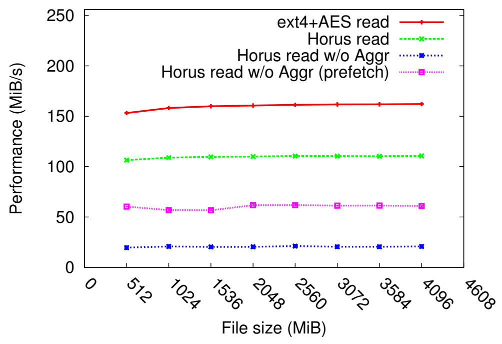
Figure 7(b) illustrates the performance for write operations. We suspect the fluctuation of the performance was caused by the distributed effect of caches. For writing, Horus write's performance was on par with non-Horus raw AES write operations in some cases, this is because the Horus overhead was negligible comparing to the slow write operation.

5.4 BTIO benchmark

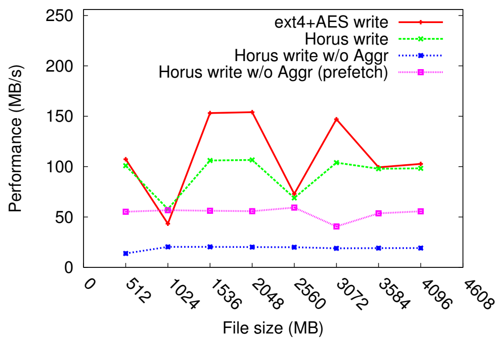
Figure 8 shows the BTIO performance benchmark [37] (from NASA Parallel Benchmarks 3.3.1 using MPICH2-1.4.1p1) that solves a block tridiagonal multi-partitioning problem. We used the Simple I/O mode and the PVFS2 version 2.8.2 file system on 16 nodes for both I/O and compute nodes. We used AES XTS mode for cryptography. AES requires 16-byte aligned blocks, but BTIO writes in some modes don't meet this requirement. Fixing this problem would require changing the BTIO code, but that would make our results incomparable to other published BTIO results. Therefore we didn't change any BTIO code and simply ignored AES errors for those modes.

Figure 8(a) and 8(b) shows the performance benchmark of BTIO Class B (Total written data: 1.7 GB) and C (6.8 GB), respectively. "PVFS2" is the BTIO runs on PVFS2 with no encryption, "AES" is with AES-XTS enabled, "Horus w/o Aggr" is the naïve version of encryption that requests all leaf keys from KDS, and "Horus" is the efficient version that requests only one aggregated range key and calculates the leaf key to be used by AES-XTS. Both "Horus w/o Aggr" and "Horus" include AES-XTS encryption/decryption.

The naïve "Horus w/o Aggr" version shows significant performance degradation. It indicates that 1) handling of many block keys is a significant burden, and 2) key distribution transactions required to request a number of block keys from KDS via network deteriorate the performance. Other simple methods that aim at strong security by managing different keys for blocks and em-

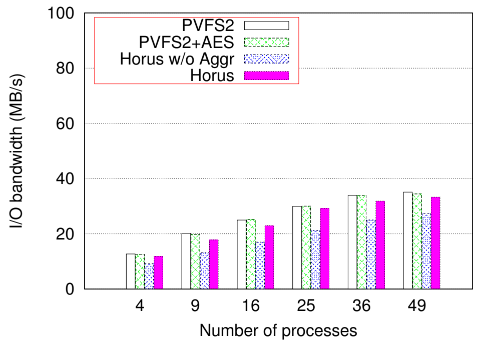


(a) Read performance

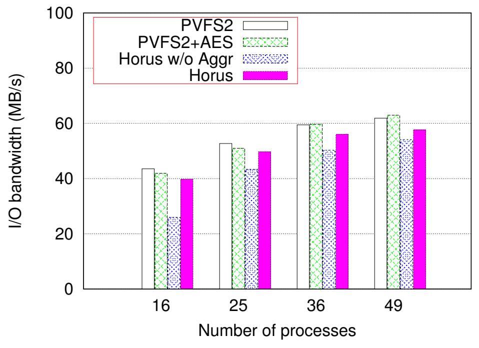


(b) Write performance

Figure 7: Performance of sequential read and write operations. A single client is requesting to a single KDS. Those "w/o Aggr" modes are included for comparison, showing the effect of disabling requesting aggregated range key but having to request every leaf block key (the naïve mode).



(a) Class B



(b) Class C

Figure 8: Performance results for NPB BTIO benchmark.

ploy a key management server have a tendency to show similar problems. In this naïve mode, the overhead can be as high as 16 MB/s (38%) in 16 processes in Class C.

In contrast, Horus mode did not show large performance degradation. The overhead was 2.14 MB/s (5.3%) in 16 processes and 5.36 MB/s (8.5%) in 49 processes in Class C.

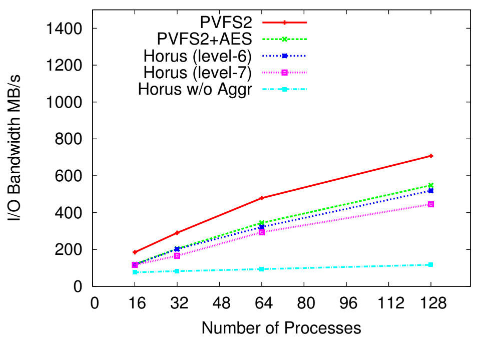
5.5 IOR benchmark

We tested the I/O performance of Horus using the IOR [20], benchmark, a well-known I/O benchmark for HPC environments. We use IOR to generate sequential parallel I/O using MPI-IO. Table 2 shows parameters used for IOR benchmarking. On our cluster, we observed that PVFS2 had better write performance than read because of large RAM caches. AES encryption overhead varied from 20% to 50%.

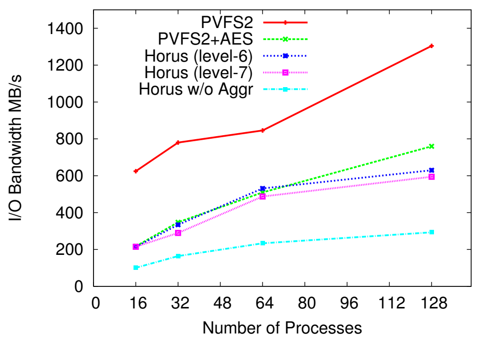
For reads, Figure 9(a) shows that Horus has about 20% overhead in comparison to PVFS2+AES, when level 7 range (2 MB) keys are requested from KDS. Notice that this overhead is negligible when level 6 range (16 MB) keys are requested. Except for 128 processes, when the test clients were saturated, similar results can be seen in Figure 9(b) for write performance. For both reads and writes, we observed that picking the right request level is important and that the naïve approach of requesting only leaf keys from KDS drastically reduces performance. Overall, IOR benchmark results showed that Horus is highly scalable and had very little overhead even for fine-grained range keys.

Table 2: Parameters used for IOR benchmark

Transfer Size	4 MB
Data Read/Write per Process	64 MB
Number of Machines	16
Number of Processes	16, 32, 64, 128
Range Key Level(Range)	6 (16 MB), 7 (2 MB), leaf (4 KB)
Encryption Block Size	4 KB



(a) IOR read performance



(b) IOR write performance

Figure 9: Performance results for IOR benchmark. AES encryption's overhead varied from 20% to 50%. Horus, when properly configured, accounts for overhead from nearly zero to about 20% (when the test clients were saturated).

6 Conclusions and Future Work

Large-scale distributed storage systems need better security for HPC, cloud computing, and SCDN systems that need to store and process sensitive data. Our design, Horus, provides such fine-grained security, allowing each client to only access the range of data assigned to it and leaving it unable to read any other data even if the client or the storage system are compromised. By keeping the Horus KDC as an independent component outside the HPC system or the cloud, the need to trust the system operators or the cloud service provider is reduced, and the damage caused by malicious insiders is more easily contained. This design also simplifies the deployment of Horus because the changes to the existing system and applications is minimal. These features are especially useful for shared HPC systems, HPC applications running in the cloud, and SCDN systems deployed in the cloud.

The biggest limitation of Horus is perhaps the reliance on authentication to find the accessible ranges for each client. Also, HPC nodes are often homogeneous, perhaps allowing an intruder to gain access to a large number of nodes at the same time by using the same vulnerability, thus, gathering a lot of range keys. Horus uses encryption and unavoidably would cause performance degradation for HPC systems, but with the introduction of new hardware AES support [29], this problem is becoming less significant every day.

For the future, we believe that Horus can be leveraged to protect checkpoint data without excessively degrading the performance of scientific computation. The local storage that will be provided in compute nodes to accelerate checkpointing performance must also be protected to prevent other applications and node from accessing the checkpoint data. Since it is necessary to provide millions of different keys for different nodes and files, Horus's scalable key management and delegation capability are beneficial.

Horus can also be leveraged to protect data stored in HDFS [32] to enforce data security in Apache Hadoop based cloud computing environments. Data can be encrypted by using Horus before being loaded into HDFS. While starting a job, the Hadoop Job Tracker needs to submit a list of worker nodes and respective ranges of data each node will process to the KDC. Worker nodes can request keys from KDC to access the required data. In this manner access from every node is restricted to the portion of data that it needs to process. The Job Tracker has to be trusted to some degree, but the accessing policy can be enforced by KDC to limit the range/volume of data, improving security.

Our prototype implementation and performance evaluation have demonstrated that Horus is a versatile and light-weight encryption solution for very large-scale storage. The performance penalty of key distribution and KHT calculation is very low, compared with other competing encryption solutions. The performance overhead of Horus is small because KHTs provide an efficient way of managing and distributing a large number of encryption keys. Key distribution and computation is also highly parallelizable, making it ideal for use in large and high performance systems. By facilitating encryption in large-scale HPC storage, Horus can greatly improve storage security while imposing little penalty on file system access.

Acknowledgments

This research was supported in part by the National Science Foundation under awards IIP-0934401 and CCF-0937938, and the Department of Energy under Award number DE-FC02-10ER26017/DESC0005417. We would like to thank the industrial sponsors of the Storage Systems Research Center (SSRC), including

EMC, Hewlett Packard Laboratories, Hitachi, Huawei, IBM Research, LSI, NetApp, Northrop Grumman, Permabit, and Samsung, for their generous support. We also thank the faculty and students of SSRC for their help and guidance, especially Arifa Nisar, Aleatha Parker-Wood, Christina Strong, Ian Adams, and Stephanie Jones for their valuable comments.

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] Amazon Web Service. AWS: using server-side data encryption. `http://docs.amazonwebservices.com/AmazonS3/latest/dev/UsingServerSideEncryption.html`.

[3] Amazon Web Service. AWS client-side data encryption. `http://aws.amazon.com/articles/2850096021478074`, Apr. 2011.

[4] 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.

[5] 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.

[6] BBC News. US urges action to prevent insider leaks, Jan. 2011.

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

[8] G. Bronevetsky and A. Moody. Scalable I/O systems via node-local storage: Approaching 1 TB/sec file I/O. Technical Report LLNL-TR-415791, Lawrence Livermore National Laboratory, 2009.

[9] P. H. Carns, W. B. Ligon, R. B. Ross, and R. Thakur. PVFS: a parallel file system for Linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, Atlanta, GA, Oct. 2000.

[10] A. M. Caulfield, L. M. Grupp, and S. Swanson. Gordon: Using flash memory to build fast, power-efficient clusters for data-intensive applications. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS09)*, Mar. 2009.

[11] 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.

[12] Department of Energy. NETL shares computing speed, efficiency to tackle barriers. *Fossil Energy Today*, 1(6):1–3, June 2012.

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

[14] X. Dong, Y. Xie, N. Muralimanohar, and N. P. Jouppi. Hybrid checkpointing using emerging nonvolatile memories for future exascale systems. *ACM Transactions on Architecture and Code Optimization*, 8(2):6:1–6:29, June 2011.

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

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

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

[18] 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.

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

[20] Lawrence Livermore National Laboratory. IOR software. `http://www.llnl.gov/icc/lc/siop/downloads/download.html`, 2003.

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

[22] 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.

[23] Los Angeles Times. Google fires employee for snooping on users, Sept. 2010.

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

[25] 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.

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

[27] 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.

[28] R. Rajendran, E. L. Miller, and D. D. E. Long. Horus: Fine-Grained Encryption-Based Security for High Performance Petascale Storage. In *Proceedings of the 6th Parallel Data Storage Workshop (PDSW '11)*, Nov. 2011.

[29] J. Rott. Intel Advanced Encryption Standard Instructions (AES-NI), Feb. 2012.

[30] 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.

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

[32] 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.

[33] M. W. Storer, K. M. Greenan, E. L. Miller, and K. Voruganti. Pergamum: Replacing tape with energy efficient, reliable, disk-based archival storage. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST)*, Feb. 2008.

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

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

[36] 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.

[37] P. Wong and R. F. V. der Wijngaart. NAS Parallel Benchmarks I/O Version 2.4. Technical Report NAS-03-002, NASA Advanced Supercomputing (NAS) Division, 2003.

[38] YADIFA. YADIFA DNS Benchmark. `http://www.yadifa.eu/benchmark`.