Appears in the proceedings of the 12th ACM International Systems and Storage Conference (ACM SYSTOR '19)

Kurma: Secure Geo-Distributed Multi-Cloud Storage Gateways

Ming Chen and Erez Zadok
Stony Brook University
{mchen,ezk}@fsl.cs.sunysb.edu

ABSTRACT

Cloud storage is highly available, scalable, and cost-efficient. Yet, many cannot store data in cloud due to security concerns and legacy infrastructure such as network-attached storage (NAS). We describe *Kurma*, a cloud storage gateway system that allows NAS-based programs to seamlessly and securely access cloud storage. To share files among distant clients, Kurma maintains a unified file-system namespace by replicating metadata across geo-distributed gateways. Kurma stores only encrypted data blocks in clouds, keeps file-system and security metadata on-premises, and can verify data integrity and freshness without any trusted third party. Kurma uses multiple clouds to prevent cloud outage and vendor lock-in. Kurma's performance is 52–91% that of a local NFS server while providing geo-replication, confidentiality, integrity, and high availability.

CCS CONCEPTS

• **Security and privacy → Management and querying of encrypted data**; • **Computer systems organization → Cloud computing**;

KEYWORDS

Multi-cloud, cloud storage gateways, storage security

ACM Reference format:
Ming Chen and Erez Zadok. 2019. Kurma: Secure Geo-Distributed Multi-Cloud Storage Gateways. In *Proceedings of The 12th ACM International Systems and Storage Conference, Haifa, Israel, June 3–5, 2019 (SYSTOR'19),* 12 pages.
https://doi.org/10.1145/3319647.3325830

SYSTOR'19, June 3–5, 2019, Haifa, Israel

ACM ISBN 978-1-4503-6749-3/19/06…$15.00
https://doi.org/10.1145/3319647.3325830

1 INTRODUCTION

Cloud storage has obvious security challenges since tenants do not control the physical media of their data. Cloud users also suffer from long latency when data need to be frequently transferred between branch offices and remote clouds. On-premises cloud storage gateways alleviate these problems by keeping sensitive data in private storage media, safeguarding against attacks in public clouds, and caching hot data locally.

Kurma is a cloud-based file system designed for organizations that have several to dozens of physical branch offices and want to share data securely among the offices. Kurma uses geo-distributed cloud-storage gateways (one per office) to collectively provide a unified file-system namespace for all offices. Each Kurma gateway is physically a coordinated cluster of on-premises machines, and provides NFS services to local clients. Kurma gateways use multiple public clouds as back-ends and use on-premises storage to cache hot data.

Kurma considers the on-premises gateways trusted, and the public clouds untrusted. Kurma stores only encrypted and authenticated file data blocks on clouds; it keeps all sensitive metadata in trusted gateways, including file-system metadata, encryption keys, and integrity metadata of data blocks. Many cloud object stores are eventually consistent, meaning they may return stale data [4, 6, 59] as with replay attacks. Kurma efficiently detects stale data using timestamps and versions. Kurma has a simple and secure key management scheme that does not need any trusted third parties.

Kurma stores data in multiple clouds to tolerate cloud failures. Kurma increases data availability across clouds using replication [61], erasure coding [43, 44], or secret sharing [35, 46]. With secret sharing, Kurma provides an additional level of security such that one compromised cloud cannot recover any part of the data. Kurma supports AWS S3, Azure Blob Store, Google Cloud Storage, and Rackspace Cloud Files.

Each Kurma gateway maintains a copy of the whole file-system metadata, so that it can still serve local clients after a network partition. Kurma minimizes its metadata size by using large data blocks and compression. Metadata changes made by a Kurma gateway are asynchronously replicated to all other gateways using Hedwig [54], a publish-subscribe system that provides *guaranteed-delivery* of large amounts of data across the Internet. Kurma provides NFS close-to-open consistency among clients connected to a common Kurma

gateway. For clients across geo-distributed gateways, Kurma provides FIFO consistency [36], trading off consistency for higher performance and availability. Thus, operations in different gateways may be conflicting. Kurma detects conflicts and provides resolution for common types of conflicts.

Kurma uniquely combines many advanced and industry-proven techniques from prior studies [8, 16, 22, 25, 35, 45, 54] to achieve high security and availability in a simple way.

The rest of this paper is organized as follows. §2 discusses the design. §3 describes the implementation of our Kurma prototype. §4 evaluates its performance. §5 discusses related work. §6 concludes and discusses future work.

2 DESIGN

We present Kurma's threat model, design goals, architecture, metadata management, security, multi-cloud redundancy, consistency, and persistent caching.

2.1 Threat Model

Our threat model reflects the settings of an organization with offices in multiple regions, and employees in these offices store and share files via Kurma gateways.

Public clouds are not trusted. Data stored in cloud may be leaked or tampered by malicious tenants and compromised providers. Transferring data to clouds is vulnerable to man-in-the-middle attacks. Eventually-consistent clouds may return stale data. Cloud outage happens [3, 57].

Clients are trusted. Clients represent internal employees and are generally trustworthy with proper access control. Kurma supports NFSv4 with advanced ACLs [50].

Kurma gateways are trusted. They provide consolidated security services. Kurma gateways can authenticate each other and establish trusted secret channels. Each gateway is a cluster of computers that fits in one access-controlled room.

2.2 Design Goals

Kurma has four goals in descending order of importance:

(1) **Strong security**: Kurma should ensure confidentiality, integrity, and freshness to both file data and metadata while outsourcing storage to clouds.
(2) **High availability**: Kurma should have no single point of failure, and be available despite network partitions and outage of a small subset of clouds.
(3) **High performance**: Kurma should minimize the performance penalty of its security features, and overcome the high latency of remote cloud storage.
(4) **High flexibility**: Kurma should be configurable in many aspects to support flexible trade-off among security, availability, performance, and cost.

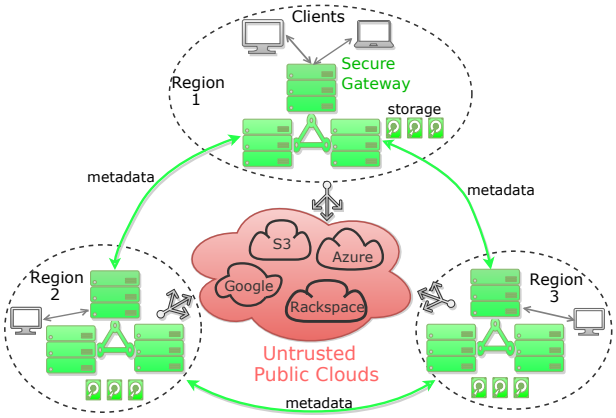


Figure 1: Kurma architecture with three gateways. Each dashed oval represents an office in a region, where there are clients and a Kurma gateway. Each gateway is a cluster of coordinated machines represented by three inter-connected racks. The green arrows connecting gateways are private secret channels for replicating file-system and security metadata. Each gateway has local storage for cache. Clocks of all machines are synchronized using NTP [41].

2.3 Architecture

Figure 1 shows Kurma's architecture. Kurma uses trusted on-premises machines and storage to build gateways that seamlessly protect data in clouds. For strong security, Kurma uses public clouds to store only encrypted file blocks, but not any metadata. Instead, Kurma store all metadata, including file block mapping and file keys, in trusted on-premises machines; and uses a secret channel between each pair of gateways for distributing metadata among gateways.

For high availability, Kurma uses multiple clouds as backends. Kurma stores file-system metadata in ZooKeeper [25], which is distributed and highly available. Each Kurma gateway runs a separate ZooKeeper instance that stores a full replica of the whole file-system metadata; thus, outage in one region will not bring down gateways in other regions.

For high performance, each Kurma gateway uses a persistent write-back cache to avoid the long latency of cloud accesses. Kurma replicates metadata asynchronously among gateways detecting and resolving conflicts as needed.

To be flexible, Kurma supports three redundancy mechanisms when storing data in multiple clouds: replication, erasure coding, and secret sharing. They enable a wide range of trade-offs among availability, performance, and costs.

Figure 2 shows the Kurma servers and their components. Each gateway has three types of servers. NFS Servers export files to clients via NFS; each NFS Server has a Cache Module

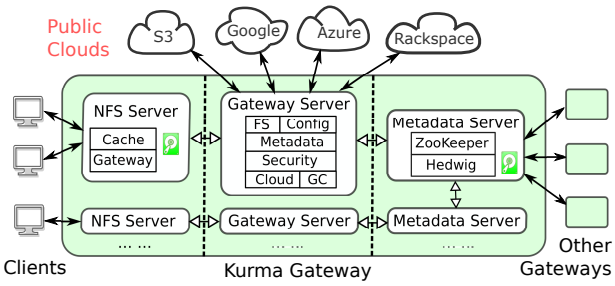


Figure 2: Kurma gateway components. A gateway consists of three types of servers as separated by dashed lines: NFS, Gateway, and Metadata Servers. Each NFS Server has a persistent Cache Module and a Gateway Module. Each Gateway Server has six modules: file system (FS), configuration (Config), metadata, security, cloud, and garbage collection (GC). Each Metadata Server has a ZooKeeper Module and a Hedwig Module. NFS Servers and Metadata Servers have local storage for data cache and metadata backups, respectively.

```
typedef  i16  GatewayID
struct ObjectID {
  i128          id;
  GatewayID     creator;
  byte          type;
}
struct Attributes {
  i64           size;
  i32           flags;
      ... ...
  i64       remote_ctime;
}
```

```
struct File {
  ObjectID              id;
  ObjectID              parent;
  Attributes            attrs;
  i32                   block_shift;
  list<i64>             block_versions;
  list<GatewayID>       block_creators;
  string                redundancy;
  list<string>          cloud_ids;
  map<GatewayID, binary> keymap;
}
```

Figure 3: Simplified Kurma data structures in Thrift. `i16` is a 16-bit integer. Thrift does not have a native `i128`; we emulated it using two `i64`s. `list` and `map` are builtin linear and associative containers, respectively. We omit common attributes such as `mode`, `uid`, and other data structures for directories and volumes.

that has a persistent cache (see §2.8). For requests to metadata and uncached data, each NFS Server uses its Gateway Module to talk to Gateway Servers' FS Module using RPCs.

Gateway Servers break files into blocks, and use its Cloud Module to store encrypted blocks as key-value objects in clouds after replication, erasure coding, or secret sharing (according to configuration). The Config Module parses configuration parameters; the Security Module performs authenticated encryption of each block; the Garbage Collection (GC) Module deletes stale data blocks from clouds and stale metadata from metadata servers. The FS Module manages file-system namespace and handles conflicts (see §2.7).

Metadata Servers run ZooKeeper to store the file-system metadata (e.g., file attributes, block versions). They also run Hedwig [54] to receive messages of metadata updates from other gateways. For each message, Hedwig notifies a responsible Gateway Server (specifically its Metadata Module) to apply the metadata update in the local gateway.

Kurma supports volumes, each of which is a separate file-system tree that can be exported via NFS to clients. Kurma assigns an NFS Server and a Gateway Server to each volume; uses ZooKeeper to coordinate the assignment. A client can mount to any NFS Server, which will either process the client's NFS requests, or redirect the requests to the responsible NFS Server using NFSv4 referrals [50].

2.4 Metadata Management

Each Kurma gateway maintains a replica of the entire file-system metadata in a local ZooKeeper instance, thus metadata operations can be processed without synchronizing with other gateways. This is safer and faster than storing metadata in clouds. Each Kurma gateway asynchronously replicates metadata changes to all other gateways.

Figure 3 shows Kurma's file-system metadata format. Kurma uniquely identifies a gateway using a 16-bit `GatewayID`; and a file-system object using an `ObjectID`. Kurma never reuses `ObjectID`s because its 128-bit `id` allows one billion machines to create one billion files per second for more than 10^{13} years. In `ObjectID`, the `creator` distinguishes objects that are created simultaneously in multiple gateways with the same `id`; the `type` tells if it is a file or a directory.

Besides common attributes (e.g., `size` and `mode`), each Kurma file-system object has a timestamp of the last update by any remote gateway and a set of flags. One such flag is to indicate if a file is only visible in one gateway but not in other gateways; it is used to resolve conflicts (see §2.7).

As shown in Figure 3, a Kurma file includes `ObjectID`s of the file and its parent, attributes, and the block shift (i.e., log_2 of the block size). `block_versions` and `block_creators` record file blocks' version numbers and creator gateways, respectively. `redundancy` encodes the redundancy type and parameters (e.g., "r-4" means replication with four copies). `cloud_ids` encodes the cloud backend; for example, "S3a" represents a bucket named "a" in Amazon's AWS S3. `keymap` stores the file's secret key (see §2.5).

Storing metadata in distributed and durable ZooKeeper [25] makes Kurma resilient to node failures and power outage. Since ZooKeeper is an in-memory store, thus Kurma, like HDFS [9] and GFS [21], also keeps its metadata in-memory so that metadata operations are fast. Kurma minimizes the memory footprint of its metadata using three strategies:

(1) Kurma uses a large block size (defaults to 1MB but configurable) so that files have fewer blocks and thus less metadata. A large block size also helps throughput because cloud accesses are often bottlenecked by network latency instead of bandwidth; and it saves

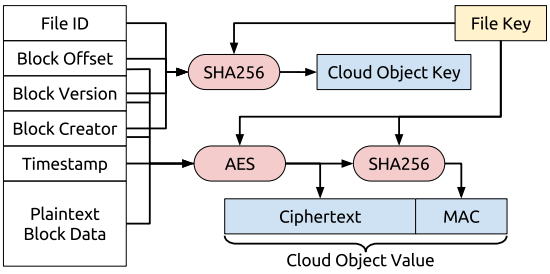


Figure 4: Authenticated encryption of a Kurma file block. AES runs in CTR mode without padding.

costs because some clouds (e.g., AWS and Azure) charge by request counts.

(2) For each block, Kurma stores only a 64-bit version number and a 16-bit `GatewayID` in ZooKeeper; stores other metadata (e.g., the offset and timestamp) in clouds; and generates the cloud object key on the fly as illustrated in Figure 4.

(3) Kurma compresses its metadata. The block version numbers are particularly compressible because most versions are small and neighboring versions are often numerically close thanks to data locality.

2.5 Security

Kurma stores each encrypted block as a key-value object: the key is derived from the block's metadata; the value is a concatenation of ciphertext and message authentication code (MAC) as shown in Figure 4. Kurma protects data integrity by embedding into each cloud object encrypted metadata including the block offset, version, creator, and timestamp. Kurma uses the offset to detect intra- and inter-file swapping attacks that swap data blocks; each file has a unique key.

To check data freshness, Kurma broadcasts block version numbers among all gateways. This is feasible in Kurma because each region has only one gateway and the total number of gateways is small. Kurma incorporates a block's version number into its cloud object key; so updating blocks does not overwrite existing cloud objects but creates new objects instead; Kurma garbage-collects old blocks lazily [11]. When a Kurma gateway reads a block during an inconsistency window of an eventually-consistent cloud, the gateway, instead of reading stale data, may find the new block missing and then fetch it from other clouds. Kurma ensures that each version of a block has a unique cloud object key: when a file is truncated, Kurma does not discard the version numbers of truncated blocks until the file is completely removed; when the truncated blocks are added back later, their version numbers are incremented from the existing values instead of starting at zero. Kurma uses `block_creators` to differentiate the same version of a block created simultaneously by multiple gateways. Block versions also help prevent replay attack: since each block version generates a unique cloud object key (see Figure 4), attackers could not tell whether two objects are two versions of the same block.

Kurma guarantees that a client always reads fresh data that contains all updates made by clients in the same region. However, Kurma cannot guarantee that a client always sees all updates made by clients in other regions. This is because network partitions may disrupt the replication of version numbers among gateways. This is a trade-off Kurma makes between availability and partition tolerance [10].

Each Kurma gateway has a master key pair, which is used for asymmetric encryption (RSA) and consists of a public key (PuK) and a private key (PrK). The public keys are exchanged manually among geo-distributed gateways by security personnel. This is feasible because one geographic office has only one Kurma gateway, and key exchange is only needed when opening a new office in a new location. This key distribution scheme does not need any trusted third parties. Knowing all the public keys makes it easy for gateways to authenticate each other, and allows Kurma to securely replicate metadata among gateways. When creating a file, the creator gateway randomly generates a 128-bit symmetric encryption file key (FK) and uses it to encrypt the file's data blocks. Then, for each Kurma gateway with which the creator is sharing the file (i.e., an *accessor*), the creator encrypts the FK using the accessor's public key (PuK) using RSA, and then generates a ⟨GatewayID, EFK⟩ pair where EFK is the encrypted FK. All the ⟨GatewayID, EFK⟩ pairs are stored in the file's metadata (i.e., `keymap` in Figure 3). When opening a file, an accessor gateway first finds its ⟨GatewayID, EFK⟩ pair in the `keymap`, and then recovers FK by decrypting the EFK using its private key (PrK). During encryption, Kurma uses the concatenation of the block offset and version as the initialization vector (IV); this avoids the security flaws of reusing IVs [18].

2.6 Multiple Clouds

By storing data redundantly on multiple clouds, Kurma can achieve high availability and tolerate cloud failures. Kurma supports three redundancy types: (1) replication, (2) erasure coding, and (3) secret sharing. They represent different trade-offs among reliability, security, space and computational overhead. (1) To tolerate the failure of f clouds, we need to replicate data over $f+1$ clouds. A write operation finishes only when we have successfully placed a replica in each of the $f+1$ clouds. A read operation, however, finishes as soon as one valid replica is read. The storage overhead and write amplification are both $(f+1)\times$. The read amplification is zero in the best case (no failure), but $(f+1)\times$ in the worst case (f failures). (2) Using erasure coding to tolerate the failure of f clouds while at least k clouds are available, Kurma breaks

each block into k parts, transforms them into $k+f$ parts using an erasure code, and then stores each part in one cloud. In the best case, a read operation has to read from k clouds but each read size is $\frac{1}{k}$ of the original block size. In the worst case, it has to read from $k+f$ clouds. The storage overhead and write amplification are both $\frac{f+k}{k}\times$. Unlike replication, erasure coding requires extra computation. Specifically, Kurma uses Reed-Solomon [44] from the Jerasure library [43]. (3) A secret sharing algorithm has three parameters: (n,k,r), where $n > k > r$. It transforms a *secret* of k symbols into n *shares* (where $n > k$) such that the secret can be recovered from any k shares but it cannot be recovered even partially from any r shares. A secret sharing algorithm simultaneously provides fault tolerance and confidentiality. It is more secure than erasure coding by preventing r (or fewer) conspiring clouds from getting any secret from the cloud objects. The storage overhead and read/write amplification of secret sharing are the same as erasure coding. Kurma's secret sharing algorithms include AONT-RS [46] and CAONT-RS [35].

Kurma always encrypts data blocks before adding redundancy. A file's redundancy type and parameters are determined by configuration upon creation; and then stored in its metadata (i.e., `redundancy` and `cloud_ids` in Figure 3).

2.7 File Sharing Across Gateways

To share files across geo-distributed gateways, Kurma maintains a unified file-system namespace by replicating and replaying metadata changes across gateways. Since file sharing is "rarely concurrent" [31], Kurma replicates metadata changes asynchronously, without waiting for replies. This asynchrony makes metadata operations fast; it also keeps a gateway available to local clients when the network connecting gateways is partitioned. In other words, Kurma trades off consistency for performance, availability, and partition tolerance [10]. This degree of trade-off is acceptable because the relaxed consistency is still the same as provided by traditional NFS servers. That is, NFS clients in a local region follow the close-to-open consistency model [32]: when a client opens a file, the client sees all changes made by other clients in the same region who closed the file before the open. The client, however, may not see changes from remote gateways until the changes propagate to the local region.

Kurma uses Hedwig [54] to replicate metadata in an "all-to-all broadcast" manner [4]. For instance, after a gateway performs a write operation, it broadcasts the incremented version numbers to all other gateways. This broadcasting is feasible because Kurma was designed for a small number of gateways. Hedwig is a pub-sub system optimized for communication across data-centers, and it protects its communication using SSL. Using pub-sub services to replicate data is common in geo-distributed systems; Facebook uses a pub-sub system called Wormhole [48] to replicate data.

Across gateways, Kurma provides *FIFO consistency* [36]: it preserves the order of operations in a single gateway, but not across gateways. This is because Hedwig delivers messages from a particular region in the same order to all subscribers globally, but it may interleave messages from different regions [56]. Kurma synchronizes dependent operations when replaying messages. For example, Kurma ensures that a preceding directory-creation operation is finished before replaying an operation that creates a file in that directory.

Conflicts may happen during the asynchronous replication; Kurma adds extra information inside Hedwig messages to detect and resolve conflicts. For example, each file-creation message contains not only the parent directory and the file name, but also the `ObjectID` of the locally created file. If a remote gateway happens to create a file with the same name simultaneously, Kurma can differentiate the two files using their `ObjectID`s.

Resolving conflicts can be complex and even requires human intervention [28, 47]. Fortunately, the majority of conflicts can be resolved automatically [45]. Kurma contains default resolvers for three common types of conflicts: (1) content conflicts when two or more gateways write to the same file block simultaneously, (2) name conflicts when two or more gateways create objects with the same name in one directory, and (3) existence conflicts when one gateway deletes or moves a file-system object (e.g., delete a directory) while another gateway's operations depend on the object (e.g., create a file in that directory). Our separate technical report [11] provides further details on Kurma's conflict resolution.

2.8 Persistent Caching

Each Kurma NFS Server has a persistent cache so that hot data can be read in the low-latency on-premises network instead of from remote clouds. The cache stores plaintext instead of ciphertext so that reading from the cache does not need decryption; this is safe because on-premises machines are trusted in our threat model. The cache is a write-back cache that can hide the high latency of writing to clouds. The cache is stored on persistent storage because stable NFS WRITEs require dirty data to be flushed to persistent storage [49] before replying. The cache also maintains additional metadata in stable storage so that dirty data can be recovered after crashes; the metadata includes a list of dirty files and the dirty extents of each file. For each cached file, the cache maintains a local sparse file of the same size. Insertions (evictions) of file blocks are performed by writing (punching holes) to the sparse files at the corresponding locations. Dirty cache items are not replicated intra- or inter-gateway.

To maintain NFS's close-to-open cache semantics, Kurma revalidates a file's persistent cache content when processing an NFS open request on the file. As discussed in §2.4, Kurma stores a file attribute that is the timestamp of the last change

made by any other remote gateway (i.e., `remote_ctime`). The cache compares its locally-saved `remote_ctime` with the latest `remote_ctime`: if they match, it means that no other gateway has changed the file, and the content is still valid; otherwise, the content is invalidated.

To allow flexible trade-off between consistency and latency, Kurma's cache uses a parameter called *write-back wait time* (*WBWT*) to control whether the write-back should be performed synchronously or asynchronously upon file close. When *WBWT* is set to zero, write-back is performed right away and the close request is blocked until the write-back finishes. When *WBWT* is greater than zero, Kurma first replies to the close request, and then waits *WBWT* seconds before starting to write-back dirty cache data to the clouds.

3 IMPLEMENTATION

We have implemented a Kurma prototype that includes all features described in the design, except for the partition of volumes among multiple NFS servers. We have tested our prototype thoroughly using unit tests and ensured that it passed all applicable `xfstests` [63] cases. We implemented the Kurma NFS server in 15,800 lines of C/C++ code, and the Gateway server in 22,700 lines of Java code (excluding code auto-generated by Thrift). We have open-sourced all of our code at https://github.com/sbu-fsl/kurma.

We implemented Kurma's NFS Servers (see Figure 2) on top of NFS-Ganesha [14, 42], a user-space NFS server. NFS-Ganesha can export files from many backends to NFS clients through its *File System Abstraction Layer* (FSAL). FSAL is similar to Linux's Virtual File System (VFS) and is also stackable [23, 64]. We implemented the Gateway Module as an `FSAL_KURMA` layer, and the Cache Module as an `FSAL_PCACHE` layer that is stacked on top of `FSAL_KURMA`. `FSAL_PCACHE` always tries to serve NFS requests from the local cache; it only redirects I/Os to the underlying `FSAL_KURMA` in case of cache miss or write back. `FSAL_PCACHE` groups adjacent small I/Os to form large I/Os so that slow cloud accesses are amortized. `FSAL_PCACHE` uses the LRU algorithm to evict blocks and ensures that evicted dirty blocks were written back first. `FSAL_KURMA` requests file-system operations to Gateway Servers using RPCs implemented in Thrift [55].

We implemented the Gateway Servers in Java. The File-System Module is a Thrift RPC server that communicates with Kurma's NFS Servers; it is implemented using Thrift's Java RPC library. The Metadata Module uses the ZooKeeper client API to store metadata; it also uses Curator [53], a ZooKeeper utility library. Before being stored into ZooKeeper, metadata is compressed using Thrift's compressing data serialization library (`TCompactProtocol`). The Metadata Module uses the Hedwig client API to subscribe to remote metadata changes and to publish local changes. The secret channels connecting Kurma gateways are SSL socket connections. The Security Module uses Java 8's standard cryptographic library. The Cloud Module includes cloud drivers for Amazon S3, Azure Blob Store, Google Cloud Storage, and Rackspace Cloud Files; it also includes a redundancy layer for replication, erasure coding, and secret sharing. We adapted the cloud drivers code from Hybris [15, 16]. Our erasure coding uses the Jerasure library [43] and its JNI wrapper [58]. Our secret sharing library is based on C++ code from CD-Store [34]; we implemented JNI wrapper for it.

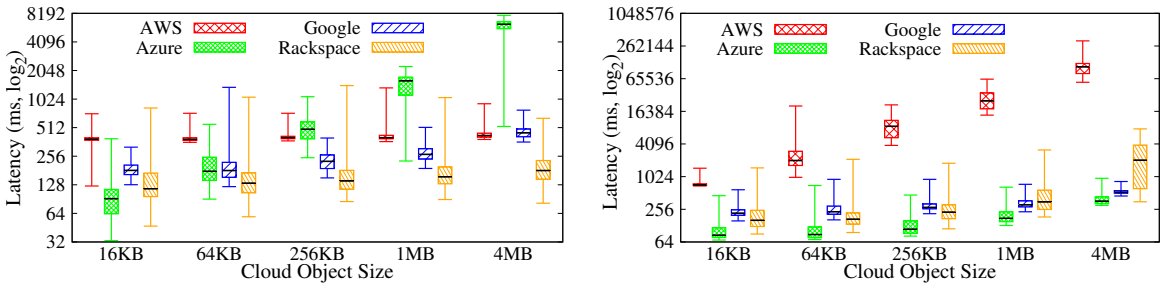
Our implementation includes five optimizations: (1) Generating a file's `keymap` needs to encrypt the file's key using slow RSA for each gateway (see §2.5). To hide the high latency of RSA encryptions, Kurma uses a separate thread to pre-compute a pool of `keymaps`, so that Kurma can quickly take one `keymap` out of the pool when creating a file. (2) To reduce the metadata size written to ZooKeeper, Kurma stores a file's `keymap` in a child `znode` (a ZooKeeper data node) under the file's `znode`. For large files, Kurma also splits their block versions and creators into multiple child `znodes` so that a write only updates one or two small `znodes` of block versions. (3) Kurma metadata operations are expensive because one ZooKeeper update requires many network hops among the distributed ZooKeeper nodes. Furthermore, a file-system operation may incur multiple ZooKeeper changes. For example, creating a file requires one creation of the file's `znode`, one creation of its `keymap` znode, and one update of its parent directory's `znode`. To amortize ZooKeeper's high latency, we batch multiple ZooKeeper changes into a single ZooKeeper transaction [25]. This almost doubled the speed of metadata operations. (4) Latencies of clouds vary significantly over time. To achieve the best performance, Kurma sorts cloud providers by their latencies every *N* seconds (*N* is a configurable parameter) and uses the *K* fastest clouds as backends where *K* depends on operations (e.g., 1 for reads). (5) To reduce the frequency of accessing the Metadata Servers, Kurma's Gateway Servers cache clean metadata in memory using Guava's `LoadingCache` [26]. The cached metadata includes attributes of hot file-system objects, block versions of opened files, and hot directory entries.

4 EVALUATION

We evaluated Kurma's security and performance. We also compared Kurma to a traditional NFS server.

4.1 Testbed Setup

Our testbed consists of two identical Dell PowerEdge R710 machines, each with a six-core Intel Xeon X5650 CPU, 64GB of RAM, and an Intel 10GbE NIC. Each machine runs Linux KVM [29] to host a set of identical VMs that represent a cluster of Kurma servers in one gateway. Each VM has two CPU cores and 4GB of RAM. Each VM runs Fedora 25 with



(a) Read (b) Write

Figure 5: Latency of reading and writing cloud objects. The five ticks of each boxplot represent (from bottom to top): the minimum, 25^{th} percentile, median, 75^{th} percentile, and the maximum. Note: both axes are in log_2 scales.

a Linux 4.8.10 kernel. To emulate WAN connections among gateways, we injected a network latency of 100ms using `netem` between the two sets of VMs; we chose 100ms because it is the average latency we measured between the US east and west coasts. We measured a network latency of 0.6ms between each pair of servers in the same gateway.

For each gateway, we set three VMs as three Metadata Servers (see Figure 2) running ZooKeeper 3.4.9 and Hedwig 4.3.0. Each gateway also has a Kurma NFS Server and a Gateway Server; the two servers communicate using Thrift RPC 0.12.3. The Kurma NFS Server runs NFS-Ganesha 2.3 with our `FSAL_PCACHE` and `FSAL_KURMA` modules; `FSAL_PCACHE` uses an Intel DC S3700 200GB SSD for its persistent cache. The Gateway Server runs on Java 8. Each gateway has another VM running as an NFSv4.1 client. For comparison, we set up a traditional NFS server on a VM. The traditional NFS server runs NFS-Ganesha with its vanilla `FSAL_VFS` module. `FSAL_VFS` exports to the client an Ext4 file system, stored on a directly-attached Intel DC S3700 200GB SSD. The traditional NFS server does not communicate with other VMs other than the client.

4.2 Security Tests

We tested and verified that Kurma can reliably detect security errors and return valid data available in other healthy clouds. To test availability, we manually deleted blocks of a file from one of the clouds, and then tried to read the file from an NFS client. We observed that Kurma first failed to read data from the tampered cloud, but then retried the read from other clouds, and finally returned the correct file contents.

For integrity tests, we injected four types of integrity errors by (1) changing one byte of a cloud object, (2) swapping two blocks of the same version at different offsets of a file, (3) swapping two blocks of the same version and offset of two files, and (4) replaying a newer version of a block with an old version. Kurma detected all four types of errors during authentication. It logged information in a local file on the secure gateway for forensic analysis; this information included the block offset and version, the cloud object key, the erroneous cloud, and a timestamp. Kurma also successfully returned the correct content by fetching valid blocks from other untampered clouds. We also tested that Kurma could detect and resolve the three types of conflicting changes made in multiple gateways (see §2.7).

4.3 Cloud Latency Tests

Kurma's cloud backends include AWS, Azure, Google, and Rackspace. Figure 5 (log_2 scale on both axes) shows the latency of these public clouds when reading and writing objects with different sizes. Kurma favors a large block size because larger blocks cost less (both AWS and Azure charge requests by count instead of size) and reduce the metadata size. Larger block sizes not only reduce cloud costs, but they also improve overall read throughputs. When the block size increased by 256× from 16KB to 4MB, the read latency of a block increased by only 1.1×, 3.1×, 1.2× for AWS, Google, and Rackspace, respectively. However, thanks to the larger block sizes, the read throughput increased by a lot: 234×, 83×, and 216× for AWS, Google, and Rackspace, respectively. However, Azure is an exception where reading a 4MB object takes 6.5 seconds and is 43 times slower than reading a 16KB object. Our measurements of Azure are similar to those reported in Hybris [16], where Azure took around 2 seconds to read a 1MB object, and around 20 seconds to read a 10MB object. Large performance variances of cloud storage in Figure 5 were also observed in other studies [16, 62].

A larger block size also improves write throughputs. When the block size increases from 16KB to 4MB, the write throughput increased by 1.9×, 76×, 82×, and 68× for AWS, Azure, Google, and Rackspace, respectively. Writes are significantly slower than reads. As shown in Figure 5(b), writing a 4MB object to AWS takes close to 2 minutes. However, the high write latency of large objects is acceptable because Kurma's

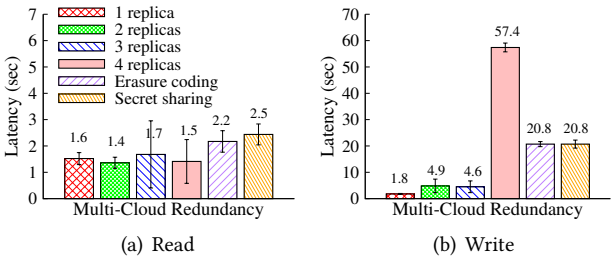


Figure 6: Latency of reading and writing a 16MB file with different redundancy configurations over multiple clouds. The persistent write-back cache is temporarily disabled. The "N replicas" configuration uses the first N clouds out of the list of Google, Rackspace, Azure, and AWS (ordered by their performance with 1MB objects). The erasure coding algorithm is Reed-Solomon with $k = 3$ and $m = 1$. The secret sharing algorithm is CAONS-RS with $n = 4$, $k = 3$, and $r = 2$.

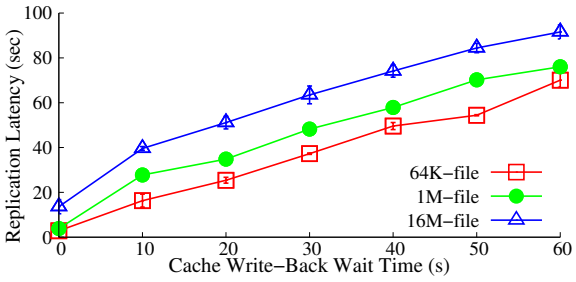


Figure 7: Latency of replicating files across gateways under different write-back wait times ($WBWT$s).

persistent write-back cache can hide the latency from clients. Therefore, Kurma uses a default block size of 1MB.

Figure 5 also shows that the latencies of different clouds can differ by up to 100 times for objects of the same size. The latency variance is high even for the same cloud. Note the large error bars and the logarithmic scale of the Y-axis. Therefore, when reading from only a subset of clouds, ordering the clouds by their speeds, and using the fastest ones can significantly improve performance. Our tests showed that ordering the clouds by their speeds can cut Kurma's average read latency by up to 54%. If not configured differently, our Kurma prototype reorders the clouds based on their speeds every minute, to decide where to send read requests to first.

4.4 Multi-Cloud Tests

For high availability, Kurma stores data redundantly over multiple clouds using replication, erasure coding, and secret sharing. Figure 6 shows the latency of reading and writing a 16MB file with different redundancy configurations. To exercise the clouds, we temporarily disabled Kurma's persistent cache in this test. The N-replica configuration uses the first N clouds out of the list of Google, Rackspace, Azure, and AWS. The list is ordered in decreasing overall performance with 1MB objects (see Figure 5). Figure 6(a) shows that reading a 16MB file takes around 1.6 seconds for all four replication configurations. This is because all N-replica configurations have the same data path for reads: fetching 16 × 1MB blocks from the single fastest cloud. Note that 1.6 seconds is smaller than 16× the read latency of a single 1MB-large object (around 0.28s as shown in Figure 5). This is because Kurma uses multiple threads to read many blocks in parallel. Both the erasure-coding and secret-sharing configurations need to read from three clouds, and thus the reading takes longer with these two configurations: 2.2s and 2.5s on average, respectively.

When writing a 16MB file, the N-replica setup writes a replica of 16 1MB-large blocks to each of N clouds. The write latency of N-replica is determined by the slowest one of the N clouds. AWS is the slowest cloud for writes, so when AWS was added as a 4^{th} replica to the 3-replica configuration, making it a 4-replica configuration, the write latency jumped to 57.4 seconds (Figure 6(b)). Both the erasure-coding and secret-sharing configurations write to four clouds; however, their write latencies are around one third of the latency of 4-replica. This is because both erasure coding and secret sharing split a 1MB block into four parts, each around 340KB large. Kurma also uses multiple threads to write blocks in parallel, so writing a 16MB-large file takes less time than sequentially writing 16 1MB-large objects.

In Figure 6, the 2-replica, erasure-coding, and secret-sharing configurations can all tolerate failure of one cloud. Among them, the 2-replicas configuration has the best performance. However, secret sharing provides extra security—resistance to cloud collusion—and has read performance comparable to the 2-replica configuration. Therefore, we used the secret-sharing configuration in the remaining tests. Note that in general, write latency is less important here because it will be hidden by the persistent write-back cache.

4.5 Cross-Gateway Replication

Kurma shares files across geo-distributed gateways by asynchronously replicating file-system metadata. Figure 7 shows the replication latency of files under different write-back wait times ($WBWT$s, see §2.8). The timer of a file's replication latency starts ticking after the file is created, written and closed in one gateway; the timer keeps ticking and does not stop until the file is found, opened, fully read and closed in another remote gateway. When $WBWT$ is zero, dirty data is synchronously written back to clouds when closing a file. So the replication latency does not include the time of writing to the clouds and thus is small: 2.9s, 3.9s, and 14s for a 64KB, 1MB, and 16MB files, respectively. When $WBWT$ is not zero,

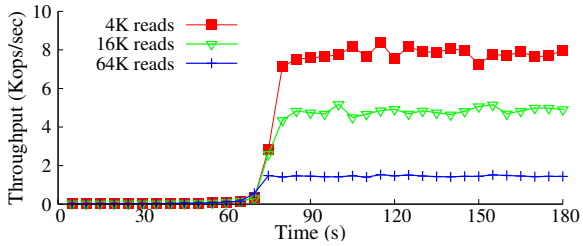


Figure 8: Aggregate throughput of randomly reading a 1GB file using 64 threads. The test starts with a cold cache. The I/O sizes are 4KB, 16KB, and 64KB.

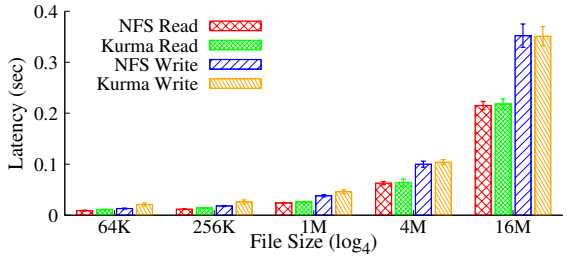


Figure 9: Latency of reading and writing files. The cache is hot during reads, and its $WBWT$ is 30 seconds.

dirty data is written back after closing the file; the replication latency increases linearly with the wait time. In Figure 7, the replication latency for larger files is higher because larger files take more time to write to and read from clouds.

4.6 Data Operations

To test Kurma's performance with large files, we created a 1GB file: this took around 200 seconds writing to clouds. We then performed random reads on the file after emptying Kurma's persistent cache. Figure 8 shows the results. For all three I/O sizes (4KB, 16KB, and 64KB), the initial throughput was merely around 20 ops/sec because all reads needed to fetch data from clouds over the Internet. The throughput slowly increased as more data blocks were read and cached. Once the whole file was cached, the throughput suddenly jumped high because reading from the cache was faster than reading from the clouds by two orders of magnitude. Afterwards, all reads were served from the cache, and the throughput plateaued. It took around 75 seconds to read the whole file regardless of the I/O size; this is because Kurma always uses the block size (1MB) to read from clouds.

To show Kurma's performance when its cache is in effect, we compared a Kurma gateway with a hot cache to a traditional NFS server. Figure 9 shows the latency results of reading and writing whole files. For 64KB files, Kurma's read latency is 22% higher and its write latency is 63% higher. This is because each Kurma metadata operation (e.g., OPEN, CLOSE, and GETATTR) involved multiple servers and took longer to process. In contrast, the traditional NFS server is simpler and each operation was processed by only one server. However, as the file size increased, Kurma's latency became close to that of the traditional NFS. This is because when the file data was cached, Kurma did not need to communicate with the Gateway Server or the Metadata Server; thus its data operations were as fast as NFS, and they amortized the high latency of the few metadata operations.

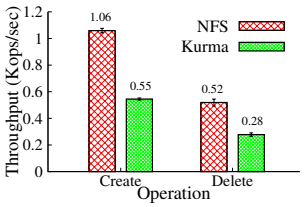


Figure 10: Throughput of creating and deleting empty files.

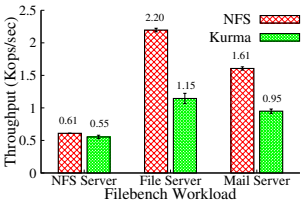


Figure 11: Throughput of Filebench workloads.

4.7 Metadata Operations

To put the performance of Kurma's metadata operations into perspective, we compared Kurma to a traditional NFS using two metadata-only workloads: creating and deleting empty files. Figure 10 shows the results. When processing metadata operations, Kurma needs to communicate with the Kurma NFS Server, the Gateway Server, and the Metadata Servers (see Figure 2). Moreover, a metadata update in ZooKeeper needs to reach a consensus over all ZooKeeper servers; in our setup with three ZooKeeper nodes, it means that at least three additional network hops. Because of these extra network hops, Kurma's throughput is 49% and 46% lower than NFS for file creation and deletion, respectively. Kurma's performance penalty in file creation is higher than that in file deletion. This is because creating a Kurma file requires extra computation to generate and encrypt the per-file secret key (see §2.5).

4.8 Filebench Workloads

Figure 11 shows Kurma's performance under Filebench workloads that are more complex than micro-workloads. The Filebench NFS-Server workload emulates the SPEC SFS benchmark [51]. It contains one thread performing four sets of operations: (1) open, entirely read, and close three files; (2) read a file, create a file, and delete a file; (3) append to an existing file; and (4) read a file's attributes. The File-Server workload emulates 50 users accessing their home directories and spawns one thread per user to perform operations similar to the NFS-Server workload. The Mail Server workload mimics the I/Os of a Unix-style email server operating on a `/var/mail` directory, saving each message as a file; it

has 16 threads, each doing create-append-sync, read-append-sync, read, and delete operations on 10,000 16KB files.

For the Filebench NFS-Server workload, Kurma's throughput is around 9% lower than NFS. That is caused by Kurma's slow metadata operations which require extra network hops to process. For example, deleting a file took only 1ms for the traditional NFS server, but around 2ms for Kurma. The File-Server workload has similar operations to the NFS-Server workload, but contains 50 threads instead of one. Many concurrent metadata updates, such as deleting files in a common directory, need to be serialized using locks. This type of serializations makes Kurma's metadata operations even slower because of longer wait time. For example, deleting a file in the File-Server workload took around 16ms for the traditional NFS server, but as long as 188ms for Kurma. Consequently, Kurma's throughput is around 48% lower than the traditional NFS server. The same is true of the multi-threaded Mail-Server workload, where Kurma throughput is around 41% lower. The high latency of metadata operations is the result of trading off performance for security and availability.

5 RELATED WORK

Kurma is related to prior studies of (1) secure distributed storage systems, and (2) cloud storage gateways.

SFS [39], SiRiUS [22], Plutus [27], and SUNDR [33] are all cryptographic file systems that protect file integrity and confidentiality with minimal trust on storage servers. These systems focused on a single-server architecture instead of a geo-distributed architecture with multiple cloud back-ends. Guarantee freshness of file-system data and metadata is difficult [52]. SiRiUS [22] ensures partial metadata freshness but not data freshness. SUNDR [33], SPORC [19], and Depot [37] all guarantee *fork consistency* and could detect freshness violations with out-of-band inter-client communication. Most of the file systems [17, 20, 52] that guarantee freshness use Merkle trees [40] or its variants [52] to detect replay attacks. SCFS [8] provides freshness without using Merkle trees, but it requires a trusted and centralized metadata service running on a cloud. Kurma does not need Merkle trees or any trusted third parties. For clients across geo-distributed gateways, Kurma guarantees that the time window of stale data is no longer than the duration of the network partition.

Cloud storage gateways give local clients a SAN or NAS interface to cloud storage. SafeStore [30], BlueSky [60], Hybris [16], and Iris [52] are examples of cloud storage gateway systems that provide data protection. SafeStore, BlueSky, and Iris have file system interfaces on the client side; Hybris provides a key-value store. However, all of them were designed for clients in one geographical region with only one gateway. In contrast, Kurma supports sharing files among clients in geo-distributed gateways. Using multiple clouds is an effective way to ensure high availability and business continuity in case of cloud failures [3]. There were several studies of multi-cloud systems [1, 16, 24, 30, 62], and they stored data redundantly on different clouds using either replication or erasure coding. Using multiple clouds can also enhances security. Since collusion across multiple clouds is less likely, dispersing secrets among them is more secure than storing all secrets on a single cloud [2, 7, 8, 35]. However, most of these multi-cloud storage systems [1, 5, 7, 16, 24], provide only key-value stores, whereas Kurma is a geo-distributed file system. Several client-side cloud stores, including CYRUS [13] and SCFS [8], also use multiple clouds as beck-ends. These systems are for personal use when files can fit in a client's local cache—whereas Kurma is designed for enterprise use.

6 CONCLUSIONS

We presented Kurma, a secure geo-distributed multi-cloud storage gateway system. Kurma keeps file-system metadata in trusted gateways, and encrypts data blocks before storing them in clouds. Kurma embeds a version number and a timestamp into each file block to ensure data freshness. Kurma tolerates cloud failures by storing data redundantly among multiple clouds using replication, erasure code, and secret sharing. Kurma provides NFS close-to-open consistency for local clients, and FIFO consistency for clients across gateways. Kurma asynchronously replicates metadata among gateways, detecting and resolving conflicts after they occur. We implemented and evaluated a Kurma prototype. Thanks to Kurma's persistent write-back cache, its performance of data operations is close to a baseline using a single-node NFS server. Kurma's throughput is around 52–91% of the baseline for general purpose Filebench server workloads. This overhead is contributed by slower metadata operations. Kurma sacrifices some performance for significantly improved security, availability, and file sharing across regions.

Limitations and future work. Kurma currently does not consider the insider problem which is addressed by systems like RockFS [38]. Kurma uses multiple clouds for high availability but is not optimized for cost efficiency. We plan to amortize the high latency of Kurma's metadata operations using NFSv4 compound procedures [12, 50].

ACKNOWLEDGMENTS

We thank the anonymous ACM SYSTOR reviewers for their valuable comments. We thank Shivanshu Goswami, Praveen Kumar Morampudi, Harshkumar Patel, Rushabh Shah, and Mukul Sharma for their help in Kurma implementation. This work was made possible in part thanks to Microsoft Azure, Dell-EMC, NetApp, and IBM; NSF awards CNS-1251137, CNS-1302246, CNS-1305360, CNS-1622832, CNS-1650499, and CNS-1730726; and ONR award N00014-16-1-2264.

REFERENCES

[1] Hussam Abu-Libdeh, Lonnie Princehouse, and Hakim Weatherspoon. 2010. RACS: a case for cloud storage diversity. In *Proceedings of the 1st ACM symposium on Cloud Computing*. ACM, 229–240.

[2] Mohammed A. AlZain, Ben Soh, and Eric Pardede. 2013. A Survey on Data Security Issues in Cloud Computing: From Single to Multi-Clouds. *Journal of Software* 8, 5 (2013), 1068–1078.

[3] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. 2010. A View of Cloud Computing. *Commun. ACM* 53, 4 (2010), 50–58.

[4] Peter Bailis and Ali Ghodsi. 2013. Eventual consistency today: Limitations, extensions, and beyond. *Commun. ACM* 56, 5 (2013), 55–63.

[5] David Bermbach, Markus Klems, Stefan Tai, and Michael Menzel. 2011. MetaStorage: A Federated Cloud Storage System to Manage Consistency-Latency Tradeoffs. In *Proceedings of the 2011 IEEE 4th International Conference on Cloud Computing (CLOUD '11)*. IEEE Computer Society, Washington, DC, USA, 452–459. https://doi.org/10.1109/CLOUD.2011.62

[6] David Bermbach and Stefan Tai. 2011. Eventual Consistency: How Soon is Eventual? An Evaluation of Amazon S3's Consistency Behavior. In *Proceedings of the 6th Workshop on Middleware for Service Oriented Computing (MW4SOC '11)*. ACM, New York, NY, USA, Article 1, 6 pages. https://doi.org/10.1145/2093185.2093186

[7] Alysson Bessani, Miguel Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. 2013. DepSky: dependable and secure storage in a cloud-of-clouds. *ACM Transactions on Storage (TOS)* 9, 4 (2013), 12.

[8] A. Bessani, R. Mendes, T. Oliveira, N. Neves, M. Correia, M. Pasin, and P. Verissimo. 2014. SCFS: A Shared Cloud-backed File System. In *USENIX ATC 14*. USENIX, 169–180.

[9] Dhruba Borthakur et al. 2008. HDFS architecture guide. *Hadoop Apache Project* 53 (2008).

[10] Eric A. Brewer. 2000. Towards Robust Distributed Systems (Abstract). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing (PODC '00)*. ACM, New York, NY, USA, 7–. https://doi.org/10.1145/343477.343502

[11] M. Chen. 2017. *Kurma: Efficient and Secure Multi-Cloud Storage Gateways for Network-Attached Storage*. Ph.D. Dissertation. Computer Science Department, Stony Brook University. Technical Report FSL-17-01.

[12] Ming Chen, Dean Hildebrand, Henry Nelson, Jasmit Saluja, Ashok Subramony, and Erez Zadok. 2017. vNFS: Maximizing NFS Performance with Compounds and Vectorized I/O. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST)*. USENIX Association, Santa Clara, CA, 301–314.

[13] Jae Yoon Chung, Carlee Joe-Wong, Sangtae Ha, James Won-Ki Hong, and Mung Chiang. 2015. CYRUS: Towards Client-defined Cloud Storage. In *Proceedings of the Tenth European Conference on Computer Systems (EuroSys '15)*. ACM, New York, NY, USA, Article 17, 16 pages. https://doi.org/10.1145/2741948.2741951

[14] Philippe Deniel, Thomas Leibovici, and Jacques-Charles Lafoucrière. 2007. GANESHA, a multi-usage with large cache NFSv4 server. In *Linux Symposium*. 113.

[15] Dan Dobre, Paolo Viotti, and Marko Vukolić. 2014. Hybris: robust and strongly consistent hybrid cloud storage. (2014). https://github.com/pviotti/hybris.

[16] Dan Dobre, Paolo Viotti, and Marko Vukolić. 2014. Hybris: Robust Hybrid Cloud Storage. In *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 1–14.

[17] J. R. Douceur and J. Howell. 2006. EnsemBlue: Integrating Distributed Storage and Consumer Electronics. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI 2006)*. ACM SIGOPS, Seattle, WA, 321–334.

[18] M. Dworkin. 2007. *Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC*. National Institute of Standards and Technology (NIST).

[19] Ariel J Feldman, William P Zeller, Michael J Freedman, and Edward W Felten. 2010. SPORC: Group Collaboration using Untrusted Cloud Resources. In *OSDI*. 337–350.

[20] K. Fu, M. F. Kaashoek, and D. Mazières. 2000. Fast and Secure Distributed Read-Only File System. In *Proceedings of the 4th Usenix Symposium on Operating System Design and Implementation (OSDI '00)*. USENIX Association, San Diego, CA, 181–196.

[21] S. Ghemawat, H. Gobioff, and S. T. Leung. 2003. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*. ACM SIGOPS, Bolton Landing, NY, 29–43.

[22] E.-J. Goh, H. Shacham, N. Modadugu, and D. Boneh. 2003. SiRiUS: Securing Remote Untrusted Storage. In *Proceedings of the Tenth Network and Distributed System Security (NDSS) Symposium*. Internet Society (ISOC), San Diego, CA, 131–145.

[23] J. S. Heidemann and G. J. Popek. 1994. File System Development with Stackable Layers. *ACM Transactions on Computer Systems* 12, 1 (February 1994), 58–89.

[24] Yuchong Hu, Henry C. H. Chen, Patrick P. C. Lee, and Yang Tang. 2012. NCCloud: Applying Network Coding for the Storage Repair in a Cloud-of-Clouds. In *Proceedings of the Tenth USENIX Conference on File and Storage Technologies (FAST '12)*. USENIX Association, San Jose, CA.

[25] P. Hunt, M. Konar, J. Mahadev, F. Junqueira, and B. Reed. 2010. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference (USENIXATC'10)*.

[26] Google. Inc. 2017. Guava: Google Core Libraries for Java 6+. (2017). https://github.com/google/guava.

[27] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. 2003. Plutus: Scalable Secure File Sharing on Untrusted Storage. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*. USENIX Association, San Francisco, CA, 29–42.

[28] J. J. Kistler and M. Satyanarayanan. 1991. Disconnected Operation in the Coda File System. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*. ACM Press, Asilomar Conference Center, Pacific Grove, CA, 213–225.

[29] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. 2007. kvm: the Linux Virtual Machine Monitor. In *Proceedings of the 2007 Ottawa Linux Symposium (OLS 2007)*, Vol. 1. Ottawa, Canada, 225–230.

[30] Ramakrishna Kotla, Lorenzo Alvisi, and Mike Dahlin. 2007. SafeStore: a durable and practical storage system. In *USENIX Annual Technical Conference*. 129–142.

[31] A. W. Leung, S. Pasupathy, G. Goodson, and E. L. Miller. 2008. Measurement and analysis of large-scale network file system workloads. In *Proceedings of the Annual USENIX Technical Conference*. USENIX Association, Boston, MA, 213–226.

[32] Chuck Lever. 2001. Close-To-Open Cache Consistency in the Linux NFS Client. (2001). http://goo.gl/o9i0MM.

[33] J. Li, M. Krohn, D. Mazières, and D. Shasha. 2004. Secure Untrusted Data Repository (SUNDR). In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. ACM SIGOPS, San Francisco, CA, 121–136.

[34] M. Li, C. Qin, and P. Lee. 2016. Convergent Dispersal Deduplication Datastore. (2016). https://github.com/chintran27/CDStore.

[35] Mingqiang Li, Chuan Qin, and Patrick P. C. Lee. 2015. CDStore: Toward Reliable, Secure, and Cost-efficient Cloud Storage via Convergent Dispersal. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '15)*. USENIX Association, Berkeley, CA, USA, 111–124. http://dl.acm.org/citation.cfm?id=2813767.2813776

[36] R. J. Lipton and J. S. Sandberg. 1988. *PRAM: A scalable shared memory*. Technical Report TR-180-88. Princeton University, Princeton, NY.

[37] Prince Mahajan, Srinath Setty, Sangmin Lee, Allen Clement, Lorenzo Alvisi, Mike Dahlin, and Michael Walfish. 2011. Depot: Cloud Storage with Minimal Trust. *ACM Trans. Comput. Syst.* 29, 4 (December 2011), 12:1–12:38.

[38] David R. Matos, Miguel L. Pardal, Georg Carle, and Miguel Correia. 2018. RockFS: Cloud-backed File System Resilience to Client-Side Attacks. In *Proceedings of the 19th International Middleware Conference (Middleware '18)*. ACM, New York, NY, USA, 107–119. https://doi.org/10.1145/3274808.3274817

[39] D. Mazières, M. Kaminsky, M. F. Kaashoek, and E. Witchel. 1999. Separating key management from file system security. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*. ACM, Charleston, SC, 124–139.

[40] Ralph C. Merkle. 1988. A Digital Signature Based on a Conventional Encryption Function. In *A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology (CRYPTO'87)*. Springer-Verlag, London, UK, 369–378.

[41] D. L. Mills. 1989. *Internet Time Synchronization: the Network Time Protocol*. Technical Report RFC 1129. Network Working Group.

[42] NFS-Ganesha 2016. NFS-Ganesha. (2016). http://nfs-ganesha.github.io/.

[43] James S Plank, Scott Simmerman, and Catherine D Schuman. 2008. Jerasure: A library in C/C++ facilitating erasure coding for storage applications-Version 1.2. (2008). http://git-scm.com.

[44] Irving S Reed and Gustave Solomon. 1960. Polynomial codes over certain finite fields. *Journal of the society for industrial and applied mathematics* 8, 2 (1960), 300–304.

[45] Peter Reiher, John Heidemann, David Ratner, Greg Skinner, and Gerald Popek. 1994. Resolving File Conflicts in the Ficus File System. In *Proceedings of the Summer USENIX Conference*. 183–195.

[46] Jason K. Resch and James S. Plank. 2011. AONT-RS: Blending Security and Performance in Dispersed Storage Systems. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies (FAST'11)*. USENIX Association, Berkeley, CA, USA, 14–14. http://dl.acm.org/citation.cfm?id=1960475.1960489

[47] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere. 1990. Coda: A Highly Available File System for a Distributed Workstation Environment. *IEEE Trans. Comput.* 39 (1990), 447–459.

[48] Yogeshwer Sharma, Philippe Ajoux, Petchean Ang, David Callies, Abhishek Choudhary, Laurent Demailly, Thomas Fersch, Liat Atsmon Guz, Andrzej Kotulski, Sachin Kulkarni, Sanjeev Kumar, Harry Li, Jun Li, Evgeniy Makeev, Kowshik Prakasam, Robbert Van Renesse, Sabyasachi Roy, Pratyush Seth, Yee Jiun Song, Benjamin Wester, Kaushik Veeraraghavan, and Peter Xie. 2015. Wormhole: Reliable Pub-Sub to Support Geo-replicated Internet Services. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. USENIX Association, Oakland, CA, 351–366.

[49] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. 2003. *NFS Version 4 Protocol*. RFC 3530. Network Working Group.

[50] S. Shepler, M. Eisler, and D. Noveck. 2010. *NFS Version 4 Minor Version 1 Protocol*. RFC 5661. Network Working Group.

[51] SPEC. 2001. SPEC SFS97_R1 V3.0. (September 2001). www.spec.org/sfs97r1.

[52] Emil Stefanov, Marten van Dijk, Ari Juels, and Alina Oprea. 2012. Iris: A scalable cloud file system with efficient integrity checks. In *Proceedings of the 28th Annual Computer Security Applications Conference*. ACM, 229–238.

[53] The Apache Software Foundation. 2018. Apache Curator. (2018). https://curator.apache.org.

[54] The Apache Software Foundation. 2018. Apache Hedwig. (2018). https://bookkeeper.apache.org/docs/master/hedwigDocs.html.

[55] The Apache Software Foundation. 2018. Apache Thrift. (2018). https://thrift.apache.org.

[56] The Apache Software Foundation. 2018. Hedwig Design. (2018). https://bookkeeper.apache.org/docs/r4.3.0/hedwigUser.html.

[57] Joseph Tsidulko. 2017. The 10 Biggest Cloud Outages of 2017 (So Far). (2017). http://www.crn.com/slide-shows/cloud/300089786/the-10-biggest-cloud-outages-of-2017-so-far.htm.

[58] Jos van der Til. 2014. Jerasure library that adds Java Native Interface (JNI) wrappers. (2014). https://github.com/jvandertil/Jerasure.

[59] Werner Vogels. 2009. Eventually consistent. *Commun. ACM* 52, 1 (jan 2009), 40. https://doi.org/10.1145/1435417.1435432

[60] Michael Vrable, Stefan Savage, and Geoffrey M Voelker. 2012. BlueSky: a cloud-backed file system for the enterprise.. In *FAST*. 19.

[61] Hakim Weatherspoon and John D Kubiatowicz. 2002. Erasure coding vs. replication: A quantitative comparison. In *International Workshop on Peer-to-Peer Systems*. Springer, 328–337.

[62] Zhe Wu, Michael Butkiewicz, Dorian Perkins, Ethan Katz-Bassett, and Harsha V. Madhyastha. 2013. SPANStore: Cost-effective Geo-replicated Storage Spanning Multiple Cloud Services. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. ACM, New York, NY, USA, 292–308. https://doi.org/10.1145/2517349.2522730

[63] SGI XFS. 2016. xfstests. (2016). http://xfs.org/index.php/Getting_the_latest_source_code.

[64] E. Zadok and J. Nieh. 2000. FiST: A Language for Stackable File Systems. In *Proceedings of the Annual USENIX Technical Conference*. USENIX Association, San Diego, CA, 55–70.