Iris: A Scalable Cloud File System with Efficient Integrity Checks

Emil Stefanov*
UC Berkeley
emil@cs.berkeley.edu

Marten van Dijk
RSA Laboratories
mvandijk@rsa.com

Ari Juels
RSA Laboratories
ajuels@rsa.com

Alina Oprea
RSA Laboratories
aoprea@rsa.com

ABSTRACT

We present Iris, a practical, authenticated file system designed to support workloads from large enterprises storing data in the cloud and be resilient against potentially untrustworthy service providers. As a transparent layer enforcing strong integrity guarantees, Iris lets an enterprise tenant maintain a large file system in the cloud. In Iris, tenants obtain strong assurance not just on data integrity, but also on data freshness, as well as data retrievability in case of accidental or adversarial cloud failures.

Iris offers an architecture scalable to many clients (on the order of hundreds or even thousands) issuing operations on the file system in parallel. Iris includes new optimization and enterprise-side caching techniques specifically designed to overcome the high network latency typically experienced when accessing cloud storage. Iris also includes novel erasure coding techniques for the first efficient construction of a dynamic Proofs of Retrievability (PoR) protocol over the entire file system.

We describe our architecture and experimental results on a prototype version of Iris. Iris achieves end-to-end throughput of up to 260MB per second for 100 clients issuing simultaneous requests on the file system. (This limit is dictated by the available network bandwidth and maximum hard drive throughput.) We demonstrate that strong integrity protection in the cloud can be achieved with minimal performance degradation.

1. Introduction

Organizations that embrace cloud computing outsource massive amounts of data, as well as workloads to external cloud providers. Cost savings, lower management overhead, and rapid elasticity are just some of the attractions of the cloud.

But cloud computing entails a sacrifice of control. Tenants give up configuration and management oversight of the infrastructure that contains their data and computing resources. In cloud storage systems today, for example, tenants can only discover corruption or loss of their data (particularly infrequently accessed data) if their service providers faithfully report failures or security lapses—or when a system failure occurs. This integrity-measurement gap creates business risk and complicates compliance with regulatory requirements.

We propose a cloud-oriented *authenticated file system* called Iris that gives tenants efficient, comprehensive, and real-time data-integrity verification. The Iris system enables an enterprise tenant—or an auditor acting on the tenant's behalf—to verify the integrity and freshness of any data retrieved from the file system while performing typical file system operations. Data integrity ensures that data has not been accidentally modified or corrupted, while freshness ensures that the latest version of the data is always retrieved (and thus prevents rollback attacks reverting the file system state to a previous version). Moreover, tenants in Iris can efficiently audit the cloud provider on a regular basis and obtain continuous guarantees about the correctness and availability of the entire file system.

Motivating scenario We envision a scenario in which a large enterprise migrates its internal distributed file system to a cloud storage service. An important requirement for our system is that enterprise users (called herein *clients*) perform the same file system operations as they typically do (e.g., file read, write, update, and delete operations, creation and removal of directories) without modifying applications running on user machines. The slowdown in operation latency should be small enough to be unnoticed by users even when a large number of clients (on the order of hundreds and even thousands) issue operations on the file system in parallel.

Design goals in Iris Iris aims to support outsourcing of enterprise-class file systems to the cloud seamlessly and with minor performance degradation. Thus the design goals of Iris stem from the most common needs of enterprise-class tenants:

- *Efficiency:* Cloud file systems need to achieve throughputs close to those offered by local file systems under thousands of operations issued concurrently by many clients. Individual file system operation latency overhead should also be minimal.

- *Scalability:* A cloud file system should be scalable to large enterprise file systems under a variety of workloads with potentially very sensitive performance requirements. The system should also be scalable to multiple clients issuing operations on the file system in parallel.

- *Transparency:* Transparency and backwards compatibility with existing file system interfaces is important to facilitate migration to the cloud seamlessly.

*This research was mostly performed while visiting RSA Laboratories. Partially supported by an NSF Graduate Research Fellowship under Grant No. DGE-0946797 and a DoD National Defense Science and Engineering Graduate Fellowship.

ACSAC '12 Dec. 3-7, 2012, Orlando, Florida USA

- *Strong integrity protection:* Data and file system meta-data retrieved from the cloud need to be both *authentic* and *fresh*. Tenants' ability to verify continuously the integrity and availability of their data with minimal bandwidth and computation is a desirable feature, as well.

Contributions of Iris

In more detail, the key technical contributions and novel elements in Iris are:

- **Authenticated file system design:** The first contribution of Iris is to provide data integrity and freshness for an enterprise-class file system in an efficient way. To that end, we design a balanced Merkle-tree data structure that authenticates both file system data and meta-data blocks. The distinctive features of our data structure design compared to other authenticated file systems is that it efficiently supports updates from multiple clients *in parallel* (without blocking) and it handles *all existing* file system operations (including delete, move and truncate) with minimal overhead. Iris further implements many optimizations for typical file system workloads (e.g., those involving sequential file accesses).

In addition, Iris is designed to overcome the main economic barrier in migrating storage to the cloud: the impact of high network latency. Iris implements novel caching techniques locally, within the enterprise trust boundary. A lightweight (possibly distributed) trusted entity called *the portal* mediates file system operations passing between the enterprise clients and cloud and caches most recently accessed blocks. We develop techniques to cache the authentication information (nodes of the Merkle tree), handle dependencies among nodes, and preserve Merkle tree consistency when multiple clients simultaneously access nodes from the (partially cached) data structure.

- **Continuous auditing of file system correctness (PoR):** Iris provides the first construction for *dynamic* Proofs of Retreivability (PoR) [18]; it enables an enterprise tenant to continuously monitor the operation of the cloud storage service and obtain strong guarantees about the correctness and availability of the *entire file system*. With a PoR, a tenant can verify the correctness and availability of large data collection stored in the cloud with low computation and bandwidth cost. While previous PoR protocols are designed for static data (e.g., archival files), our protocol is the first to efficiently support *dynamic PoR* protocols over the entire file system. One of the key innovations in Iris is the design of a sparse randomized erasure code over the file system data and metadata. The new erasure code is specifically crafted to hide the code parity structure (typically revealed by other codes during file updates) and be resilient against a potentially adversarial cloud. It enables recovery when corruptions are detected through auditing.

- **End-to-end design and implementation:** One of our main contributions is the end-to-end design and full implementation of Iris consisting of 25,000 lines of code. We show through our performance evaluation that the caching mechanism in Iris is effective in achieving low latency for file system operations (similar to LAN latencies). Moreover, Iris achieves high throughput (up to 260MB for 100 clients issuing simultaneous requests on the file system in our local testbed), with the bottleneck given by the available network bandwidth and hard drive throughput. Finally, we demonstrate that the overall cost of adding strong integrity protection to Iris is minimal.

2. Related Work

File systems with integrity support: Early cryptographic file systems were designed to protect data confidentiality [6] and the integrity of data [29] in local storage. Later cryptographic networked file systems provided different integrity guarantees. TCFS [8] and SNAD [23] provide data integrity by storing a hash for each file data block. A number of systems construct a Merkle tree over files in order to authenticate file blocks more efficiently (e.g., [14, 13, 19, 4, 24, 25]).

Many cryptographic file systems to date provide data integrity, but do not authenticate the file system directory structure (or meta-data), e.g., [19, 24, 25]. Others, while authenticating both file system data and meta-data, do not provide strong freshness guarantees. SiRiUS [16] does not ensure data freshness, but only partial meta-data freshness by periodically requiring clients to sign meta-data entries. SUNDR [21] implements a property called "fork consistency" that detects freshness violations only when clients communicate out of band. More recently, SPORC [12] supports the building of collaborative cloud applications, enabling clients to recover from malicious forks performed by untrusted cloud servers. Depot [22] reconciles malicious forks even in the presence of faulty clients.

To the best of our knowledge, few cryptographic file systems provide freshness of both file system data and meta-data. SFSRO [14] and Cepheus [13] build a Merkle tree over the file system directory tree. While this approach efficiently supports file system operations like moving or deletion of entire directories, it results in an unbalanced authentication data structure and thus has a high authentication cost for directories with many entries. Athos [17] constructs a balanced data structure that maps the directory tree of the file system in a set of node relations represented as a skip list. Athos abstracts away the hierarchical structure of the directory tree, however, and doesn't provide efficient support for some existing file system operations, e.g., garbage collection. Moreover, its primary, prototyped design handles only a single client. FARSITE [4] is a peer-to-peer storage system that uses a distributed directory group to maintain meta-data information. Meta-data freshness is guaranteed when more than two thirds of the directory group members are correct. Data freshness is provided by storing hashes of file Merkle trees in the directory group.

Other systems provide data integrity guarantees for key-value stores. Venus [28] implements strong consistency semantics for a key-value store with malicious storage in the back-end. CloudProof [26] provides a mechanism for clients to verify the integrity and freshness, as well as other properties of cloud-stored data.

PoRs/PDPs: A *Proof of Retrievability* (PoR) [18] is a challenge-response protocol that enables a cloud provider to demonstrate to a client that a file is retrievable, i.e., recoverable without any loss or corruption. *Proofs of data possession* (PDP) [5] are related protocols that only detect a large amount of corruption in outsourced data. Most existing PDP [5] and PoR [18, 27, 7, 10] protocols are designed for static data, i.e., infrequently modified data.

Dynamic PDP protocols have been proposed by Erway et al. [11], but they were not designed to handle typical file system operations. For instance, Erway et al. [11] support operations like insertion in the middle of a file, but do not efficiently support moving and deleting entire files or directories. The CS2 system [20] designs and implements an efficient dynamic PDP protocol, as well as techniques for searching over encrypted data.

Several papers ([30] and [31]) claim to construct dynamic PoRs, but in fact only provide dynamic PDP schemes. To the best of our knowledge, designing efficient dynamic PoR protocols is extremely challenging and has stood as an open problem in the community.

3. System model and overview

Iris is designed as an enterprise file system using back-end cloud storage. Clients in Iris (enterprise users) issue file system operations intermediated by Iris and relayed to the public cloud. An important design consideration is that heavy caching on the enterprise side is strictly necessary. There are several reasons for this. First, if local caching is not performed, the cost of network transfer to and from the cloud will far outweigh any storage costs savings ([9] points to the extremely high cost of network transfer). Second, without local caching individual operation latency will be prohibitive for the system to be usable.

Existing network file systems are not designed with similar requirements in mind. For instance, NFS is not optimized for high network latency scenarios [15]. Moreover, most cloud storage systems available today (e.g., Amazon S3) export a key-value store interface and employ a flat namespace. Our system is unique in providing a file system interface to enterprise clients (for compatibility with existing applications), and at the same time ensuring low operation latency. In addition, our main goal is to support integrity protection of both file system data and meta-data and continuous verification of full file system correctness and availability with minimum overhead.

We describe here Iris's architecture, threat model, and give an overview of our solution and technical challenges.

3.1 System architecture

In our architecture (shown in Figure 1), a trusted portal residing within the enterprise trust boundary intermediates all communication between enterprise clients and the cloud. The portal caches data and meta-data blocks recently accessed by enterprise clients. Cached blocks are evicted once the cache is full and they are not utilized by a pending operation. The portal is also responsible for checking data integrity and freshness for all file system operations (with the *integrity layer* component). Data integrity ensures that data retrieved from the cloud has been written by authorized clients and has not been accidentally modified or corrupted at the cloud side. A stronger property, data freshness, ensures that data accessed by a client during a file system operation is always the latest version written to the cloud by any client.

The portal offers a *portal service* to clients issuing file system operations, and communicates to the cloud through the *storage interface* component. The auditing component issues challenges to the cloud periodically to verify the correctness and availability of the entire file system. The portal plays a central role in recovering from data corruptions: The portal caches error-correcting information (or more concisely, *parities*) for the full file system. When corruption is detected through the auditing protocol, these parities enable recovery of lost or corrupted data. Parities are backed up to the cloud on a regular basis (e.g., once a day or once a week).

To scale to large organizations with tens of thousands of clients, the portal needs to be distributed internally using a tool to ensure consistency of distributed caches (e.g., memcached [3]). For purposes of our prototype detailed in Section 6, we have instantiated the portal on a single server machine and show that it can scale up to 100 clients simultaneously executing sequential workloads in parallel on the file system.

The cloud maintains the distributed file system, consisting of all files and directories belonging to enterprise users. Iris is designed to use any existing cloud storage system transparently in the back end without modification. In addition, the cloud also stores the MACs and Merkle tree necessary for authenticating data, as well as the checkpointed parity information needed to recover from potential corruptions at the portal. As an additional resilience measure, the parity information could be stored on a different cloud or replicated internally within the enterprise.

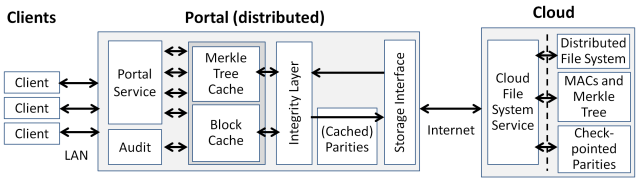

Figure 1: System architecture.

3.2 Threat model

Iris treats the portal, which is controlled by the enterprise, as a trusted component, in the sense that it executes client file system operations faithfully. No trust assumption is required on clients: They may act arbitrarily within the parameters of the file system. (The file system may enforce access-control policies on clients through the portal, but such issues lie outside the scope of Iris.)

The cloud, on the other hand, is presumed to be potentially untrustworthy. It may corrupt the file system in a fully Byzantine manner. The cloud may alter or drop file system operations transmitted by the portal; it may corrupt or erase files and/or metadata; it may also attempt to present the portal with stale, incorrect, and/or inconsistent views of file system data. The objective of the portal in Iris is to detect the presentation of *any invalid data by the cloud*, i.e., immediately identify any cloud output that reflects a file system state different from that produced by a correct execution of the operations emitted by the portal.

3.3 Solution overview and challenges

Iris consists of two major components:

Authenticated file system: As already described, the first challenge we address in building an authenticated enterprise-class file system is the high cost of network latency and bandwidth between the enterprise and cloud. Another challenge is efficient management and caching of the authenticating information. Integrity and freshness verification should be extremely efficient for existing file system operations and induce minimal latency.

Iris employs a two-layer authentication scheme. In its lower layer, it stores on every file block a message-authentication code (MAC)—generated by the portal when a client writes to the file system. These MACs ensure data integrity. To ensure freshness, it is necessary to authenticate not just data blocks, but also their *versions*. Each block has an associated version counter that is incremented every time the block is modified. This version number is bound to the file-block's MAC: To protect against cloud replay of stale file-blocks (rollback attacks), the counters themselves must be authenticated.

The upper layer of the authenticated data structure in Iris is a balanced Merkle-tree-based structure that protects the integrity of the file-block version counters. This data structure embeds the file system directory tree, and balances each directory for optimization. Attached to each node representing a file is a sub-tree containing file-block version counters. The root of the Merkle tree stored at the portal guarantees the integrity and freshness of both data and meta-data in the file system.

This Merkle-tree-based structure has two distinctive features compared to other authenticated file systems: (1) *Support for existing file system operations:* Iris maintains a balanced binary tree over the file system directory structure to efficiently support existing

file system calls; and (2) *Support for concurrent operations:* The Merkle tree supports efficient updates from multiple clients operating on the file system in parallel. Iris also optimizes for the common case of sequential file-block accesses: Sequences of identical version counters are compacted into a single leaf. We detail the data structure in Section 4, and the Merkle tree caching mechanism in Section 6.

Auditing protocol: Iris enables the enterprise tenant to continuously monitor and assess the correctness and availability of the entire file system through the auditing protocol. The auditing protocol in Iris is an instantiation of a PoR protocol and, in fact, the first dynamic PoR protocol supporting data updates. Previous PoR protocols have been designed for static data (files that do not undergo modifications). In any PoR, the tenant samples and checks the correctness of random data blocks retrieved from the cloud to detect any large-scale data corruption. To recover from small-scale damage, parity information computed with an erasure code needs to be maintained over the data.

The main challenge in designing a dynamic PoR protocol is that the erasure code structure, i.e., mapping of data blocks to parity blocks, must be randomized to prevent an adversarial server from introducing targeted, undetectable file corruptions. File updates are most problematic as they partially reveal the code structure (in particular the parity blocks corresponding to updated file blocks). At the same time, file updates should be efficient and involve only a small fraction of parity blocks.

We overcome this challenge with two techniques. First, we design Iris to cache parity information locally at the portal (and only checkpoint it to the cloud at fixed time intervals). As the cloud does not perceive individual file updates, but only parity modifications aggregated over a long time interval, the cloud cannot easily infer the mapping from file blocks to parity blocks. Second, we design a new sparse, binary code structure that combines randomly chosen blocks from the file system into a codeword. The code supports updates to the file system very efficiently through binary XOR operations. Its sparse structure supports very large file systems. This novel code construction is carefully parameterized to optimize local storage at the portal side, update cost, and bandwidth and computation in the auditing protocol. We describe the auditing protocol and the erasure code construction in Section 5.

4. Authentication in Iris

We describe in this section how Iris provides strong data protection, including integrity and freshness, for both file system data and meta-data. The authentication scheme in Iris is based on Merkle trees, and designed to support existing file system operations. In addition, random access to files for both read and write operations is a desirable feature (offered by existing file systems like NFS) that we also choose to implement. The tenant needs to maintain at all times the root of the Merkle trees for checking the integrity and freshness of data retrieved from the cloud. For reducing operation latency, recently accessed nodes in the tree are also cached at the portal (the caching mechanism is described in Section 6).
Figure 2 depicts the main components of our tree-based structure used for authentication:

Block-level MACs: To provide file-block integrity, we store a MAC for each file block, and combine block MACs from the same file in a *MAC file*. We choose to store MACs for each file block (instead of a single MAC for each file) to support random accesses to files. Block MACs are computed by the portal when a client writes to the file system. For providing freshness, we need to bind a unique version number to each file block every time it's updated and include the version number in the block MAC. To protect against rollback attacks (in which clients are presented with an old state of the file system), version numbers will have to be authenticated as well.

File version trees: We construct a *file version tree* per file that authenticates version numbers for all file blocks in a compressed form. Briefly, the file version tree compresses the versions of a consecutive range of blocks into a single node, storing the index range of the blocks and their common version number. File version trees are optimized for sequential access to files. For instance, if a file is always written sequentially then its file version tree consists of only one root node. The compacted version tree essentially behaves as a range tree data structure. An example of a compacted tree is shown in Figure 3.

Directory trees: To authenticate file system meta-data (or the directory structure of the file system), the file system directory tree is transformed into a Merkle tree in which every directory is mapped to a *directory subtree*. We have chosen to map our authenticated data structure onto the existing file system tree in order to efficiently support file system operations like delete or move of entire directories. To support directories with large number of files efficiently, we create a balanced binary tree for each directory that contains file and subdirectory nodes in the leaves, and includes intermediate, empty internal nodes for balancing. Nodes in a directory tree have unique identifers assigned to them, chosen as random strings of fixed length. A leaf for each file and subdirectory is inserted into the directory tree in a position given by a keyed hash applied to its name and its parent's identifier (to ensure tree balancing).

At the leaves of the directory tree, we insert the file version trees in compacted form, as described above. Internal nodes in the Merkle tree contain hash values computed over their children, as well as some additional information, e.g., node identifiers, their rank (defined as the size of the subtree rooted at the node), file and directory names.

Our Merkle tree supports the following operations. Clients can insert or delete file system object nodes (files or directories) at certain positions in the tree. Those operations trigger updates of the hashes stored on the path from the inserted/deleted nodes up to the root of the tree. Deleted subtrees are added to the free list, as explained below. Clients can verify a file block version number, by retrieving all siblings on the path from the leaf corresponding to that file block up to the root of the tree. Searches of files or directories in the tree can also be performed, given absolute path names.

We also implement an operation randompath-dir-tree for directory trees. This feature is needed to execute the challenge-response protocols of the auditing component in Iris. A (pseudo)-random path in the tree is returned by traversing the tree from the root, and selecting at each node a child at random, weighted by rank. In addition, the authentication information for the random path is returned, so the tenant can verify that the path has been chosen pseudo-randomly.

With this Merkle tree construction, we authenticate both file system meta-data, as well as file block version numbers. Together with the file block MACs, this mechanism ensures data integrity and freshness, assuming that the portal always stores the root of the Merkle tree.

Free list: As an optimization, we also maintain in the data structure a *free list* containing pointers of nodes deleted from the data structure, i.e., subtrees removed as part of delete or truncate operations. The aim of the free list is to defer garbage collection of deleted nodes and support remove and truncate file system operations efficiently. We omit further details due to space limitations.

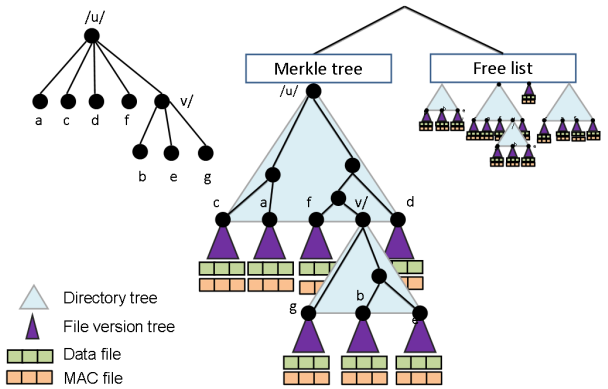

Figure 2: Authenticated tree. A file system directory on the left and its mapping to the Merkle tree on the right.

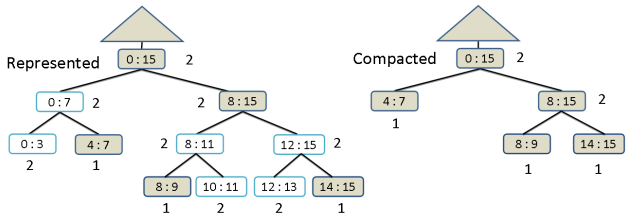

Figure 3: File version tree for a file with 16 blocks. Blocks 0-3 and 10-13 have been written twice, all other blocks have been written once. White nodes on the left are removed in the compacted version on the right. Version numbers are adjacent to nodes.

5. Auditing protocol

The authentication mechanism in Iris presented in the previous section can be used to verify the correctness of all blocks retrieved from the file system during the course of normal operations issued by clients. A challenging question that we address in this section is how can the enterprise verify infrequently accessed blocks and detect even small amounts of corruptions spread throughout the file system. We are particularly interested in offering strong assurances to the enterprise about the correctness and availability of *the entire file system*. An important requirement is that auditing of correctness should be performed with minimal bandwidth and computation. For instance, downloading a substantial fraction of the file system to verify its correctness would not be an acceptable solution. In addition, a recovery mechanism is needed to reconstruct the original data once corruptions are detected.

Several different protocols that address to some extent this question have been proposed in the literature. PoR protocols provide strong assurances about availability of data outsourced to the cloud, and a recovery mechanism, but they have only been designed for static data (files that do not undergo modifications). PDP protocols, while supporting updates to data, ensure only *detection* of a certain amount of data corruption, but do not implement a recovery mechanism. To the best of our knowledge, our solution here is the first *dynamic PoR protocol* over an entire file system, supporting updates and providing an efficient recovery mechanism in case data corruption is detected.

We start by presenting at a high level how existing PoR protocols work, and then describe the challenges of adapting these ideas to a dynamic setting. We then discuss our main insights and contributions in constructing a dynamic PoR protocol.

5.1 Static PoR protocols

In a PoR protocol, the tenant encodes a single file with an error-correcting code (ECC) and stores the encoded file in the cloud. The encoded file contains the original file and some *parity blocks*, redundant blocks computed with the ECC that are needed in recovering from corruption. To ensure correctness and availability of the data, the tenant periodically challenges the cloud for a few randomly selected file blocks, and verifies their correctness. Through this auditing protocol, the tenant can detect large-scale corruption to the file (exceeding a certain fixed threshold). Small corruptions, while not detectable through sampling, can be recovered from the redundancy embedded in the encoded file.

An important parameter in a PoR is the recovery-failure probability ρ. This is the probability, assuming that the cloud replies correctly to all challenges during an audit, that the tenant can't recover the file from the cloud's storage. The size and frequency of challenges in a PoR may be calibrated to achieve a target parameter ρ given the file size, and error-correcting code parameters.

5.2 Challenges for dynamic PoRs

The main challenge in adapting a static PoR protocol to a dynamic setting is the construction of an error-correcting code with several required properties. As a reminder, the error-correcting code is used to recover from corruptions once the auditing protocol detects missing or corrupted data at the cloud. An additional requirement our system has compared to previous PoR protocols is that it needs to recover from corruptions of both data and meta-data in the entire file system (while previous PoR protocols have been designed for single files).

Our first observation is that we can use in our system an erasure code instead of a more expensive error-correcting code. The reason is that Iris's main service is authentication of file system blocks, and, therefore, the portal can verify the correctness of file blocks and Merkle tree nodes during recovery and determine the positions of corrupted blocks. We present the remaining challenges in achieving an efficient dynamic PoR protocol:

Challenge 1: Update efficiency The erasure code has to support updates to the file system efficiently. In particular a modification to a file block or Merkle-tree node should require the update of only a small number of parity blocks. Additionally, it would be preferable to use cheap Galois field arithmetic in the parity computation, such as $GF(2)$ which essentially consists of XOR operations. Higher order Galois field arithmetic (as employed by Reed-Solomon codes, for instance) is too expensive to sustain our desired throughput of several hundred megabytes per second.

This requirement excludes upfront the use of maximum-distance separable (MDS) codes. While such codes are attractive for their correction capability, a parity block in an MDS codes depends on all message blocks, and therefore updates to the codeword are quite impractical.

Thus we must use a non-MDS code, with a lower error-correction capability. For instance, we might *stripe* the file system, that is, partition it into a number of smaller components, called stripes, and apply an erasure code individually to each stripe (striping is a common technique employed in most storage systems today). With this approach, updates would be more efficient as an update to a file block or Merkle tree node would involve updating only parity

blocks within a single stripe.

Challenge 2: Hiding code structure Nevertheless, striping introduces a problem. When a client updates a block of the file system along with the corresponding stripe parities, it *reveals code-structure information* to the cloud, namely the correspondence between the file blocks and the parity blocks. A malicious cloud can then create a targeted corruption against the file system, e.g., it can corrupt a single stripe and its corresponding parity blocks. Such corruption, being focused, will be hard to detect by sampling (since sampling detects only a large amount of corruption).

We overcome this challenge with two key techniques:

1. *Cache parities at the portal* We cache the parity information at the enterprise side and only transmit parities to the cloud at regular time intervals for back up (e.g., at the end of the week). With this approach, the cloud does not perceive individual updates to the file system, but only the aggregate parity structure over a large number of updates and can not infer the exact code structure. Moreover, updates are extremely efficient if parities are stored in main memory at the portal.
2. *Randomize code structure* Even when parities are stored at the portal, it is important that the stripe structure is not revealed to the cloud to avoid targeted corruptions. To enforce this, we randomize the assignment of file blocks to stripes.

If these two design principles are employed, it might seem that after caching the parities locally and randomizing the assignment of file blocks and tree nodes to stripes, any erasure code could be used for computing the parity blocks within a stripe. But our system has to overcome another subtle challenge:

Challenge 3: Variable-length encoding Typically, the code parameters for an erasure code, including the message size, and the size of parity information are fixed and known in advance (before encoding is performed). However in Iris we need to compute parity blocks over an entire file system data and meta-data blocks without knowing in advance the total size of the file system. At the same time, we have to enforce a randomization of the mapping of file system blocks to parity blocks at any given time. Therefore, approaches in which new parity blocks are created as more data is added to the file system in a streaming fashion (e.g., LDPC codes) would not be applicable here.

New sparse randomized erasure code construction. Our solution is to set an upper bound on the total size of the file system, and design a novel erasure code construction that is *sparse* in the sense that it supports incremental updates to the codeword very efficiently, even when only a fraction of the maximum size is used by the file system. The construction randomizes the mapping of file system blocks to parity blocks, and uses binary XOR operations. The size of the parity information is also constrained to fit into the main memory of typical servers today (an important consideration for efficient updates). We are able to prove for this construction an exponentially small bound for the recovery-failure probability.

If the file system needs to be expanded, the error correcting codes can be rebuilt, but a more bandwidth efficient solution would be to double the ECC data structure when the file system doubles in size.

5.3 Our erasure code

Parameter overview: We first set an upper bound for the entire file system size, denoted n. In our example parameterization, n is the number of 4KB blocks needed for a file system of size 1PB. Our erasure code construction is scalable up to that size, but once the file system exceeds the upper bound, the code parameters need to be changed and the file system has to be re-encoded.

To correct a fraction α of erasures, the storage for parities must be at least $s \geq \alpha n$ blocks—a coding-theoretic lower bound. Here s is limited by the sizes of current memories to about $s = O(\sqrt{n})$ for practical file system sizes and thus $\alpha = \Omega(1/\sqrt{n})$. (To obtain a probabilistic guarantee that at most an α-fraction of all stored file blocks is missing or corrupted, the tenant must challenge $c = O(1/\alpha) = O(\sqrt{n})$ randomly selected file blocks.)

To support updates efficiently we split the huge codeword into $m \approx \alpha n$ stripes; each stripe being a codeword itself with p parities. With high probability, given an α-fraction of erasures, each stripe is affected by only $O(\log n)$ erasures. Thus to correct and recover stripes, we need $p = O(\log n)$ parity blocks per stripe, leading to $s = O(\alpha n \log n) = O(\sqrt{n} \log n)$ memory. Each write only involves updating $u = O(\log n)$ parities within the corresponding stripe. By using a sparse parity structure, though, we are able to reduce u to $O(\log \log n)$.

Details on our erasure-code construction: Our erasure code is a sparse one based on efficient XOR operations. Although the new construction is probabilistic in that successful erasure decoding is not guaranteed for any number of erasures, its main advantage is that it is a binary efficient code scalable to large codeword lengths.

The portal computes parities over both file blocks and Merkle tree nodes when block values are updated by a client operation. For the purpose of erasure coding, we view data blocks or tree nodes as identifier-value pairs $\delta = (\delta_{id}; \delta_{val})$, where δ_{id} is a unique identifier (a unique block ID in the file system) and $\delta_{val} = (\delta_1, \ldots, \delta_b)$ is a sequence of b bits denoting the change in block value. (We assume all blocks are initialized with 0.) To randomize the mapping from data blocks to parity blocks, we use a keyed hash function $H_k(.)$ that maps an identifier δ_{id} to a pair (θ_{ind}, θ), where θ_{ind} is a random stripe index and $\theta = (\theta_1, \ldots, \theta_p)$ is a binary vector of p bits. The randomization is graphically depicted in the full version of this paper [1].

The 1s in vector θ indicate the parity bits that need to be updated. Each update modifies at most u of the p parities of the stripe to which δ belongs. That is, $H_k(\delta_{id})$ is designed to produce a binary random vector θ of length p with at most u entries equal to 1. For $u = O(\log p) = O(\log \log n)$ this leads to a sparse erasure code that still permits decoding, but entails relatively few parity updates.

Encoding: We maintain a parity matrix $P[i]$ for each stripe i, $1 \leq i \leq m$. To change the value of block δ_{id} with δ_{val}, the portal computes $H_k(\delta_{id}) = (\theta_{ind}; \theta)$; constructs $A = \delta_{val} \otimes \theta = \{\delta_i \theta_j\}_{i \in [1,b], j \in [1,p]}$; and updates $P[\theta_{ind}] \leftarrow P[\theta_{ind}] \oplus A$. The change in parity structure is shown graphically in the full version of this paper [1].

Since vector θ has at most u non-zero positions, the number of XOR operations for updating a block is u. The total storage for all parities is $s = bpm$ bits.

Decoding: Erasure decoding of the multi-striped structure involves decoding each stripe separately. Gaussian elimination is performed m times, each time computing the right inverse of a $(\leq p) \times p$ matrix–at a cost of at most $p^2 = O((\log n)^2)$ XOR operations. As an additional benefit of our construction, decoding can be done in place, and thus within memory at the portal.

Analysis: The full version of this paper [1] provides a detailed analysis. E.g., with a block size of 4KB, 5KB communication per challenged block, 5.8GB total communication per challenge-response round, 16GB of main memory at the portal for parity storage, and 1PB file system size, we achieve recovery failure probability $\rho \leq 0.74\%$.

5.4 Erasure-coding for Dynamic PoR

We now explain how our erasure code functions in Iris.

PoR encoding and update: During encoding, the portal constructs two parity structures: the *data parity structure* constructed over the file system data blocks (including the data blocks in the free list) and the *meta-data parity structure* over the meta-data blocks (internal nodes in the data structure comprising the Merkle tree and free list).

The challenge-response protocol: The portal challenges the cloud to return a set of c (again $c = O(\sqrt{n})$ randomly selected file system data blocks, including data blocks from the free list. These blocks are all leaf nodes in the authenticated data structure containing the Merkle tree and free list. As an optimization, the portal sends a seed from which the challenge set is derived pseudo-randomly.

The c selected random blocks together with the authenticating paths from the authenticated data structure are transmitted back to the portal. The portal verifies the correctness of the responses by performing two checks. First, it verifies the integrity and freshness of the selected blocks, checking the block MACs and the path to the root in the authenticated data structure. Second, it verifies that the blocks have been correctly indexed by the challenges according to the node ranks/weights. (This proves that the file system data blocks are selected with uniform probability.) As a byproduct of these checks the challenge-response protocol also verifies the integrity and freshness of the meta-data blocks (internal nodes in the authenticated data structure). We can immediately infer that if a fraction α of file system data blocks don't verify correctly, then at most a fraction α of internal nodes in the Merkle tree and free list are either missing or corrupted.

Recovery: See the full version of this paper [1] for the recovery algorithm.

6. Implementation

Our implementation of Iris is a 25,000-line end-to-end system with all integrity checking in place. The system is fully asynchronous and never holds a lock while waiting for network or disk I/O operations. The code runs in user space as a transparent layer that can take advantage of any existing storage system at the cloud provider. Our implementation uses the open-source .NET framework Mono, which is advantageously platform-independent: Iris can run on Linux, Windows, and MAC OS.

Our implementation includes the Portal, a simple Cloud storage server, and clients that run traces and benchmarks, as depicted in the detailed system architecture in Figure 1.

6.1 Cloud

The cloud stores not only regular file system data, but also authenticating meta-data, including MAC files and our Merkle tree authenticated data structure, as well as checkpointed parities needed for recovery. The repositories for these data types are shown at the top of Figure 1.

The portal performs reads and writes to the various data repositories by invoking their respective cloud-side services. The *Cloud File System Service* handles requests for file blocks, MAC files, and the Merkle tree (stored in our implementation in an NTFS file system). Operations on file blocks are executed asynchronously at the portal. Sequential access operations to the same file can potentially arrive out of order at the cloud. (Re-ordering can occur in transit on the network, as our portal and cloud machines are each equipped with three network cards.) To reduce disk spinning, the Cloud File System Service orders requests to the same file in increasing order by block offset.

6.2 Portal

The portal interacts with multiple clients which issue file system calls to the *Portal Service*. The portal executes client operations in parallel: Each operation is executed in a thread pool as a user-scheduled task with asynchronous steps. When an operation is waiting for a long running step such as disk or network I/O, the task is paused and the current thread switches to another task. This allows thousands of simultaneously active operations to be handled by the thread pool with a small number of threads. In our setup, the thread pool had 16 threads—one for each virtual CPU core, for maximum parallelism.

Operations don't interact directly with the cloud, but instead with the Merkle Tree and Block Caches. All data and meta-data requested by the caches is downloaded from the cloud via the *Storage Interface* in the portal. While in use by an active operation, blocks and nodes are retained in the cache. Prior to being cached, however, blocks and nodes downloaded from the cloud are checked for integrity by the *Integrity Checker* components.

Our implementation benefits from multi-core functionality available in most modern computers. Operations performed on active blocks in the cache are split into atomic operations (e.g., hash update for a tree node, check MAC for a data block or compact nodes in file version trees). These are inserted into various priority queues maintained at the portal. Multiple threads seize blocks from these queues, lock them and execute the atomic operations. Operations are always started in order, but may complete out of order. However, our implementation ensures that the effect of the operations on the system is the same as if they were executed by a single thread in order. If multiple clients issue conflicting operations simultaneously, they are executed in the order in which they arrive to the Portal. It is the responsibility of the clients to perform locking out-of-band.

Distributing the Portal. For scalability, the portal can be distributed across multiple machines. Each portal machine would then be responsible for a subtree of the file system. When clients first mount the file system, they can contact any one of the portals to get the assignment of portal machine to subtrees. As the file system changes over time, a subtree may grow or shrink substantially, and then the subtree assignment will need to be rebalanced by splitting a subtree and copying one part of it onto another portal. Our current implementation does not support this, but we would like to point out that even with a single portal, Iris can achieve a throughput of up to 260 MB/s, which already exceeds the bandwidth to the cloud for many enterprises. Additional challenges (e.g., caching to reduce latency) arise when the portal is *geographically* distributed, but these are out of the scope of this paper.

6.2.1 Merkle tree cache

The Merkle Tree Cache in the portal is Iris's most complex component. Much of the design effort and complexity of Iris lies in the caching strategy for recently accessed portions of the tree. We designed a generic, efficient Merkle Tree Cache that ensures consistency across thousands of simultaneous asynchronous client operations.

When an operation accesses the cache, it first locks it using a mutex and unlocks it when it's done. All of the operations are designed such that they access the cache for a very short period of time for tasks such as changing the value of a few fields of a Merkle tree node. To ensure a high degree of parallelism, the Merkle tree mutex is never locked while an operation waits for a long running step such as network or disk I/O.

When executing operations in parallel, a real challenge is to handle dependencies among tree nodes and maintain data structure

consistency and integrity. We do this by imposing several orderings of operations. Nodes are brought into the cache in a top-down order and are evicted in a bottom-up order. The top-down ordering is necessary because when a node is read from the untrusted storage, it can only be verified once all of its ancestors have also been cached in and verified. Likewise, a node can only be written out to the untrusted storage after the hash of its subtree has been computed. If multiple nodes in a sub-tree are modified, the Merkle Tree Cache will only hash the shared path to the root once, thereby significantly reducing the number of hashes that need to be performed.

Phases. To enforce the ordering, each node is always in one of the following phases: Reading, Verifying, Neutral, Compacting, UpdatingHash, or Writing. A node always traverses these phases in order and only after its parent or children have reached a certain phase. For example, a node only enters the verifying phase after its parent has completed the verifying phase. The Reading and Verifying phases are applied top-down and the Compacting, UpdatingHash, and Writing phases are applied bottom-up. When a node is in the Neutral phase, it is in the cache and available to be used by operations.

Pinning. Operations oftentimes need to access multiple nodes. For example, a WriteFile operation needs to access the path in the version tree that descends all the way to the version node corresponding to a specific block. The operations first *pin* all of the nodes they need and then proceed to execute. If a node is needed by an operation and is not currently in the cache, the operation is paused and resumed when all of its pinned nodes have been loaded into the cache. Once a node is pinned, it is not cached out until it is unpinned (e.g., when the operation completes). A node may be pinned multiple times, in which case it must be unpinned the same number of times until it is considered in the unpinned state and may be cached out.

If a node is pinned, its ancestors, sibling, and siblings of the ancestors are automatically *indirectly* pinned. This is necessary because if the node is modified, the indirectly pinned nodes will be needed when updating the hashes of the path to that node.

Eviction. When the cache reaches its maximum allowed size, it repeatedly evicts least-recently-used (LRU) leaf nodes, causing a bottom-up wave of evictions. Evicting a node consists of transitioning its phase from the Neutral to Compacting. The node then goes through the UpdatingHash and Writing phases until it is finally removed from the cache. If a node and its subtree were not modified, then the UpdatingHash and Writing phases are skipped. If all of the nodes are pinned, then new operations block until the currently executing operations complete and unpin more nodes.

6.2.2 Other components

The Block Cache functions much like the Merkle Tree Cache except that blocks don't have parents/children so there are no dependencies between blocks.

The Merkle Tree and Block Caches keep track of two items per node/block: The old and new data. The old data is the value of the node/block when it was fetched from the cloud. The new data is its value after it was (possibly) modified by an operation. When a node/block is evicted, the portal computes the difference of the byte representations of the old and new data and updates the parities.

Another component of the portal is the auditing module. This service, periodically invoked by the portal, transmits a PoR challenge to the cloud and receives and verifies the response, consisting simply of a set of randomly selected data blocks in the file system and their associated Merkle tree paths. The portal also maintains a repository of *Parities* to recover from file system corruptions detected in a PoR, seen in the portal cache module in Figure 1. Parities undergo frequent modification: Multiple parities are updated with every file-block write. Thus, the Parities repository sits in the main memory of the portal.

The portal can include a checkpointing service that backs up data stored in the main memory at the portal to local permanent storage. To enable recovery in the event of a portal crash, checkpointed data can be periodically transmitted to the cloud (with a MAC for integrity). While we have not implemented this component, it can rely on well-known checkpointing techniques.

7. Experimental evaluation

We ran several experiments to test different aspects of Iris. We first describe our setup and then present our results. Two machines ran the full end-to-end system implementation described in Section 6: The Portal and the Cloud.

Portal Computer. The Portal computer has an Intel Core i7 processor and 12 GB of RAM. The experiments were run on Windows 7 64-bit installed on a rotational disk, but no data was written to the Portal's hard drive for the purpose of our experiments.

Cloud Computer. The Cloud computer has seven rotational hard drives with 1TB of storage each. The file system and MAC files reside on these disks. The disks are used as separate devices and are not configured as a RAID array. This configuration mimics a cloud where each disk could potentially be on a separate physical machine. The operating system (Windows 7 64-bit) runs on an separate additional hard drive to avoid interfering with our experiment.

Networking. Because our file system can handle very large throughput, we used three 1Gbps cables to connect the two computers. Each computer had one network port on the motherboard and two additional network cards. After accounting for networking overhead, the 3 Gbps combined connections between the two computers can handle about 280 MB/s of data transfer as our experiments show.

Configuration. In our configuration, write operations originate from clients (simulated as threads on the Portal). Then they are processed by the Portal and multiplexed over the three network connections. Finally, data reaches the Cloud computer and is written to the appropriate disk. Reads are similarly processed, but the data flow is in the opposite direction (from the Cloud machine to the Portal).

Simulated Latency. To obtain more realistic results, we deliberately simulated 20ms round-trip time (RTT) latency between the clients and Portal, and 100ms RTT latency between the Portal and Cloud. This setting aims to resemble the scenario where the clients and Portal are both part of the same corporate network and the Cloud is a data center located elsewhere on the same continent.

7.1 Workloads

To evaluate Iris, we used the following workloads. Each workload was recorded as a trace and played back exactly under different parameterizations of our system.

- **Tar/Untar** (directory structure): Benchmarks access and modify operations on a tarball consisting of the entire Linux kernel source (420 MB, 37,000 files, and 2,300 directories).
- **IOZone** (various file access patterns): IOZone [2] benchmark of combining various operations (reread/rewrite, random read/write, backwards read, and strided read).
- **Sequential Read/Write** (throughput): Measures the performance of sequentially reading/writing ten files simultaneously, each of size 10 GB.
- **Random Read/Write** (seeks): Measures the performance of

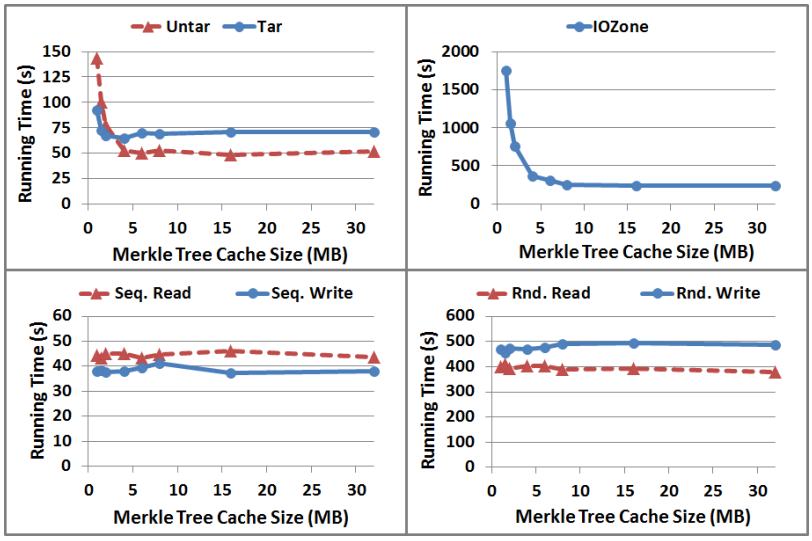

Figure 4: Workloads under different Merkle Tree Cache sizes. Time for the workload to complete vs the Merkle Tree Cache size.

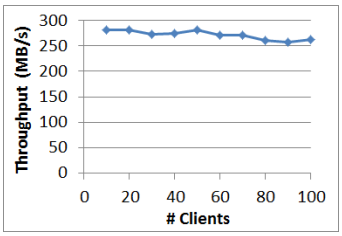

Figure 5: Avg sequential read & write speed.

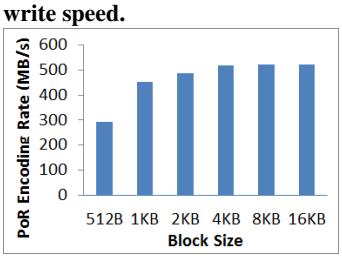

Figure 6: PoR Encoding Rate.

	Total Latency (ms)		Network I/O		Cloud Disk I/O		Portal Processing	
Portal Cache:	Hot	Cold	Hot	Cold	Hot	Cold	Hot	Cold
Create file in directory of depth 0	20.0	20.0	20.0	20.0	0.0	0.0	0.0	0.0
Create file in directory of depth 1	20.0	144.0	20.0	120.0	0.0	9.6	0.0	14.4
Create file in directory of depth 2	20.0	254.0	20.0	220.0	0.0	16.5	0.0	17.5
Create file in directory of depth 3	20.0	363.0	20.0	320.0	0.0	23.4	0.0	19.6
List directory with 10 files at depth 1	27.7	678.9	20.0	620.3	0.0	32.6	7.7	26.0
Write 1 MB file at depth 1, wait completed	24.8	138.8	20.0	120.0	0.0	0.0	4.8	18.8
Read 1 MB file at depth 1	20.0	284.2	20.0	220.0	0.0	43.7	0.0	20.5

Figure 7: Latency for different operations in Iris.

randomly reading/writing ten files simultaneously, each of size 1 GB. Reads and writes are uniformly random, and trigger seeks with almost every operation. For the random read workload, the file is first randomly written and then only the random read portion of the trace is benchmarked.

7.2 Results

Our experimental results show how Iris performs under the above workloads on the full end-to-end system described in Section 6. We note that even with seven hard drives for storage and three 1 Gbps network links between the Portal and Cloud, under no workload was the Portal the bottleneck. Depending on the workload, the limiting factor was either the network or hard drives.

Varying the Merkle Tree Cache Size: The parallel Merkle Tree Cache is crucial for the performance of our system. The cache allows the Portal to perform file operations without having to read and write entire Merkle tree paths from the server for each operation. The asynchronous cache also allows for pausing operations that are waiting to retrieve Merkle tree nodes while other operations actively use the cache.

Multiple paths can be loaded into the Merkle Tree Cache at once while maintaining consistency. In order to demonstrate the usefulness of the cache, in this experiment we varied its size (i.e., how many nodes it can hold at once) and we timed each of the workloads under different cache sizes. The results are in Figure 4.

Interpretation: As demonstrated in the figure, the Tar, Untar, and IOZone workloads greatly benefit from having a Merkle tree cache of 5 to 10 MB (about 10,000 to 20,000 nodes), whereas the sequential and random read/write workloads are mostly unaffected by the cache size.

The reason is quite simple: The Tar, Untar, and IOZone benchmarks frequently revisit the same part of the Merkle tree. For example, the Tar/Untar workloads often read/write multiple files within the same directory (and hence their Merkle tree paths share many nodes). Likewise, the random write portion of the IOZone benchmark creates a file with a large uncompactable Merkle tree which is then read sequentially and the sequential read portion of the workload yields an in-order traversal of the Merkle tree that is significantly sped up by the cache.

On the other hand, the sequential read and write workloads generate version tree nodes that are quickly compacted. Hence the Merkle Tree Cache only needs to hold a few dozen nodes at a time. The random read/write workloads are extremely intensive on the Cloud's disks. Almost every operation causes a seek, so the Cloud's disks are the bottleneck. Because the random read/write operations are executed very slowly by the Cloud's disks and the Portal parallelizes requests for the Merkle tree nodes, there is plenty of time for the Merkle tree nodes to be fetched without delaying the workload.

Scalability: In Iris, all operations are handled by the same thread pool and each file has its own queue of pending/active operations. From the Portal's perspective, there is little difference between each operation being issued by a different client and all operations being issued by the same client. Most of the overhead of having multiple clients comes from having to manage multiple TCP sockets and

their associated buffers.

We wanted then to show that Iris can easily scale to 100 clients accessing it simultaneously. To maximize both strain on the Portal's CPU and the number of cryptographic operations performed, each client generated a sequential access pattern. (With more seek-intensive access, the bottleneck would be disk seeks on the Cloud.)

We averaged the sequential read and sequential write speeds for 10 to 100 clients. Figure 5 shows the results. As can be seen, Iris consistently reads/writes at 250 MB/s to 280 MB/s. The slight performance degradation for 100 clients is due to the fact that many files are accessed at once and that causes a larger portion of disk seeks.

Latency: Figure 7 shows the latency for several basic operations in Iris. The latency is measured under two scenarios: when the portal cache is hot and cold. A hot cache means that the cache already contains all of the data (Merkle tree nodes and blocks) necessary to perform the operation on the portal alone. A cold cache means that all of the data has been evicted from the portal's cache.

The bulk of the latency (over 84%) comes from the portal-cloud and client-portal network latencies. Our results show that the latency introduced by the portal for integrity checking and cache management (denoted as portal processing time) is much smaller in comparison: less than 14% for a cold cache and less than 29% for a hot cache.

The 1 MB read operation takes about half of the time of the 1 MB write operation because the portal notifies the client that the write operation has completed while uploading the file to the cloud in the background. For the read operation, the portal must first read the file from the cloud.

The high cold cache latency for high depth operations (e.g., create depth 3 and list directory) is due to the fact that each file is represented as a separate node in the Merkle tree and tree paths are fetched one node at a time. It should be noted that this latency can be significantly reduced by having the portal fetch all nodes in a path in parallel or grouping multiple files into a single file node.

PoR Encoding Rate: Finally, we measure the rate at which the Portal can perform erasure-encoding for file system recovery if auditing detects corruption. Figure 6 shows encoding speeds for data blocks of different sizes. As can be seen, the PoR encoding rate is sufficiently fast in order to sustain a throughput of 500 MB/s for 4 KB blocks.

8. Conclusions

We have presented Iris, an authenticated file system designed to outsource enterprise-class file systems to the cloud. Iris goes beyond basic data-integrity verification to achieve two stronger properties: File freshness and retrievability. Using a lightweight, tenant-side portal as a point of aggregation, Iris efficiently processes asynchronous requests from multiple clients transparently, i.e., with no underlying file system interface changes.

Iris achieves a degree of end-to-end optimization possible only through a carefully crafted, holistic architecture, one of the systems's major contributions. Iris's architecture also relies on several technical novelties: The authenticating data-structure design and management, caching techniques, sequential-file-access optimizations, and a new erasure code enabling the first efficient dynamic PoR protocol.

Acknowledgements

We'd like to extend our thanks to Roxana Geambasu for her insightful comments on a previous draft of the paper and the anonymous reviewers for all their feedback and suggestions.

9. References

[1] Full version, `http://eprint.iacr.org/2011/585.pdf`.

[2] IOzone filesystem benchmark. www.iozone.org. 2011.

[3] `www.memcached.org`.

[4] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. Usenix, 2002.

[5] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song. Provable data possession at untrusted stores. In *14th ACM CCS*, pages 598–609, 2007.

[6] M. Blaze. A cryptographic file system for Unix. In *Proc. First ACM Conference on Computer and Communication Security (CCS 1993)*, pages 9–16, 1993.

[7] K. Bowers, A. Juels, and A. Oprea. Proofs of retrievability: Theory and implementation. In *Proc. ACM Cloud Computing Security Workshop (CCSW 2009)*, 2009.

[8] G. Cattaneo, L. Catuogno, A. D. Sorbo, and P. Persiano. The design and implementation of a transparent cryptographic file system for Unix. pages 199–212, 2001.

[9] Y. Chen and R. Sion. To cloud or not to cloud? musings on costs and viability. In *ACM Symposium on Cloud Computing (SOCC)*, 2011.

[10] Y. Dodis, S. Vadhan, and D. Wichs. Proofs of retrievability via hardness amplification. In *Proc. 6th IACR TCC*, volume 5444 of *LNCS*, pages 109–127, 2009.

[11] C. Erway, A. Kupcu, C. Papamanthou, and R. Tamassia. Dynamic provable data possession. In *Proc. ACM Conference on Computer and Communications Security (CCS 2009)*, 2009.

[12] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten. Sporc: Group collaboration using untrusted cloud resources. In *Proc. OSDI*, 2010.

[13] K. Fu. Group sharing and random access in cryptographic storage file systems. Master's thesis, Massachusetts Institute of Technology, 1999.

[14] K. Fu, F. Kaashoek, and D. Mazieres. Fast and secure distributed read-only file system. *ACM Transactions on Computer Systems*, 20:1–24, 2002.

[15] R. Geambasu, J. P. John, S. D. Gribble, T. Kohno, and H. M. Levy. Keypad: An auditing file system for theft-prone devices. In *Proc. European Conference on Computer Systems (EuroSys)*, 2011.

[16] E. Goh, H. Shacham, N. Modadugu, and D. Boneh. SiRiUS: Securing remote untrusted storage. In *Proc. Network and Distributed Systems Security Symposium (NDSS 2003)*, pages 131–145, 2003.

[17] M. T. Goodrich, C. Papamanthou, R. Tamassia, and N. Triandopoulos. Athos: Efficient authentication of outsourced file systems. In *Proc. Information Security Conference 2008*, 2008.

[18] A. Juels and B. Kaliski. PORs: Proofs of retrievability for large files. In *Proc. ACM Conference on Computer and Communications Security (CCS 2007)*, pages 584–597, 2007.

[19] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus: Scalable secure file sharing on untrusted storage. In *Proc. 2nd USENIX Conference on File and Storage Technologies (FAST)*, 2003.

[20] S. Kamara, C. Papamanthou, and T. Roeder. CS2: A searchable cryptographic cloud storage system. Technical Report MSR-TR-2011-58, Microsoft, 2011.

[21] J. Li, M. Krohn, D. Mazieres, and D. Shasha. Secure untrusted data repository. In *Proc. 6th Symposium on Operating System Design and Implementation (OSDI)*, pages 121–136. Usenix, 2004.

[22] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: Cloud storage with minimal trust. In *Proc. OSDI*, 2010.

[23] E. Miller, D. Long, W. Freeman, and B. Reed. Strong security for distributed file systems. In *Proc. 1st USENIX Conference on File and Storage Technologies (FAST)*, 2002.

[24] A. Oprea and M. K. Reiter. Integrity checking in cryptographic file systems with constant trusted storage. In *Proc. Usenix Security Symposium 2007*, 2007.

[25] R. Pletka and C. Cachin. Cryptographic security for a high-performance distributed file system. In *Proc. 24th IEEE Conf. on Mass Storage Systems and Technologies (MSST 2007)*, 2007.

[26] R. A. Popa, J. Lorch, D. Molnar, H. J. Wang, and L. Zhuang. Enabling security in cloud storage SLAs with CloudProof. In *Proc. 2011 USENIX Annual Technical Conference (USENIX)*, 2011.

[27] H. Shacham and B. Waters. Compact proofs of retrievability. In *Proc. ASIACRYPT*, volume 5350 of *LNCS*, pages 90–107, 2008.

[28] A. Shraer, C. Cachin, A. Cidon, I. Keidar, Y. Michalevsky, and D. Shaket. Venus: Verification for untrusted cloud storage. In *Proc. Workshop on Cloud Computing Security*, 2010.

[29] C. A. Stein, J. H. Howard, and M. Selzer. Unifying file system protection. In *Proc. USENIX Annual Technical Conference*, 2001.

[30] Q. Wang, C. Wang, J. Li, K. Ren, and W. Lou. Enabling public verifiability and data dynamics for storage security in cloud computing. In *Proc. 14th European Symposium on Research in Computer Security (ESORICS 2009)*, 2009.

[31] Q. Zheng and S. Xu. Fair and dynamic proofs of retrievability. In *Proc. 1st ACMM Conference on Data and Application Security and Privacy (CODASPY)*, 2011.