UPSS: A Global, Least-Privileged Storage System with Stronger Security and Better Performance

Arastoo Bozorgi[a], Mahya Soleimani Jadidi and Jonathan Anderson[b]
Department of Electrical and Computer Engineering, Memorial University, St. John's, NL, Canada

Keywords: Cryptographic Filesystem, Distributed Filesystem, Private Sharing, Redaction, Private Version Control.

Abstract: Strong confidentiality, integrity, user control, reliability and performance are critical requirements in privacy-sensitive applications. Such applications would benefit from a data storage and sharing infrastructure that provides these properties even in decentralized topologies with untrusted storage backends, but users today are forced to choose between systemic security properties and system reliability or performance. As an alternative to this *status quo* we present *UPSS: the user-centric private sharing system*, a cryptographic storage system that can be used as a conventional filesystem or as the foundation for security-sensitive applications such as redaction with integrity and private revision control. We demonstrate that both the security and performance properties of UPSS exceed that of existing cryptographic filesystems and that its performance is comparable to mature conventional filesystems — in some cases even superior. Whether used directly via its Rust API or as a conventional filesystem, UPSS provides strong security and practical performance on untrusted storage.

1 INTRODUCTION

Across a broad spectrum of domains, there is an acute need for private storage with flexible, granular sharing. Environments as diverse as social networking, electronic health records and surveillance data management require both strong cryptographic protection and fine-grained sharing across security boundaries without granting overly-broad access. Existing systems provide coarse security guarantees or strong performance properties, but rarely both. Fine-grained, flexible, high-performance sharing of default-private data is still a challenging problem.

What is needed is a mechanism for *least-privileged* storage that facilitates *simple discretionary sharing* of arbitrary subsets of data, providing strong confidentiality and integrity properties on commodity cloud services from untrusted providers. In the previous years, some cryptographic filesystems have been developed that store user data on untrusted storage providers. However, they cannot provide strong security properties nor flexible data sharing. For example, EncFS (Team, 2018) and CryFS (Messmer et al., 2017) are cryptographic filesystems that leave metadata unprotected, or in the latter one, everything is encrypted with one key. TahoeFS (Wilcox-O'Hearn and Warner, 2008) is another cryptographic filesystem with strong security properties, but its design does not allow flexible and fine-grained data sharing.

In this paper, we have built UPSS: the user-centric private sharing system, which is a "global first" cryptographic filesystem with no assumptions of trustworthiness for storage infrastructure or even on common definitions of user identities. Relying on key concepts from capability systems (Dennis and Van Horn, 1966), distributed systems, log-structured filesystems and revision control, we have developed a new approach to filesystems that offers novel features while being usable in ways that are compatible with existing applications.

UPSS makes several key contributions to the field of privacy-preserving filesystems. First, unlike cryptographic filesystems that entangle user and group identifications and device specification with access controls, UPSS stores all data as encrypted blocks on untrusted block stores including local, network, or cloud block stores, without any mapping between the blocks or blocks to block owners. Granular access controls are then defined by higher level applications according to application semantics. Traditional access control modalities such as Unix permissions can be implemented by systems using UPSS, as in the case of our FUSE-based interface, but they are not

[a] https://orcid.org/0000-0002-7059-9501
[b] https://orcid.org/0000-0002-7352-6463

Bozorgi, A., Jadidi, M. and Anderson, J.
UPSS: A Global, Least-Privileged Storage System with Stronger Security and Better Performance.
DOI: 10.5220/0012306600003648
Paper published under CC license (CC BY-NC-ND 4.0)
In *Proceedings of the 10th International Conference on Information Systems Security and Privacy (ICISSP 2024)*, pages 660-671
ISBN: 978-989-758-683-5; ISSN: 2184-4356
Proceedings Copyright © 2024 by SCITEPRESS – Science and Technology Publications, Lda.

encoded in the shared cryptographic filesystem itself. This decoupling allows the filesystem to be global-first and local-second.

Second, all UPSS blocks can be accessed by cryptographic capabilities (Dennis and Van Horn, 1966) called block pointers consist of block names and their decryption keys that reduces the burden of key management and simplifies naming; a block pointer is enough to fetch, decrypt and read a block, with no central key management required. Block pointers enable flexible data sharing at the block level among mutually-distrustful users. They also enable per-block encryption rather than per-file or per-filesystem encryption, which provides a stronger security model.

Third, UPSS enables aggressive and safe caching by defining a multi-layer caching block store consists of other block stores that guarantee data consistency between all block stores. The caching block store prioritizes applying the operations on faster block stores on the cache hierarchy and processes the operations on slower block stores in the background. Therefore, the caching block store becomes available immediately despite the number of layers in the hierarchy or the slowness of higher-level block stores. This provides performance that exceeds cryptographic filesystems by factors of 1.5–40×.

UPSS' system model and design is described in Section 2. Performance is evaluated in Section 4 via three case studies comparing UPSS to existing filesystems: local filesystems (Section 4.2), network filesystems (Section 4.3) and global filesystems (Section 4.4). The related works are discussed in Section 5 and we conclude our paper in Section 6.

2 UPSS SYSTEM MODEL AND DESIGN

UPSS is a *cloud-first* private storage and sharing system. Rather than a local cryptographic filesystem that projects POSIX assumptions (e.g., file ownership, user identification and trusted devices) into the cloud, UPSS starts with the assumptions of untrusted storage and user-directed sharing via cryptographic capabilities (Dennis and Van Horn, 1966). UPSS can be exposed via FUSE (fus, 2019) as a conventional POSIX filesystem, allowing performance comparison to existing local filesystems, network filesystems and global filesystems, but its most exciting capabilities are exposed directly through a Rust API.

In this section, we review key elements of the UPSS design, which was seeded in (Bozorgi et al., 2019; Bozorgi, 2020) as prototype ideas, and describe new design elements that enable practical per-

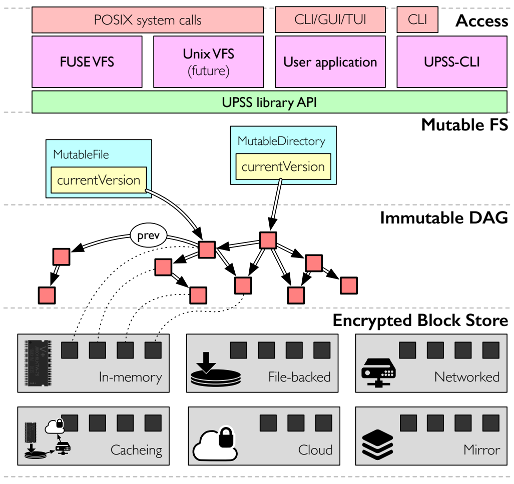


Figure 1: The layered structure of UPSS.

formance that had previously been envisioned as future work. These elements are visible across four layers shown in Figure 1: untrusted storage (Section 2.1), an immutable copy-on-write DAG of blocks (Section 2.2), mutable filesystem objects (Section 2.3) and two user-visible filesystem interfaces (Section 2.4).

2.1 Untrusted Storage

Like all filesystems, UPSS ultimately stores data in fixed-size blocks on persistent media. Block sizes are all multiples of common physical sector sizes and are set by the backing store rather than the client. UPSS uses a default block size of 4 kiB that can be overridden on a per-store basis. Unlike other filesystems, all UPSS blocks are encrypted in transit and at rest: plaintext blocks is only held in memory and never stored to persistent media. Rather than using per-file or per-filesystem encryption keys, each block is encrypted with a key k_B derived from its plaintext and named by a cryptographic hash n_B of its ciphertext. The 2-tuple (n_B, k_B) constructs a *block pointer* as given in eq. (1).

$$\begin{aligned} k_B &= h(B) \\ n_B &= h\left(E_{k_B}\{B\}\right) \end{aligned} \tag{1}$$

In this equation, B represents the plaintext contents of a block, which contains user content and random padding to fill out the fixed-size block, h is a cryptographic hash function and E is a symmetric-key encryption algorithm. A block pointer is a cryptographic capability (Dennis and Van Horn, 1966) to fetch, decrypt and read a block's contents, though not to modify it, as blocks are immutable. Changing a single byte in the block would change a block's encryption key k_B, which would change the encrypted version of the block, which would change its name

n_B. As a matter of practical implementation, serialized block pointers also contain metadata about their hashing and encryption algorithms (typically SHA3 (Dworkin, 2015) and AES-128 (Dworkin et al., 2001)).

Deriving a symmetric encryption key from a block's contents is an example of *convergent encryption* (Douceur et al., 2002; Li et al., 2013; Agarwala et al., 2017). Convergent encryption is a symmetric-key encryption technique in which identical ciphertexts are produced from identical plaintexts. This technique affords two benefits: a reduced burden of key management and the possibility of block (rather than file) level data deduplication (Satyanarayanan et al., 1990; Douceur et al., 2002). Deduplication is an important feature for global-scale information sharing systems in which many users may share the same content with others. By deduplication, only two extra 4 KiB meta blocks are required to ingest a 1 GB file to UPSS for the second time with the same content. However, convergent encrypion and deduplication bring with themselves some risks that are discussed in Section 3.

2.1.1 Block Stores

A narrow API including `read`, `write`, `block_size` and `is_persistent` methods is implemented by several types of block stores shown in Figure 1: in-memory (non-persistent), file-backed, networked, cloud via Amazon S3 (Amazon Web Services, Inc., 2020) or Azure blob storage (Microsoft, Inc., 2023), caching and mirror. The caching and mirror block stores consist of multiple stores, that accomplish different tasks. The former enables caching (Section 2.1.2) and the latter handles replication (Section 2.1.3), both at the block level.

When an encrypted block is stored in a block store, the block store responds with a block name n_B derived using that store's preferred cryptographic hash algorithm. A block's name can be used to retrieve the block in the future without any further authorization — it is a cryptographic *capability* (Dennis and Van Horn, 1966). This approach allows block stores to be oblivious to user identities and content ownership. Instead, it is a *content-addressed store*. The operator of a block store cannot view plaintext content or even directly view metadata such as file sizes or directory-file relationships. Inference of these relationships is discussed in Section 3, which also describes the stronger privacy and security properties that UPSS provides relative to other cryptographic and conventional filesystems.

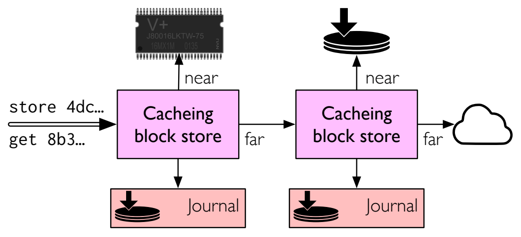


Figure 2: A caching hierarchy of untrusted block stores.

2.1.2 Caching

The caching block store consists of two other near and far stores and a journaling mechanism. A near store can be an in-memory block store that processes the operations faster than a far store that can be a file-backed, networked, cloud, mirror, or another caching block store. Note that both near and far stores can be any block stores. By having the caching block store, UPSS enables building a cache hierarchy as shown in Figure 2. For storing an encrypted block, the caching block store stores the block to the near store and journals it to an on-disk file. The journaled blocks will be processed in the background to be stored to the far store. For reading, the caching block store tries to read the block from the near store and if it does not exist (e.g., the near store is an in-memory store which has been cleared), the block is read from the far store. The confidentiality and immutability of blocks in a block store enable aggressive yet safe caching, even with remote storage on untrusted systems. This makes UPSS achieve better performance results as discussed in Section 4.

A challenging problem with caching data in any information system is handling inconsistencies; a block's content can be updated in a cache while not in other locations. However, UPSS avoids any cache inconsistencies and reduces this problem to a version control problem by the immutable nature and cryptographic naming of the stored blocks. A block may be present within or absent from a store, but it cannot be inconsistent between two stores: even the smallest inconsistency in content would cause the blocks to have different cryptographic names.

2.1.3 Data Availability via Replication

The mirror block store handles data replication across multiple block stores. For storing an encrypted block, the mirror block store replicates the block to all block stores in parallel and returns the block name upon successful replication. For reading, the mirror block store queries the block by its name from all block stores in parallel and returns the block from a block store that responds faster and ignores other block store responses.

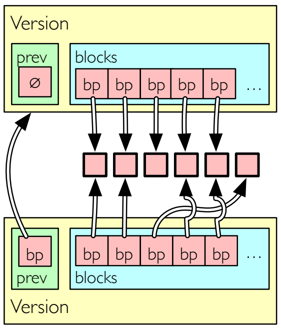


Figure 3: Two sequential `Version` objects that reference two common blocks and one diverging block.

2.2 Immutable DAGs

UPSS uses directed acyclic graphs (DAGs) of immutable blocks to represent files and directories. Relationships among blocks are specified by `Version` objects that describe arbitrary-length collection of immutable blocks, each accessible by their block pointers. As shown in Figure 3, multiple `Version` objects can reference underlying immutable blocks, facilitating the copy-on-write modification of files and directories described in Section 2.3. `Version` objects are themselves stored in UPSS blocks, allowing them to be named according to *their* cryptographic hashes. For files smaller than 100 kiB, a `Version` fits in a single UPSS block. A `Version` may contain a block pointer to a previous `Version`, and thus a `Version` can be used as a Merkle tree (Merkle, 1979) (more precisely, a Merkle DAG) that represents an arbitrary number of versions of an arbitrary quantity of immutable content.

This use of Merkle DAGs reduces the problem of data consistency to that of version control: it is possible for two files to contain different blocks, but not two variations of the "same" block. It is left to the user of these immutable DAGs to provide mutable filesystem objects using copy-on-write (CoW) semantics and to ensure that new blocks are appropriately pushed to backend block stores.

2.3 Mutable Filesystem Objects

Conventional mutable filesystem objects (files and directories) are provided by UPSS by mapping arrays of bytes into mutable `Blob` objects. These objects maintain copy-on-write (CoW) references to underlying blocks and versions. Non-traditional objects such as structured binary key-value data structures are also possible using multiple blobs and versions.

A `Blob` manages an array of bytes via copy-on-write block references, starting from an empty sequence of blocks and permitting operations such as truncation, appending and random-access reading and writing. A `Blob` accumulates edits against an immutable `Version` in an "edit session" (Asklund et al., 1999) until a file or directory is persisted into a new immutable `Version`. This allows UPSS to accumulate write operations and batch them into aggregate CoW operations.

Files and directories are both backed by `Blob` objects, and both can be explicitly persisted to backing storage via API calls `persist()` and `name()`, which persists an object and returns its block pointer. A file version can be named by a block pointer to its `Version` object which represents the file's content and, optionally, history. A directory is represented as a sequence of directory entries, each of which maps a unique, user-meaningful name to a filesystem object (file or directory). A directory can be persisted by serializing its entries into a `Version` that is named by a block pointer. Thus, directories are also Merkle DAGs that reference the lower-level Merkle DAGs of other file and directory objects.

Cryptographic hashes are computed and blocks encrypted when files and directories are persisted, making persisting one of the most expensive operations in UPSS. Tracking chains of `Version` objects in addition to content makes both the time and storage requirements for persistence superlinear. It is, therefore, only done when requested via the API or, in the case of UPSS-FUSE, every 5 s. Based on our measurement results, the total space b_t required in a block store to store b bytes of content follows Equation (2).

$$b_t = (1.09 + 0.001613\,)\,b \qquad (2)$$

2.3.1 Mutation and Versioning

Naming all filesystem objects by block pointers to `Version` structures introduce new challenges to handling modifications. Whenever a file or directory is modified in a directory hierarchy, a new block pointer is generated that should be updated in the object's parent entries , and this update should be applied up to the root directory. In order to handle updates efficiently, every file and directory object keeps an `Updater` object, which is a reference to its parent in-memory object. Upon modification and persisting, an object notifies its `Updater` about its new block pointer and the parent object is modified to reflect the child's new version. Similar requirements for updating of parents exist in other CoW filesystems such as ZFS (Bonwick et al., 2003), but the case of a global CoW filesystem such as UPSS is more challenging than that of local filesystems. In a local CoW filesystem, it is possible for the filesystem implementation to be aware of all concurrent uses of a parent directory, including by multiple users. In a global filesystem, however, not all uses of a parent directory are visible to a local host. UPSS therefore, treats every update to a filesystem subtree as a potential versioning operation, allowing new directory snapshots to be created and shared as described in section 2.3.2; versions can be integrated at the level of filesystem interfaces as described in section 2.4.

2.3.2 Snapshot and Sharing

As a copy-on-write filesystem, UPSS provides cheap snapshots of previous versions. UPSS creates snapshots whenever requested via `sync(2)`, `fsync(2)` or the UNIX `sync(1)` command, or in case of FUSE (filesystem in user space) wrapper, by querying a directory's cryptographic name with POSIX extended attributes, or in case of UPSS API, by calling `persist()` or `name()` methods. Also, extended attributes can be queried to retrieve the cryptographic hash or serialized block pointer of any UPSS file or directory. Exposing serialized block pointers to users facilitates sharing of file and directory snapshots from user to user, including sharing from UPSS FUSE wrapper snapshots to users employing the UPSS CLI. Also, this allows users to check integrity guarantees over file and directory Merkle DAGs, facilitating blockchain-like applications.

As a user-empowering *sharing* system, these snapshots can be quickly shared with other users for read-only access: user *a* need only share the block pointer to a file or directory with user *b*, and user *b* will be able to retrieve the content from a block store and decrypt it. Since block pointers correspond to immutable blocks, user *b* cannot modify the shared block. Upon modification, a new block is generated with a new block pointer and user *a* still has access to the unmodified shared block.

2.4 File Access Interfaces

Users can access an UPSS filesystem via a variety of interfaces, including a Rust API which can be compiled to WebAssembly, a command-line interface and a FUSE (Filesystems in Userspace) interface. Unlike many filesystems, any UPSS directory can be treated as the root directory of a filesystem. Within a directory hierarchy, a user may persist any subdirectory to retrieve a block pointer to an immutable version of it. That version may then be used as the basis for further filesystem operations including mounting, mutating and further sharing. When new versions of files and directories are generated, parent directories are updated until a new root directory version is created. Storing the block pointer of that new root directory is

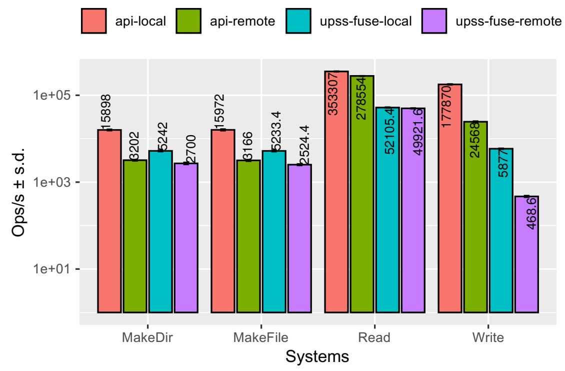


Figure 4: UPSS performance when accessed via its API and via UPSS-FUSE connected to a local or remote block store. The average number of operations per 60 seconds is reported for five runs; error bars show standard deviation.

the responsibility of an UPSS client (API, CLI, FUSE or future native VFS implementation). The UPSS CLI and FUSE clients both store this information in a local passphrase-protected file as per PKCS #5 Version 2.0 (Kaliski, 2000) for interoperability.

Direct API invocation provides clear performance benefits when compared to FUSE-based wrapping. As shown in Figure 4, directly invoking the UPSS API yields higher performance than using a FUSE wrapper with the same storage backend. For two of the four microbenchmarks described in Section 4.1, the cost of the FUSE wrapper exceeds that of the cost of communicating with a remote blockstore via direct UPSS API.

3 SECURITY MODEL

The judicious use of full-filesystem, per-block convergent encryption allows UPSS to employ untrusted storage backends that can scale to the largest of workloads without revealing user data or metadata. Its user-agnostic approach allows it to be employed within applications and in a range of uses from a local filesystem to a global sharing system.

Other cryptographic filesystems take a variety of approaches to encryption. EncFS (Team, 2018) and NCryptFS (Wright et al., 2003) employ encryption for file content but not the filesystem itself, e.g., directory structure. CryFS (Messmer et al., 2017) protects the filesystem with a single encryption key. This implies strong filesystem boundaries and precludes safe subset sharing: the unit of possible sharing in the system is a filesystem, not a file. It also increases the value of one specific encryption key, making it a more attractive target for attackers. UPSS, in contrast, encrypts data using per-block keys derived from block content, removing the need for separate storage of keys and re-

SCIENCE AND TECHNOLOGY PUBLICATIONS

ducing the value of any single encryption key. Since keys are not re-used, they are not secret from authorized users.

Commonly cloud-based storage allows providers to examine users' plaintext; backend block store providers in UPSS can only see a sea of encrypted blocks, as encryption is performed on the client side. This approach ensures that metadata such as file sizes and directory structures are not revealed explicitly to storage providers. Providers might perform traffic analysis to infer relationships among various blocks, but only at significant computational cost. This threat could be addressed by the use of oblivious transfer techniques such as ORAM, but other large-scale distributed systems that support such techniques have disabled them due to cost (uta, 2020b).

Convergent encryption, first used in Farsite (Adya et al., 2002), can also introduce risks that are not present in traditional cryptosystems. Convergent encryption is a deterministic encryption model, but the traditional objective of indistinguishability under chosen plaintext attack (IND-CPA) forbids determinism in encryption; it can therefore reveal whether a given plaintext has previously been encrypted and stored in the content addressable storage if an attacker encrypts a plaintext, presents it to a block store and uses timing or other response information to determine whether that block has previously been stored. Worse still, naïve systems could allow an attacker to guess variations on a known format (`user=1000`, `user=1001`, etc.) to test whether any such variations have previously been stored.

UPSS addresses these concerns by appending random padding to plaintext blocks to bring them to the fixed block size. Small blocks of user data, those that most need protection from guessing attacks, are padded with high-entropy random bits. Full blocks of user data, such as content from shared media files, do not require padding, allowing such files to enjoy deduplication. Confirmation of a large existing block is still possible, but not via guessing attacks due to the block sizes involved, and no user can be associated with the content. By default, any block of plaintext data that is smaller than the fixed block size will have random padding appended. A typical AWS credential file with a known access key will have 72 B of data that could be known to the attacker, 40 B of Base64-encoded secret key that the attacker would like to guess and, in a 4 kiB block, 3,984 B of random padding.

UPSS does not explicitly represent users or user identities. This allows applications or clients to bring their own user model to the filesystem and avoid the limitations of system-local users, as in systems that project local filesystems to a multi-system context, e.g., multi-user EncFS (Leibenger et al., 2016)). Multiple users on separate systems can share a backend block store without interference, but the common block store permits efficient sharing among systems and users.

4 PERFORMANCE EVALUATION

In this section, we demonstrate the practicality of UPSS as a local filesystem (Section 4.2), a network filesystem (Section 4.3) and a global filesystem (Section 4.4). Although UPSS achieves its best performance when accessed via API rather than FUSE, employing the FUSE interface allows us to directly compare its performance with the performance of extant systems. These performance comparisons are completed using a suite of microbenchmarks and one FileBench-inspired macrobenchmark.

4.1 Benchmark Description

We have compared the performance of UPSS with other systems using both custom microbenchmarks and a Filebench-inspired benchmark. All benchmarks were executed on a 4-core, 8-thread 3.6 GHz Intel Core-i7-4790 processor with 24 GiB of RAM and 1 TB of ATA 7200 RPM magnetic disk, running Ubuntu Linux 4.15.0-72-generic. Remote block stores, where employed, used machines with different configurations as described in Section 4.3.

For microbenchmarking, we evaluated the cost of creating files and directories and reading and writing from/into on-disk local and remote block stores. For evaluating file and directory creation, we generated a user-defined number of files and directories, added them to an ephemeral root directory and persisted the results into file-backed block stores. To evaluate read and write operations, we generated 1000 files filled with random data of size 4 KiB, the natural block size of our underlying storage, select a file randomly and performed sequential read and write operations on it.

We also implemented a macrobenchmark that simulates more complex behaviour. In this benchmark, we selected a file randomly from a set of files and performed 10 consecutive read and write operations with different I/O sizes: 4 KiB, 256 KiB, 512 KiB and 1 MiB. The building blocks of this benchmark were inspired by the Filebench framework (fil, 2016), but Filebench itself could not produce the fine-grained timing information used to produce the figures shown in Sections 4.2.2, 4.3 and 4.4.1.

4.2 UPSS as a Local Filesystem

Direct usage of the UPSS API requires program modification — and, today, the use of a specific programming language. In order to expose the benefits of UPSS to a wider range of software, we have implemented a *filesystem in userspace (FUSE)* (fus, 2019) wrapper that exposes UPSS objects to other applications via a hook into the Unix VFS layer. The challenge here is picturing UPSS's global view of encrypted blocks to a local view of files and directories that can be accessed via FUSE inode numbers. To tackle this, UPSS-FUSE uses a mapping from FUSE inode numbers to in-memory UPSS objects to service VFS requests. This allows conventional applications to access an UPSS directory mounted as a Unix directory with POSIX semantics, though there is one unsupportable feature: hard links. Hard links are defined within the context of a single filesystem, but UPSS is designed to allow any directory to be shared as a root directory of a filesystem. Owing to this design choice, it is impossible to provide typical hard link semantics and, e.g., update all parents of a modified file so that they can perform their own copy-on-write updates (see Section 2.3). Therefore, we do not provide support for hard links — a common design choice in network file systems such as NFS.

The UPSS-FUSE wrapper exposes an ephemeral plaintext view of an UPSS's directory underneath a Unix mount point, allowing conventional file and directory access, while keeping all data and metadata encrypted at rest in a local or remote block store (see Section 2.1). Unlike existing cryptographic filesystems such as NCryptFS (Wright et al., 2003) and EncFS (Leibenger et al., 2016; Team, 2018), no plaintext directory structure is left behind in the mount point after the filesystem has been unmounted.

4.2.1 Consistency

In order to provide data consistency, UPSS-FUSE requests that UPSS persist a "dirty" — i.e., modified — root directory every five seconds, or after a tunable number of dirty objects require persisting. As described in Section 2.3, persisting a `Directory` object causes its versioned children to be recursively persisted (if dirty), after which the cryptographic block pointer for the new root directory version can be stored in the UPSS-FUSE metadata file. This root block pointer is the only metadata that UPSS-FUSE needs to mount the filesystem again. The block pointer size is 80 bytes as the defualt hashing and encryption algorithm in UPSS are SHA3-512 and AES-128 respectively. As in other copy-on-write filesystems, the cost of persisting an entire filesystem depends on the amount of "dirty" content in the filesystem. The trade-off between the demand for frequent data synchronization and the requirement for more frequent — though smaller — persistence operations is illustrated in Figure 5.

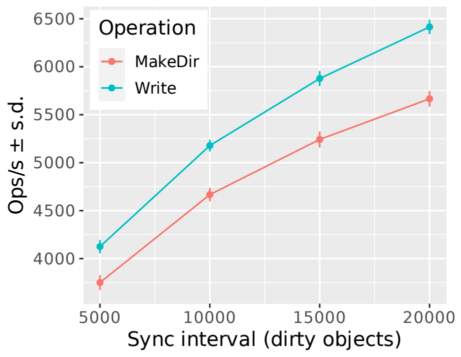


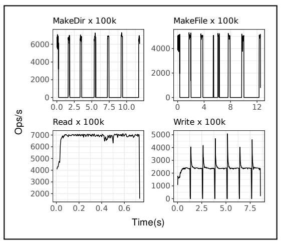
Figure 5: Performance of 4 kiB operations vs sync frequency (in number of dirty objects) over five runs.

4.2.2 Performance Comparisons

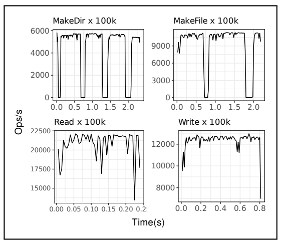
To illustrate the performance of UPSS when used as a conventional local filesystem, we compared UPSS-FUSE with the cryptographic filesystems CryFS (Messmer et al., 2017) and EncFS (Leibenger et al., 2016; Team, 2018), also based on FUSE, as well as the mature, heavily-optimized ZFS (Bonwick et al., 2003). ZFS is not a cryptographic filesystem designed for fine-grained confidentiality, but it does share some design elements with UPSS: it is a log-structured filesystem with copy-on-write updates that uses cryptographic hashes to name blocks. In contrast to UPSS-FUSE, ZFS has been extensively optimized over the past two decades to become a high-performance, widely-deployed filesystem.

We mounted each of these four filesystems on different paths in the Linux host referenced in Section 4.1 and ran four microbenchmarks to test their speed in creating empty directories (**MakeDir**), creating empty files (**MakeFile**), reading randomly select files sequentially including 4 KiB of data (**ReadFile**) and writing random data to files (**WriteFile**).

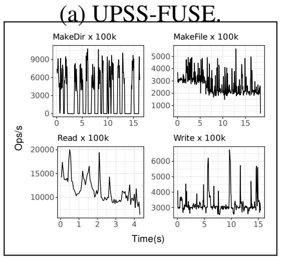
Each of these four operations was run 100k and the behaviour of the filesystems were reported in Figure 6. In these plots, the *x*-axis represents the time needed to complete all 100k operations. UPSS outperforms EncFS and CryFS for all operations, with performance especially exceeding these existing systems in the critical read and write benchmarks. As might be expected, ZFS significantly outperforms UPSS in all benchmarks, with read performance 3× and write performance 10.9× faster than UPSS-FUSE. In UPSS-FUSE, creating files and di-



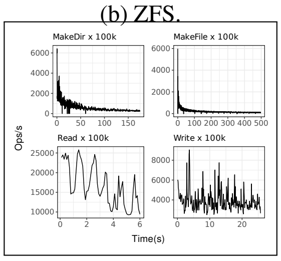
(a) UPSS-FUSE.



(b) ZFS.



(c) EncFS.



(d) CryFS.

Figure 6: Performance comparison of UPSS-FUSE with CryFS, EncFS and ZFS. Benchmarks were run for 100 kops.

rectories have the same cost, as they are both backed by empty collections of blocks. We also note that UPSS-FUSE performs $1.47-41.6\times$ more operations per second in various benchmarks than CryFS and EncFS while also providing stronger security properties (see Section 5). This is due to our design choice that the requests are served from the mapped in-memory objects that are persisted periodically, if dirty. Therefore, expensive persist operations can be done quickly: with little accumulation of dirty state, less synchronous persistence work is required.

These plots show the bursty nature of real filesystems, and in the case of CryFS, they reveal performance that scales poorly as the number of requested operations increases. Much of the bursty nature of these plots derives from how each filesystem synchronizes data to disk. For example, by default, ZFS synchronizes data every 5 s or when 64 MiB of data has accumulated to sync, whichever comes first. Similarly, to provide a fair comparison, UPSS-FUSE is configured to synchronize after 5 s or 15,000 writes (close to 64 MiB of data when using 4 KiB blocks). These periodic synchronizations cause performance to drop, even on dedicated computers with quiescent networks and limited process trees.

4.2.3 Macro-Benchmark

We ran the macrobenchmark described in Section 4.1 on UPSS-FUSE, CryFS, EncFs and ZFS, to evaluate UPSS-FUSE in a simulation in which consecutive read and write operations with different I/O sizes are performed on different files. The results are reported in Figure 7. As in our microbenchmarks,

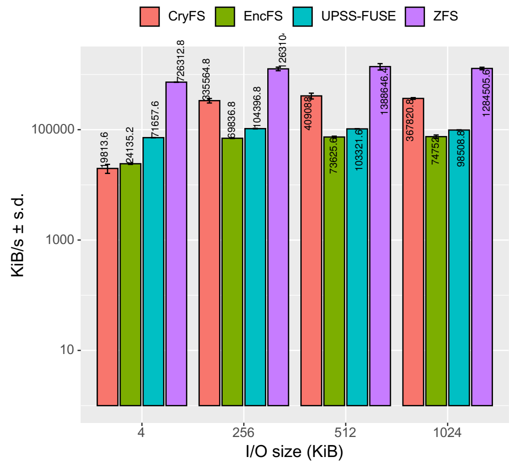
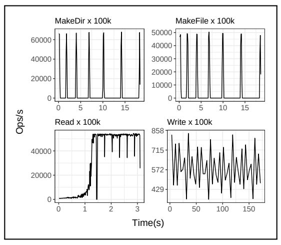


Figure 7: Performance of CryFS, EncFS, UPSS-FUSE and ZFS for the macrobenchmark. The numbers are the average of KiB of I/O per second for five runs, each 60 seconds along with their standard deviations.

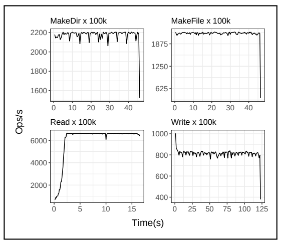
ZFS outperforms the other filesystems for different I/O sizes. UPSS-FUSE achieved better results than CryFS and EncFS for the 4 KiB case. However, as the I/O size increases, CryFS outperforms UPSS-FUSE. The larger the I/O operation, the more fixed-sized blocks are generated by UPSS-FUSE, each of which needs to be encrypted with a different key and persisted. In CryFS, however, all the fixed-size blocks related to a file are encrypted with the same symmetric key. This causes better performance for larger files, but at the same time makes CryFS inapplicable to the partial sharing and redaction use cases that can be supported by UPSS. UPSS has been designed for small block sizes (typically 4 kiB), as decades of research has shown that filesystems mostly contain small files (Rosenblum and Ousterhout, 1992; Baker et al., 1991; Lazowska et al., 1986).

4.3 UPSS as a Network Filesystem

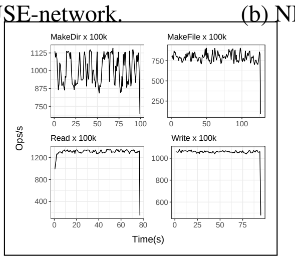
Although UPSS can be used as a local filesystem, it is primarily designed as a system for sharing data across networks with untrusted storage providers. UPSS' use of encrypted block stores, in which confidentiality and integrity of these blocks' content are assured by clients and not servers, allows us to build a block store in which a centralized server exploits high-quality network links to transfer large numbers of encrypted blocks — the data plane — regardless of what block pointers are shared between users — the control plane. This design is amenable to multi-layer caching, as described in Section 2.1. Thus, we have compared the performance of UPSS-FUSE when connected to a remote block store to that of



(a) UPSS-FUSE-network.



(b) NFS.



(c) SSHFS.

Figure 8: Performance comparison of UPSS-FUSE-network, NFS and SSHFS. Benchmarks were run for 100 kops.

SSHFS (Team, 2020) and the venerable NFS (Shepler et al., 2003).

4.3.1 Performance Comparison

As in Section 4.2.2, we evaluated the performance of UPSS by mounting an UPSS-FUSE filesystem in a Unix mount point and comparing it to other filesystems using four microbenchmarks. In this section, however, we connected our UPSS-FUSE filesystem to a remote block store and compared our performance results against two other remote filesystems: the FUSE-based SSHFS (Team, 2020) and the venerable NFS (Shepler et al., 2003). Similar to Section 4.2.2, one comparison filesystem is primarily designed for security and the other has higher performance after a long history of performance optimization.

The remote block store server was run on a 4-core, 2.2 GHz Xeon E5-2407 processor with 16 GiB of RAM and 1 TB of magnetic disk, running FreeBSD 12.1-RELEASE. The client machine, that runs UPSS-FUSE, is a 4-core, 3.5 GHz Xeon E3-1240 v5 processor with 32 GiB of RAM and 1 TB of magnetic disk, running Ubuntu Linux 16.04. The client and server were connected via a dedicated gigabit switch. Figure 8 shows the behaviour of the benchmarked filesystems when executing 100k **MakeDir**, **MakeFile**, **Read** and **Write** operations.

UPSS outperforms SSHFS and even NFS for **MakeDir**, **MakeFile** and **Read** operations and for **Write**, it achieves comparable results. For the **Read** benchmark, UPSS-FUSE has a slow start as encrypted blocks are read from the remote block store and are loaded into memory. After files are loaded into memory, other read operations are served from the in-memory objects. This causes UPSS-FUSE to be about 5× faster than NFS in the **Read** benchmark, validating UPSS-FUSE's approach to encrypted block storage and the safe and aggressive caching it enables.

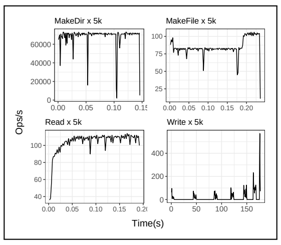
4.4 UPSS as a Global Filesystem

In addition to local and network filesystem, UPSS-FUSE can also be connected to untrusted cloud storage providers. To do so, we have implemented an UPSS block store backed in the Amazon S3 service (Amazon Web Services, Inc., 2020) and compared its performance with S3FS (Gaul et al., 2020), Perkeep (Lindner and Norris, 2018) and UtahFS (uta, 2020a; uta, 2020b).

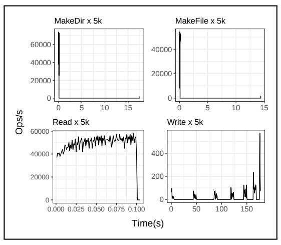
4.4.1 Performance Comparison

We mounted UPSS-FUSE backed with the Amazon block store (with and without local caching), S3FS, Perkeep and UtahFS in different Unix mount points and compared them using our four microbenchmarks. S3FS allows Linux and macOS to mount an Amazon S3 bucket via FUSE without any security properties. Perkeep, formerly called Camlistore, is a FUSE-based cryptographic filesystem that can be backed by memory, local or cloud storage. UtahFS which is in its initial stage of development, stores encrypted data on untrusted cloud storage. We mounted UtahFS without Path ORAM that hides the access patterns, as it degrades the performance (uta, 2020b). Having the Path ORAM enabled, the **Write** benchmark runs 18.59× slower. We configured Perkeep and UtahFS to use an Amazon S3 account for our evaluation.

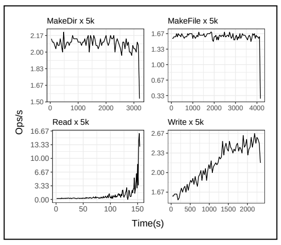
We ran the benchmarks discussed in Section 4.2.2 with 5k **MakeDir**, **MakeFile**, **Read** and **Write** operations and the behaviours of UPSS-FUSE-network, S3FS, Perkeep and UtahFS during time are reported in Figure 9. In all of these cases, Amazon S3's response time is the bottleneck. To have a fair comparison, we ran the benchmarks for UPSS-FUSE with and without caching. With caching enabled, we write the encrypted blocks in a caching block store and journal the blocks to an on-disk file, then we write to Amazon S3 bucket by processing the journal using a background thread. This makes a large difference in the number of operations that can be done by UPSS-FUSE as a global filesystem in comparison with S3FS, Perkeep and UtahFS (Figure 9a). In Figure 9b, we disabled caching and persisted the content just before the benchmark script is finished so



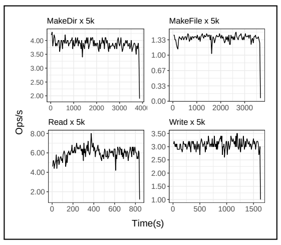
(a) UPSS-FUSE-global.



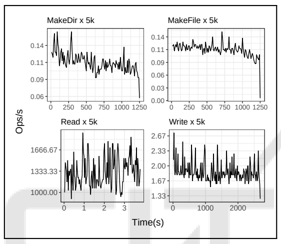
(b) UPSS-FUSE-global(full-sync).



(c) Perkeep.



(d) S3FS.



(e) UtahFS.

Figure 9: Performance comparison of UPSS-FUSE-global, S3FS, Perkeep and UtahFS. Owing to long read and write delays for comparison filesystems, benchmarks were run for 5 kops rather than 100 kops.

that the content is ready to be read from the Amazon block store. Even without caching and having the content persisted to the Amazon block store, UPSS-FUSE outperforms the other three filesystems by factors of 10–8,000. These results show that the cryptographic foundation of UPSS provides, not just strong security properties, but a foundation for aggressive caching that would be unsafe in a system that does not use cryptographic naming.

5 RELATED WORK

The CFS (Dabek et al., 2001), Coda (Satyanarayanan et al., 1990), Ivy (Muthitacharoen et al., 2002), and FARSITE (Adya et al., 2002) filesystems provide availability for user data stored on dedicated servers in a distributed environment along with other features such as disconnected operations, content-addressable storage and log-structured systems. Similar to UPSS-FUSE, FARSITE, which is a decentralized network filesystem, uses convergent encryption (Douceur et al., 2002; Li et al., 2013; Agarwala et al., 2017) to protect user data. As CFS, Coda and Ivy are non-cryptographic filesystems, they cannot rely on untrusted storage servers. On the other hand, the access control lists in FARSITE, which is a cryptographic fileystem, is not completely decoupled from user data; therefore, higher level applications cannot define their own policies, as it is possible in UPSS.

Several filesystems have been designed for untrusted cloud settings, such as NCryptFS (Wright et al., 2003), EncFS (Team, 2018; Leibenger et al., 2016), OutFS (Khashan, 2020) and CryFS (Messmer et al., 2017). NCryptFS and EncFS are cryptographic filesystems, which protect content by encrypting files, but leave filesystem metadata such as the directory structure unprotected. CryFS and OutFS solve this problem by splitting all filesystem data into fixed-size blocks and encrypting each block individually. PLUTUS (Kallahalla et al., 2003), VDisk (Wires and Feeley, 2007) and TVFS (Catuogno et al., 2014) also apply per block encryption. CryFs uses one key for all encryptions, but OutFS generates separate keys per file.

Ori (Mashtizadeh et al., 2013), IPFS (Benet, 2014) and Perkeep (Lindner and Norris, 2018) (formerly known as Camlistore) connect multiple devices with a filesystem that users can access anywhere. Both Ori and IPFS reduce the data inconsistency problem to a version control problem by storing new versions of files upon modification. Similar to UPSS-FUSE, Perkeep can be backed by a memory store, a local store or a cloud account. However, none of Ori, IPFS or Perkeep provide a mechanism for sharing redacted file and directory hierarchies. Moreover, Perkeep leaves the directory structure unprotected on the backing service.

MetaSync (Han et al., 2015) and DepSky (Bessani et al., 2013) are synchronization services that store confidential data on untrusted cloud storage providers. However, these two systems cannot be used as a platform for novel applications that UPSS can support and they just synchronize multiple cloud services.

Tahoe (Wilcox-O'Hearn and Warner, 2008) and UtahFS (uta, 2020a) are cryptographic filesystems with the goal of storing user data on untrusted storage servers. As in UPSS, Tahoe and UtahFS store content encrypted in Merkle DAGs and provide access control by cryptographic capabilities. Unlike UPSS, however, Tahoe's replica-oriented design lends itself more readily to storage of shared immutable data than to the use cases of a general-purpose filesystem.

6 CONCLUSION

UPSS: the user-centric private sharing system provides data availability, strong confidentiality and integrity properties while relying only on untrusted backend storage (local or remote). Data is encrypted at rest, named cryptographically and store within a content-addressable *sea of blocks*, so no file or directory structure can be discerned directly from the contents of an encrypted block store. Cryptographic capabilities are used to authorize access to arbitrarily-sized DAGs of files and directories without centralized access control. Convergent encryption enables data de-duplication for large files among even mutually-distrustful users while avoiding the common pitfalls of the technique for small, low-entropy files.

UPSS wraps copy-on-write operations with a conventional filesystem API, accessible directly as a library or proxied via a FUSE interface. Although UPSS-FUSE's performance is lower than that of direct API usage, it exceeds that of comparable cryptographic filesystems and is within an order of magnitude of that of the mature copy-on-write filesystem ZFS. When using remote storage, UPSS's performance exceeds that of UtahFS, Google's Perkeep and even, for some benchmarks, unencrypted NFS.

UPSS demonstrates that it is possible to achieve both strong security properties *and* high performance, backed by untrusted local, remote or global storage. UPSS's performance is comparable to — or, in some cases, superior to — mature, heavily-optimized filesystems. Adoption of UPSS will lay the foundation for future transformations in privacy and integrity for applications as diverse as social networking and medical data storage, providing better opportunities for users — not system administrators — to take control of their data.

REFERENCES

(2016). Filebench - A model based filesystem workload generator. https://github.com/filebench/filebench.

(2019). FUSE (Filesystem in Userspace). https://github.com/libfuse/libfuse/releases/tag/fuse-3.9.0.

(2020a). UtahFS. https://github.com/cloudflare/utahfs/releases/tag/v1.0.

(2020b). UtahFS: Encrypted File Storage. https://blog.cloudflare.com/utahfs.

Adya, A., Bolosky, W. J., Castro, M., Cermak, G., Chaiken, R., Douceur, J. R., Howell, J., Lorch, J. R., Theimer, M., and Wattenhofer, R. P. (2002). FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. *ACM SIGOPS Operating Systems Review*, 36(SI):1–14.

Agarwala, A., Singh, P., and Atrey, P. K. (2017). DICE: A dual integrity convergent encryption protocol for client side secure data deduplication. In *2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 2176–2181. IEEE.

Amazon Web Services, Inc. ((Accessed on February 28, 2020)). Amazon Simple Storage Service. "https://aws.amazon.com/s3".

Asklund, U., Bendix, L., Christensen, H. B., and Magnusson, B. (1999). The unified extensional versioning model. In *System Configuration Management*, pages 100–122, Berlin, Heidelberg. Springer Berlin Heidelberg.

Baker, M. G., Hartman, J. H., Kupfer, M. D., Shirriff, K. W., and Ousterhout, J. K. (1991). Measurements of a distributed file system. In *Proceedings of the thirteenth ACM Symposium on Operating Systems Principles*, pages 198–212.

Benet, J. (2014). IPFS: content addressed, versioned, P2P file system. *arXiv preprint arXiv:1407.3561*.

Bessani, A., Correia, M., Quaresma, B., André, F., and Sousa, P. (2013). DepSky: dependable and secure storage in a cloud-of-clouds. *ACM Transactions on Storage (TOS)*, 9(4):1–33.

Bonwick, J., Ahrens, M., Henson, V., Maybee, M., and Shellenbaum, M. (2003). The Zettabyte file system. In *Proc. of the 2nd Usenix Conference on File and Storage Technologies*, volume 215.

Bozorgi, A. (2020). *From online social network analysis to a user-centric private sharing system*. PhD thesis, Memorial University of Newfoundland.

Bozorgi, A., Jadidi, M. S., and Anderson, J. (2019). Challenges in Designing a Distributed Cryptographic File System. In *Cambridge International Workshop on Security Protocols*, pages 177–192. Springer.

Catuogno, L., Löhr, H., Winandy, M., and Sadeghi, A.-R. (2014). A trusted versioning file system for passive mobile storage devices. *Journal of Network and Computer Applications*, 38:65–75.

Dabek, F., Kaashoek, M. F., Karger, D., Morris, R., and Stoica, I. (2001). Wide-area cooperative storage with CFS. In *ACM SIGOPS Operating Systems Review*, volume 35, pages 202–215. ACM.

Dennis, J. B. and Van Horn, E. C. (1966). Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–155.

Douceur, J. R., Adya, A., Bolosky, W. J., Simon, P., and Theimer, M. (2002). Reclaiming space from duplicate files in a serverless distributed file system. In *Proceedings of the 22nd International Conference on Distributed Computing Systems*, pages 617–624. IEEE.

Dworkin, M. (2015). SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. Federal Inf. Process. Stds. (NIST FIPS), National Institute of Standards and Technology.

Dworkin, M. J., Barker, E. B., Nechvatal, J. R., Foti, J., Bassham, L. E., Roback, E., and Jr., J. F. D. (2001). Advanced Encryption Standard (AES). Federal Inf.

Process. Stds. (NIST FIPS), National Institute of Standards and Technology.

Gaul, A., Nakatani, T., and rrizun (2020). S3FS: FUSE-based file system backed by Amazon S3). https://github.com/s3fs-fuse/s3fs-fuse/releases.

Han, S., Shen, H., Kim, T., Krishnamurthy, A., Anderson, T., and Wetherall, D. (2015). MetaSync: File synchronization across multiple untrusted storage services. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 83–95.

Kaliski, B. (2000). PKCS #5: Password-Based Cryptography Specification Version 2.0. RFC 2898.

Kallahalla, M., Riedel, E., Swaminathan, R., Wang, Q., and Fu, K. (2003). Plutus: Scalable secure file sharing on untrusted storage. In *2nd USENIX Conference on File and Storage Technologies (FAST 03)*.

Khashan, O. A. (2020). Secure outsourcing and sharing of cloud data using a user-side encrypted file system. *IEEE Access*, 8:210855–210867.

Lazowska, E. D., Zahorjan, J., Cheriton, D. R., and Zwaenepoel, W. (1986). File access performance of diskless workstations. *ACM Transactions on Computer Systems (TOCS)*, 4(3):238–268.

Leibenger, D., Fortmann, J., and Sorge, C. (2016). EncFS goes multi-user: Adding access control to an encrypted file system. In *2016 IEEE Conference on Communications and Network Security (CNS)*, pages 525–533. IEEE.

Li, J., Chen, X., Li, M., Li, J., Lee, P. P., and Lou, W. (2013). Secure deduplication with efficient and reliable convergent key management. *IEEE Transactions on Parallel and Distributed Systems*, 25(6):1615–1625.

Lindner, P. and Norris, W. (2018). Perkeep (née Camlistore): your personal storage system for life. https://github.com/perkeep/perkeep/releases.

Mashtizadeh, A. J., Bittau, A., Huang, Y. F., and Mazieres, D. (2013). Replication, history, and grafting in the Ori file system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 151–166. ACM.

Merkle, R. (1979). *Secrecy, authentication, and public key systems*. PhD thesis.

Messmer, S., Rill, J., Achenbach, D., and Mü ller Quade, J. (2017). A novel cryptographic framework for cloud file systems and CryFS, a provably-secure construction. In *IFIP Annual Conference on Data and Applications Security and Privacy*, pages 409–429. Springer.

Microsoft, Inc. ((Accessed on January, 2023)). Azure Blob Storage. "https://azure.microsoft.com/en-us/products/storage/blobs/ ".

Muthitacharoen, A., Morris, R., Gil, T. M., and Chen, B. (2002). Ivy: A read/write peer-to-peer file system. *ACM SIGOPS Operating Systems Review*, 36(SI):31–44.

Rosenblum, M. and Ousterhout, J. K. (1992). The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)*, 10(1):26–52.

Satyanarayanan, M., Kistler, J. J., Kumar, P., Okasaki, M. E., Siegel, E. H., and Steere, D. C. (1990). Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459.

Shepler, S., Callaghan, B., Robinson, D., Thurlow, R., Beame, C., Eisler, M., and Noveck, D. (2003). RFC3530: Network File System (NFS) Version 4 Protocol.

Team, E. (2018). EncFS: an Encrypted Filesystem for FUSE. https://github.com/vgough/encfs/releases/tag/v1.9.5.

Team, S. (2020). SSHFS (a network filesystem client to connect to ssh servers). https://github.com/libfuse/sshfs/releases.

Wilcox-O'Hearn, Z. and Warner, B. (2008). Tahoe: the least-authority filesystem. In *Proceedings of the 4th ACM International Workshop on Storage Security and Survivability*, pages 21–26.

Wires, J. and Feeley, M. J. (2007). Secure file system versioning at the block level. In *proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 203–215.

Wright, C. P., Martino, M. C., and Zadok, E. (2003). NCryptfs: A Secure and Convenient Cryptographic File System. In *USENIX Annual Technical Conference, General Track*, pages 197–210.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS