Secure Untrusted Data Repository (SUNDR)

Abstract

We have implemented a secure network file system called SUNDR that guarantees the integrity of data even when malicious parties control the server. SUNDR splits storage functionality between two untrusted components, a block store and a consistency server. The block store holds all file data and most metadata. Without interpreting metadata, it presents a simple interface for clients to store variable-sized data blocks and later retrieve them by cryptographic hash.

The consistency server implements a novel protocol that guarantees close-to-open consistency whenever users see each other's updates. The protocol roughly consists of users exchanging version-stamped digital signatures of block server metadata, though a number of subtleties arise in efficiently supporting concurrent clients and group-writable files. We have proven the protocol's security under basic cryptographic assumptions. Without somehow producing signed messages valid under a user's (or the superuser's) public key, an attacker cannot tamper with a user's files—even given control of the servers and network. Despite this guarantee, SUNDR performs within a reasonable factor of existing insecure network file systems.

1 Introduction

Nobody wants malicious attackers tampering with his files. This basic and obvious fact underlies much of the way people manage data. Important file systems must be kept on secure servers in machine rooms to which only authorized people have access. Only highly trusted administrators can perform mundane tasks such as backup and hardware maintenance. Conversely, people must address the constant threat of attackers gaining administrative privileges on servers. Otherwise, a number of readily available "rootkits" allow attackers who penetrate a system to replace core operating system utilities with altered versions that open back doors and conceal evidence of the security breach.

Data security is often viewed as the problem of building a better fence around storage servers—limiting the number of people with server access, disabling unnecessary daemons that might be remotely exploitable, and staying current with security patches to minimize the window of vulnerability to software flaws. This approach has two drawbacks. First, experience has shown in many cases that people do not build high enough fences. Second, perhaps more important, high fences are inconvenient; they restrict the ways in which people can manage data.

An alternative approach is to reduce the security needs of file servers. This paper presents SUNDR, a secure network file system designed to do exactly this. Assuming only the existence of digital signatures and a collision-resistant hash function, SUNDR's protocol provably guarantees the integrity and consistency of file system data, *even when malicious parties entirely control the server*. Unlike previous Byzantine-fault-tolerant file systems [4, 22] that distribute trust but assume that a threshold fraction of servers are honest, SUNDR assumes no on-line trusted parties. To tamper with a user's files without being detected, an attacker must either compromise the user's client while the user is logged in, or else otherwise produce valid digital signatures under the user's public key. In particular, the superuser's private signature key can be stored off-line when not in use, making it extremely difficult for an attacker to gain super-user access to the file system.

SUNDR does not currently address the issue of storage reliability. Of course, an attacker can always physically damage a server or wipe its disks. However, SUNDR stores all long-lived data in an append-only block log. Thus, it could use append-only storage [27] to gain resilience to network attacks. Incremental off-site backups are also easily implementable to survive physical compromise. Moreover, because SUNDR does not trust the file server, after a disaster, any lost file system data can safely be recovered from other untrusted sources that might have the data, such as clients' file caches.

SUNDR's security model gives people more options for managing their data than current systems. For instance, organizations can outsource data storage without fear of the server operators tampering with data. It also offers a vast improvement over current file system security, An attacker who compromises a SUNDR server cannot tamper with file contents. Our prototype implementation gives performance within a reasonable factor of the popular NFS file system, making SUNDR practical despite its

increased security.

The next section gives an overview of the SUNDR protocol. The following two sections describe SUNDR's implementation and how we tuned the protocol to give acceptable performance. Section 5 evaluates the performance of our implementation. Section 6 discusses related work, and Section 7 concludes.

2 Protocol

The SUNDR protocol provably guarantees the integrity and consistency of file system contents, preventing untrusted server opertors or malicious parties who compromise a server from undetected tampering with data. This section gives an overview of the protocol and its security properties, and explains how SUNDR's semantics differ from those of ordinary Unix file systems. A more detailed description of the protocol and a proof of its security were presented in [1].

At the highest level, every file in a SUNDR file system belongs to either a user or a group. For the purposes of this paper, a user is a principal logged into a single SUNDR client. (Human users logged into multiple terminals function as separate SUNDR users in the same SUNDR group.) Each user has a public signature key. To change a file, one must produce a message signed by the file's owner or, for group-owned files, by a member of the group that owns the file. As a special case, the superuser can sign any data structure.

All users know each other's public keys and group membership. Currently, this information is manually distributed to clients, though in the future we envisage managing keys and groups through superuser-owned files in the file system. The server does not have any user's signature key, and thus cannot make unauthorized changes to the file system. Moreover, the server operator does not need the superuser key. In fact, the superuser's signature key need not be anywhere on-line unless a privileged user is accessing the file system as an administrator.

2.1 Data structures

Figure 1 shows the basic SUNDR data structures. Every file is identified by a $\langle \mathrm{principal}, \text{i-number} \rangle$ pair, where principal is the user or group that owns the file and i-number is a per-user or per-group inode number. (Unlike traditional file systems, SUNDR allows files owned by different users to have the same i-number.) Directory entries map file names onto $\langle \mathrm{principal}, \text{i-number} \rangle$ pairs. A per-principal data structure called the *i-table* maps each active i-number to a collision-resistant SHA-1 [8] hash of the file's inode. We call this value the file's *i-hash*. Inodes themselves contain SHA-1 hashes of file data blocks and indirect blocks, an approach taken from the SFSRO [11] read-only file system.

SUNDR forms a cryptographic hash tree [15] from each i-table. The root of this tree is called the *i-handle*. Given an i-handle and all appropriate intermediary data structures, one can verify any block of any file in the i-table. Thus, securely retrieving file system contents boils down to the problem of first obtaining the latest i-handle of each user and group, then retrieving any needed data blocks by their SHA-1 hash. The latter functionally is conceptually simple to implement, while the former requires a somewhat complex consistency protocol.

2.2 Consistency protocol

Traditional network file systems provide close-to-open consistency. If one user modifies and closes a file, another user who subsequently opens and reads the file should see the data most recently written there. Concurrent operations can be ordered at the discretion of the server, but all clients must agree on the order of any conflicting operations.

SUNDR achieves consistency by embeding i-handles in digitally signed *version structures* through which clients can verify that the file system has been delivering close-to-open consistency. As long as users see the effects of each other's operations, they are guaranteed to be getting close-to-open consistency.[1] There remains the possibility of a malicious server entirely concealing some users' actions from others, if neither collection of users expects anyone from the other to have accessed the file system. However, such attacks are very likely to be discovered quickly, as once a malicious server has "forked" the state of the file system, it can never again allow deceived users to see any evidence of each other's on-line activity.

Version structures use a technique somewhat similar to version vectors [5] to detect inconsistencies. Each user and group has a version number. A version structure contains the latest version number of every user and group at the time of the file system operations it reflects. Two version structures are compatible if one contains at least as high a version number as the other for all principals. When creating a version structure, users increment both their own version numbers and the version numbers of any group whose i-tables they are modifying. Both opens and closes require new version structures to be computed. If the server fails to deliver close-to-open consistency, users will detect this upon seeing each other's incompatible version structures. Figure 2 gives an example. While the ba-

[1] More precisely, if user A sees the effect of an operation user B performed at time t, then, at least until t, A and B had close-to-open consistency with repect to each other. Moreover, anyone who sees a subsequent operation by A must also have had close-to-open consistency with respect to B until t.

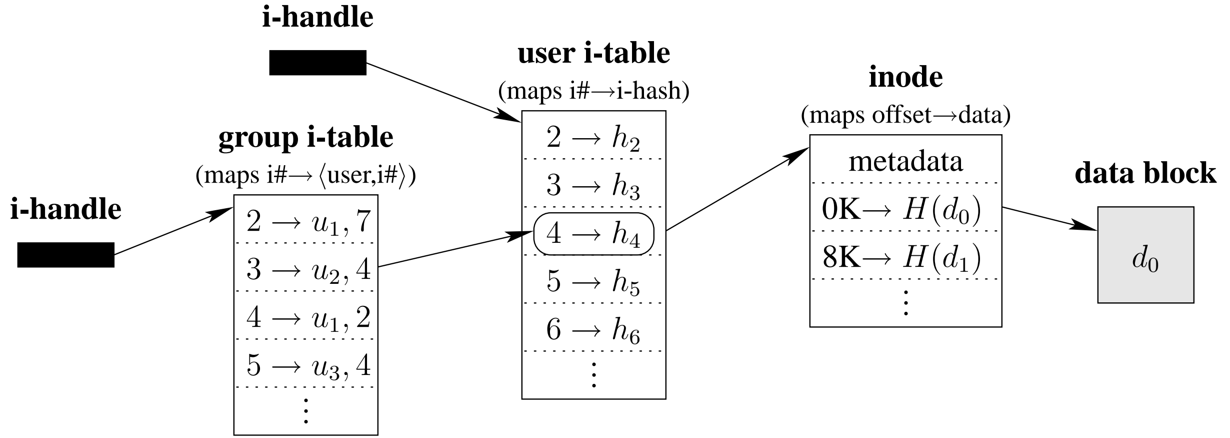

Figure 1: Basic SUNDR data structures. An i-handle is the root of a hash tree containing a user or group i-table. A group i-table maps group inode numbers to user inode numbers. A user i-table maps a user's inode numbers to i-hashes. An i-hash is the hash of an inode, which in turn contains hashes of file data blocks.

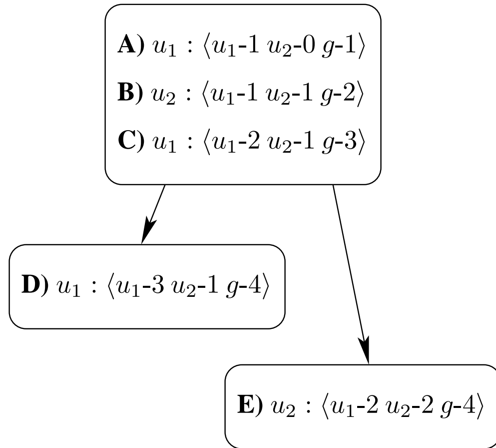

Figure 2: Example version structures. In **A–C**, users u_1 and u_2 both modify group g's i-handle and increment its version number. u_1 again modifies g in **D**, but the malicious server conceals this update from u_2, forking the file system state. u_2 subsequently signs a version structure incompatible with u_1's, which the two users will discover should they ever again see any of each other's updates.

sic idea is simple, a significant complication is that under concurrent updates, SUNDR users need to compute and sign their own version structures without necessarily having access to all other users' latest version structures.

To allow concurrent updates, the SUNDR consistency protocol consists of two RPCs, UPDATE and COMMIT, illustrated in Figure 3. Both RPCs contain messages signed under users' public keys. Intuitively, UPDATE declares a user's intent to perform some operation on the file system. It returns the precise state of the file system at the time the update is to have taken place. COMMIT contains the user's version structure, certifying the order that the server has assigned to operations. Clients wait for and verify users' signed version structures before reading the effects of any declared modifications, so as to verifying the consistency provided by the server.

In more detail, when a user makes a file system call, the client first sends the server a signed *update certificate* including the user's version number in the forthcoming version structure and a list of changes or *deltas* the user wishes to make to the file system. If the user is simply looking up a file name in a directory or opening a file for reading, the list of deltas is empty. Otherwise, when changing the state of the file system, deltas may be of four types:

- Set file $\langle \mathrm{user}, \mathrm{i\#} \rangle$ to i-hash h.
- Set group file $\langle \mathrm{group}, \mathrm{i\#} \rangle$ to $\langle \mathrm{user}, \mathrm{i\#} \rangle$.
- Set/delete entry *name* in directory $\langle \mathrm{princ.}, \mathrm{i\#} \rangle$.
- Pre-allocate a range of group i-numbers (pointing them to unallocated user i-numbers).

The effects of all deltas are deterministic, so that multiple clients applying the same deltas to the same i-handles in the same order will always produce the same result.

In response to an update message, the server sends back the latest signed version structure that it has for every i-handle, plus a list of signed update certificates for operations not yet reflected in the version structure list. We call the collection of version structures returned the *version structure list*, or VSL, and the list of update certificates the *pending version list*, or PVL. An honest server totally orders all file system operations by the time at which it receives the corresponding update certificate. An operation with update certificate x occurs after every operation

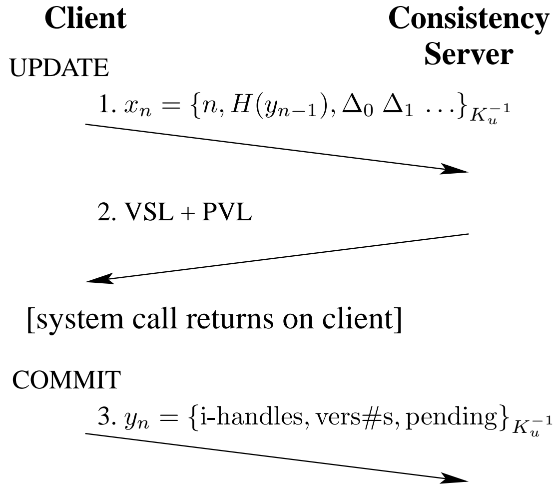

Figure 3: SUNDR consistency protocol. The client declares its intent to fetch or modify file system content with a signed update certificate. The server responds with the latest version structure for each i-handle plus any pending update certificates. The client then computes new i-handles and sends a signed version structure to the server.

reflected in the VSL and PVL returned for x, and before any operation whose UPDATE RPCs return x in the PVL.

When the client receives the reply to its update certificate, it checks the PVL for any read-after-write conflicts. If it is not trying to read a file that another client is currently in the process of modifying, the client can return immediately from the file system call. Otherwise, it must wait for any pending conflicting operations to move from the PVL to the VSL. In either case, the client computes a new version structure based on the VSL and PVL it has received. This version structure contains the user's i-handle, the i-handles of any groups the user has modified, the current latest version number of every user and every group, and a list of pending operations in the PVL, including for each pending operation a SHA-1 hash of its forthcoming version structure without the i-handles. When computing group i-handles, the client must reflect any previous modifications to the group in the PVL.

A proof of the consistency protocol was presented in [1], but is beyond the scope of this paper. At a high level, however, its security follows from two properties. First, once two users have signed incompatible version structures, they can never again sign compatible ones without detecting the attack. Second, once a pending operation has appeared in the PVL, it will remain there until the corresponding version structure (or a later one) is signed by the user making the update. Thus, if server misbehavior makes two users' version structures incompatible, neither user can complete any system call that conflicts with an operation by the other that was pending at the time of the attack. It is this last property that makes it safe for one user to apply another user's deltas to group i-tables.

2.3 Semantics

SUNDR's semantics differ from traditional Unix file systems in several ways. The implementation currently does not keep track of time of last access ("atime") for files, as atime cannot be tracked securely by an untrusted server. Other file times, including the i-node change time ("ctime"), are set by clients, allowing users who hack their client software to set ctimes of files they own to arbitrary values. SUNDR also has no equivalent of the Unix "sticky bit" on directories.

While Unix files have both an owner and a group, group-writable files in SUNDR do not have a comparable notion of owner—the user who last wrote the file functions as its owner. The process of changing the permissions on a SUNDR file also functions more like making a new copy of the file; it requires write permission on the parent directory and does not affect other hard links to the file. Fortunately, SUNDR's content-hash based block store allows such copies to be made cheaply. The owner of a directory also can delete any entries in that directory—including non-empty directories to which he or she does not have write permission.

SUNDR's semantics were primarily driven by its security requirements. Though they depart significantly from traditional file systems in some areas, the new semantics offer several benefits. In particular, it is often useful to know which user has last written a group-writable file. Furthermore, if the SUNDR block server implemented disk quotas, it could limit users by the actual number of new blocks they write, whether to group-owned or user-owned files. The fact that users may be unable to delete subdirectories of their own directories is somewhat of an annoyance in Unix; giving them this right does not appreciably weaken security, as users anyway have the ability to rename directories they cannot remove. Note that except for the ctime value, malicious users who violate the protocol can only affect files in ways they equivalently could through the system call interface.

We emphasize also that SUNDR's focus is on guaranteeing the integrity and consistency of data. The current implementation does not directly address the questions of privacy, availability, or reliability, but SUNDR is entirely compatible with these goals. A number of previously developed cryptographic storage techniques could be applied to achieve some degree of privacy in SUNDR. Extensive work has also been done on ensuring data availability in the face of disk and server failures. The fact that STORE RPCs can safely be reordered would make it easy

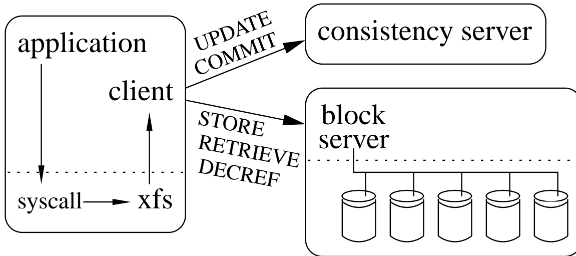

Figure 4: Implementation overview

to replicate the block server, while the consistency protocol already replicates the PVL and and VSL on clients. SUNDR also facilitates reliability by storing long-lived data in an append-only log, making the server amenable to backup on write-once media and incremental transfer to off-site replicas. Because the server is not trusted, a damaged file system can be reconstructed from all available sources of data, including file caches on untrusted clients.

3 File system implementation

Figure 4 shows the overall architecture of the SUNDR implementation. The client is implemented at user level, using a modified version of the **xfs** device driver from the ARLA [28] file system on top of a slightly modified FreeBSD kernel. Server functionality is divided between a consistency server and a block server. The consistency server implements the UPDATE and COMMIT RPCs described in the previous section. The only file system data it ever sees are the i-handles embedded in version structures.

Most file system data is stored on the block server. The block server is a user-level program that reads and writes raw disks through character devices so as to manage its own cache and control the order in which writes go to disk. The interface primarily consists of three RPCs, STORE, RETRIEVE, and DECREF. STORE stores a block of data at the server. RETRIEVE takes a SHA-1 hash as an argument and returns a block with that hash (assuming one has been stored). DECREF informs the server that the client will no longer attempt to retrieve a particular SHA-1 hash, so that the server can discard the corresponding block if no one else has stored it. A few less fundamental RPCs are not shown in Figure 4—for instance to support authenticated communication so that an attacker cannot impersonate the block server and begin discarding newly stored blocks.

The remainder of this section describes the implementation of the client and consistency server, and explains how we optimized the SUNDR protocol to make the file system perform acceptably. The next section gives the details of our high-performance block server implementation.

3.1 File system client

The **xfs** device driver used by SUNDR is designed for whole-file caching. When a file is opened, **xfs** makes an upcall to the SUNDR client asking for the file's data. The client returns the identity of a local file that has a cached copy of the data. All reads and writes are performed on the cached copy, without further involvement of SUNDR. When the file is closed (or flushed with *fsync*), if it has been modified, **xfs** makes another upcall asking the client to write the data back to the server. Several other types of upcalls allow **xfs** to look up names in directories, request file attributes, create/delete files, and change metadata.

As distributed, **xfs**'s interface posed two problems for SUNDR. First, **xfs** caches information, such as local file bindings, which it uses to satisfy some requests without upcalls. In SUNDR, some of these requests require interaction with the consistency server for the security properties to hold. We therefore modified **xfs** to invalidate its cache tokens immediately after getting or writing back cached data, so as to ensure that the user-level client gets control whenever the protocol requires an UPDATE RPC.

Second, some system calls that should require only a single interaction with the SUNDR consistency server result in multiple kernel vnode operations and **xfs** upcalls. For example, the system call "`stat ("a/b/c", &sb)`" results in three **xfs** GETNODE upcalls (for the directory lookups) and one GETATTR. The whole system call should require only one UPDATE RPC. Yet if the user-level client does not know that the four upcalls are on behalf of the same system call, it must check the freshness of its i-handles four separate times with four UPDATE RPCs.

To eliminate unnecessary RPCs, we modified the FreeBSD kernel to keep a count of the number of invocations of system calls that might require an interaction with the consistency server. We increment the counter at the start of every system call that takes a pathname as an argument (e.g., `stat`, `open`, `readlink`, `chdir`). The SUNDR client memory maps this counter and records the last value it has seen. If **xfs** makes an upcall that does not change the state of the file system and the counter has not changed, then the client can use its cached copies of all i-handles.

3.2 Protocol optimizations

Any implementation of the SUNDR protocol faces a number of potential performance bottlenecks. Digital signatures can be costly to compute and verify, yet are on the critical path for every RPC to the consistency server. Recomputing hash trees for every file system modification is

expensive and requires large amounts of bandwidth and storage from the block server. Group i-tables in particular are expensive to update, as different users must transfer new hash nodes back and forth through the block server, and a new i-table could potentially force clients to flush their name caches.

The first requirement for decent performance was to reduce the cost of digital signatures in SUNDR. Our implementation does so in three ways—by choosing an efficient signature algorithm, by reducing the total number of signatures required, and by moving signatures and verifications out of the critical path wherever possible. We chose the Esign [18] signature algorithm for SUNDR because it is over an order of magnitude faster than more popular schemes such as RSA. While there are no known attacks on the version of Esign we are using, a flaw was recently found in its proof of security and better variant proposed [12] with a correct proof. We intend to switch to the newer variant. All experiments reported in this paper use 2,048-bit public keys, which require a work factor of well over 2^{80} to break using the fastest known algorithms.

To get verification out of the critical path, the consistency server also processes and replies to an UPDATE RPC before verifying the signature on its update certificate. If the signature fails to verify, the server reverts any effects of the RPC (before other clients can see them) and drops the TCP connection to the forging client. This behavior is acceptable because only a faulty client would forge signatures. With this optimization, the consistency server's verification of one signature overlaps with the client's computation of the next.

The second class of optimizations reduces the cost of processing i-table hash trees. First, because group i-tables are particularly expensive to modify, SUNDR minimizes the number of times group i-handles change through a level of indirection; group i-table entries map to user i-table entries which map to i-hashes. Thus, a group i-handle doesn't change when a group-writable file or directory is modified multiple times by the same user, a common case. When creating group-writable files, clients pre-allocate group i-numbers in batches, pointing them to unused ranges of user i-nodes. 256 group i-numbers can be pre-allocated with a single delta in an update certificate.

As a further optimization for both user and group i-tables, SUNDR avoids recomputing i-handles on every update. Instead, it specifies a principal's i-table with two hashes embedded in version structures—an i-handle, which is still the root of a hash tree, and a *log hash*, which is the SHA-1 hash of a vector of deltas to be applied to state of the hash tree. The COMMIT RPC takes a log as well as a version structure, and the UPDATE RPC returns a log for each i-handle. Clients only periodically compute new i-handles to clear the log hash.

Log hashes have a number of benefits. They amortize computation of new new i-handles over many updates. They save bandwidth to the block server by requiring fewer hash tree blocks to be stored. For group i-tables, they allow one client to apply another's update certificate deltas without fetching any blocks from the server. Most importantly, they allow clients to validate cached information without fetching i-table blocks from the server. Both name cache entries and cached file contents are valid so long as the relevant i-handles haven't changed and the particular file is not affected by any deltas in the hashed log.

3.3 Future optimizations

Finally, we mention several optimizations that we would like to implement but have not yet. To reduce the number of signatures required, the SUNDR client might avoid signing version structures by eliding COMMIT RPCs. (This is safe, as from other clients' point of view an omitted COMMIT is equivalent to two COMMITs arriving back-to-back at the server.) After completing an UPDATE RPC, the client can return immediately to **xfs**, delaying computation of the new version structure. If, within 2 milliseconds, no further UPDATE RPCs are required, the client can then sign a version structure and make a COMMIT RPC. If, on the other hand, the client immediately makes another UPDATE RPC, it can avoid signing the first version structure or making the first COMMIT RPC. Under workloads such as unpacking tar archives, this optimization might save the signing cost of several consecutive version structures.

Bandwidth to the consistency server could also be considerably reduced. UPDATE RPCs currently return the entire VSL and PVL. However, a client will generally already have many of the entries in these lists. The consistency server should therefore return only VSL and PVL entries that are new since the last UPDATE reply it has sent to a particular client. Similarly, both UPDATE replies and COMMIT RPCs currently send the entire delta log with each i-handle, when only new entries need be sent. Even version structures need not be sent in their entirety, as most of their content other than i-handles is implicit in the order of PVL entries and the groups they affect. On systems with many users, the messages exchanged with the consistency server should be proportional in size to the number of active users and groups, while currently they are proportional to the total number of principals.

Another class of possible optimizations target the block server. SUNDR currently breaks files into fixed-size 8K blocks, stores the blocks at the block server, and puts their SHA-1 hashes into inode and indirect block data structures. Because the block server handles variable-

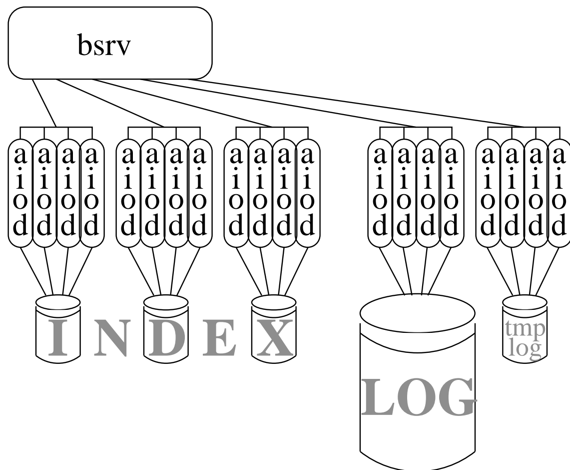

Figure 5: Block server architecture

sized blocks efficiently, there is no reason not to compress blocks at the client, saving both network bandwidth and storage at the server. Moreover, recent work [17] suggests that by breaking files into carefully-constructed variable-sized blocks, one can increase the number of blocks common across files. Such commonality would reduce the number of block transfers needed, save space at the block server, and furthermore increase the effectiveness of the block server's cache.

4 Block server implementation

In SUNDR, all data and most metadata are stored on and retrieved from the block server, called *bsrv*. *bsrv* presents clients with an extremely simple interface. Clients send the server data blocks and later retrieve them by SHA-1 hash. Since the client actually implements most file system functionality, the block server cannot interpret or prioritize blocks. Moreover, the server does not have any user signature keys, and thus does not even have permission to repair the file system should something go wrong. Thus, while traditional file systems write many types of data asynchronously, crash recoverability requires that *bsrv* write every block to stable storage before returning to the client. In addition, block server performance is critical to good file system performance in SUNDR.

4.1 Background: Venti

bsrv is certainly not the first content-hash-based block server. In fact, *bsrv*'s design takes its inspiration from the Venti [19] block server. Venti appends variable-sized blocks back-to-back on a large log disk or RAID array, and then indexes them by SHA-1 hash on a fast SCSI disk. An index entry is stored in a logical block on disk determined entirely by the SHA-1 hash. Overflow entries in Venti spill over onto the next block.

Whereas we follow this general design, three properties make Venti ill-suited for SUNDR:

1. Venti buffers writes, and does not guarantee they have reached stable storage until a special and expensive sync RPC is made. The sync RPC requires synchronous disk seeks on the log disks to differentiate new blocks from redundantly stored ones. Moreover, since blocks are not sector aligned, calling sync frequently would result in multiple writes to the same sector of the log disk, which the operating system cannot schedule simultaneously.

2. In the absence of a crash, all writes to Venti are permanent. However, SUNDR generates many short-lived blocks, such as i-table hash tree nodes. Storing these permanently would waste log space, while indexing them would increase contention for the log disks.

3. Venti's reported write throughput is only 5–7 MBytes/sec, while for SUNDR we wanted a block server capable of absorbing data at fast Ethernet rates. Writes are more important than reads in our application because client caches are expected to satisfy most read requests.

To overcome these problems, *bsrv* introduces a temporary log to ensure immediate durability of data while also improving performance in several ways.

4.2 *bsrv* architecture

Figure 5 shows *bsrv*'s general architecture. The block server requires at least three disks, a large log disk (or RAID array), a small temporary log disk, and one or more index disks. As an example configuration, this paper reports on a server that uses high-capacity IDE disks for the log and temporary log, while striping the index over four fast SCSI disks.

bsrv accesses disks through the raw character device interface. Because Unix devices have a blocking interface, it uses a number of helper processes to issue concurrent I/O requests, thereby keeping multiple disks busy and enjoying better disk scheduling through SCSI tagged command queuing on the index disks. *bsrv* spawns four instances of the helper program, called *aiod*, for each disk on the system. It communicates with them through large regions of shared memory. Each group of four *aiod*s also listens on a pipe. To initiate an I/O, *bsrv* writes to the appropriate pipe an 8-byte pointer to a command block in shared memory. When the I/O is complete, *aiod* writes

the status into the command block and notifies *bsrv* by sending the command block's address back over a unix-domain socket. In order to avoid waking up four kernel-level threads each time *bsrv* writes to a pipe, the *aiod* processes use an *flock* lockfile to ensure only one process at a time blocks reading a given pipe. (*flock* is specifically implemented to avoid waking threads up unnecessarily.)

The *bsrv* interface consists of the following five RPCs:

STORE (*header*,*block*)
RETRIEVE (*hash*)
DECREF (*hash*)
CSTORE (*hash*)
PSTORE (*header*, *block*)

The STORE RPC takes a data block preceded by a block header and writes them both to stable storage if the server does not already have a copy of the block. The header has information encapsulating the owner and creation time of the block, as well as fields that could be used to store the encoding method and decoded length of compressed blocks. The header additionally contains the block's SHA-1 hash, so as to save the server from recomputing hashes of redundantly stored blocks. (Of course, *bsrv* verifies SHA-1 hashes of new blocks so as to prevent malicious clients from corrupting its index.)

The RETRIEVE RPC retrieves a block from the server given its SHA-1 hash. It also returns the first header STOREd with the particular block.

DECREF informs the server that a block with a particular SHA-1 hash is no longer needed and can be discarded. *bsrv*'s dereferencing semantics are conservative. When a block is first STOREd, the server establishes a short window (one minute by default) during which it can be deleted. If a client STOREs then DECREFs a block within this window, the server marks the block as garbage and does not permanently store it. If two clients store the same block during the dereference window, the block is also marked as permanent. Once a block has been marked permanent, it can never be deleted from the block server.

Two more RPCs exist that are not currently used by the SUNDR client. CSTORE, short for "conditional store," functions like a read that doesn't return the data—it simply tells the client whether or not the server is storing a block with a particular SHA-1 hash. A client could use this RPC to save bandwidth when storing blocks that are likely to exist already at the server.

PSTORE, short for "permanent store," is not intended to be called by clients, but is exposed for the purposes of testing and comparison. PSTORE's semantics differ from STORE only in that PSTOREd blocks cannot be DECREFed. PSTORE's performance characteristics are quite different from STORE's, however—PSTORE bypasses the temporary log and is equivalent to a store followed by a sync in Venti.

We now describe how *bsrv* uses its disks.

4.3 Temporary Log

When a client issues a STORE RPC, the server will immediately store the given block to the temporary log subsystem. In our implementation, this subsystem comprises a small raw partition on an IDE disk, and an in-memory cache that mirrors the contents of the temporary log. By the time it responds to the RPC, the server has either written the block to the temporary log disk or located a copy of the block on a disk in the system, thus guaranteeing durability in the event of a crash. Keeping a redundant copy of the block in memory assures us that we need read from the temporary log only to support crash recovery. Assuming the temporary log disk does not serve requests generated by other processes, it need rarely seek. Rather, an overwhelming portion of disk traffic will be sequential writes, thus enjoying the same performance characteristics as a database log.

Finally, we should note that all writes to the temporary log are aligned along sector boundaries. This wastes, on average, half a sector of disk space per block. The space overhead is of little concern given that the server continually recycles the temporary log and that the temporary log is very small relative to the capacity of modern IDE disks. Aligning writes to sector boundaries avoids the need for successive writes of the same sector, which the operating system cannot issue concurrently and which therefore cost a disk rotation.

4.4 Permanent Log Subsystem

The server periodically processes the temporary log, aging out those blocks that have exceeded their dereference window. If a block has been dereferenced but not accessed in any other way, the server will delete the block from the temporary log, being certain not to enter the block into any of the various caching systems. Otherwise, the server PSTOREs the block. PSTORE will write a block to the permanent log *provided that* the block does not already exist on the server. Blocks are organized in the permanent log along four-byte boundaries to optimize storage density. Once a block is written to the permanent log, it is read-only and immutable.

Since the server processes the temporary log in batches, it will write blocks to the permanent log in batches. For a batch that consists of n blocks, the server will usually require only nb/m sequential writes to the log disk, where b is the average block size, and m is the maximum write size. On our system, $b = 8k$ and m = 64k, thus we need only write to disk once every 8 blocks.

Our server uses a large raw IDE hard disk for its permanent log. Along the lines of Venti's architecture, we

further subdivide the IDE disk into a sequence of smaller, more manageable *arena*s. Within an arena, blocks are stored from the lowest disk offset to the highest, and block directory structures are stored from the highest disk offset down. A block directory structure consists of the block header and its disk offset on the permanent log. It is used to facilitate the reconstruction of the index in the case of a crash.

The log subsystem also includes an LRU cache. Once data blocks are written to or retrieved from the log disk, they are stored in an in-memory LRU block cache.

4.5 Index Subsystem

The index subsystem serves to locate blocks on the permanent log, keyed by their SHA1 hashes. An ideal index would be a simple in-memory hash table mapping 20-byte SHA1 hashes to an 8 byte disk offset. If we assume that the average block stored on the system is 8K, we see that the index must have roughly $1/128$ the capacity of the log disk. Although at present, such a ratio of disk to memory is obtainable with commodity components, we are not convinced that memory will be able to maintain such a ratio with hard disks in the future.

Instead, we adopt Venti's strategy of implementing the index as a disk-resident hash table, striped over an array of high-speed SCSI disks. We hash 20-byte SHA1 hashes down to $\langle$*index-disk-id*, *index-disk-offset*$\rangle$ pairs. The disk offsets point to sector-sized on-disk data structures called *buckets*, which contain 15 *index-entries*, sorted by SHA1 hash. *index-entries*, in turn map SHA1 hashes to offsets on the permanent data log.

The server might access the index subsystem during PSTORE, CSTORE, and RETRIEVE. Assuming the server has not found the requested block in the temporary log or the block LRU cache, it will look up the block's SHA1 hash in the index. If it finds the hash in the index, it will trigger an immediate RPC response to PSTORE and CSTORE. In the case of RETRIEVE, it will then access the *index-entry* to find the block's location on the permanent log, read and then return the block to the client. If the server does not find the hash in the index, it will, in the case of PSTORE, write the block to the permanent log and then update the index to reflect where the block was stored, using quadratic hashing when index blocks overflow. In the case of CSTORE, the server will respond false to the RPC. In the case of RETRIEVE, the server will reply that the block was not found.

As the server receives PSTORE, CSTORE and RETRIEVE RPCs from the client, it should expect a random ordering of SHA1 hashes. Thus, each RPC will require one random seek across an index disk, and a PSTORE that is storing a block not already on the system will need two. As in Venti, we use caching and striping to mitigate the effects of this bottleneck. After the server accesses or updates an *index-entry*, it will store it in an LRU cache. Furthermore, striping the index across multiple disks allows concurrency among lookups and limits the amount of space needed on each disk, hence shortening the range over which the index disks heads need to seek.

Doing a large majority of our stores via STORE and the temporary log allows us to make more efficient use of the index subsystem. Before moving a batch of blocks from the temporary log to the permanent log, the server sorts them by the offset on the index disk of their associated bucket. That is, we map each block to a SHA1 hash, each SHA1 hash to a bucket offset, and sort. As a result, the index disk arms can service a whole batch of index reads in one sweep. We use the same technique for scheduling lazy-writes to the index disk after we have written a block to the permanent log disk. The sorting supports index disk efficiency, but the system ensures that blocks are stored on the log in the order in which they arrived. Scattering blocks would be detrimental to read performance for large multi-block files.

4.6 Crash Recovery

To recover from a crash or an unclean shutdown, the system first assures the consistency of the permanent log and index subsystems. It scans the most recently stored data blocks to determine if the data and directory portions are consistent in the open arena. It then scans the open arena's directory and ensures there is an index-entry for every data block. Because the server updates the index lazily after a PSTORE, a fair number of unwritten index entries and hence out-of-date buckets may exist. Finally, *bsrv* process the temporary log by PSTOREing all its blocks to the permanent log.

5 Performance

Our primary goals in testing our system were to ensure that the added security benefits of the SUNDR system do not come at too dear a price. One might expect that SUNDR's consistency protocol and frequent use of cryptographic primitives would drastically reduce file system responsiveness. Instead we have found that our system offers performance competetive with more conventional network file systems.

In this performance analaysis, we compare SUNDR's overall performance to NFS versions 2 and 3. We report the performance of the older NFS 2 protocol because it writes data through to disk on file closes, more closely resembling SUNDR's durability guarantees. We also perform microbechmarks to better understand our

application-level results, and to support our claims that our block server outperforms a Venti-like architecture in our setting.

5.1 Experimental Setup

We carried out our experiments on three 3Ghz Pentium IVs running FreeBSD 4.7. All machines are connected with 100Mbit, full-duplex, switched Ethernet. We measured TCP throughput between clients at 11.21Mbyte/sec, and round-trip times at 0.110 msec. All machines have 3GB of RAM and Seagate Cheetah 18GB SCSI hard drives, which spin at 15K RPM. We used four of these drives in our block server as index disks, and added 2 Western Digital Caviar 180GB 7200 RPM EIDE hard drives for temporary and permanent log disks.

Our cryptographic implementations use GNU Multi-precision Library version 4.1.2 for large integer arithmetic. We collected our microbenchmark timing information using standard *gettimeofday* calls, hence we expect small time dilations on all of our results. To measure user-level application results in Section 5.2.4, we used the standard *time* utility in Unix.

5.2 Microbenmarks

We perform a series of micorbenchmarks on both the SUNDR block server, the SUNDR client, and the combined system.

5.2.1 SUNDR Block Server

Our SUNDR block server currently can store up to 175GB of data, but in our experimental setup, we partitioned our index disks so that we could comfortably accommodate up to 2TB of data. That is, for every 8K block stored on the server, we require approximately 64 bytes of space on the index disks. An index entry only takes up around 32 bytes of disk space, but we would expect performace to degrade if the index becomes more than half full. Thus, our index array must have a capacity of $2^{41}2^{6}/2^{13}$ or 16GB to meet out storage needs. Given our modest 4 disk index array (Venti uses 8), we need only use 4GB of each disk to accommodate the index.

We have tuned many other block server parameters to optimize performance, and scalability. For instance, we allow SUNDR clients to make interleaved asynchronous requests to the server but only with an available a maximum window of 40 RPCs. This is both for flow control and fairness. We have set our dereference window to 60 seconds unless otherwise noted. For the purposes of the microbenchmarks, we have turned off the block cache and have set the index entry cache to 100,000 entries. Our temporary log is 720MB and was not overrun in the course of experiments. In terms of 100 Mbit Ethernet, the SUNDR block server can process the temporary log as fast as blocks can be sent over the network.

	Rabin		Esign	
	1024 bits	1280 bits	2048 bits	6000 bits
Sign	3656	6424	169	695
Verify	27	34	120	575

Figure 7: Sign and verify times for Rabin and Esign Signatures, in μsecs

Figure 6 shows our measurements of the block server's throughput. STORE outperforms PSTORE by 106%. For batches of data that can be transmitted in an amount of time less than the dereference window, the write throughput is limited only by the network bandwidth. For very large block batches, the server will start to process the temporary log while the client is still transmitting data. The server processes the temporary log at 11.72 Mbytes/s while never accessing the underlying disk. However, the operation does place demands on the CPU, thereby restricting server bandwidth available to SUNDR clients. Indeed, we see a 17% decrease in write throughput while the server is processing the temporary log. Even under these conditions, STORE still outperforms PSTORE by 72%.

In terms of write latency, STORE performs siginficantly better than PSTORE, responding to RPCs in 23% of the time it takes for a PSTORE under quiescent conditions, and 29% when simultaneously processing the temporary log.

Our block server does not perform well when asked to RETRIEVE randomly scattered blocks. Note that the Index LRU cache is of little help. The bottleneck is the IDE log disk, which is slow to make random seeks. This weakness is not particular to our block server; we would see the same behavior in Venti. In the context of SUNDR, slow random RETRIEVEs should not affect overall system performance if the client aggressively caches blocks and reads large files sequentially.

5.2.2 Client Profiling

We next performed a high-level profile of the SUNDR client, so as to better understand our other micro and macrobenchmarks. Figure 7 shows the performance of our underlying signature scheme in comparison to a a more conventional scheme – Rabin Signatures [20]. We use 2048 bit Esign keys in SUNDR, which can be broken with a work factor greater than 2^{80}. In other words, an adversary with infinite storage could more easily find a SHA1 hash collision that break a 2048 bit Esign key. We also provide

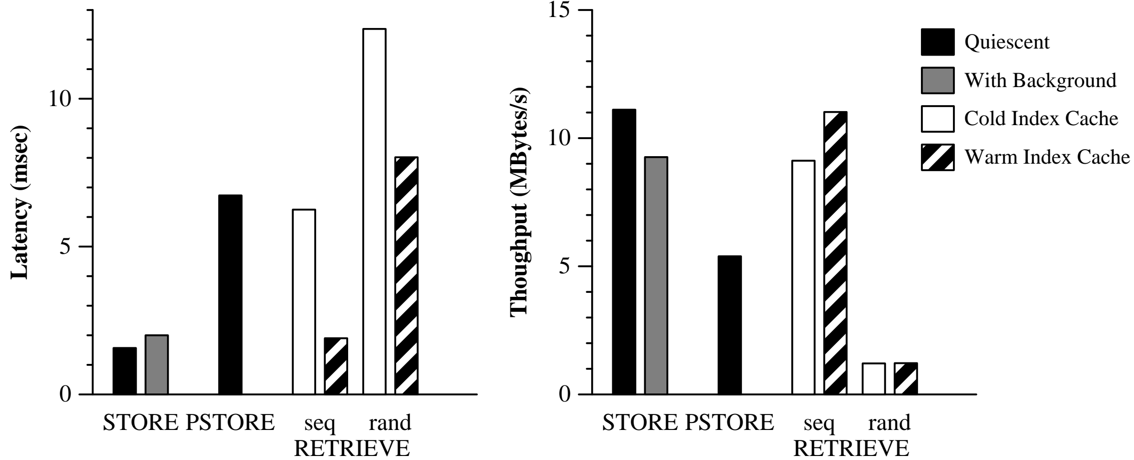

Figure 6: Throughput and Latencies for the SUNDR block server. Throughputs were collected over 10,000 asynchronous operations on 8k blocks, with 40 outstading RPCs. Latencies were collected over 5,000 back-to-back synchronous operations on 8k blocks. The block cache was cold in all experiments.

benchmarks for 6000 bit keys, which can be broken with a work factor of 2^{128} [12].

Indeed, the cost of signing and verifying messages is cheap compared to the latencies of various RPCs along the critical path. Figure 8 shows the RPC timelines of the file operations used in the LFS small file benchmark [23]. A **create** operation consists of synchronous calls to *open*, *write* and *close* system calls. SUNDR's semanitcs guarantee that the data is securely stored on the block server when *close* returns. The *write* operation is handled locally at the **xfs** layer and does not involve any remote RPCs or cryptographic operations. A **read** operation consists of synchronous calls to *open*, *read* and *close*. In this experiment, almost all of the expensive operations are handled in the *open* call. Since **xfs** does not currently support chunked file retrieves, SUNDR fetches the entirety of the file at the open call. In our experiments, all file data is cached on the client side, so no RETRIEVE calls to the block server are needed. Finally, an **unlink** operation consists only of a system call to *unlink*. In this figure, note that COMMIT is not on the critical path of any SUNDR system calls and therefore should not affect overall latency.

Figure 8 shows our prototype's efforts to interleave RPCs and to delay those not on the critical path until after returning to the user-level caller. In particular, system calls do not wait for COMMIT RPCs to return. Moreover, even in small writes, the client sends data blocks to the block store concurrently, and waits for them to return only in the case of *close* and *fsync*. This increases concurrency without compromising our consistency guarantees.

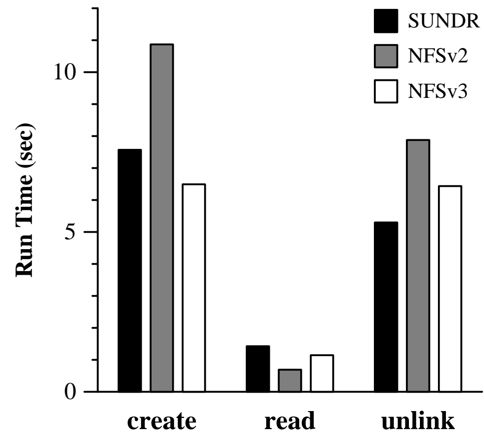

Figure 9: LFS Small File Benchmark. 1000 operations on files with 1k of random content.

5.2.3 LFS Small File Benchmark

Next, we ran our system using the LFS small file benchmark and measured end-to-end performance. Before doing so, we made a slight adjustment to the benchmark suite, and randomized the contents of the files written to disk. To write equivalent files would give SUNDR's block architecture an unfair advantage. In this and other benchmarks, we compare SUNDR to NFSv2 and NFSv3. Indeed, NFSv3 offer better performance, but NFSv2's semantics guarantee that a client's data has been written to stable storage on close. Since SUNDR offers the same guarantee, we believe a comparison to NFSv2 is a fairer one; we include results for NFSv3 regardless.

Figure 9 details our results. SUNDR significantly outperforms NFSv2 in the **create** and **unlink** stages of the

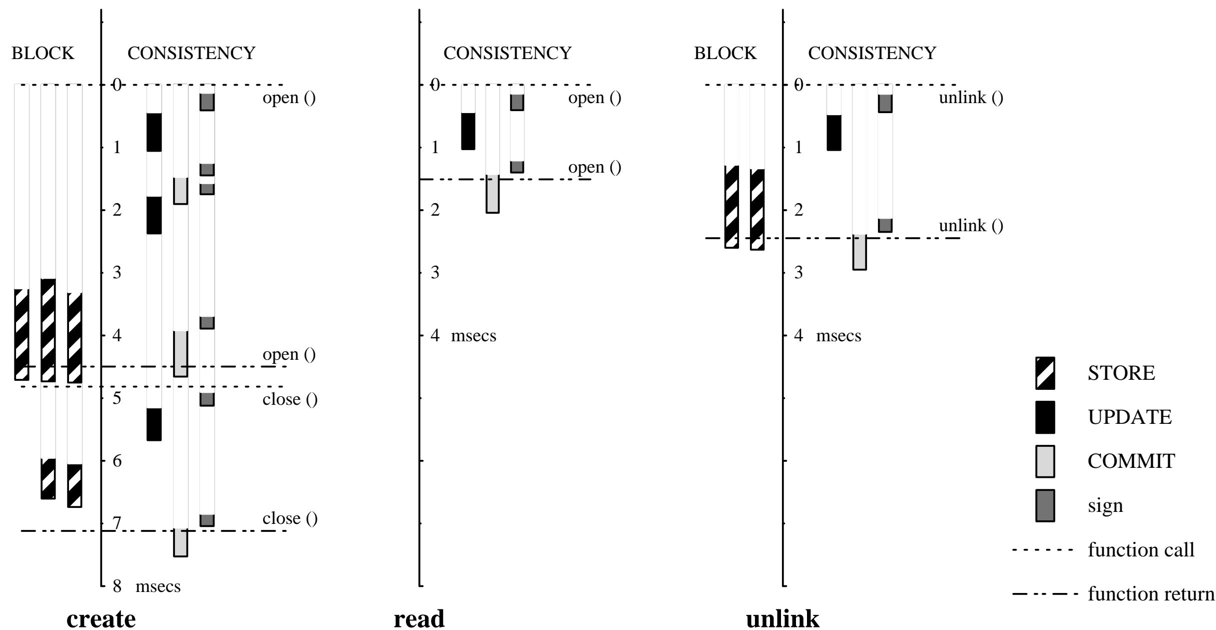

Figure 8: Timeline for typical LFS small benchmark operations, in msecs. Calls on the left side of the axis are RPCs to the block server. Calls on the right side are involved with file system consistency, and are either RPCs to the consistency server or local signatures. All files are 1k.

experiment — by 30% and 18% respectively — thus is competetive with NFSv3. SUNDR is noticably slower in the **read** phase, primarily because of latency introduced by the consistency protocol. In particular, *open* must wait for UPDATE to return before returning to the caller.

5.2.4 Application-Level Benchmarks

Figure 10 shows SUNDR's performance in copying, uncompressing, untarring, configuring, compiling, installing, and removing the Apache webserver distribution (`apache_1.3.27.tar.gz`). We first note that during the experiment, *bsrv* processed 16,872 8k data blocks, of which 8,023 (48%) were dereferenced before they could be committed to the permanent log. The configure, compile and untar operations in particular involve many temporary files and sequential changes to filesystem metadata. Thus, in these benchmarks, clients frequently DECREF blocks within milliseconds of STOREing them.

SUNDR is competetive with NFSv2 in copying the source archive from local disk to the SUNDR partition; its completes this operation 110 msecs, while NFSv2 does the same in 87 mescs. NFSv3 actually performs slower at this stage, completing the *cp* in 199 msecs. SUNDR's performance is similar to NFSv2's in the unzip, untar, compile and cleanup portions of the build process. The *config* and *install* segments clearly present a problem for us. We see several explanations for this. Most intuitively, many operations in these procedures reference the file system three times: once to load an executable shell script in the build directory, once to read an input file, and once to store to an output file. Hence, we see a three-fold magnification of performance bottlenecks. Moreover, our current implementation of SUNDR does not handle concurrent **xfs** upcalls optimally. Our prototype sometimes serializes these requests for convenience. Future versions of SUNDR will seek to improve small file reads, and upcall concurrency to perform more comptetively in this and similar settings.

6 Related Work

Recently, there has been growing interest in peer-to-peer storage systems comprised of potentially untrusted nodes. Farsite [3] investigated the possibility of spreading such a file system across people's unreliable desktop machines. Several new distributed hash tables such as Chord [26] and Pastry [24] show the potential to scale to millions of separately administered volunteer nodes, with CFS [6] layering a read-only file system on top of such a highly distributed architecture. These systems could potentially replace the SUNDR block server, provide automated offsite backup, or be used to coordinate cooperative caching amongst mutually distrustful clients.

A number of file systems in the past have used cryptographic storage to keep data secret in the event of a

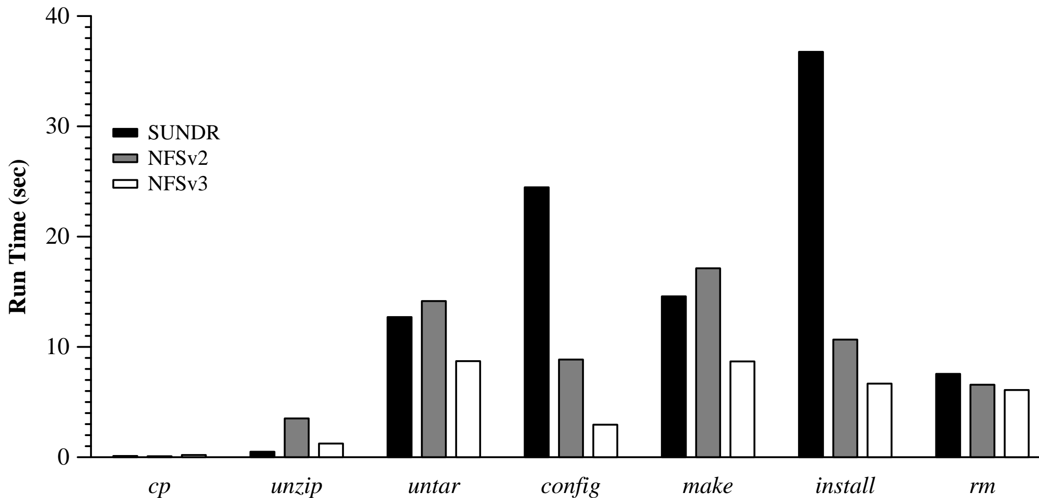

Figure 10: Installation procedure for `apache_1.3.27.tar.gz`.

server compromise. The swallow [21] distributed file system used client-side cryptography to enforce access control. Clients encrypted files before writing them to the server. Any client could read any file, but could only decrypt the file given the appropriate key. Unfortunately, one could not grant read-only access to a file. An attacker with read access could, by controlling the network or file server, substitute arbitrary data for any version of a file.

CFS [2] allows users to keep directories of files that get transparently encrypted before being written to disk. CFS does not allow sharing of files between users, nor does it guarantee freshness or integrity of data. It is intended for users to protect their most sensitive files from prying eyes, not as a general-purpose file system. Cepheus [9] adds integrity and file sharing to a CFS-like file system, but trusts the server for the integrity of read-shared data. SNAD [16] can use digital signatures for integrity, but does not guarantee freshness. PFS [25] is an elegant scheme for checking the integrity of a file system stored on an untrusted disk. With minor modifications, PFS could make strong freshness guarantees. However, PFS is really a local file system designed to reside on untrusted, potentially remote disks. Users on multiple clients cannot simultaneously access the same file system. Plutus [13] is a cryptographic storage system in which clients use public key encryption to hide the contents and properties of their files from the server. Plutus does not guarantee file authenticity or freshness; clients trust the server to store their data properly.

The Byzantine fault-tolerant file system, BFS [4], uses replication to ensure the integrity of a network file system. As long as more than $2/3$ of a server's replicas are uncompromised, any data read from the file system will have been written by a legitimate user. SUNDR, in contrast, does not require any replication or place any trust in machines other than a user's client. If data is replicated in SUNDR, only one replica need be honest for the file system to function properly. However, SUNDR provides weaker freshness guarantees than BFS, because of the possibility that a malicious SUNDR server can fork the file system state if users have no other evidence of each other's on-line activity.

Pond [22] is a prototype of the OceanStore system, a large-scale data store that offers certain consistency and privacy guarantees. Pond trusts an "inner core" of Byzantine fault-tolerant servers to reliably order user file operations. Pond uses "heartbeat" messages, distributed throughout a peer-to-peer network of "secondary replicas," to convey a loose notion of file freshness to its clients. If a client desires a stronger assurance of a file's freshness, it must ask the inner core of servers to sign a cryptographic nonce. This is very expensive for the servers, considering they must generate a threshold signature, which the authors report consumes an order of magnitude more computational time than a conventional signature. Pond cannot very well increase the number of core servers to distribute load amongst more servers, as this will only make matters worse: the complexity of its consistency protocol is quadratic in the number of primary servers. In SUNDR, by constrast, the consitency server only verifies signatures. File consistency is provably guaranteed by the clients, who sign with a standard and computationally practical signature scheme.

SUNDR uses hash trees, introduced in [15], to verify a file block's integrity without touching the entire file system. Duchamp [7], BFS[4], SFSRO [10] and TDB [14] have all made use of hash trees for comparing data or checking the integrity of part of a larger collection of data.

7 Conclusions

SUNDR is a general-purpose, multi-user network file system that never presents applications with incorrect file system state, even when data is stored on an untrusted server. SUNDR's protocol provably guarantees data integrity and consistency without assuming any on-line trusted parties. By reducing the amount of trust required for file servers, SUNDR both increases people's options for managing data and improves their file security. Performance measurements of our implementation show that while SUNDR's security comes at a cost, the protocol is still practical. Moreover, we suggest several possible optimizations that are not yet part of the implementation.

References

[1] Anonymous. Citation omitted for blind review. In *Some peer-reviewed conference*, 2002. The full version appeared as a technical report.

[2] Matt Blaze. A cryptographic file system for unix. In *1st ACM Conference on Communications and Computing Security*, pages 9–16, November 1993.

[3] William J. Bolosky, John R. Douceur, David Ely, and Marvin Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop pcs. In *SIGMETRICS*, pages 34–43, 2000.

[4] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, pages 173–186, New Orleans, LA, February 1999.

[5] Jr. D. Stott Parker, Gerald J. Popek, Gerard Rudisin, Allen Stoughton, Bruce J. Walker, Evelyn Walton, Johanna M. Chow, David Edwards, Stephen Kiser, and Charles Kline. Detection of mutual inconsistency in distributed systems. *IEEE Transactions on Software Engineering*, SE-9(3):240–247, May 1983.

[6] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with cfs. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 202–215, Chateau Lake Louise, Banff, Canada, October 2001. ACM.

[7] Dan Duchamp. A toolkit approach to partially disconnected operation. In *Proceedings of the 1997 USENIX*, pages 305–318. USENIX, January 1997.

[8] FIPS 180-1. *Secure Hash Standard*. U.S. Department of Commerce/N.I.S.T., National Technical Information Service, Springfield, VA, April 1995.

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

[10] Kevin Fu, M. Frans Kaashoek, and David Mazières. Fast and secure distributed read-only file system. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, 2000.

[11] Kevin Fu, M. Frans Kaashoek, and David Mazières. Fast and secure distributed read-only file system. *ACM Transactions on Computer Systems*, 20(1):1–24, February 2002.

[12] Louis Granboulan. How to repair esign. Cryptology ePrint Archive, Report 2002/074, 2002. `http://eprint.iacr.org/`.

[13] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus: Scalable secure file sharing on untrusted storage. In *2nd USENIX conference on File and Storage Technologies (FAST '03)*, San Francisco, CA, April 2003.

[14] Umesh Maheshwari and Radek Vingralek. How to build a trusted database system on untrusted storage. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, San Diego, October 2000.

[15] Ralph C. Merkle. A digital signature based on a conventional encryption function. In Carl Pomerance, editor, *Advances in Cryptology—CRYPTO '87*, volume 293 of *Lecture Notes in Computer Science*, pages 369–378, Berlin, 1987. Springer-Verlag.

[16] Ethan Miller, Darrell Long, William Freeman, and Benjamin Reed. Strong security for distributed file systems. In *Proceedings of the 20th IEEE International Performance, Computing, and Communications Conference*, pages 34–40, Phoenix, AZ, April 2001.

[17] Athicha Muthitacharoen, Benjie Chen, and David Mazières. A low-bandwidth network file system. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 174–187, Chateau Lake Louise, Banff, Canada, October 2001. ACM.

[18] Tatsuaki Okamoto and Akira Shiraishi. A fast signature scheme based on quadratic inequalities. In *Proceedings of the 1985 IEEE Symposium on Security and Privacy*, pages 123–132, Oakland, CA, April 1985.

[19] Sean Quinlan and Sean Dorward. Venti: a new approach to archival storage. In *First USENIX conference on File and Storage Technologies (FAST '02)*, Monterey, CA, January 2002.

[20] Michael O. Rabin. Digitalized signatures and public key functions as intractable as factorization. Technical Report TR-212, MIT Laboratory for Computer Science, January 1979.

[21] David Reed and Liba Svobodova. Swallow: A distributed data storage system for a local network. In A. West and P. Janson, editors, *Local Networks for Computer Communications*, pages 355–373. North-Holland Publ., Amsterdam, 1981.

[22] Sean Rhea, Patrick Eaton, and Dennis Geels. Pond: The OceanStore prototype. In *2nd USENIX conference on File and Storage Technologies (FAST '03)*, San Francisco, CA, April 2003.

[23] M. Rosenblum and J. Ousterhout. The design and implementation of a log-structured file system. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 1–15, Pacific Grove, CA, October 1991. ACM.

[24] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Middleware*, 2001.

[25] Christopher A. Stein, John H. Howard, and Margo I. Seltzer. Unifying file system protection. In *Proceedings of the 2001 USENIX*. USENIX, June 2001.

[26] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM '01 Conference*, San Diego, California, August 2001.

[27] John D. Strunk, Garth R. Goodson, Michael L. Scheinholtz, Craig A.N. Soules, and Gregory R. Ganger. Self-securing storage: Protecting data in compromised systems. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, San Diego, October 2000.

[28] Assar Westerlund and Johan Danielsson. Arla—a free AFS client. In *Proceedings of the 1998 USENIX, Freenix track*, New Orleans, LA, June 1998. USENIX.