# SUNDR: Secure Untrusted Data Repository

Jinyuan Li

May 7, 2003

## 1 Introduction

Since NFS was invented in the mid 80's, it has been widely used in all areas. However, not like its traditional local storage system counterpart, NFS is more easily attacked while data is transferred over the wire or stored on remote servers. Since then, a lot of research works have been carried out to improve the security of NFS. Most of them placed their trust on servers, i.e. they assume servers always exhibit fail-and-stop behaviors. Unfortunately, this assumption does not always hold for the following reasons. First, many people have access to data who don't need it. Any people with administrator access, even though he has no role in the maintain of file servers, can tamper files or make modifications without being detected. Secondly, stored with all data, severs are more attractive targets for attackers. They can easily install a lot of available rootkits to replace core operating system utilities with altered versions the open back doors and conceal the evidence of security breach.

Hence, we have designed and implemented a secure network file system called SUNDR that gives users integrity and consistency guarantees even when malicious parties control servers. In order to facilitate users to detect misbehavior, SUNDR splits its server functionality into two mutually untrusted components, a block server and a consistency server. The block server holds all file data and most meta data. Without interpreting any data structure, it presents a simple and clean interface for clients to store variable-sized data blocks and later retrieve them by cryptographic hash.

The consistency server implements a novel protocol to guarantee close-to-open consistency. The protocol roughly consists of exchanging each user's version structure, including the digest of each user's file system data, through which any user can retrieve its whole content and verify its integrity. We have proven the security of the protocol under the basic cryptographic assumptions. Without somehow producing valid signed messages under user's public key or encountering cryptographic hash collides, an attacker can not tamper user's file without being detected – even given control of the servers and network. Despite these security guarantees, SUNDR performs within a reasonable factor of existing insecure network file systems.

The next section gives the detailed design of SUNDR file system. The following two sections describe SUNDR's implementation and performance evaluation. Section 5 describes the related work, and Section 7 concludes.

# 2 System Architecture

In this section, we briefly describe the architecture of SUNDR. Unlike most of previous works, SUNDR does not assume servers to be trusted, so additional care is taken to design its architecture.

## 2.1 System model and architecture

In traditional network file systems, end-users are *untrusted* and control their own clients. In contrast, servers are *trusted* and reflect only authorized operations. All communications between servers and clients are mutually authenticated, therefore for clients, they know the response from server is authentic.

In SUNDR, both the block server and the consistency server are untrusted, Particularly, we aim to provide integrity and consistency in the current design and implementation. Secrecy is not addressed in this work now, though we have carried some research work on it. In next two subsections, we will describe how SUNDR provides the two guarantees separately.

## 2.2 Integrity: Digest of file system

The idea here for ensuring integrity of SUNDR file system is not new, and has been used in several previous works. This goal is achieved by generating the digest of the whole file system. First, each file/directory (Directory entries map file names onto {principal, i-number} pairs) is chucked into fixed-sized (e.g. 8k) blocks, each of which then is hashed. Then all these hash values are organized into a SUNDR *virtual inode* and this virtual inode is hashed. We call this hash value file's *i-hash*. Secondly, a per-principal data structured called *i-table* maps each active i-number to a collision-resistant SHA-1 hash of file's i-hash.

SUNDR forms a cryptographic tree 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 retrieve any block of any file in the i-table and verify its integrity. Thus, securely retrieving file system boils down to the problem of first obtaining the latest i-handle of each user, then retrieving any needed data block by its SHA-1 hash. The latter functionality is conceptually simple to implement, while the former one needs a somewhat complex consistency protocol.

## 2.3 Freshness: a novel consistency protocol

We describe the SUNDR consistency protocol step by step. First we begin with a simplified bare-bones protocol that provides consistency with a global lock. Then we will show how to improve concurrency by removing this restriction.

### 2.3.1 Consistency protocol barebone

In order to achieve close-to-open consistency in traditional network file systems, SUNDR embeds i-handle in digitally signed *version structure* through which clients can verify the file system has been delivering close-to-open consistency. As long as users see the the effects of each other's operations, they are guaranteed to be getting close-to-open consistency. Otherwise there remains a possibility that an attack has been carried out on this server. However, if there does exist such a attack, the malicious server can no longer present that user any other's on-line activity. This kind of attacks can be easily detected by out-of-band communications.

The key part in version structure is vection vectors. Each user or group has a version number. A version structure contains the latest number of every user and group version number of every user and group at the time of the operation it reflects. Two version structures are compatible if one contains at least as high a version number of every user or group as the other for all principals. When creating/updating a version structure, users increment both their own version numbers, the version number of any group whose i-table they are modifying and all other's version numbers according to all other's latest version structures retrieved from the server. Both opens and closes requires 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. While the bare-bones protocol is simple, significant change to protocol is needed when adopting the protocol to concurrent updates. SUNDR users need to compute and sign their version structure without accessing all other users' latest version structures.

### 2.3.2 Improve concurrency

To allow concurrent updates, SUNDR consistency protocol consists of two RPCs, UPDATE, COMMITVST. Both RPCs contain messages signed under users' public keys. Intuitively, UPDATE declares a user's intent to perform some operation to the file system. It returns the precise state of the file system at the time update is to have take place and all the pending operations (declared in all other users' UDPATE) at that time. COMMITVST contains the user's version structure, certifying the order that the server has assigned to the operation. (The order is defined as the arriving time of UPDATE to the consistency server.)

In more detail, whenever 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 called *delta* the user wishes to make to the file system. If the users does not make change to the file system, the list of deltas is empty.

Since the effect of apply a single delta to the file system is deterministic. so the 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 for every i-handle, plus a list of signed update certificates for operations not yet reflected in version structure list. We call the collection of version structures returned *version structure list*, or VSL, and a list of update certificates the *pending version list*, or PVL. An honest server totally orders file system operations by the time which it receives the corresponding update certificate.

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 from 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 structures based on the VSL and PVL it has received. This version structure contains the new user's ihandle, any group's i-handle the user has updated in this round, the current latest version number of every user and every group, and a list of pending operation in the PVL, including for each pending operation a SHA-1 hash of its forthcoming version structure without i-handle. When computing group i-handles, the client must reflect any previous modification to the group in the PVL.

Interested reader can refer to [] for a complete proof the security correctness of the protocol. At a higher level, however, the intuition for the security comes as the following, in traditional distributed file systems, version vectors are used to detect update conflicts. However, in SUNDR, the server can order all the events with it discretion, upon which all users agree with, so there should be no update conflicts and the server can deliver all users close-to-open consistency. If the server fails to deliver this consistency, users' version vectors can't be total ordered as in traditional distributes file system.

## 2.4 Deal with catastrophe

We only discuss the crash recovery for the consistency server and clients here, The issue for the block server could be found the original paper.

### 2.4.1 Crash recovery for the consistency server

First of all, the consistency server needs to log all the latest version structures. In order to avoid the overhead, we have another lightweight backup server that resides behind the consistency server and two backup RPCs. When the consistency server receives an version structure, it calls BACKVST, passing the version structure and its corresponding user id.(or user's public key[1]) The backup server stores this version structure with the id as its index.(So, it can be implemented from the block server with little modification) When the consistency server recovers, it retrieves all the latest version structures by RECOVST, which takes a user id, and returns its version structure.

However, this is not enough. When the server crashes after any client sends UPDATE , but before COMMITVST, it should remember the UPDATE. Fortunately the consistency server can retrieve PVLs from all clients, from which the server can reconstruct the PVL. If the client also crashes at the same time and its UPDATE does not appear in the PVL, it is okay for the system to forget it.

### 2.4.2 Crash recovery for clients

Since the version structure is stored on the consistency server, the issue for clients to recover from crash is greatly simplified. The client only needs to log the current version number on its local disk. When it recovers later, it retrieves its latest version structure from the consistency server and matches the number with the version number embedded in the version structure.

[1]If we use a P2P storage or else instead of a dedicated backup server.

### 2.5 Future works

There are several possible extension to the current work. First, we can adopt LBFS's chucking algorithm to reduce bandwidth requirement. This has the potential of reducing response time and server load as showed in Pastiche. Second, data secrecy concern is addressed, and this has been explored by Luigi in last summer. Third, we can improve the availability, either by replicating block server, or co-operative cache management. However, this doesn't address the scalability of the consistency server. Fortunately, since the overhead with the consistency server is much lower than that of the block server, it won't be the bottleneck of the whole system before we successfully replicate the block server.

## 3 Implementation

We only sketch how we implement SUNDR here, more details can be found in []. Fig **??** shows the overall architecture of the SUNDR implementation. The client is implemented at user-level, using a modified version of xfs device driver from ARLA on top of a slightly modified FreeBSD kernel. The whole system is implemented with libasync provided by SFS.

## 4 Performance Evaluation

### 4.1 Experiment Setup

We carry out the experiment on three 3Ghz Pentium IVs running FreeBSD 4.7. All machines are connected 100Mbit, full-duplex, switched Ethernet.

### 4.2 LFS small benchmark

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 server an unfair advantage. In this and other benchmarks, we compare SUNDR to NFS v2 and v3. Indeed, NFSv3 offers 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 1 details our result, SUNDR significantly outperforms NFSv2 in the create and unlink stages of experiment - by 30% and 18% respec-

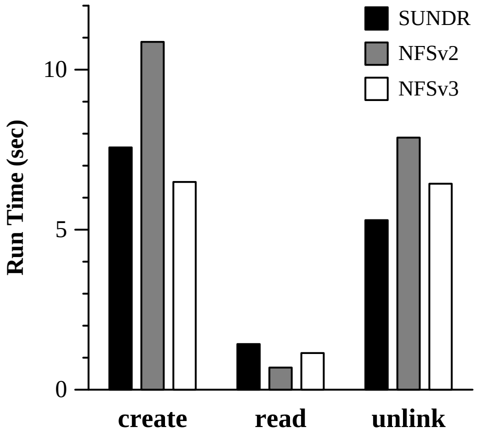


Figure 1: LFS Small File Benchmark. 1000 operations on files with 1K of random content.

tively - thus is competitive with NFSv3. SUNDR is noticeably slower in the read phase, primarily because of the latency introduced by the consistency protocol.

### 4.3 Application-level benchmark

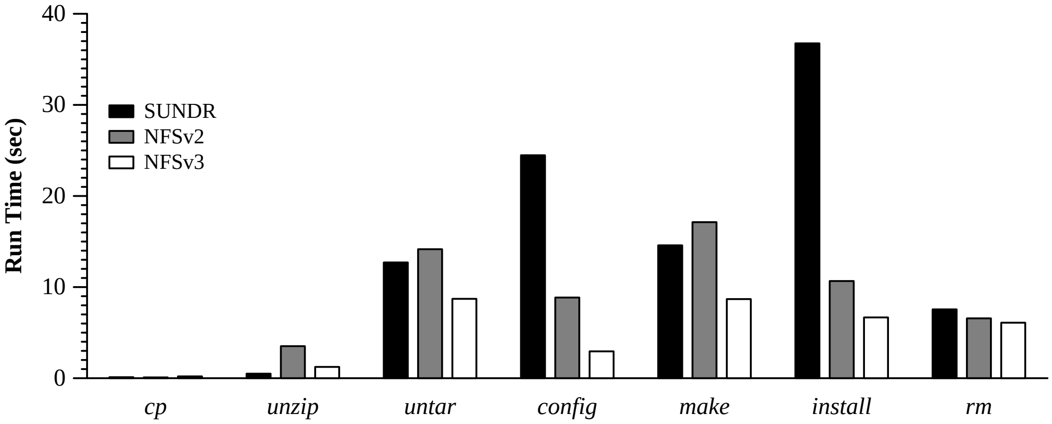


Figure 2: Install procedure for `apache_1.3.27.tar.gz`.

Figure 2 shows SUNDR's performance in copying, uncompressing, untarring, configuring, compiling, installing and removing the Apache web server distribution.

SUNDR is competitive with NFSv2 in copying the source archive from the local disk to the SUNDR partition. 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 by far.

# 5 Conclusion

SUNDR differs from all previous works in that it does not assume any trusted component on servers. It is a general-purpose, group readable-writable file system that provides trusted end-users with integrity and consistency guarantees. It achieves this goal by a provably secure consistency protocol. Despite the complexity of the consistency protocol and some expensive cryptographic operations involved, SUNDR performs within a reasonable factor of the current insecure NFS. Moreover, several possible optimizations might further improve the performance.

# 6 References

Omitted here.