Secure Untrusted Data Repository (SUNDR)
Li, Krohn, Mazieres, and Shasha

Why did I assign this paper?

- very different threat model, one that is becoming more and more relevant because of cloud. still learning the fundamental limitations of what can be done with integrity, consistency, confidentiality, and performance in this world.
    - fork consistency is one such thing
- two cool tricks to add to your systems toolbox
    - hash trees. let you cheaply track integrity via hashing of a large system, without paying the cost of rehashing entire system on every update.
    - venti-like block store. lets you validate integrity of blocks by naming blocks by hash, and simultaneously gives you single-instance-store properties.

Motivation

- Cannot trust that programs will remain uncompromised, but we need to find a way to get work done anyway.
- Here, file server might be compromised, but still need clients to be able to verify the integrity of their data

Q: what is the threat model in this paper? (what kinds of attacks are they worried about?)

- compromised server corrupting data? [this paper]
- compromised server leaking data? [no – how would you defend? encrypt files]
- compromised, but authorized, client injecting bogus data? [no – how would you defend? can't really – just need ability to audit and rollback. LOCKSS.]
- repudiation? (i.e., I disavow knowledge of a modification.) [yes, but not goal.]
- availability attacks? [not really – need replication, offline access]

Q: how does this compare against byzantine fault tolerance models?

- BFT: replicate file server across 3F+1 machines, assume no more than F replicas are simultaneously faulty
- How? Cryptography.

Strawman

- Each authorized user has pub/private key pair
    - and FS administrator has superuser pub/priv keypair, allowing them to authorize users
    - file server needs NO keys
- File operations are signed by the authorized key; signature convolves file operation with the history of the filesystem that preceeds it – essentially fixing in stone that history.
    - Mental model (implementation differs): file system stores history of operations
    - An authorized user that interacts the file system state remembers their last operation and its signature
        - So, when that user comes back for another operation, it can verify the history of the file system up to and including its own last operation
        -

fetch(f_2) user A sig	mod(f_3) user B sig	fetch(f_3) user A sig	mod(f_2) user A sig	fetch(f_2) user B sig

- To fetch a file, user:
    - acquires global lock
    - downloads entire history of filesystem
    - validates each users' most recent signature
    - checks that its most recent operation is in the downloaded history
    - traverses history to construct FS, validates each operation is permittable

- Invariant:
    - **Client A accepts history of a filesystem iff it observes its last operation in that history**
    - Implications:
        - Server cannot introduce arbitrary operations; it can only drop last operation.
            - Cannot introduce "gaps" in history
            - Cannot forge new operations
        - Possible for server to "fork" history on by dropping last operation
        - But by doing so, server must make sure forks are permanent – otherwise clients will catch on
        - Time stamp box – every N seconds touch a file – bounds fork partition time before discovery.

Walk through properties this gives you, and fork attack:

user A:	fetch(f_2) user A sig	mod(f_3) user B sig	fetch(f_3) user A sig	mod(f_2) user A sig

user B:	fetch(f_2) user A sig	mod(f_3) user B sig	fetch(f_3) user A sig	fetch(f_2) user B sig

Serialization in strawman

Do this (send entire file system history on every access, verify and sign entire history). Single lock to get serialized operations.

Q: Why do you need to serialize?
- because signature has to be over entire previous history. Need to fix order of mutations in order to know what that history is. So, for an op to commit, it needs to know all preceeding committed ops as well as mutation that will happen. Not safe to "commit" op until writer has verified history – everybody logjams.

Q: why do "fetches" need to be in history?
- want to commit user's view of the system every time they get a new view, so entangle view on fetches as well as updates!
- bad news for read-mostly clients: state grows with # operations.

Serialized SUNDR

- Cannot store/ship entire history, or afford "giant lock".
- Fix "entire history" problem with hash trees
- Instead of signing full history, you:
  - Aggregate file state of user/group into single hash value using hash tree (i-handle)
  - Tie i-handles to each other using version vectors
  - Signatures are of user's ihandle and set of version vectors
- Still have fetch operations causing changes, but bounded to one update per client, rather than entire fetch history in file system history log
  - basically carries forward version-vector like contour of what each user sees. FS state grows with # files, # users, but not # operations.

Main data structure:

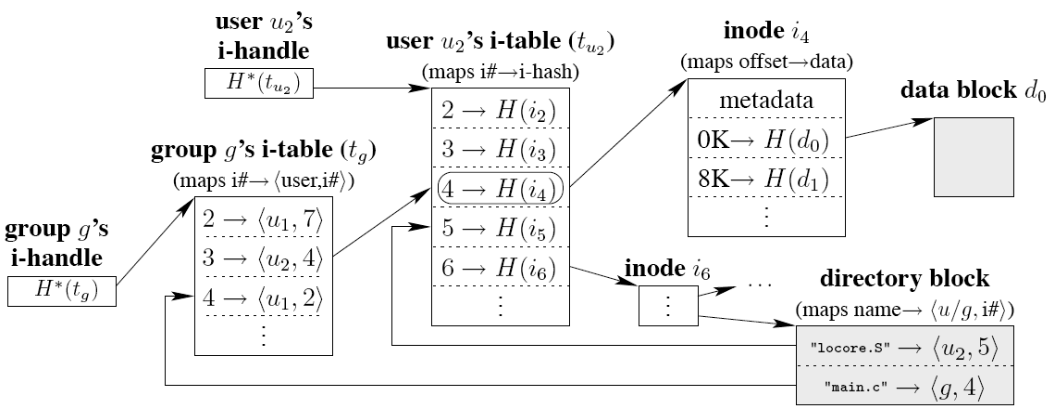


Figure 2: User and group i-handles. An *i-handle* is the root of a hash tree containing a user or group *i-table*. (H denotes SHA-1, while H^* denotes recursive application of SHA-1 to compute the root of a hash tree.) A *group i-table* maps group inode numbers to user inode numbers. A *user i-table* 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.

Pretty much like FFS, except with:

- Hash trees
- Extra level of indirection between inumber and inode (why? I think to allow mutations of file without requiring directory block mutations.)
- Group tables: to allow groups of people the right to modify files. Group tables point to user table – last writer.
    - Q: why don't group i-tables contain hashes? (yet user itables do?)
    - A: group itables point to user i-tables, so that users can modify same file multiple times without having to update the group i-table. If had hash, would need to update group i-table.
    - Q: what are implications of not having hash in group i-table?
    - A: Not much. Group declares a list of files, and who last modified the file. Doesn't declare anything about file content – go a separate path through. No need to entangle.

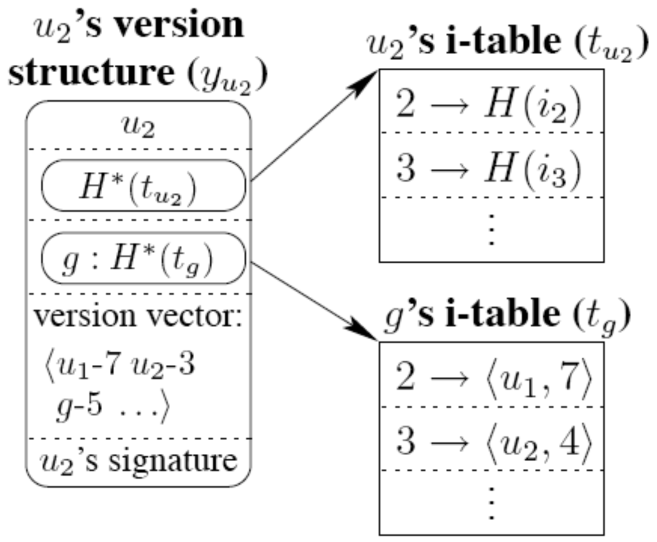
Versioning structure: VSL (version structure list).

- Is that user's view of state of filesystem it knows about.
- Is a list of version structures.

A version structure from a user u contains:

- Hash of user's itable – state of files that it has permission to modify (owns).
- Hash of group itables – state of files that groups it's a member of can modify
- Version vector: last known version vector for all files/groups in system
    - State of filesystem as it knew it
- Signature of all of the above

VSL captures state of filesystem

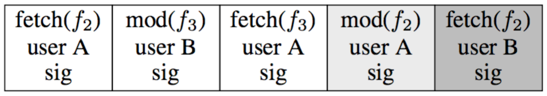

u_2's version structure (y_{u_2})
u_2
$H^*(t_{u_2})$
$g : H^*(t_g)$
version vector:
$\langle u_1$-7 u_2-3 g-5 $\ldots\rangle$
u_2's signature
u_2's i-table (t_{u_2})
$2 \rightarrow H(i_2)$
$3 \rightarrow H(i_3)$
$\vdots$
g's i-table (t_g)
$2 \rightarrow \langle u_1, 7\rangle$
$3 \rightarrow \langle u_2, 4\rangle$
$\vdots$

On read:

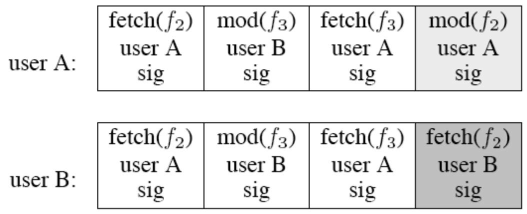
1. Fetch VSL (list of version structure)
2. Create new version structure by copying its previous ihandle into new version structure. (where did it learn its previous ihandle? It is required to remember its last version structure.)
3. For each principle in the system, copy version number of that principle from that principle's version vector into the new version structure
   $(z[p] \leftarrow Yp[p])$
4. Set $z[u] = z[u] + 1$, and for each g it is a member of, $z[g] = z[g] + 1$
5. Check VSL for consistency, then commit it.
   a. Make sure VSL contains old version structure of the client
   b. Make sure VSL unioned with new version structure has a total ordering over <= (version numbers <= for each)
   =➔ this is what tests for fork attacks:

   imagine this fork attack:

truth:



fetch(f_2) user A sig	mod(f_3) user B sig	fetch(f_3) user A sig	mod(f_2) user A sig	fetch(f_2) user B sig

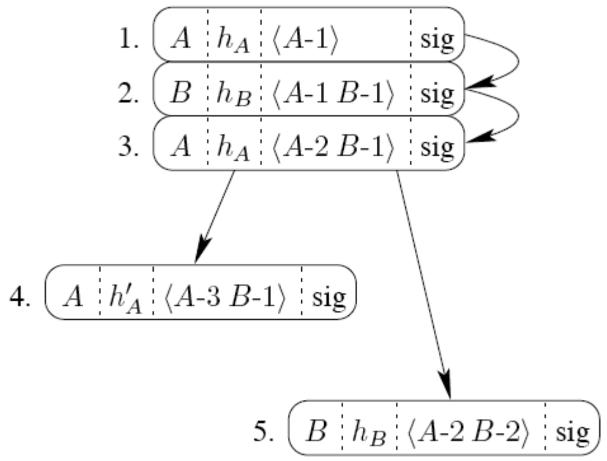


user A:

fetch(f_2) user A sig	mod(f_3) user B sig	fetch(f_3) user A sig	mod(f_2) user A sig

user B:

fetch(f_2) user A sig	mod(f_3) user B sig	fetch(f_3) user A sig	fetch(f_2) user B sig


1. A | h_A | $\langle A\text{-}1\rangle$ | sig
2. B | h_B | $\langle A\text{-}1\ B\text{-}1\rangle$ | sig
3. A | h_A | $\langle A\text{-}2\ B\text{-}1\rangle$ | sig
4. A | h'_A | $\langle A\text{-}3\ B\text{-}1\rangle$ | sig
5. B | h_B | $\langle A\text{-}2\ B\text{-}2\rangle$ | sig

Q: in strawman, user signs entire state of file system – all content, all previous signatures. In serialized, user only signs its belief of other users version vectors, but doesn't sign other users' signatures or content. Why is this OK?

A: strawman is stronger than it has to be. Property is that user should become aware of other users' modifications, so make sure timeline of system goes forward – in other words, need to:

(a) detect fork attack
(b) detect corruption of data that user cares about

strawman also adds: (c) detect corruption of other users' data that I don't care about.

Concurrent SUNDR

- Fix big lock by letting people "predeclare" operations before receiving VSL
- Avoids need to grab lock, download VSL, verify and modify, commit
    - Instead, send intended operations, and receive VSL and set of pending op versions
    - Integrate the two to look for consistency
    - Details subtle but in paper

Implementation

- Block store: indexed by hash
- Consistency server: stores the data structures

Consistency server

- Must do durable store of VSL / PVL before responding to client RPCs
    - Implies file is committed when server writes VSL/PVL to disk, not when RPC response is sent
    - Crash recovery issues?
    - NVRAM to absorb these sync writes

Block store

- Must synchronously store contents before returning, since cannot repair filesystem after crash (no keys)
- Two kinds of disks:
    - Log disk for appending blocks -- fast in spite of sync writes
    - Fast disk for indexing log – small writes, needs low seek time. 15k RPM disk, multiple disks, heavy in-memory cache.

Performance

- Does OK compared to NFS, some wrinkles here and there
    - Randomly issued reads: seek across log
    - Crypto: use fast crypto algorithsm to get out of critical path

General questions:

- tech trends in sundrs favor, or against them?
    - CPU: signatures getting faster
    - sync disk writes: NVRAM, SSDs helping out

- workload trends in sundrs favor, or against them?
    - depends on setting
    - less appropriate for large scale (# of users) deployments
    - makes perfect sense for CVS repositories with low write / commit rates