```
6.824 2021 Lecture 18: Secure Untrusted Data Repository (SUNDR) (2004)

Why are we reading this paper?
  Logical generalization of Frangipani, but
    Strong threat model: fully compromised ("Byzantine") server.
  Powerful techniques for achieving security with untrusted servers.
    Not just file systems.
  Similar ideas show up in decentralized systems, git, blockchains, etc.
    Keybase (acquired by zoom) is directly influenced by SUNDR
  Intersection with 6.858
    Solutions requires signing, hashing, etc.
    [lab 5 in 6.858]

Setting.
  Network file server.
    Imagine cloud storage service, ala Dropbox / Google Drive / ...
    But the server is mostly a block store; the clients implement the file system
    Like Franginpani
  Clients accessing files over the network.
    RPC protocol.
  Server is "Byzantine".
    Fully controlled by the adversary.
    Could respond to client RPCs in any way adversary chooses.
    Could even collude with compromised users.
    .. except constrained by crypto:
       cannot fake signatures from users that aren't compromised.
  Powerful threat model: encompasses wide range of possible real attacks.
    Bugs in server software.
    Administrators that have weak passwords.
    Physical break-ins at data center.
    Malicious or bribed server operators.

Potential goals:
  Confidentiality.
    Adversary cannot get file contents.
  Integrity.
    Adversary cannot trick clients into getting wrong file contents.
  Availability.
    Adversary can't prevent clients from accessing their files.

This paper's focus: integrity.
  Motivating use case: source code for project with many developers.
  Bad if server can insert backdoor into source code,
                can hide security fixes from clients, etc.
  Serious problem.
    Paper mentions Debian server compromised in 2003.
    SourceForge compromised in 2011.
      [ http://sourceforge.net/blog/sourceforge-attack-full-report ]
    Canonical (Ubuntu) compromised in 2019.
      [ https://hub.packtpub.com/canonical-the-company-behind-the-ubuntu-linux-distribution-was-
hacked-ubuntu-source-code-unaffected/ ]

Example scenario: team of developers working on the Zoobar web app.
  Source code stored in shared network file system.
  User A edits auth.py to authenticate users with MIT certificates.
  User B edits bank.py so users transfer real Techcash money.
    Plausible because users are now authenticated with MIT certs.
  User C packages up the source code and deploys it.

  Obvious undesirable outcome: adversary gives arbitrary code to user C.
  Subtle undesirable outcome: bank.py change without the auth.py change!

Naive design: sign file contents.
  Use existing network file system (say, Dropbox) as a starting point.
```

```
    Not expecting any security guarantees from it.
  Each file contains data, Sig(SK_writer, data).
  When user writes to file, client signs with user's key.
  When user reads file, client checks signature.
    Assuming clients know public keys of all users.
  Along the lines of what we might do based on last lecture on messaging.

What could a malicious server do in this naive design?
  Cannot send arbitrary source code to user C.
  Can send one file's data instead of another file's data.
    Possible fix: include filename in the signature.
  Can send the old contents of all files.
  Can selectively send any version of any file.
  Can claim that a file doesn't exist.
  Not good for our Zoobar scenario:
    Adversary could cause C to deploy bank.py without auth.py changes.

Need more sophisticated design for a file system.
  Consistent versions of files: bank.py change requires auth.py change.
  Latest version of files: should not be missing any changes.
  Integrity of directory contents: should not be missing any files.
  Permissions: users might not have permissions to modify every file.

Big idea in SUNDR: log of operations.
  File system state is determined by the log of operations by users.
  Server is responsible for storing this log.
  Clients interpret the log.
  Simplifies thinking about integrity of complex file system state!

Strawman design: section 3.1 from the paper.
  Log entries: fetch or modify, user, sig.
    Signature covers the entire log up to that point.
  Client step:
    Download log (other clients now wait).
    Check the log:
      Correct signatures in each entry, covering log prefix.
      This client's last log entry is present.
    Construct FS state based on logged operations.
    Append its operation and sign new log.
    Upload log (other clients can now proceed).
  Inefficient but simple to reason about.
  Example scenario:
    A: mod(auth.py), sig
    B: mod(bank.py), sig
    C: fetch(auth.py), sig
    C: fetch(bank.py), sig

Crucial that signature covers all previous operations in the log.
  Prevents adversary from cherry-picking just the bank.py change.
  If adversary drop auth.py change from log, B's signature does not verify.

Could an adversary sign a fake log entry?
  How do clients know the public keys of authorized users?
  1. File system stores the public key of each file/directory owner.
  2. Must specify owner when a file/directory is created.
  3. All clients must know public key of root directory owner.
  Clever design: file system integrity ensures integrity of public keys.
  If any mod() in the log is unauthorized, clients will reject the log.
    That is, mod(f) must be signed by f's owner.

What would happen if C did not log its fetch?
  Malicious server could give a stale view of the log.
  When C fetches auth.py, server gives log before mod(auth.py).
  When C fetches bank.py, server gives log with mod(auth.py) and mod(bank.py).
  Effectively, server pretends like C had a race with A + B.
```

```
    Outcome: C uses auth.py from first fetch and uses bank.py from second fetch
      Oops; that is what we were trying to prevent

    With fetches in the log, when C fetches bank.py, its fetch for
    auth.py must be in the log, and a client can detect the attack.
      The server cannot pretend that reading to auth.py happened
      concurrently with mod to auth.py and bank.py.

  How does logging the fetch help?
    Server can still tell C that no modifications happened!
    But then C logs its fetch based on that log prefix.
    Now the server cannot send mod(auth.py), mod(bank.py).
    "Forking" attack.

  Hard to do better than fork consistency.
    Section 3.2.
    Server pretends there are N universes.
    Each client only sees its own universe.
    No way to detect this forking attack without out-of-band help.
    But fork consistency is a strong guarantee: server can never merge forks.

  Fork consistency is pretty good.
    Consistent view within each fork.
    Leaves strong trace of attack: impossible to cover up a fork.
    Likely to be detected.

  Detecting forks with out-of-band communication.
    Clients A and B compare their view of each other's latest log entries.
    Not a fork: A's view is newer than B's view.
      While A and B were comparing their views, other changes were happening.
      A looked at the log after B looked at the log.
    Fork: neither view is a prefix of the other view.

  "Timestamp box".
    Designated user is responsible for updating some file every 5 seconds.
    If client sees these updates, it's in the same "fork" as the timestamp box.
    Transitive: all clients that see these upates are consistent with each other.
    What if client does not see the updates?
      Option 1: Timestamp box might not be issuing any updates.
        Timestamp computer could be broken or disconnected.
        Malicious server could be ignoring packets from the timestamp box.
      Option 2: Forking attack, client is in a different fork from timestamp box.

  Strawman is not practical: log keeps growing.
    Must check all signatures in the log.
    Interpreting log gets slow.

  Insight 1: only need to check last signature of each user.
    When X signed its latest log entry, it must have validated log.
    Log contains X's previous log entries.
    So earlier signatures by user X were already checked by X.

  Insight 2: each user signs a snapshot of its part of the FS state.
    Partition file system state by file/directory owner.
    When user X makes a change, user signs the resulting FS state.
    Suffices to check latest signature by X (see insight 1).
      No need to check signatures on all previous operations.
    Snapshot contains files/directories that X can modify anyway.
      No problem if X makes up an arbitrary snapshot of FS state.

  Figure 2: data structures for per-owner FS snapshots.
    Each file/directory is named by <owner, i#>.
      Directories map a name to an <owner, i#>.
      Effectively just partitions all files based on owner.
    Owner maintains i-table mapping i# to inode state.
```

```
    Hashes make this table compact.
      Inode contains hashes of data blocks.
      i-table contains hash of each inode.
      i-handle is the hash of the i-table.
      Hash ensures integrity: modifying data/inode changes hash.

  How to maintain fork consistency with i-tables?
    Risk: could malicious server give out an old i-table?
    Need to make sure we have a consistent set of i-handles.

  Idea: signed version vectors.
    Version vector: user -> how many operations that user performed.
    Version structure: signed i-handle together with version vector.

  Zoobar example:
    A mod(auth.py):
      sign new VS
        A's i-handle contains new auth.py
        version vector: {A: 1, B: 0}
    B mod(bank.py):
      ask server for the latest VS of every user
      sign new VS
        B's i-handle contains new bank.py
        version vector: {A: 1, B: 1}
        includes A's bumped count!

  What happens if server hides auth.py from C?
    B's VS refers to A version 1, but no VS from A with that version!
    Forking attack.

  What happens if server hides auth.py from B?
    B's VS will refer to A version 0.
    If C sees VS'es from A and B, will detect forking attack.
    Version vectors not strictly ordered.

  Remaining issue: concurrency.
    Cannot stop the world while one client is appending to the log.
    But allowing concurrent changes leads to conflicting version vectors.
    Indistinguishable from a server performing a forking attack.
    Full SUNDR protocol: update certificates.  See paper for details.

  Summary.
    Hard problem: integrity despite compromised servers.
    Thinking about a log of operations simplifies the problem.
    Fork consistency.
    Optimizations: hashing, partition by owner/writer, version vectors.
```