Store, Forget, and Check: Using Algebraic Signatures to Check Remotely Administered Storage

International Conference on Distributed Computing Systems 2006

Thomas Schwarz    Ethan Miller

presented by Ryan Carr
Introduction

P2P remote storage and backup systems are growing in popularity
  - e.g. Oceanstore, Intermemory, FarSite, etc.

Trusting your data to random people on the internet carries some risks
  - Nodes are frequently unreliable, and sometimes malevolent

How can we make sure peers are actually storing our data?
Possible Scheme

Periodically request a random block of data from peer, verify against local copy.
Possible Scheme

Periodically request a random block of data from peer, verify against local copy.

- Problems with this scheme:
  - Need to have local copy of data
  - Uses lots of bandwidth
Another Possible Scheme

Request MD5/SHA-1 of random ranges of data.
Another Possible Scheme

Request MD5/SHA-1 of random ranges of data.

- Still has problems:
  - Eventually we’ll run out of new challenges
  - Unless we keep a local copy
The "Ideal" Scheme

- Detects small changes/corruptions in the data
- Uses small challenges and responses
  - Saves on bandwidth
- Allow challenger to test unpredictably
  - Storing all possible responses requires more memory than storing the real data
- Challenger need not have a copy of the original data
  - Users can confirm their data is still there if they have lost/deleted it locally
  - Administrators can find malicious entities in the system
In This Paper

We consider an "ideal" scheme that uses

- *Erasure codes* to provide data robustness
  - Let us correct small changes in the data.
- *Algebraic signatures* to verify peers are honest
  - Only two small messages are exchanged per challenge
  - Challenger does not need to have the original data
Erasure Codes take a message of $m$ symbols and add $k$ symbols to create a message of length $n = m + k$.

- Obtaining *any* $m$ symbols lets you reconstruct message.
- This is an $m/n$ erasure code ($m/n$ is the "code rate").

**Example:** Adding a parity bit to a word of data lets us detect 1-bit errors.

- This is an 8/9 erasure code.
Galois Fields

This paper defines its erasure code using *Galois Fields*. This is a field with finitely many elements.

- Adding, subtracting, multiplying, or dividing any two elements in the field produces an element in the field.

**Example:** \( \mathbb{Z} \mod 5 \) as a Galois Field...

- \((3 + 4) \mod 5 = 2\)
- \((2 \times 3) \mod 5 = 1\)
- etc.
\( \alpha \) is called a *primitive* of a Galois field with \( s \) members if

- \( \alpha^{s-1} = 1 \)
- For \( i = 1 \) to \( s - 2 \), \( \alpha^i \neq 1 \).

**Example:** \( \mathbb{Z} \mod 5 \) as a Galois Field...

- 2 is a primitive:
  - \( 2^1 \mod 5 = 2 \)
  - \( 2^2 \mod 5 = 4 \)
  - \( 2^3 \mod 5 = 3 \)
  - \( 2^4 \mod 5 = 1 \)
Our Galois Fields

In the paper, Galois Field members are all bit-strings of length 8 ($2^8$ symbols in the field).

- Addition is the same as XOR.
- Multiplying two strings produces a third string.
  - Addition and multiplication interact naturally
  - e.g. $ac + bc = (a + b)c$
- If $\alpha$ is a primitive...
  - $\alpha$ is some bit string.
  - $\alpha^2, \alpha^3, ..., \alpha^{2^8-1}$ are all different bit strings.
Definition

Let $X = x_0, x_1, \ldots, x_{N-1}$ where $x_i$ is a word of data. Our Algebraic signature is defined as:

$$\text{sig}_\alpha(X) = (x_0 \cdot \alpha^0) \oplus (x_1 \cdot \alpha^1) \oplus \ldots \oplus (x_{N-1} \cdot \alpha^{N-1})$$

This is just another bit string.
Key Property

Let $X$ and $Y$ be collections of data words.

$$\text{sig}_\alpha(X) \oplus \text{sig}_\alpha(Y) = (x_0 \cdot \alpha^0) \oplus \ldots \oplus (x_{N-1} \cdot \alpha^{N-1}) \oplus (y_0 \cdot \alpha^0) \oplus \ldots \oplus (y_{N-1} \cdot \alpha^{N-1})$$

$$= ((x_0 \oplus y_0) \cdot \alpha^0) \oplus ((x_1 \oplus y_1) \cdot \alpha^1) \ldots$$

$$= \text{sig}_\alpha(X \oplus Y)$$

The parity of the signatures is equal to the signature of the parity.
Assume $D_1, D_2, ..., D_m$ are data blocks. Use a $m/n$ erasure code to generate $k = n - m$ parity blocks $P_1, P_2, ..., P_k$

- $P_i$ is generated using function $\rho_i(D_1, D_2, ..., D_m)$
- $\rho_i$ uses XORs to generate a different block for each $i$.

Store $D_1, D_2, ..., D_m$ and $P_1, P_2, ..., P_k$ in the remote storage system.
Naive Protocol

1. Store data across distributed systems

\[ D_0 \quad D_1 \quad D_2 \quad P_0 \]

2. Challenge sites to prove they hold the data

\[ <8,4,5> \]
Naive Protocol

3. Sites respond with signatures of requested data

4. Data originator (challenger) verifies the signatures

Verify: Parity(SIG(D₀), SIG(D₁), SIG(D₂)) == SIG(P₀)
Naive Protocol

- Node not storing our data has a $2^{-l}$ chance of guessing correctly.
- Can tolerate $k$ missing blocks and still reconstruct data.
- Can *correct* $\lfloor k/2 \rfloor$ corrupted blocks.
- What if nodes are colluding?
Handling Collusion

Collusion can be foiled by *blinding* the parity blocks with a pseudo-random bit string.

- Seed number for a stream cipher (e.g. RC4) chosen by data owner when data is put into the system.
- Seed is kept secret, used to generate \( R_1, R_2, \ldots, R_k \).
- Data owner stores \( D_1, D_2, \ldots, D_m \) and \( P_1 \oplus R_1, P_2 \oplus R_2, \ldots, P_k \oplus R_k \).
Nodes storing $D_1, D_2, \ldots, D_m$ return signatures as before.
Node storing $P_i \oplus R_i$ returns $\text{sig}_\alpha(P_i \oplus R_i)$.

- Challenger uses seed to regenerate $R_i$, calculates $\text{sig}_\alpha(R_i)$.
- Challenger computes $\text{sig}_\alpha(P_i \oplus R_i) \oplus \text{sig}_\alpha(R_i) = \text{sig}_\alpha(P_i)$
- Challenger checks that $\text{sig}_\alpha(P_i) == \text{sig}_\alpha(D_1) \oplus \text{sig}_\alpha(D_2) \oplus \ldots$
Implementation

Variations of this algorithm tested on two Windows machines:

- Desktop with 3 GHz dual core Pentium, 512 MB RAM
- Laptop with 2 GHz Centrino, 1 GB RAM

Calculating a 32-bit signature for every 512 bits of data:

- Laptop’s throughput was 900 MBps
- Desktop’s throughput was 700 MBps
- Both machines fell to 40 MBps when data had to be read from disk

Algorithm is fast, transfer between disk/memory is the main bottleneck.
Developed a method for verifying our data is stored correctly in a P2P remote storage system. Used *algebraic signatures* with *erasure codes*, which have many advantages:

- Does not require challenger to have a copy of the data
- Allows verification to be done with two small messages, saving bandwidth
- Allows for unpredictable challenges
- Can detect and correct small errors
- Can reconstruct original data even if multiple nodes go down
- Implementations can be very fast

This could allow us to create very large-scale, verifiable distributed storage systems.