# Chapter 2

# Storage Outsourcing

*Elaine Shi*

In this chapter, we will use cloud outsourcing as a motivating theme to describe several simple and elegant cryptographic protocols. Cloud computing allows users to store files and retrieve them anywhere and anytime. An obvious concern is security — if the cloud server is compromised or hacked into, then data on the server may be tampered with or leaked. We will describe various cryptographic protocols that aim to provide **integrity**, **availability**, and **privacy**.

Note that most of these cryptographic constructions actually have much broader applications outside storage outsourcing. To aid understanding, however, we will describe these schemes in the application context of the storage outsourcing.

## 2.1 Integrity: Merkle Hash Tree

Imagine that Alice is a teaching assistant for Applied Cryptography, she stores students' grades on a Google cloud server. When she retrieves the grades from the server, it is important to make sure that the grades be correct. How can we achieve this?

**Strawman idea.** A naïve idea is for Alice to store a checksum locally, and whenever she retrieves the grade file, the checks whether the retrieved file matches the checksum. For example, $G = (g_1, g_2, ..., g_m)$ be the grades of $m$ students. Alice could store the checksum $\mathsf{CheckSum}(G) := \Sigma_{i \in m} g_i \bmod P$, where $P$ is a sufficiently large prime integer.

Such a checksum scheme may be good enough to defeat random errors, but is not strong enough to defeat adversarially generated errors. In particular, an adversary can easily modify the grades in a way that preserves the sum mod $P$, e.g., by adding 10 to one student's grade and subtracting 10 from another.

In cryptography, we would like to develop schemes that protect against adversarially introduced errors. No matter how the server deviates from honest behavior, our schemes should always be able to detect it except with negligible probability.

To accomplish this task, we will first introduce a new cryptographic primitive.

## Collision Resistant Hash Function (CRH)

**Definition 2.1.** A family of functions $\mathcal{H} = \{H_i : D_i \to R_i\}_{i \in I}$ is a family of collision-resistant hash functions (CRH) if :

1. (ease of sampling) there is a probablistic polynomial time algorithm $\mathsf{Gen}$ that samples an instance efficiently: $\mathsf{Gen}(1^n) \in I$

2. (compression) $\forall i \in I, |R_i| < |D_i|$

3. (ease of evaluation) Given $x, i \in I$, the computation of $H_i(x)$ can be done in probabilistic polynomial time.

4. (collision resistance) for all non-uniform probabilistic polynomial time Turing machine $A$, there exists a negligible function $\epsilon$ such that $\forall n \in \mathbb{N}$,

$$\Pr[i \leftarrow \mathsf{Gen}(1^n); x, x' \leftarrow A(1^n, i) : H_i(x) = H_i(x') \land x \neq x'] < \epsilon(n)$$

Note that since $\forall i \in I, |R_i| < |D_i|$, i.e., the output of the hash is shorter than the input, collisions must exist by the pigeon-hole principle. The simplest way to find a collision is to brute-force enumerate all possibilities, then a collision is guaranteed. However, if the security parameter $n$ is large enough, no computationally bounded adversary should be able to find collisions in say, 10,000 years.

We can construct collision-resistant hash functions from cryptographic assumptions, such as Discrete Logarithm. More details can be found in Chapter 5 of Pass and Shelat's textbook [PS10]. In practice, we often use SHA-256, SHA-384 and SHA-512 [NIS02] as heuristic candidates for collision resistant hashes. These functions are known to have certain good properties.

**Using a CRH to ensure integrity.** Suppose that Alice wishes to store a file $F$ on a Google server. Alice samples a hash function $H$ from a CRH family $\mathcal{H}$ (using a sufficiently large security parameter $n$). Alice now uploads $F$ to the server but stores the cryptographic digest $H(F)$ locally. Notice that the file $F$ can be much larger than the cryptographic digest $H(F)$. When Alice gets back a file $F'$ from the server, she accepts the file if $H(F) = H(F')$. If a malicious server can mislead Alice to ever accept a wrong file, we must be able to leverage the server to find a hash collision, thus contradicting the collision resistance property. In other words, the digest $H(F)$ cryptographically binds to the file $F$ assuming that the adversary (i.e., the server) is computationally bounded.

### Storing Multiple Files

Now consider a more challenging scenario. Instead of storing a single file, Alice has a large database of files denoted $F_1, F_2, \ldots, F_N$. She wishes to store all these files on the server, but every time she is interested in reading only one file — e.g., think of each file as an email.

**Strawman idea 1.** One naive idea is to use the same scheme as before, that is, Alice stores the digest $H(F_1, \ldots, F_N)$ locally. However, every time she retrieves a file of interest $F_i$, she would have to download all other files in order to verify the correctness of $F_i$. Clearly this requires $\Theta(N)$ cost for reads, and thus is too expensive (e.g., the entire database of files can be terabytes in size but each file is small).

**Strawman idea 2.** Another straightforward idea is for Alice to store $N$ digests, $H(F_1)$, $H(F_2), \ldots, H(F_N)$ locally. In this way, Alice need not retrieve any additional data if she is only interested in reading $F_i$. However, this scheme requires that Alice stores $\Theta(N)$ digest data locally, which is also expensive especially if Alice is using her weak mobile phone as the client.

We will now describe a new approach called the Merkle Hash tree, which was first invented by Ralph Merkle [Mer89].

### Merkle Hash Tree

We now explain how a Merkle hash tree works (see Figure 2.1). Suppose that we divide the entire storage into $N$ blocks, $\mathsf{Block}_1, \mathsf{Block}_2, \ldots, \mathsf{Block}_N$ — one can think of each block as a file if all files were equal size. Without loss of generality, we will assume that $N$ is a power of 2 — since otherwise we can

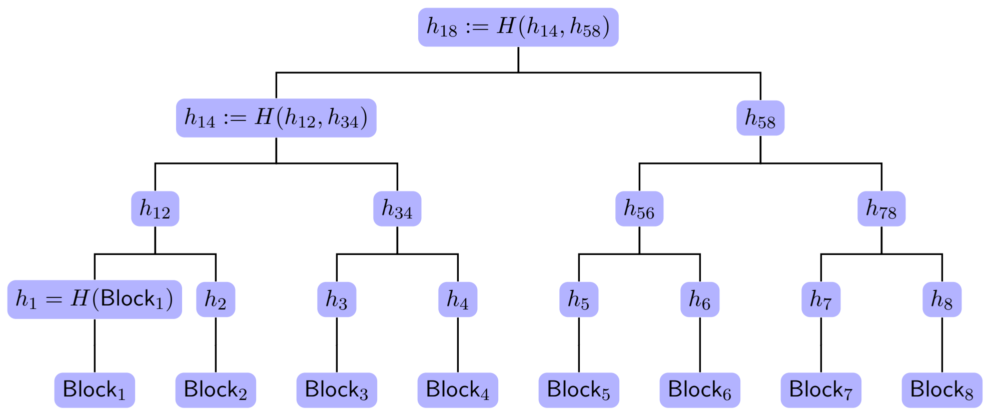

Figure 2.1: A Merkle hash tree containing 8 blocks.

always round it up to the nearest power of 2 incurring only a constant factor blowup. These $N$ original blocks correspond to the leaves of the Merkle hash tree.

Now, we first hash each block individually, i.e., for $i \in [N]$, $h_i := H(\mathsf{Block}_i)$. Next, for other level in the tree, its hash value is computed as the hash of its two children. For example, in Figure 2.1, $h_{14} := H(h_{12}, h_{34})$. Finally, the client stores the root digest and the server stores the entire Merkle hash tree including the hashes of all internal nodes.

### Read

Whenever a client wants to read a specific block, besides the block, which hashes must the client retrieve in order to verify correctness of the block? For example, in Figure 2.2, say, the client wishes to retrieve $\mathsf{Block}_3$. It is not hard to see that if the client additionally retrieves all the yellow hashes, it can verify the correctness of $\mathsf{Block}_3$ — in particular, the client would be able to recompute all the red hashes in the tree representing the path from $\mathsf{Block}_3$ to the root.

The above observation can be generalized into the following read procedure. To read a block (e.g., $\mathsf{Block}_3$ in Figure 2.2), the client performs the following actions:

1. Draw the path from the root to the block she wants to read (e.g., red nodes in Figure 2.2), henceforth referred to as the *Merkle path*.

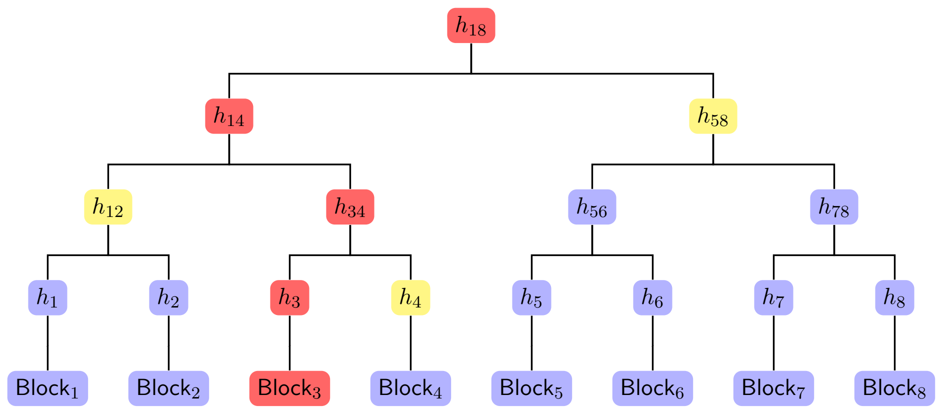

Figure 2.2: Example: read operation of a Merkle hash tree

2. Fetch every sibling node (e.g., yellow nodes in Figure 2.2) of the Merkle path, henceforth referred to as the *Merkle proof*.

3. Reconstruct all hashes on the Merkle path (e.g., read nodes in Figure 2.2).

4. Compare the root hash reconstructed in this manner against the locally stored digest, and accept the block if they are equal.

It is not hard to see that each read operation results in reading $O(\log N)$ additional hashes.

Without going into formal security definitions, we provide an informal argument why the above Merkle tree construction preserves the integrity of any data fetched. Consider the example in Figure 2.2 where the client wishes to read $\mathsf{Block}_3$ — henceforth we use $\mathsf{Block}_3$ to denote the correct value of the block. Suppose that the server can succeed in convincing the client into accepting a wrong value denoted $\mathsf{Block}'_3 \neq \mathsf{Block}_3$. We now argue that we must be able to leverage such a server to find a hash collision. Suppose that the server succeeded in deceiving the client into accepting the wrong value $\mathsf{Block}'_3$ by returning the Merkle proof $\{h'_{12}, h'_4, h'_{58}\}$. Henceforth all unprimed values denote the correct values, whereas primed values denote the values actually received or reconstructed by the client. Consider the path from the root to $\mathsf{Block}_3$. We know that the reconstructed root hash $h'_{18}$ must be the same as the client's locally stored value $h_{18}$. Thus one of the following must be true:

1. either $H(\mathsf{Block}'_3) = h_3$;

2. or $h'_3 := H(\mathsf{Block}'_3) \neq h_3$, but $H(h'_3, h'_4) = h_{34}$;

3. or $h'_3 := H(\mathsf{Block}'_3) \neq h_3$, and $h'_{34} := H(h'_3, h'_4) \neq h_{34}$, but $H(h'_{12}, h'_{34}) = h_{14}$;

4. or $h'_3 := H(\mathsf{Block}'_3) \neq h_3$, and $h'_{34} := H(h'_3, h'_4) \neq h_{34}$, and $h'_{14} := H(h'_{12}, h'_{34}) \neq h_{14}$, but $H(h'_{14}, h'_{58}) = h_{18}$.

In all of the above cases, it is not hard to see that we have found a hash collision. Recall again that a hash collision always exists — but here we show that if a probablistic polynomial time server can find one with significant probability, then we can leverage such a server to find collisions *efficiently*, thus violating the collision resistance property of the hash function.

## Write

So far we have focused our discussions on reads. Now we consider how to support write operations. During a write operation, the client must update its locally stored root digest. To achieve this, the client performs the following:

1. First, perform a read operation on the block being updated (e.g., $\mathsf{Block}_3$ in Figure 2.2), fetching the Merkle proof (e.g., yellow nodes in Figure 2.2) consisting of all siblings of the Merkle path. Verify that the fetched block and the Merkle proof are correct by reconstructing the root hash and comparing the reconstructed value with the locally stored digest.

2. Next, given the Merkle proof and the new value of the block to be updated, it is not hard to see that the client has sufficient information to compute the new root hash. The client stores the new root hash as the updated digest.

3. The client sends the new block to the server, and the server updates all hashes along the corresponding Merkle path accordingly.

It is not hard to see that each write operation involves retrieving $O(\log N)$ hashes. We also point out that with writes, a Merkle hash tree scheme not only guarantees *integrity*, but also *freshness*, i.e., if the client ever accepts a block fetched, then the accepted block must reflect the last value written.

### Applications and Real-World Adoption

Merkle hash trees (and variant constructions) have been adopted in various applications to ensure the authenticity of data storage, such as revision control systems [git], Bitcoin [Nak09], secure processors [SCG$^+$03, TML$^+$00, MAB$^+$13, AGJS13], outsourced cloud storage [zfs, tah, GPTT08].

## 2.2 Availability: Proof of Retrievability (PoR)

Alice stores or backs up her data in the cloud (e.g., Dropbox). Since Alice might be paying Dropbox subscription for the storage service, Alice would like to make sure that the Dropbox is indeed storing all ofher data, and that no data is lost. How can Alice be sure of this?

**Strawman solution.** A trivial solution is for Alice to store a Merkle tree digest of all her data. Every month, Alice performs an audit where she downloads each and every block in the dataset along with every block's Merkle proof. In this way, Alice is able to verify the correctness of every block downloaded, and thus at the end of the audit, Alice is sure that the server must be storing all of her blocks.

An obvious drawback of this strawman scheme is that it is very expensive, especially if Alice is outsourcing terabytes of data.

Can we have a solution where the cost of making an audit is sublinear in the total data size?

**Another attempt: probablistic checking.** As a second try, our idea is to perform probabilistic checking rather than downloading the entire dataset during an audit. Alice still stores the Merkle digest of the dataset. Now, say, every month, Alice picks a random subset of $k$ blocks, and challenges the server to return the $k$ blocks as well as their Merkle proofs. Alice checks the correctness of the downloaded blocks against its local digest, and she is satisfied with the audit if all checks succeed.

We now analyze this probabilistic checking scheme.

- **Scenario 1: the server has lost many blocks.** Such a probabilistic checking scheme is indeed great at detecting a cheating server that has lost many blocks. As a simple example, consider that the server has lost half of the blocks (the analysis below generalizes to any constant fraction). The probability that such a cheating server escapes detection is the following, since for each of the $k$ random challenges, the server succeeds in answering the challenge with probability $\frac{1}{2}$.

$$\Pr[\text{server escapes detection}] = \frac{1}{2^k}$$

This means that if $k$ is our security parameter, then the probability that such a cheating server can survive an audit is negligible in $k$, and Alice is happy.

- **Scenario 2: the server has lost a small number of blocks.** Unfortunately, this probabilistic checking scheme would frequently fail to detect a server that has lost only a small number of blocks.

  For example, consider a server that has lost exactly 1 block. Such a cheating server can survive an audit with the following probability where $N$ denotes the total number of blocks outsourced:

  $$\Pr[\text{server escapes detection}] = (1 - \frac{1}{N})^k$$

  In particular, note that for each of the $k$ challenges, the server can succeed in answering the challenge with probability $1 - \frac{1}{N}$.

  Now, even when Alice samples a large number of blocks, say $k = \frac{N}{2}$, we claim that the server can still escape detection with $O(1)$ probability, since

  $$(1 - \frac{1}{N})^{\frac{N}{2}} = \left((1 - \frac{1}{N})^N\right)^{\frac{1}{2}} \approx \exp(-0.5) = O(1)$$

  This is unsatisfying, since we would like a scheme with strong security: even when the server has lost only a single block, Alice can detect it in an audit except with negligible probability.

## Soundness Amplification with Erasure Code

As shown above, the simple probabilistic checking scheme does not achieve strong enough soundness, i.e., a cheating server that has lost one block can escape detection with constant probability. Erasure codes or error-correcting codes are often adopted for amplifying soundness in various theoretical and practical applications.

A natural idea is the following: let $N$ denote the total number of blocks in the original dataset. We now encode $N$ original blocks into $2N$ code-blocks, with a redundancy blowup of 2. In particular, we will leverage an erasure coding scheme with the following property: given any $N$ out of $2N$ code-blocks, we can recover the entire dataset of size $N$. For the time being, let us assume that such an erasure coding scheme does exist. If our dataset has been encoded in this way, then this means that for the server to actually cause any data loss, it must have lost at least half of the code-blocks (otherwise, there is sufficient information left to reconstruct the entire dataset). As we argued earlier, our simple probabilistic checking scheme can almost always (i.e., except with negligible failure probability) detect a cheating server that has lost half of the blocks! Therefore, it suffices to show how to construct an erasure coding scheme satisfying the aforementioned property.

**Erasure code.** An $(N, M)$-erasure code has two deterministic algorithms — henceforth we assume that all blocks and code-blocks are bit-strings of length $\ell$, and we omit writing $\ell$ explicitly.

Encode: takes $N$ data blocks $\mathsf{Block}_1, \mathsf{Block}_2, ..., \mathsf{Block}_N$, and outputs $M \geq N$ code-blocks $C_1, C_2, ..., C_M$.

Decode: takes any $N$ code-blocks, and outputs $N$ decoded blocks.

We say that $(N, M)$-erasure coding scheme is correct iff for any set of $N$ blocks $\{\mathsf{Block}_i\}_{i\in[N]}$, for any index set $S \subseteq [M]$ of size $N$, let $\{C_i\}_{i\in[M]} \leftarrow \mathsf{Encode}(\{\mathsf{Block}_i\}_{i\in[N]})$, then it must hold that $\mathsf{Decode}(\{C_i\}_{i\in S}) = \{\mathsf{Block}_i\}_{i\in[N]}$.

We present a very simple $(N, 2N)$-erasure code construction. The idea is to rely on polynomial interpolation. Consider a uni-variate polynomial of degree $d - 1$ operating over a finite field $\mathbb{F}_p$ where $p$ is a prime, i.e., all coefficients are integers mod $p$, and all arithmetic is performed mod $p$. One important observation is that if we know the polynomial evaluated at $d$ places, then we can reconstruct the polynomial efficiently.

Therefore, we can consider the following erasure coding scheme. Let $a_0, a_1, ..., a_{N-1} \in \mathbb{F}_p$ denote the values of the data blocks where $p > 2^\ell$ is a prime. We now consider the polynomial

$$P(x) = a_{N-1}x^{N-1} + a_{N-2}x^{N-2} + ... + a_1 x + a_0 \bmod p$$

Now we can construct an erasure coding scheme as follows:

$\mathsf{Encode}(a_0, a_1, ..., a_{N-1})$: Let $P$ be the above defined polynomial. Output $(i, P(i))$ for $i \in [2N]$.

$\mathsf{Decode}(C_1, C_2, \ldots, C_N)$: Reconstruct the polynomial given $(C_1, \ldots, C_N)$ where each $C_i$ is of the form $C_i := (x_i, y_i)$ where $x_i \in [2N]$ and $y_i := P(x_i)$. Output the coefficients of $P$.

One way to reconstruct the polynomial is by expressing the problem as a system of linear equations as follows:

$$\begin{bmatrix} 1, x_1, x_1^2, \ldots, x_1^{N-1} \\ 1, x_2, x_2^2, \ldots, x_2^{N-1} \\ \vdots \\ 1, x_N, x_N^2, \ldots, x_N^{N-1} \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{N-1} \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix},$$

and to recover the polynomial's coefficients, we just need to solve this linear system. It turns out that matrices of the above form are called Vandermonde matrices, and square Vandermonde matrices are invertible as long as all of the $x_i$'s are distinct [van].

**Proof-of-retrievability with erasure code.** We can construct a proof-of-retrieveability scheme as follows:

- **Setup.** The client leverages an $(N, 2N)$-erasure coding scheme to encode the original $N$ blocks into $2N$ code-blocks. It then builds a Merkle tree over the $2N$ code-blocks, and remembers the root digest. It then sends all the $2N$ code-blocks as well as the Merkle tree to the server.

- **Audit.** To audit, the client chooses $k$ random indices from $[2N]$, and challenges the server to return the corresponding $k$ code-blocks along with their Merkle proofs. The client checks the returned blocks as well as their Merkle proofs against the local root digest, and accepts if all pass the check.

**Claim 2.2** (Informal.)**.** *Let $k$ be a super-logarithmic function in the security parameter. Then, with all but negligible probability, as long as the client accepts during an audit, the server must have stored enough information to recover all $N$ original blocks.*

Note that if the client also needs to read the data every now and then besides performing audits, the client can additionally outsource a cleartext copy that is not encoded (along with a separate Merkle tree), in order to support authenticated reads.

### Supporting Dynamic Writes

So far, we have assumed that the dataset is static and that the client need not perform dynamic writes to the data. Due to the introduction of the erasure coding scheme, performing writes appears difficult. A standard erasure coding scheme such as the one mentioned earlier performs global encoding, such that every code-block may depend on all original data blocks. Unfortunately this is bad news for writes since every time the client updates a block, all code-blocks must be recomputed!

Fortunately, it turns out that there are known techniques to support dynamic writes efficiently with very little additional overhead. For additional reading, we refer the readers to Shi et al. [SSP13].

### Acknowledgments

Thanks to Siqiu Yao and CS6832 students for creating an initial scribe.

## 2.3 Privacy: Oblivious RAM

*Elaine Shi*

The setting: you have a large amount of private data (e.g., your genomic data) stored on an untrusted server. While standard encryption techniques allow the client to hide the contents of the data from the server, the server can still observe access patterns to the data. Through such access patterns, the server can potentially infer sensitive information about your private data. For example, through *frequency* and *co-occurrence* information, the server may be able to infer what genomic algorithm (e.g., medical test) is being executed on the genomic data. It is also helpful to think of access pattern leakage through a programming language perspective: for example, the following program has an `if`-branch dependent on secret inputs (e.g., think of the secret input as the last bit of a secret key) Thus by observing whether memory location `x` or `y` is accessed, one can infer which branch is taken.

```
if (s) {
  mem[x]
} else {
  mem[y]
}
```