Fully Dynamic Data Management in Cloud Storage Systems with Secure Proof of Retrievability*

Nam-Su JHO†(a), Daesung MOON†, and Taek-Young YOUN††(b), Nonmembers

SUMMARY: For reliable storage services, we need a way not only to monitor the state of stored data but also to recover the original data when some data loss is discovered. To solve the problem, a novel technique called HAIL has been proposed. Unfortunately, HAIL cannot support dynamic data which is changed according to users’ modification queries. There are many applications where dynamic data are used. So, we need a way to support dynamic data in cloud services to use cloud storage systems for various applications. In this paper, we propose a new technique that can support the use of dynamic data in cloud storage systems. For dynamic data update, we design a new data chunk generation strategy which guarantees efficient data insertion, deletion, and modification. Our technique requires $O(1)$ operations for each data update when existing techniques require $O(n)$ operations where $n$ is the size of data.

key words: cloud storage, dynamic data, proof of retrievability

1. Introduction

In these days, the use of remote storage for maintaining valuable information is increasing due to the rapid progress of network bandwidth. Along with the growth of the use of remote storage, clients’ concern regarding the security of storage services is also increasing [2]. In cloud storage services, clients entrust their data to remote storage servers and delete entrusted data in their local storage. Since clients do not possess the data, the integrity of stored data is very important for reliable cloud storage services.

For cloud storage services, we need to consider two scenarios that can break the security of the services. **Inside Adversary:** A malicious service provider could be an adversary. Though it is assumed that service providers are not actively breaking the integrity of clients’ data, sometimes, some service providers could lose clients’ data due to their carelessness or the use of cheap storage. The malicious servers are also called as an honest-but-curious adversary. In general, the security against inside adversaries can be ignored only if clients can detect the service provider’s fault.

**Outside Adversary:** Differently from an inside adversary who is a legitimate service provider, outside adversaries can mount any attack since they have nothing to lose even if their attacks are detected by any entity including clients. An outside adversary may try to take clients’ data stored in service provider’s storage or interrupt valid connection between the service provider and clients. The first mentioned attack can be relatively easily countered since we can use an encryption scheme to prevent the adversary from getting visible information. When the goal of an outside adversary is to compromise or delete stored data, simply encrypting stored data is not enough, instead, we need a way to monitor the current state of the stored data to ensure that outsourced data are intact. However, the latter cannot be easily countered since the adversary can easily mount the attack without losing any asset of the adversary. Is the reason why such an attack is harmful and hard to encounter.

In clients’ viewpoint, two techniques are required to guarantee the fact that the stored data can be recovered whenever the owner of the data want to retrieve.

- Verifiable proof of integrity for stored data
- Data recovery when a part of data is damaged

To check the integrity of data stored in remote cloud servers, provable data possession (PDP) and proof of retrievability (PoR) have been proposed to permit users to audit the current state of data stored in remote storage [1], [4]–[7], [9]–[11]. Note that PDPs and PoRs are very similar in terms of the functionality of integrity check. The main difference between them is the retrievability of stored data. In PDPs, only the integrity check is supported, but PoRs support original data recovery when some corrupted data blocks are discovered. However, in this paper, we consider PoRs to be unable to provide the retrievability since an outside adversary can spoil the data so that the original data cannot be recovered. In other words, we say that PoRs also cannot support the retrievability against outside adversaries.

If a storage service meets the first requirement, we can guarantee the security against an inside adversary since clients can test if the service provider maintains the original data. However, we still cannot cover the security against an outside adversary. If an outside adversary breaks the storage system of a cloud service provider and maliciously modifies the original file, the first requirement doesn’t work for the threat. In this case, the only thing the client can do is to identify the attack by verifying the current state of the stored data. To cover the case, we need the second requirement which permits clients to recover the original data from damaged data. To support the functionality, a high-availability...
and integrity layer (HAIL) for cloud storage has been proposed [3], which permits users to recover the original data by the help of storage servers.

Until now, a number of schemes have been proposed the security against inside adversaries and outside adversaries. Most of them are designed for static data where the data are not modified after uploading. However, we need to use dynamic data in many cloud-based applications. For example, if we use a cloud system for smart work, we may want to modify document files stored in cloud storage. Due to the complexity of algorithm and protocol design for dynamic data, relatively few schemes have been proposed to cover dynamic data in cloud storage environment [4–6], [10], [11]. Among existing techniques for the security against outside adversaries [3], [8], only a few schemes support dynamic data [8].

In this paper, we propose a new architecture for maintaining dynamic data in distributed storage servers, which provides (1) verifiable proof of integrity and (2) original data recovery if limited number of storage servers are damaged. For the verifiable proof of integrity, we devise an efficient tag generation function that returns a short tag for a number of data blocks. The tag is an accumulator which is computed as a linear sum of the data blocks. Since the linearity is suitable for dynamic updates, our technique supports efficient verifiable proof of integrity suitable for dynamic data. For the retrievability, we use error-correcting code as in many existing techniques. Differently from existing techniques where a chunk is composed of consecutive data blocks, consecutive data blocks are distributed in our construction.

2. Preliminaries

In this section, we will describe the adversarial model and the system model of our technique. As we explained in the introduction, we consider two types of adversaries, inside adversaries and outside adversaries, which are defined as following:

- **Security against Inside Adversary**: An inside adversary’s malicious behavior can be detected by clients so that the adversary cannot try to mount any attack that causes the loss of stored data.
- **Security against Outside Adversary**: An outside adversary’s attack may not be prevent, but the damage should be recoverable.

Recall that the main goal of our work is to support cloud storage services to resist against inside and outside adversaries.

As seen in Fig. 1, in the proposed system, we consider three types of storage servers, servers for storing original data, servers to maintain some additional information for data recovery, and a server to manage dynamically changed data status. Let $n$ be the total number of servers in the proposed cloud storage systems. Let $n'$ be the number of servers for original data and $n''$ be the number of additional servers for data recovery. Since we need one server for managing dynamic data status, $n = n' + n'' + 1$. Note that the number of additional servers for data recovery is determined by the level of error tolerance against outside adversaries who break a number of servers so that clients cannot recover their original data. In our setting, clients can recover their data until $n''/2$ servers are broken by outside adversaries. For this reason, we assume that $n''$ is an even number. Differently from some existing works, CSP accesses to $n' + 1$ storage servers unless a part of their data stored in $n'$ storage servers for original data are not harmed.

3. Proposed Construction

3.1 Initial Data Encoding

The goal of the initial step is to generate chunks to distribute the original data to multiple servers. Let $b$ be the size of a block. We assume that the bit-size of data is always dividable by $b$. Let $F$ be the input which is composed of $\ell$ blocks. We assume $n$ storage servers where one server is in charge of maintaining information for data dynamic, $n'$ servers store original data, and $n''$ servers maintain additional information for supporting original data recovery. To resists against the corruption of $\text{cor}$ servers, we will set $n'' = 2\text{cor}$. Let $\text{params}$ be the set of input parameters of encoding function.

Then, the input file $F$ is divided into $\ell$ blocks of size $b$. We expect that the original file is distributed to $n'$ servers and they have the same sized chunk. Hence, we firstly test if $\ell$ is a multiple of $n'$ or not. If so, we use the input file as is, and set $M = F$. Otherwise, we append some redundant blocks at the end of the file $F$ and set it $M$. Then the number of blocks in $M$ is a multiple of $n'$. Here, the same value is assigned to all redundant blocks. In Algorithm 1, we initialize redundant blocks as NULL, but any value can be used for them. $m_i$ means the $i$-th bit of $M$.

From step 7 to 9, original data blocks are divided into $n'$ chunks $CH_i$ for $i = 1, \ldots, n'$. In our construction, $\alpha$ blocks are assigned to each chunk and the difference of any two adjacent blocks’ indexes is $n'$. Then we initialize the chunk $CH_0$ which contains some redundant information to support dynamic data and the proof of retrievability. Detailed explanation for the initialization function $\text{Init}()$ will be given in Sect.3.1.1. From step 16 to 22,
Algorithm 1 Initial Data Encoding

Require: $F$: input file composed of $t$ blocks of size $b$-bits, $n$: total number of servers in the cloud storage systems, $n'$: number of servers for original data, $n'' = n - n' - 1$: number of additional servers for data recovery, $corr(n''/2)$: robustness against corruption of storage servers, $params$: input parameters of encoding functions for server corruptions.

Ensure: $n''$ chunks to be stored in distributed cloud storage servers
1: /* Input File Segmentation */
2: if $t$ is a multiple of $n' + 1$ then
3:   $a = t/n'$ and $M = F$
4: else
5:   $a = \lceil t/n' \rceil$ and $M = F[n_{a+1}]\cdots[n_{a+\alpha}]$ where $n_i = \text{NULL}$ for $i = t+1, \ldots, an'$
6: end if
7: for $i$ from $1$ to $n'$ do
8:   $CH_i = m_i||m_{i+1}||\cdots||m_{i+\alpha-1}n'$
9: end for
10: /* Initialize $CH_0$ */
11: for $i$ from $1$ to $a$ do
12:   run Init(stat_i)
13: end for
14: $CH_0 = \text{stat}_1||\text{stat}_2||\cdots||\text{stat}_n$
15: /* Encoding */
16: for $i$ from $1$ to $a$ do
17:   $TM_i = m_{i+\alpha-1}||m_{i+\alpha}||\cdots||m_{i+\alpha-1}$
18:   $\{e_{i1}, e_{i2}, \ldots, e_{i\nu_1}\} = \text{Encode}(params, TM_i)$
19: end for
20: for $i$ from $1$ to $n''$ do
21:   $CH_{i\ell} = e_{\ell1}||e_{\ell2}||\cdots||e_{\ell\nu}$
22: end for
23: return Encoded chunks $\{CH_0, \ldots, CH_{n''-1}\}$

$n''$ chunks are generated by applying an encoding function for server corruption resistance. Here, we use an encoding function which takes $i$-th blocks of $n' + 1$ chunks $\{CH_0, CH_1, \ldots, CH_n\}$ as input and returns $n''$ blocks as an output. $n''$ blocks of the output are assigned to the $i$-th blocks of $n''$ chunks $\{CH_{i'+1}, \ldots, CH_{i'+\alpha-1}\}$. Then, Algorithm 1 returns $\{CH_0, \ldots, CH_{n''-1}\}$ for given input values.

3.1.1 State Initialization

We want to note that the state initialization function is assumed to be executed by the cloud service provider instead of the owner of the data. We assume the above scenario for both the efficiency in client-side cost and the simplicity of explanation of the initial data encoding algorithm. However, if the owner of the data does not fully trust CSP, the client can compute the state information and send it to CSP. The main difference between two approaches is the entity that updates state information for each data update operation. The entity who generates the information will update it too. From now, we will describe proposed algorithms under the assumption that the cloud service provider generates and updates state information. We want to emphasize that the role can be easily shifted to the owner of the data.

We use state information to manage the status of data blocks stored in the same row in $\alpha$ storage servers for original data. For example, we store $\alpha$-bit stat_i: $\{m_{1+i\ell}, m_{2+i\ell}, \ldots, m_{\alpha+i\ell}\}$ as the $i$-th status information. Different from other servers, the server for status information should maintain a little bit longer data since it supports dynamic data management and reliable retrievability proof at the same time. For two functionalities, four values are required including $s_i$, $n_i$, $r_i$, and $au_i$. Among them, two values $s_i$ and $n_i$ are used for managing dynamic updates. Note that, along with dynamic updates, some data blocks could be deleted or inserted. In our construction, we cover such dynamic modifications by managing meaningful blocks in a row, which means that some meaningless blocks may be included in some rows. The role of $\{s_i, n_i\}$ is to manage the status. From now, we explain the role of each value for our construction. The first $1$-bit information $s_i$ indicates whether there is a meaningless block or not. If all blocks are valid, we set $s_i = 1$, otherwise $s_i = 0$. Using the second $\nu$-bits information $n_i$, we count the number of invalid blocks. So the second value influences the number of storage servers for original data. We can support $2^{\nu-1}$ storage servers since $\nu$-bits information covers so far as to the case where only one block is valid among $2^{\nu-1}$ blocks. The role of $\{r_i, au_i\}$ is to support retrievability proof. Though it is possible to support more useful functions such as public verifiability, in this section, we describe secret-key based simple construction which does not support the functionality due to the simplicity of explanation. In the next section, we will give a modification which can support the functionality. In secret-key based approach, the owner of the file randomly chooses a secret key $sk$ with $\alpha$ secret values $a_1, \ldots, a_\alpha \in GF(2^\nu)$ to generate authentication code. Then the authentication code for $\nu$ data blocks $\{m_{1+i\ell}, m_{2+i\ell}, \ldots, m_{\alpha+i\ell}\}$ is computed as

\[
au_i = \text{prng}(sk, r_i) + a_1m_{1+i\ell} + \ldots + a_\alpha m_{\alpha+i\ell} \mod f(x)
\]

where prng is a pseudo-random number generator which guarantees collision resistance and $f(x)$ is an irreducible polynomial defining $GF(2^\nu)$. In practice, we can use $i = r_i$ instead of a randomly chosen value for the simplicity and efficiency of the initialization procedure. Since the goal of $r_i$ in computing prng$(sk, r_i)$ is to guarantee the collision resistance, the use of a sequential number is also acceptable. For the approach, the data owner should maintain the recently used counter.

When we need to use outer code to recover the original data from damaged data chunks stored in $n$ distributed storage servers, two values $r_i$ and $au_i$ are not needed since they are useful only for retrievability proof. Hence, the following value can be used as $d$-bits input for outer code: $s_i||n_i||0^{d-\nu}$ where $0^{d-\nu}$ means $d - (1 + \nu)$ zeros in bit. If we use a sequential number for prng, we can concatenate the counter at the end of $n_i$ and fill zeros so that the size to be $d$-bits. As a result, we can guarantee that the size of data blocks in all storage servers for outer code is $b$-bits.

3.1.2 Encoding

We use an encoding function $\text{Encode}()$ to resist the corruption of storage servers. Since we deal with dynamic data which is updated frequently, it is desirable to store original
data as it is. In general, encoding functions such as error-correcting codes that support data recovery transform the original data to encoded data so that the original data should be re-constructed from encoded data. Some error-correcting codes permit users to store the original data without transformation of the original form. In the literature, such schemes are called systematic codes. So, in our construction we use a systematic code, especially a systematic Reed-Solomon code will be used as the encoding function.

3.2 Update

Before discussing update procedures, we want to emphasize that we will focus on data blocks instead of dealing with all data blocks including the blocks in additional servers for data recovery. Since we can easily update additional data blocks for recovery by running the encoding function, we will explain the update method only using original data blocks and status information blocks for the simplicity of the explanation.

In all data update procedures, the owner of a file makes a query to CSP so that the service provider to control data blocks stored in storage servers according to the user’s request. To support a fully dynamic data update, we need to ensure that the service provider can manage data blocks in a systematic way. For given update queries, the service provider generates an update query for each storage server to perform the update operation. For each update query, the service provider generates an update query to update the data block in the storage server.

For each update query, the client makes a request packet to the service provider. The request packet contains the status information of the data block to be updated. The service provider updates the data block in the storage server according to the request packet. For each update query, the service provider generates an update query to update the data block in the storage server.

For each update query, the client makes a request packet to the service provider. The request packet contains the status information of the data block to be updated. The service provider updates the data block in the storage server according to the request packet. For each update query, the service provider generates an update query to update the data block in the storage server.

For each update query, the client makes a request packet to the service provider. The request packet contains the status information of the data block to be updated. The service provider updates the data block in the storage server according to the request packet. For each update query, the service provider generates an update query to update the data block in the storage server.

3.2.1 Modify Query

A modification query is easy to treat since the request packet given from the data owner is almost identical with the request packets will be sent to storage servers. To modify a data block in a storage server, the data owner may compute the new status information of the data block and a new status information of the data block.

Algorithm 2 Modification Algorithm for CPS

Require: \( m^* \): new data block to be stored, \([i_1, i_2]\): position of the data to be modified \((i_1\text{-th server}, i_2\text{-th block})\).
End require

Ensure: Generate the query to be sent to a storage server.
End ensure

1: Define an empty set \( QM \) for query messages
2: Generate \( qm_{i_1} = Q_m(m^*, i_2) \)
3: Add \( \{qm_{i_1}, i_1\} \) to \( QM \)
4: return \( QM \)

3.2.2 Delete Query

The goal of a deletion query is to remove a data block from stored data blocks. Differently from the simple purpose of the query, we have to carefully deal with deletion queries since possible scenarios to consider are not simple due to the existence of NULL data blocks. So, we will explain the delete query processing according to the possible scenarios.

In all data update procedures, the owner of a file makes a query to CSP so that the service provider to control data blocks stored in storage servers according to the user’s request. To support a fully dynamic data update, we need to ensure that the service provider can manage data blocks in a systematic way. For given update queries, the service provider generates an update query for each storage server to perform the update operation. For each update query, the service provider generates an update query to update the data block in the storage server. For each update query, the client makes a request packet to the service provider. The request packet contains the status information of the data block to be updated. The service provider updates the data block in the storage server according to the request packet. For each update query, the service provider generates an update query to update the data block in the storage server.

For each update query, the client makes a request packet to the service provider. The request packet contains the status information of the data block to be updated. The service provider updates the data block in the storage server according to the request packet. For each update query, the service provider generates an update query to update the data block in the storage server.

For each update query, the client makes a request packet to the service provider. The request packet contains the status information of the data block to be updated. The service provider updates the data block in the storage server according to the request packet. For each update query, the service provider generates an update query to update the data block in the storage server.

For each update query, the client makes a request packet to the service provider. The request packet contains the status information of the data block to be updated. The service provider updates the data block in the storage server according to the request packet. For each update query, the service provider generates an update query to update the data block in the storage server.

For each update query, the client makes a request packet to the service provider. The request packet contains the status information of the data block to be updated. The service provider updates the data block in the storage server according to the request packet. For each update query, the service provider generates an update query to update the data block in the storage server.
3.2.3 Insert Query

For a query, the data owner makes a query message \( Q_m(i, t_{1, 2}) \) and sends it to the CSP with a set composed of i-th data blocks in \( n' + 1 \) servers for state information and original data. To insert a new message, the CSP checks if there is a room for the message in i-th row or not. If \( \text{MSB}(\text{stat}_{i, 2}) = 1 \), then we can see that there are \( \text{MSB}(\text{stat}_{i, 2}) = 1, v \) invalid blocks assigned by NULL. In this case, the new message can be inserted in an i-th block of a storage server instead of existing NULL. If \( i_1 \) is the index of the last valid block among \( n' \) i-th blocks, the new data block will be stored in i-th block of \( \text{Srv}_{i_{1+1}} \).

Note that the condition can be checked by the following \( n' - \text{MSB}(\text{stat}_{i, 2}) = i_1 \). Otherwise, if the i-th block in \( \text{Srv}_{i_{1+1}} \) is valid, we move i-th data block in \( \text{Srv}_{i_{i+1}} \) to \( \text{Srv}_{i_{1+1}} \) for all \( \alpha = i + 1, \ldots, n' - \text{MSB}(\text{stat}_{i, 2}) = 1, v \). This procedure is described from step 2 to 14 in Algorithm 4. When all i-th blocks are valid, i.e., \( \text{MSB}(\text{stat}_{i, 2}) = 1 \), we don’t have room to store the new message in any i-th block of storage servers. So, we set up one more blocks for all storage servers. If \( i_1 = n' \), we simply insert \( m' \) at the end of i-th block of \( \text{Srv}_{i_{1}} \) and insert NULL at the end of i-th block of \( \text{Srv}_{i_{1}} \) for \( \alpha = 2, \ldots, n' \). The simple procedure is described from step 15 to 21 in Algorithm 4. If all i-th blocks are valid and \( i_2 \) is less than \( n' \), we will divide i-th blocks into two groups,

\[
\{m_{1+n'(i_2-1)}, \ldots, m_{i_1+n'(i_2-1)}, m', \text{NULL}, \ldots, \text{NULL} \}_{n'-i_2-1}
\]

and

\[
\{m_{i_1+1+n'(i_2-1)}, \ldots, m_{n'i_2}, \text{NULL}, \ldots, \text{NULL} \}_{i_2}
\]

and store the first group in i-th row and insert the later at the end of i-th row. The process is described from step 22 to 35 in Algorithm 4 for detail.

3.3 Decoding

To use entrusted data, the data owner may retrieve distributed chunks from \( n' + 1 \) servers and reconstruct the data. In Algorithm 5, we describe the decoding procedure. Intuitively, we refer i-th state information to select valid blocks among i-th row of \( n' \) storage servers for original data and reconstruct valid strings from the blocks. In the same way, we
Algorithm 5 Data Decoding

Require: \(\{CH_0, \ldots, CH_t\}\): encoded chunks of state information and original data, \(\alpha\): number of blocks in chunks

Ensure: Reconstructed data \(F\) (Note that \(F\) could be a modified one for the firstly stored original version.)

1: Initialize partial string \(ps\)
2: Initialize data structure \(F\) for reconstructed file
3: for \(i\) from 1 to \(\alpha\) do
4: if \(MSB(CH_0[i], 1) = 1\) then
5: \(\text{Append } CH_1[i]\|\cdots\|CH_t'[i] \text{ to } F\)
6: else
7: \(tmp = n' - MSB(CH_0[i] \ll 1, v)\)
8: \(\text{Append } CH_1[i]\|\cdots\|CH_{tmp}[i] \text{ to } F\)
9: end if
10: end for
11: return reconstructed file \(F\)

Algorithm 6 Remove Noise Blocks

Require: \(\{CH_0[i_1, i_2], \ldots, CH_r[i_1, i_2]\}\): a set of data blocks where \(i_1\) and \(i_2\) are indexes of the first block and the last block of the interval to be updated

Ensure: Noise-removed sub-chunks \(\{CH_0[i_1, i_1], \ldots, CH_r'[i_1, i_2]\}\) where \(i_1\) is an index of the last block such that \(i_1 < i_2\)

1: Initialize data structure \(F\) for reconstructed file
2: Run Algorithm 5 for \(\{CH_0[i_1, i_2], \ldots, CH_r'[i_1, i_2]\}\) and \(i_2 - i_1 + 1\) is computed as input where \(CH_1[i_1, i_2] = CH_1[i_1]\|\cdots\|CH_1[i_2]\) and \(CH_1[j]\) is the \(j\)-th block of the chunk \(CH_1\). Note that in the proposed scheme, data is separated into small fragments and scattered in multiple data servers. Therefore, the adversary who wants to obtain the original stored data has to attack or collude all of the data servers. Practically, increasing the number of servers or shredding into smaller fragments is sufficient to make an outside adversary hard to retrieve the original data. If we consider the higher security model, in which even the manager of data servers can be an adversary, then the inside adversary easily can collect all fragments stored in the data servers. For the case of the secrecy of the data is highly important, the user can choose that managing the data encoding procedure by himself and utilizing multiple storage providers so that one adversary collects all data fragments is not easy. Moreover, there are alternative solutions to guarantee the secrecy of the stored data, for example by inserting arbitrary-size random noise between data fragments. For these reasons, we concentrate on preventing modification of stored data, which are summarized as following two theorems.

**Theorem 1.** The proposed scheme is secure against a poison attack.

With the poison attack, we mean that an adversary tries to change a specific part of the stored data without detection of the user. If an adversary succeeds in a poison attack,
then the user believes that the changed data is the original one. Note that changing a specific part of the stored data into other contents can be a more serious problem than data leakage. Changing the number of items or promised time due, etc., in the stored data can damage to real-life of the user. In the proposed scheme, this poison attack from outside or even inside attacker can not succeed with meaningful probability.

Proof. In the proposed scheme, the user checks the authentication code

\[ au = prng(sk, r) + a_1m_1 + \cdots + a_em_e, \]

is correct while the data \((au, m_1, \ldots, m_e)\) is retrieved from the server. Therefore, the adversary who tries poison attack needs to find out another tuple \((au', m'_1, \ldots, m'_e)\) satisfying the equation

\[ au' = prng(sk, r) + a_1m'_1 + \cdots + a_em'_e, \]

where \(prng(sk, r)\) is a pseudo-random number and \(sk, r\) and \(a_i\)'s are all secret to the adversary.

Without loss of generality, we define a poison attacker, \(POI\), as a polynomial time algorithm on the finite field \(GF(2^b)\). For randomly chosen secret variables \(\{a_i\} \in GF(2^b)\) and input values \(\{au, m_1, \ldots, m_e\}\), \(POI\) outputs \(\{au', m'_1, \ldots, m'_e\}\), where \(au' \in GF(2^b), m'_i \in GF(2^b)\). If the output of \(POI\) satisfies the equation

\[ au' = au + a_1(m_1 - m'_1) + \cdots + a_em_e(m_e - m'_e), \]

then \(POI\) wins the game. The advantage of the \(POI\) can be defined as

\[ \mathcal{ADV}(POI) = |Pr[POI wins the game] - 1/|GF(2^b)||. \]

We claim that if there exists no polynomial time algorithm \(POI\) with non-negligible advantage, then the proposed scheme is secure against poison attack.

Note that \(sk\) is the secret key which is kept by the user and \(r\) is a randomly selected nonce whenever the user calculates the authentication code. Since the same value of \(r\) is never used for different authentication codes, it is assumed that no polynomial time algorithm can distinguish \(prng(sk, r)\) from a random string. It means that the adversary cannot construct a system of equations about \(a_i\)'s even if the adversary can access the whole stored data. Therefore the possible strategy for an adversary is to choose random tuple \((au', m'_1, \ldots, m'_e)\).

Note that if the the equation

\[ au' = au + a_1(m_1 - m'_1) + \cdots + a_em_e(m_e - m'_e) \]

has unique solution in \(GF(2^b)\) for any given tuples \((au, m_1, \ldots, m_e)\) and \((m'_1, \ldots, m'_e)\). It implies that the probability of that randomly chosen tuple \((au', m'_1, \ldots, m'_e)\) satisfies the equation

\[ au' = au + a_1(m_1 - m'_1) + \cdots + a_em_e(m_e - m'_e), \]

is \(1/|GF(2^b)|\). Finally, if all \(a_i\)'s are secret values and \(prng(sk, r)\) is never repeated, then the \(\mathcal{ADV}(POI)\) is negligible. It concludes the proof.

\[ \square \]

Theorem 2. The proposed scheme mitigates deletion of stored data.

Proof. In this theorem, we consider another type of attack in which the adversary deletes stored data to disrupt the proper use of the storage service. Note that an attack of this type can be executable only by the outside attackers. Since any adversary cannot modify the stored data without detection, the inside attacker has no merit for deleting data which is maintained by himself. For the outside attackers, it is hard to delete all stored data. In the proposed scheme, the embedded error-correcting code guarantees that a user can recover the deleted data where the attacked storage is less than \(n''/2\), where \(n''\) is the system parameter.

\[ \square \]

4.2 Performance

The main merit of our system is that we need to modify limited blocks for each query, which means that only a few computational cost is required for each query. The technique in [8] also can support dynamic date update queries, but it requires \(O(n)\) operations for each update query, which means that our construction is very efficient in terms of data update since it requires \(O(1)\) operations. So, for the computational complexity, we will focus on the cost of encoding and decoding operations which are performed on the full data differently from other operations that treat only a few blocks. To analyze the efficiency of two main data processing operations, encoding and decoding, we compare their computational complexities with well-known cryptographic operation, the AES-CBC encryption. Note that, for comparison, we choose AES-CBC since it is one of well-known cryptographic function which is used for dealing large-scaled data. In the experiment, we repeat 1000 times of each operation for 100 MB data. As seen in Table 1, our technique guarantees sufficient efficiency.

In our technique, we have to determine the size of some parameters, including the number of servers, the size of a shred, and the size of a random noise. Our technique can support any parameters. So, we can choose a set of parameters for given application. For example, the number of servers will be determined by the expected number of corrupted server and the number of servers can be used. The size of a shred will be determined by the size of a unit data which is changed for each data update. Note that, the goal of

<table>
<thead>
<tr>
<th>Table 1</th>
<th>Comparison.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
<td>Time Complexity</td>
</tr>
<tr>
<td>AES-CBC Encryption</td>
<td>61.9 ms</td>
</tr>
<tr>
<td>Encoding of Our Technique</td>
<td>21.9 ms</td>
</tr>
<tr>
<td>Decoding of Our Technique</td>
<td>28.5 ms</td>
</tr>
</tbody>
</table>
our work is to give stronger security without using encryption schemes, and stronger security can be supported if more servers are used, smaller shreds are used, or larger random numbers are used.

5. Conclusion

In this paper, we proposed a new technique for using dynamic data in cloud storage systems. Specifically, our construction includes the following features. For dynamic data, we design a new data chunk generation strategy. Our chunk generating algorithm guarantee efficient data insertion, deletion, and modification due to the structure of data chunk. Note that existing chunking techniques cannot support data updates efficiently. For each data update, existing techniques require $O(n)$ operations where $n$ is the size of data. Due to the suitable data structure of chunks, our technique requires $O(1)$ operations for update queries.

References


