CloudEFS: Efficient and Secure File System for Cloud Storage

Clemens Zeidler
Department of Computer Science
The University of Auckland
Auckland 1142
New Zealand
Email: clemens.zeidler@auckland.ac.nz

Muhammad Rizwan Asghar
Department of Computer Science
The University of Auckland
Auckland 1142
New Zealand
Email: r.asghar@auckland.ac.nz

Abstract—**Although outsourcing data to the cloud has many advantages, cloud computing introduces new privacy and security requirements on how data is stored and accessed. To ensure data confidentiality, use of encryption is a common strategy. A problem with this strategy is that for large data it becomes difficult to access and update data, and to ensure data integrity and data provenance without decrypting all the data. The problem becomes severe when it comes to accessing and modifying large files using mobile devices with limited processing and bandwidth capabilities. In this paper, we present *CloudEFS*, a novel storage framework to efficiently access and update large encrypted data. A new cached hash algorithm enables efficient updates of data integrity and provenance information. *CloudEFS* provides improved privacy by hiding not only content but also metadata such as data size, file count, file structure and file history.**

I. Introduction

Cloud storage is a convenient and reliable solution for storing and processing data at very attractive costs. However, outsourcing data to the cloud raises serious privacy concerns. To ensure data confidentiality, we can employ encryption techniques [1]. Once encryption techniques are employed, it becomes difficult to access and update data in an efficient manner. Another potential issue is access control management where the data could be shared among a set of users. That is, only authorised users must be allowed to access and process the data stored in the cloud. Since the data is shared, it is essential for users to know the origin of the data and verify the chain of integrity. In other words, to verify data integrity and origin, integrity and provenance information should be maintained [2], [3].

While data encryption, data integrity and provenance information are techniques to fulfil privacy and security requirements, it has not been investigated how these techniques perform for dynamic data, *i.e.,* data that is edited or updated frequently. In particular, there are two performance-related problems. First, making only small edits in plaintext may result in significant changes in the corresponding ciphertext [4]. Second, integrity and provenance information is usually calculated using a cryptographic hash function [2], [3]. However, similar to the first problem, only a small data modification makes it necessary to recalculate the data hash. For large dynamic data, re-encryption and data hashing become expensive. The problem becomes more severe when it comes to accessing and modifying large files using mobile devices with limited processing and bandwidth capabilities. That is, accessing or updating large files on mobile devices is inefficient.

One solution to store and synchronise encrypted dynamic data efficiently is to split data into dynamic chunks [5], [6], [4]. However, the problem of how data integrity or provenance information can be updated and verified efficiently has not been thoroughly investigated. Another problem is that existing secure storage solutions leak metadata such as directory structure, file names, the number of files or file sizes in one way or another.

In this work, we propose *CloudEFS*, a novel storage framework that makes it possible to access and update large encrypted data. We introduce a general purpose cached hash function that allows updating hash values efficiently when the data is modified. More specifically, this makes it possible to efficiently calculate integrity and provenance data for large dynamic data. In *CloudEFS*, data is stored in chunk containers. Content and chunk containers are stored as chunks. Based on chunk containers, we describe a file system object model to represent files, directories and data history. Compared to other storage solutions, we do not reveal sensitive information, such as directory structure or file number and size.

Our contributions can be summarised as follows:

- A storage model for cloud storage with enhanced privacy, where the content and its associated metadata do not reveal sensitive information.
- We introduce a cached hash function to efficiently calculate a hash value of dynamic data that gets modified frequently. This, in turn, speeds up verification of data integrity and provenance.
- We introduce chunk containers to store data which is divided into chunks. Chunk containers make access and data modification efficient.

II. Motivating Scenarios and Requirements

In this section, we describe two motivating scenarios based on which we derive the requirements for *CloudEFS*. The first scenario motivates why efficient, integrity-preserving data operations and management of data history are necessary for storing large scientific dataset in the cloud. The second scenario motivates why for some systems it is not only essential to protect file content from the cloud but also equally important to hide the file system information including file size, file count and directory structure.

***CloudEFS* for Scientific Data.** Let us assume a researcher collects a large scientific dataset. She leverages a cloud-based storage for storing this dataset. We consider that the dataset is dynamic, which might require frequent updates. Since she is collaborating with a number of fellow researchers, this dataset will be shared among them. Fellow researchers are interested in investigating some new aspects whenever the dataset gets updated. As desired in scientific experimentations, a researcher should be able to revert to earlier version of the dataset at any time. Moreover, fellow researchers must be able to verify data integrity and provenance, *e.g.,* to prevent a malicious party (say a cloud service provider, an untrusted collaborator or any third part) to manipulate research results. The study results must be kept confidential. For this reason, the dataset is encrypted. A suitable storage system for such a scenario must allow the owner of the large dataset to efficiently update the encrypted dataset and provide integrity and provenances information. This becomes challenging since re-encrypting the whole large dataset or calculating integrity information becomes too expensive, in particular when the dataset gets updated frequently. On the other hand, authorised users must be able to efficiently access or synchronise the updated dataset and verify integrity and provenance information.

***CloudEFS* for Healthcare Systems.** In this scenario, we assume a cloud-based data storage system for storing medical data of patients. To ensure the privacy of the patients, medical data is stored encrypted. Current cloud storage solutions that use data encryption, unfortunately, leak information about the file system to the cloud service provider, *e.g.,* the file number and size, or the directory structure. This information could be used to analyse what medical examination has been performed. For example, knowing a file size or the count of files could reveal certain examinations, *e.g.,* X-ray and MRI images. This information is sensitive as it may allow an adversary to guess the medical conditions of a patient. For this reason, not only the file content but also the directory structure and the file size must be protected from curious cloud stores.

From both scenarios, we infer the following requirements that *CloudEFS* must fulfil:

- **Privacy:** The content as well as metadata such as directory structure, file number and size, and data provenance must be kept secret from the cloud.
- **Authenticity and Integrity:** Users must be authenticated and must be able to ensure and verify data integrity and data provenance. The cloud storage should support a version control system to keep data history with options for the user to easily revert to earlier data revisions.
- **Efficiency:** Accessing and updating a dataset as well as ensuring and verifying integrity and provenance information must be efficient.

III. System Model

In the system model, we assume the following entities (see Figure 1):

- **Cloud Service Provider (CSP).** The CSP is responsible to store data in a key-value store, *a.k.a. Chunk Store*. The CSP ensures that only authorised users can read or append data.
- **Data Owner.** This entity represents a user who owns the data stored at the CSP and regulates access to the data by deploying access policies at the CSP. The data owner is a client of the CSP.
- **User.** A user is an entity that can access or modify data at the CSP according to deployed access policies.

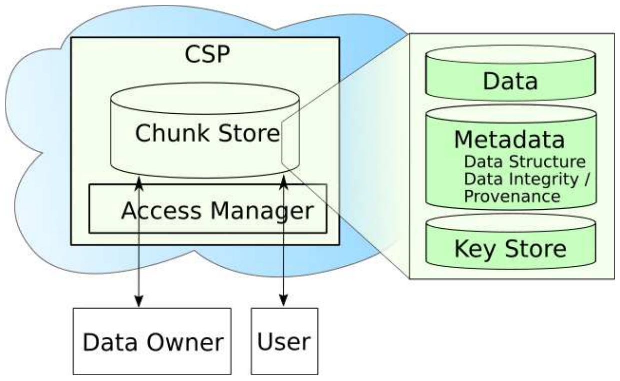

Fig. 1. A Cloud Service Provider (CSP) hosts the Chunk Store and is responsible for managing access to the chunk store in which the data is stored in forms of chunks. For data owners and users, access to the data is regulated through the Access Manager. The data includes the actual data (*e.g.,* files), the metadata and a key store. The metadata contains structural information, *e.g.,* directory structure, data integrity and provenance information. The actual data and the metadata are encrypted using encryption keys that are stored in an encrypted form in the key store.

We assume that the CSP is honest-but-curious as is considered in scenarios where data is stored in outsourced environments [7]. This means users cannot trust the CSP to guarantee confidentiality. For this reason, users who are authorised to modify data encrypt sensitive data locally before sending to the CSP. We consider that the data is updated dynamically.

Furthermore, we assume that an adversary (say a CSP or third party) may tamper with the data or its provenance to deceive data owners or users. Moreover, we consider that there are mechanisms in place for ensuring availability.

IV. *CloudEFS*: Proposed Storage Framework

In this work, we propose *CloudEFS*, a storage framework that aims at fulfilling requirements listed in Section II. More specifically, data owners or users can store their data in the cloud in a privacy-preserving manner. Authorised users can get access to the data if corresponding access policies are satisfied. These policies are enforced by the Access Manager at the CSP. *CloudEFS* enables user authentication and ensures integrity of the data and its provenance. In order to make access and verification efficient, in *CloudEFS*, we divide each data item (*e.g.,* file) into a set of data chunks. *CloudEFS* also supports management of a version control system to keep data history. Like data, data history is stored securely.

As we can see in Figure 1, a CSP hosts the chunk store and is responsible for managing access to the chunk store in which the data is stored in the form of chunks. For data owners and users, access to the data is regulated through the Access Manager. The data includes the actual data (*i.e.,* files or directories), the metadata and a key store. The metadata contains structural information, *e.g.,* directory structure, data integrity and provenance information. The actual data and the metadata are encrypted using encryption keys that are stored in an encrypted form in the key store [8], [9].

In the following, we describe how *CloudEFS* stores data in a chunk store. We first introduce the chunk store, which is used to store encrypted data chunks (Section IV-A). We then discuss how data of arbitrary size is split into a set of chunks (Section IV-B). To efficiently verify integrity of the data, which could be updated dynamically, we introduce a novel, general purpose cached hash function (Section IV-C). We describe how data chunks are encrypted to achieve confidentiality (Section IV-D). We also explain how data chunks are managed in a chunk container (Section IV-E).

A. Chunk Store

The chunk store is a database of key,value pairs, capable of storing an arbitrary chunk of data, *e.g.,* a chunk of a file, where each chunk is identified using a unique ID or key. In the following, we call a piece of data stored in the chunk store a *chunk* and the corresponding ID the chunk ID, denoted as: ID_{chunk}. A chunk can have an arbitrary size. As described later, all chunks in the chunk store are encrypted.

Although there could be a number of options for uniquely identifying a chunk, without loss of generality, we suggest a collision-resistant hash function H to calculate the chunk ID: ID_{chunk} [6]. This choice has various advantages: First, no entity is needed to generate unique identifiers because chunk keys are most likely to be globally unique; Second, storage integrity of the stored (encrypted) chunks can easily be verified; Third, if an adversary modifies a chunk, the modification can easily be detected, *e.g.,* when reading the chunk [10]. Note, the encryption-independent integrity of the plaintext is discussed later (Section V).

B. Data Chunking

In this section, we describe how data is split into a set of *data chunks*. As we describe later, we use this technique to split not only data, such as files, but also metadata that describes a set of chunks.

In this work, we consider a chunking algorithm C that takes some data as input and returns a sequence of chunk boundaries. Technically, a chunk boundary denotes a starting and ending indices of a chunk in the data. According to chunk boundaries, the data is split into n chunks. A naive approach for identifying chunk boundaries is to compute fixed sized chunks. However, we assume dynamic data that could be edited (say files could be modified by updating or removing existing content and adding new content). A chunking algorithm that produces fixed-size chunks is not suitable for *CloudEFS* since an insertion or deletion of a single byte would affect all the following chunks. On the other hand, in *CloudEFS*, we require C to produce boundaries that result in a stable set of chunks for similar data. For example, if a file is only slightly modified, unlike the naive approach, both versions of the file should share a large set of equal chunks [11].

Common chunking algorithms use a Rolling hash function $r(W)$ [12], [5]. $r(W)$ produces a pseudorandom fingerprint on a sliding window W. We use the following trigger condition to find chunk boundaries:

$$r(W) \; mod \; D < D/c$$

Assuming the input data is random, c is the average chunk size, *i.e.,* on average, for every c bytes, $r(W)$ triggers a chunk boundary. D is an arbitrary value with $c < D < max(r(W))$ and is used to select $r(W)$ values in the range $[0, D)$.

To avoid very small and very large data chunks, we apply a minimum and maximum chunk size threshold: namely T_{min} and T_{max} [5]. In the following, we assume a large T_{max} so that in practice it does not affect the chunk size distribution. The total average chunk size can then be approximated to: $\overline{s} = T_{min} + c$. This leads to a chunking algorithm:

$$C(\overline{s}, T_{min}, T_{max})$$

that produces chunks with an average size $\overline{s}$ in the range $[T_{min}, T_{max}]$ [5].

If the input data is too short to trigger a chunk boundary, we pad the data with random bytes unless a boundary gets triggered. This is done to prevent adversaries to learn the data size of smaller data. Similarly, the last data chunk is padded.

C. Cached Hash Function

In the following, we define a general-purpose, collision-resistant *cached hash function*. As its name suggests, the cached hash algorithm makes it possible to cache partial results of the hashing process. These partial results are less affected by small edits in the data. This stable property of partial results makes it possible to reuse them and efficiently calculate the hash value of modified data.

A cached hash function works by organising data chunks in a Merkle tree [13]. The top hash of this *cached hash tree* is used as a hash value for the data as illustrated in Figure 2a. It is known that Merkle trees are as collision-resistant as the underlying hash function [13], *i.e., H*.

To keep the cached hash tree stable when only parts of the data are modified, the tree nodes are chunked using a similar chunking mechanism as described in Section IV-B. Thus, only a small set of nodes are affected when a small modification is made.

The cached hash Merkle tree is built as follows. After splitting the initial data into chunks, using the chunking algorithm $C(\overline{s}, T_{min}, T_{max})$, the chunk hashes h are combined in a single node n_h. Using a different node chunking

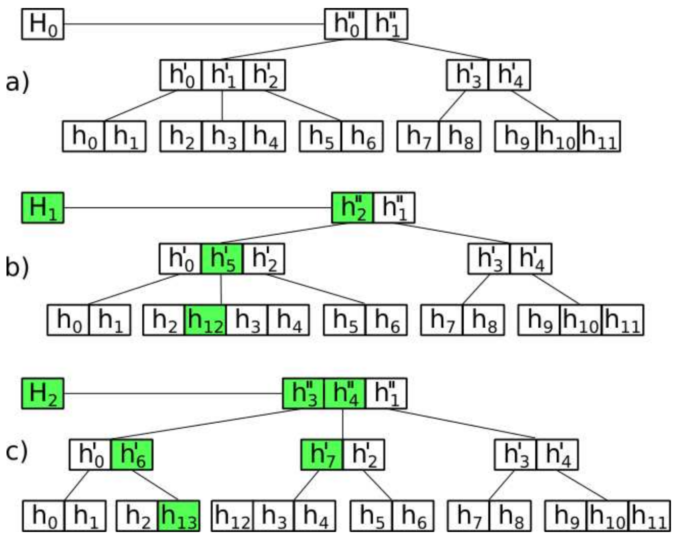

Fig. 2. The hashes of the data chunks are organised in a Merkle tree, where a hash output of a chunk i is represented using h_i. Intermediate nodes of the tree are built based on children nodes. The top (or root) hash of the tree provides us the hash of the data item. a) H_0 represents the top hash of the Merkle tree of the first data version. b) H_1 denotes the second data version, where a new chunk 12 is inserted between chunk 2 and chunk 3. The insertion of a new chunk in the tree does not trigger any split at the node level. c) H_2 shows a tree with the insertion of a new chunk 13 between chunk 2 and chunk 12. The insertion of the corresponding hash h_{13} results in a split at the node level and at the parent level. The right subtree does not need to be recomputed.

algorithm $C_{node}(n_h)$, the node n_h is then recursively split into cached hash tree nodes till the root node cannot be split further (see Figure 2a). The hash value h' for a cached hash tree node with n entries $h_0, \cdots, h_{n-1}$ is defined as $h' = H(concatenate(h_0, \cdots, h_{n-1}))$.

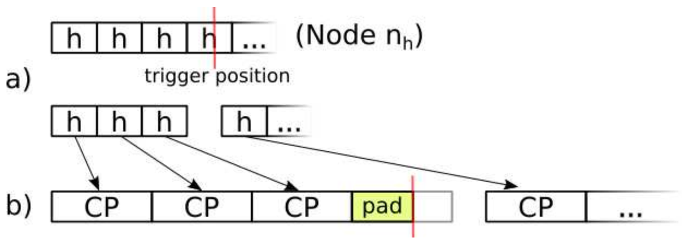

Fig. 3. a) A chunking algorithm C_{node} is applied to a node n_h. The hash value h that causes C_{node} to trigger is the first h value of the next chunk. b) Chunk pointers (CP) point to data chunks or chunk container nodes. In a chunk container node, hash values h from a cached chunk tree node are mapped to chunk pointers (CP). For privacy reasons, the chunk pointer nodes are padded.

The node chunking algorithm $C_{node}(n_h)$ is defined as follows. As described later in Section IV-E, for privacy reasons, the size of cached hash tree nodes needs to be scaled relative to the size of data chunks. To achieve this, $C_{node} = C(k \cdot \bar{s}, k \cdot T_{min}, k \cdot T_{max})$ with a scaling factor k is used to find split points within a given cached hash tree node n_h. When $C_{node}(n_h)$ triggers a chunk boundary, the node n_h is split before the hash value $h_s \in n_h$ that triggered the chunk boundary. Thus, h_s is the first hash in the next chunk (see Figure 3a).

Figures 2b and 2c illustrate how insert operations only affect some parts of the cached hash tree. This makes it efficient to calculate the top hash when data is edited since only a few nodes need to be considered.

D. Chunk Encryption

To ensure data confidentiality, data chunks are encrypted. Although there are a number of options to achieve data confidentiality, convergent encryption [14] could be applied to store chunks efficiently. In convergent encryption, the encryption key is the data chunk hash $H(chunk)$ and thus same data chunks results in the same ciphertext. However, convergent encryption is vulnerable to certain attacks, *e.g.,* dictionary and other attacks [15], [16]. We use a different encryption strategy that uses a private key K in combination with the hash $H(chunk)$ of the encrypted chunk. The chunk is encrypted with a pseudorandom permutation function:

$$E(K, H(chunk), chunk).$$

that results into the same ciphertext for same chunks when using the same K [4].

E. Chunk Container

Recall that we split data into a set of data chunks and encrypt these chunks. In the following, we describe how these data chunks are managed in a *chunk container*. We have two design goals for a chunk container: First, reading from a chunk container or writing to a chunk container must be efficient. Second, chunks to chunk container mapping and the structure of the chunk container must be kept confidential.

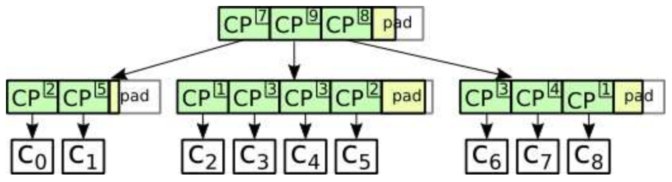

Fig. 4. Data chunks are organised in a chunk container. The size value of a chunk pointer (CP) (displayed in the right-top corner of a CP) makes it easy to seek to a certain data position. For example, the total size of the data stored in the chunk container is: $(2+5)+(1+3+3+2)+(3+4+1) = (7+9+8) = 24$. The position 12 is located in chunk C_4; C_4 starts at position $7+1+3 = 11$ and the size of C_4 is 3.

A chunk container is organised in a height balanced tree with its leaf nodes pointing to data chunks. Each *chunk container node* is a chunk that is encrypted and stored in the chunk store. A chunk container node contains a list of *chunk pointers*, which either point to other chunk container nodes or to data chunks. In this way, to access a file, only the root chunk container node is required to navigate to all associated data chunks (Figure 4). The chunk container tree structure (Figure 4) mirrors the data's cached hash tree structure (Figure 2), *i.e.,* every chunk container node has a corresponding node in the cached hash tree (Figure 3).

A chunk pointer (Figure 5) contains a reference ID_{chunk} to an encrypted chunk in the chunk store. To decrypt the

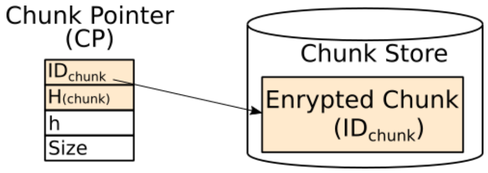

Fig. 5. A chunk pointer (CP) points to an encrypted chunk stored in the chunk store. The CP contains the chunk hash $H(chunk)$ to decrypt the referred chunk, a chunk data integrity value h, and a data size value.

referred chunk, the chunk pointer contains $H(chunk)$ (see Section IV-D). The chunk pointer also contains the corresponding h value from the cached hash tree. Furthermore, the chunk pointer contains a data size value. Since chunk pointers are stored in encrypted chunk container nodes, no information on how chunks are related is revealed.

If the chunk pointer refers to a data chunk, the size value gives the size of the data chunk. If the chunk pointer refers to a child node, the size value gives the sum of the size values of the child's chunk pointers. This makes it easy to seek and modify data in the chunk container tree (Figure 4).

Like normal chunk pointers, we define a chunk container pointer which points to the root node of the chunk container. The only difference to a normal chunk pointer is that the size value represents the height of the chunk container tree.

The size of a CP, *i.e.,* the size to store a CP, is larger than the size of a cached hash tree value h. This results in "large" chunk container nodes with sizes a multiple of the chunk pointer size. This raises a privacy issue since an attacker may distinguish between data chunks and chunk container nodes and may deduce further information from that.

For this reason, the average size of chunk container nodes is adjusted to match the average size of data chunks. To achieve this, the k scaling factor in the cached hash chunking algorithm $C_{node}(k \cdot \overline{s}, k \cdot T_{min}, k \cdot T_{max})$ is set to $k = size(h)/size(CP)$ (Section IV-C). Cached hash tree nodes are split before the trigger point and thus chunk container nodes are generally too small. For this reason, we pad the chunk container nodes with random data with a random length in the range $[0, size(CP))$ (Figure 3b).

V. File System Object Model

Chunk containers can hold an arbitrary amount of encrypted data. In this section, we use chunk containers to define a file system object model. This model, similar to other successful models[1] [17], [18], manages files and directories as well as integrity and provenance information. Compared to previous work, we require that all objects are encrypted and that the information how objects are related is not leaked.

There are three types of objects: namely files, directories and commits. A commit marks a version of the current directory structure. A commit contains a reference to previous parent commits and to a root directory while directories can contain files or subdirectories. As illustrated in Figure 6, objects can refer to objects from previous commits, this is what makes the object model storage efficient [17]. The linked list of commit objects is called a branch.

[1]Git:git-scm.com

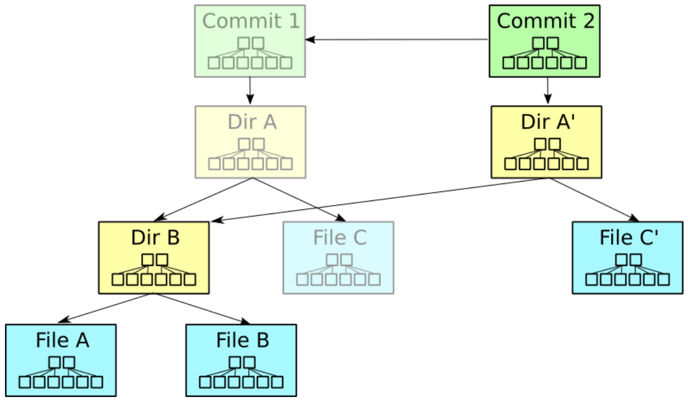

Fig. 6. The file system structure is organised in directory and file objects. Commit objects are used to refer to the certain version of a directory structure. Objects can be reused between versions. For example, in Commit 2 only File C has been modified and the unchanged directory Dir B is reused from the previous version. From the latest commit, all previous versions can be accessed.

Every object in our object model is stored in a chunk container. Furthermore, every type of object defines a plain data hash H_{obj}. This hash value uniquely identifies an object based on the containing plain data and is used to build a specialised Merkel tree [18]. This means that H_{obj} of a commit depends on the H_{obj} values of the parent commits and the root directory. The H_{obj} value of a directory depends on the hash values of all its children while H_{obj} of a file object is simply the cached hash of the file chunk container.

A. Integrity and Provenance

Using the object model described above, it becomes very easy to ensure the integrity and to provide provenance information. Given the hierarchical structure, knowing the commit hash H_{obj} is enough to verify the integrity of all object belonging to this commit as well as of all parent commits.

Provenance information can easily be integrated into a commit object. To do so, the user who is about to add a new commit compiles a message that contains the hashes of the direct parent commits and the current root directory. The message is then signed. In this way, the user can ensure what changes in the current version has been made relative to a previous version, *i.e.,* the changes are the differences to the previous version.

Since H_{obj} only depends on plain object data, our model allows re-encryption without losing integrity and provenance data.

B. Key Management

To manage and share a large set of encryption keys, we propose to store keys securely at the CSP [19], [8]. This can be done by storing keys in a "key store" branch. The key store branch is protected using the branch key K_{branch}, *i.e.,* all objects in the key store branch are encrypted with K_{branch}.

In this way, a user only needs to know K_{branch} to gain access to all user's keys.

C. Access Control

In general, it is desirable that only eligible contacts are able to access/add chunks from/to the chunk store [20]. However, if the number of chunks is large, managing access becomes expensive both in terms of storing the access policy as well as in terms of verifying that a contact has access to a set of chunks. Furthermore, if not properly protected, the CSP can learn who has access to which chunks.

We propose a simple access model. The user just has to prove that she has read and/or write access to a certain branch. Once read or write access is granted, a user is allowed to access all chunks in that branch.

Since all chunks are encrypted with a private key and stored under the hash $H(chunk)$, guessing an existing chunk without knowledge of the private key is only possible with a negligible probability. However, if the private key is known to the user, the user can launch confirmation of data attacks [15]. For this reason, it is important that a private key is not used for chunks that are not supposed to be shared with users who know the private key.

To revoke user access to certain objects, we propose a lazy revocation strategy [1], [21]. For this, we assume that the user already knows the file content before the access is revoked. Thus, we only need to ensure that the user cannot access newer file versions. The lazy revocation scheme works simply by changing the encryption key for the file. Using this strategy all new chunks of a modified version of the file cannot be located or decrypted by the user.

D. Data Sharing

At this stage, we did not discuss how users can keep track of the latest commit objects, which are needed to access the latest changes. One solution is that the user who made a change is responsible for notifying all other users about the newest commit via a secure channel. However, this approach may not always be practical and we propose a simpler solution.

In *CloudEFS*, the CSP maintains a branch log for each branch. When making a change, the user encrypts the chunk container pointer of the latest commit and appends the encrypted pointer to the branch log. This way all users, who have access to the branch, can query and decrypt the latest chunk container pointer. The disadvantage is that the CSP learns about changes made to a particular branch.

VI. Performance Analysis

In this section, the performance of the cached hash function (as described in Section IV-C) is evaluated. First, we analyse how the cached hash function performs without caching. Second, we measure the cached hash performance when data updates are made.

For our analysis, we used a Java implementation of the cached hash algorithm. We used a Rabin rolling hash function for the chunking algorithm C [12] and SHA256 for the hash function H. For the maximum chunk size, we chose $T_{max} = 512kB$ (see Section IV-B). As a minimum chunk size, we chose $T_{min} = 2kB$, which is consistent with previous work [11]. We performed our measurements on a machine running Linux having 1.9Ghz CPU and 8GB of RAM.

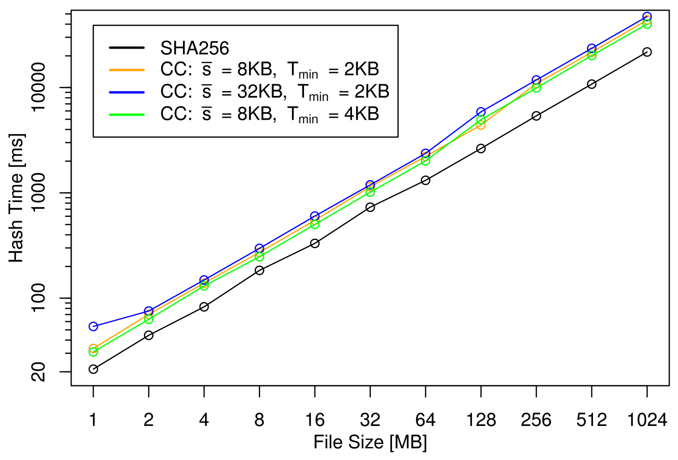

Fig. 7. Performance analysis of the cached hash algorithm without caching.

In our first analysis, we compared the performance of the cached hash algorithm and the default Java implementation of SHA256. We applied both hash functions to random data. To minimise IO latency, data was kept in memory. Figure 7 shows the average times for two different average chunk sizes $\bar{s}$. SHA256 function performs better. This is expected since the cached hash function applies the Rabin rolling hash function in conjunction with a SHA256 hash. Interestingly, a larger average chunk size $\bar{s}$ decreases the performance slightly, although fewer nodes are needed for the cached hash tree. We assume that the reason for this is that, in the range $(0, T_{min})$, no Rabin rolling hash is applied and a test with $T_{min} = 4KB$ improves the performance, which confirms this assumption (see Figure 7).

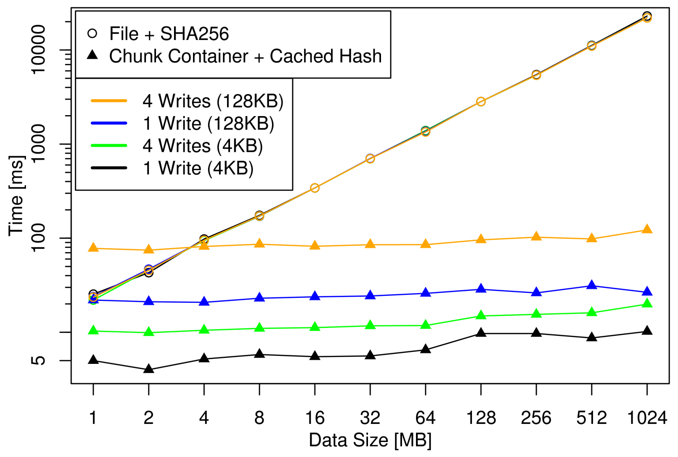

Fig. 8. Performance of the hash calculation using the cached hash function and SHA256 after performing random write operations. Variables are the number of writes and the size of modified data.

Next, we perform random writes to data and calculate the new data hash using the cached hash and the SHA256 algorithm. For the calculation of the cached hash value, we use a Java implementation of a chunk container that stores data on disk. Every write to the chunk container automatically updates the data hash. For SHA256, data is modified in an ordinary data file and then the file hash is calculated. To simulate a more realistic scenario, we perform multiple write operations before starting the hash calculation. However, due to the nature of a chunk container, the cached hash is automatically updated after each write operation. Figure 8 shows that the performance of a chunk container is generally lower than the SHA256 performance and scales very well to large data sizes, *i.e.,* the time to update data in a chunk container only grows slightly. For large files, calculating the SHA256 value makes writes to a file very expensive.

VII. Discussion

In this section, we discuss how *CloudEFS* fulfils the requirements defined in Section II.

Privacy. The confidentiality of all data and metadata is preserved since all data is stored in encrypted chunk containers. Sizes of data chunks are chosen such that they reveal little about the actual data size. That is, data that is too small to trigger a chunk boundary is padded and chunk container nodes are made in such a way that they are indistinguishable from data chunks. However, using a data depending chunking algorithm always leaks a controlled amount of information [4]. *CloudEFS* stores equal chunks only once and thus does not reveal information about repeating data.

When analysing access pattern, an attacker can gain information about the stored data, *i.e.,* what chunks may belong to the same data. If a whole file is accessed, the size of the file is revealed [22]. However, in *CloudEFS*, data can be accessed selectively, which makes it more difficult for an attacker to correctly map chunks to a certain file. A solution to mitigate this problem is to retrieve additional random chunks that do not belong to the actual data [23].

Integrity and Authenticity. Integrity and provenance is provided by the used object model (Section V). The provenance information includes the identity of the user who made a change. Thus, the data owner and the users can verify if unauthorised changes were made to the data. However, the CSP can revert to a previous storage stage, which may get unnoticed by the users.

Efficiency. The chunk container allows accessing and updating large data efficiently without touching the whole data and the cached hash algorithm allows efficient hash calculation for dynamic data. A drawback of the cached hash function is the overhead when calculating the initial data hash. However, the cached hash tree is reused to set up a chunk container and thus to split data into chunks. This is required for storage efficiency [24] and reduced network bandwidth [11]. A drawback of the chunk container tree structure is that multiple requests are necessary to navigate to a data chunk. However, for sequential operations, this can be mitigated by reading upcoming chunk container nodes ahead of time.

VIII. Related Work

Splitting dynamic data into multiple chunks that are stable against small data modification has various advantages and has been used to efficiently transfer files across a network [11], [5]. By storing duplicated chunks only once, data and the data history can be stored more efficiently [17], [4]. Even for fixed sized chunks, this can help to reduce storage requirements of dynamic data [17], [24]. We store data chunks in a key-value store database and it has been shown that such a store can efficiently be used for distributed file systems [25].

For large chunked files the required chunk map can become very large [26]. This makes accessing a file slow since the whole chunk map needs to be retrieved. In comparison, chunk containers allow users to efficiently seek to a certain file position only accessing required chunks.

To ensure confidentiality, data chunks need to be encrypted. To manage encryption keys, they can efficiently be derived from a master key [21]. Another method to encrypted data chunks is convergent encryption, which has the property that same chunks have the same ciphertext [27]. This property is particularly used to perform de-duplication of encrypted data for chunk based storage solutions as well as to find duplicate encrypted files from multiple users [6], [28], [29], [15], [30], [14]. In our work, we use a convergent encryption technique that uses an additional private key [4].

There are many storage solutions that ensure data confidentiality by employing encryption techniques [31], [32], [1]. Even in distributed cloud storage environments, data confidentiality can be ensured [33], [34]. However, above solutions do not attempt to hide directory structure, file size or the number of files. The distributed file system BRAVE [35] uses an object store to store encrypted file and directory objects. However, the size and number of objects are not hidden. Blind Storage scatters blocks of files on the storage server and even mixes blocks of multiple files. While initially the number and sizes of stored files are hidden, a file size is revealed when a file is accessed [23]. Storer *et al.* use a chunk store to store encrypted chunks of a file [10]. However, the used metadata store reveals file names, the number of files, information about access rights, and the size of an encrypted chunk map indirectly reveals the size of the file. In contrast, we never reveal the existence and size of a file.

While full disk encryption systems, such as the Linux Unified Key Setup (LUKS)[2], VeraCrypt[3] or BitLocker[4], provide a transparent block encryption, they have various disadvantages when used in a cloud environment. Basically, data within one partition or storage file is encrypted with a single key, which can be problematic if data is shared between multiple users. Furthermore, data integrity or provenance information is not provided.

[2]https://gitlab.com/cryptsetup/cryptsetup/wikis/home

[3]https://veracrypt.codeplex.com/

[4]https://technet.microsoft.com/en-us/library/cc732774.aspx

IX. Conclusion

Updating large encrypted data can become inefficient since small changes in the plaintext can result in significant changes in the corresponding ciphertext. In order to provide integrity and provenance information, calculating the hash of large data is time-consuming. In this work, we presented *CloudEFS*, a novel storage framework for storing encrypted dynamic data and ensuring data integrity and provenance. Large dynamic data can efficiently be accessed and modified. To update integrity and provenance information, we presented a general purpose cached hash function to efficiently calculate the hash of updated data. *CloudEFS* provides improved privacy by concealing not only the content but also the metadata such as file sizes and file structure. Our performance analysis shows that the cached hash function is only a constant factor slower than the underlying hash function (such as SHA265). However, it performs significantly better when updating the data hash of modified data.

In future, we plan to evaluate the effectiveness of *CloudEFS* for real-world applications.

References

[1] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu, "Plutus: Scalable secure file sharing on untrusted storage." in *FAST*, vol. 3, 2003, pp. 29–42.

[2] H. Hacigümüş, B. Iyer, and S. Mehrotra, "Ensuring the integrity of encrypted databases in the database-as-a-service model," in *Data and Applications Security XVII*, ser. IFIP International Federation for Information Processing, S. De Capitani di Vimercati, I. Ray, and I. Ray, Eds., 2004, vol. 142, pp. 61–74.

[3] M. R. Asghar, M. Ion, G. Russello, and B. Crispo, "Securing data provenance in the cloud," in *Open Problems in Network Security*, ser. Lecture Notes in Computer Science, J. Camenisch and D. Kesdogan, Eds., 2012, vol. 7039, pp. 145–160.

[4] D. Leibenger and C. Sorge, "A storage-efficient cryptography-based access control solution for subversion," in *Proc. of the 18th ACM symposium on Access control models and technologies*, 2013, pp. 201–212.

[5] K. Eshghi and H. K. Tang, "A framework for analyzing and improving content-based chunking algorithms," *Hewlett-Packard Labs Technical Report TR*, vol. 30, p. 2005, 2005.

[6] D. Bhagwat, K. Eshghi, D. D. Long, and M. Lillibridge, "Extreme binning: Scalable, parallel deduplication for chunk-based file backup," in *2009 IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems*, 2009, pp. 1–9.

[7] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, G. Pelosi, and P. Samarati, "Preserving confidentiality of security policies in data outsourcing," ser. WPES '08, 2008, pp. 75–84.

[8] C. Zeidler and M. R. Asghar, "Privacy-preserving data sharing in portable clouds," in *Proc. of the 6th International Conference on Cloud Computing and Services Science*, 2016, pp. 274–281.

[9] G. Gheorghe, M. R. Asghar, J. Lancrenon, and S. Ghatpande, "SPARER: Secure cloud-proof storage for e-health scenarios," 2016.

[10] M. W. Storer, K. Greenan, D. D. Long, and E. L. Miller, "Secure data deduplication," in *Proc. of the 4th ACM international workshop on Storage security and survivability*, 2008, pp. 1–10.

[11] A. Muthitacharoen, B. Chen, and D. Mazieres, "A low-bandwidth network file system," in *ACM SIGOPS Operating Systems Review*, vol. 35, 2001, pp. 174–187.

[12] M. O. Rabin *et al.*, *Fingerprinting by random polynomials*, 1981.

[13] R. C. Merkle, *A Digital Signature Based on a Conventional Encryption Function*, 1988, pp. 369–378.

[14] J. R. Douceur, A. Adya, W. J. Bolosky, P. Simon, and M. Theimer, "Reclaiming space from duplicate files in a serverless distributed file system," in *Distributed Computing Systems, 2002. Proceedings. 22nd International Conference on*, 2002, pp. 617–624.

[15] P. Puzio, R. Molva, M. nen, and S. Loureiro, "ClouDedup: Secure deduplication with encrypted data for cloud storage," in *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*, vol. 1, Dec 2013, pp. 363–370.

[16] S. Halevi, D. Harnik, B. Pinkas, and A. Shulman-Peleg, "Proofs of ownership in remote storage systems," in *Proc. of the 18th ACM conference on Computer and communications security*, 2011, pp. 491–500.

[17] S. Quinlan and S. Dorward, "Venti: A new approach to archival storage." in *FAST*, vol. 2, 2002, pp. 89–101.

[18] A. J. Mashtizadeh, A. Bittau, Y. F. Huang, and D. Mazières, "Replication, history, and grafting in the ori file system," in *Proc. of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13, 2013, pp. 151–166.

[19] L. Ferretti, M. Colajanni, and M. Marchetti, "Distributed, concurrent, and independent access to encrypted cloud databases," *Parallel and Distributed Systems*, vol. 25, no. 2, pp. 437–446, February 2014.

[20] V. Kher and Y. Kim, "Decentralized authentication mechanisms for object-based storage devices," in *Security in Storage Workshop, 2003. SISW '03. Proceedings of the Second IEEE International*, Oct 2003, pp. 1–1.

[21] W. Wang, Z. Li, R. Owens, and B. Bhargava, "Secure and efficient access to outsourced data," in *Proc. of the 2009 ACM Workshop on Cloud Computing Security*, ser. CCSW '09, 2009, pp. 55–66.

[22] M. Dawoud and D. T. Altilar, "Privacy-preserving data retrieval using anonymous query authentication in data cloud services," in *Proc. of the 6th International Conference on Cloud Computing and Services Science*, 2016, pp. 171–180.

[23] M. Naveed, M. Prabhakaran, and C. A. Gunter, "Dynamic searchable encryption via blind storage," in *Security and Privacy (SP), 2014 IEEE Symposium on*, 2014, pp. 639–654.

[24] C. Constantinescu, J. Glider, and D. Chambliss, "Mixing deduplication and compression on active data sets," in *2011 Data Compression Conference*, 2011, pp. 393–402.

[25] E. Pavlidakis, S. Mavridis, G. Saloustros, and A. Bilas, "KVFS: An HDFS library over NoSQL databases," in *Proc. of the 6th International Conference on Cloud Computing and Services Science*, 2016, pp. 360–367.

[26] D. Meister, A. Brinkmann, and T. Süß, "File recipe compression in data deduplication systems," in *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*, 2013, pp. 175–182.

[27] M. Bellare, S. Keelveedhi, and T. Ristenpart, *Message-Locked Encryption and Secure Deduplication*, 2013, pp. 296–312.

[28] F. Rashid, A. Miri, and I. Woungang, "A secure data deduplication framework for cloud environments," in *Privacy, Security and Trust (PST), 2012 Tenth Annual International Conference on*, July 2012, pp. 81–87.

[29] J. Xu, E.-C. Chang, and J. Zhou, "Weak leakage-resilient client-side deduplication of encrypted data in cloud storage," in *Proc. of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ser. ASIA CCS '13, 2013, pp. 195–206.

[30] J. Liu, N. Asokan, and B. Pinkas, "Secure deduplication of encrypted data without additional independent servers," in *Proc. of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15, 2015, pp. 874–885.

[31] C. P. Wright, M. C. Martino, and E. Zadok, "NCryptfs: A secure and convenient cryptographic file system," in *In Proceedings of the Annual USENIX Technical Conference*, 2003, pp. 197–210.

[32] V. Kher and Y. Kim, "Securing distributed storage: Challenges, techniques, and systems," in *Proc. of the 2005 ACM Workshop on Storage Security and Survivability*, ser. StorageSS '05, 2005, pp. 9–25.

[33] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao, "OceanStore: An architecture for global-scale persistent storage," *SIGPLAN Not.*, vol. 35, no. 11, pp. 190–201, Nov. 2000.

[34] M. Selimi and F. Freitag, "Tahoe-LAFS distributed storage service in community network clouds," in *Big Data and Cloud Computing (BdCloud), 2014 IEEE Fourth International Conference on*, Dec 2014, pp. 17–24.

[35] B. C. Reed, M. A. Smith, and D. Diklic, "Security considerations when designing a distributed file system using object storage devices," in *Security in Storage Workshop, 2002. Proceedings. First International IEEE*, 2002, pp. 24–34.