UP-MLE: Efficient and Practical Updatable Block-Level Message-Locked Encryption Scheme Based on Update Properties

Shaoqiang Wu[1], Chunfu Jia[1], and Ding Wang[1]

Nankai University, Tianjin 300350, China

Abstract. Deduplication is widely used to improve space efficiency in cloud storage. The Updatable block-level Message-Locked Encryption (UMLE) has been proposed to achieve efficient block-level updates for a deduplication system. However, the update design of the current UMLE instantiation adopts a static structure, which does not fit in with real update scenarios. This paper analyzes the File System and Storage Lab (FSL) Homes datasets that are widely used in deduplication research and reveals two interesting properties: i) Updated blocks are more likely to be updated again; ii) Updated blocks are always clustered in files. Based on these properties, we propose and implement an efficient and practical UMLE scheme. Experiments on real-world datasets show that our update algorithm is 24.85% more efficient than its foremost counterpart, increasing the space overhead by $\leq 0.39\%$.

Keywords: Deduplication · Message-locked encryption · Updatable.

1 Introduction

Deduplication is an important technology that cloud providers adopt to enhance their space efficiency by removing redundant duplicates. For data privacy, users incline to encrypt their data before outsourcing. However, conventional encryption algorithms convert identical plaintexts into different ciphertexts, making it difficult for clouds to find duplicates for encrypted data. *Convergent Encryption* (CE) [5], where the hash of a message acts as the encryption key, ensures identical ciphertexts if the messages are identical. Therefore, CE is regarded as an appropriate method to implement encrypted deduplication. Bellare *et al.* [3] proved the security of CE and proposed a new cryptographic primitive named *message-locked encryption* (MLE) for secure deduplication.

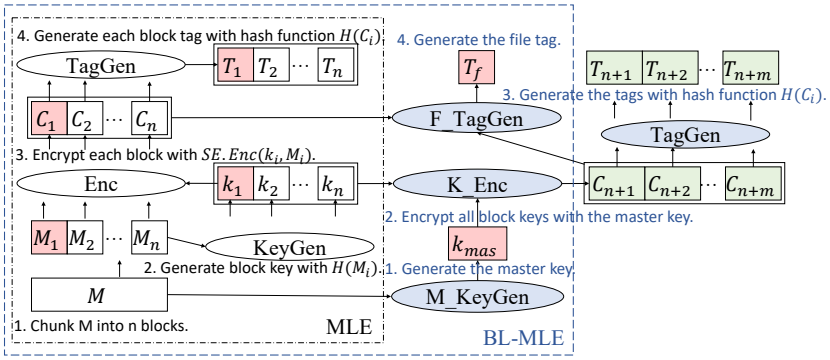
Generally, there are two granularities for data deduplication [15]: File-level deduplication and block-level deduplication. File-level deduplication [23] directly treats files as the basic units of deduplication. It is straightforward but cannot deduplicate redundant blocks across different files. While in block-level deduplication [4,9,12], files are divided into blocks as the deduplication units. Block-level deduplication is more fine-grained and has a better deduplication effect.

However, block-level deduplication generates a block-level MLE key[1] for each outsourced block, resulting in an enormous number of block keys for users to

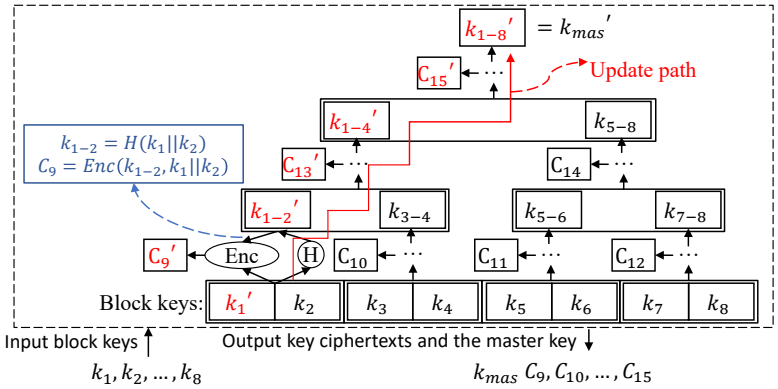
[1] We refer to the block-level MLE key as the block key for readability.

store and manage. As shown in the blue dashed box in Fig. 1(a), Chen *et al.* [4] proposed a *block-level MLE* (BL-MLE), which encrypts block keys with a master key, derived from the whole data. Then the encrypted block keys are stored in the cloud. In this encryption mechanism, the storage and management of numerous block keys are transparent to users, and they only need to hold one master key to access outsourced data.

However, BL-MLE focuses on static files. To support efficient file updates, Zhao *et al.* [22] proposed the "*(Efficiently) Updatable (Block-Level) MLE*" (UMLE) based on BL-MLE. UMLE improves the master key generation algorithm and the block key encryption algorithm (M_KeyGen and K_Enc represented by the blue shaded ellipse in Fig. 1(a)) and introduces update algorithms. As shown in Fig. 1(b), UMLE iteratively executes MLEs for each node along the tree trunk



(a) An overview of the updatable block-level message-locked encryption.



(b) The master key generation and block key encryption of UMLE19 [22]. Block keys are located in the leaf nodes ($k_{i,i\in[1,8]}$). UMLE19 recursively hashes ($H(\cdot)$) child nodes to derive the master key (k_{mas}) and uses the parent node to encrypt ($Enc(\cdot)$) the child nodes in each recursive layer. Assuming the first file block is modified, UMLE19 only needs three hashes to update the master key (k'_{1-2}, k'_{1-4}, k'_{1-8}) and three re-encryptions to update the key ciphertexts (C'_9, C'_{13}, C'_{15}).

Fig. 1: Revisiting the updatable block-level MLE (UMLE) [22].

and takes the root node as the master key. When a file block is modified, it needs to decrypt nodes along the path from the root to the to-be-modified leaf and re-execute iterative MLEs along the reverse path. For updating a block, its update cost is the logarithmic level of $|B| \log_{|B|}(|M|)$ (where $|B|$ and $|M|$ is the size of the block and the file, respectively).

We find that the UMLE scheme in [22] (UMLE19[2]) can be improved to realize higher update efficiency. This improvement is motivated by our analysis of the block-level update property of the real-world datasets. Firstly, we explore the re-modification fraction, which means the percentage of modified blocks that are modified again. Our analysis result shows that the re-modification fraction is up to 80.82% (while the average modification fraction is just 8.60%). This indicates that modified blocks are more likely to be modified. Therefore, we say that the block-level update has a temporal locality. Secondly, we find that the percentage of modified blocks does not increase linearly with the increase in the chunking size, indicating that the modified blocks are clustered. That is, the block-level update has spatial locality. Moreover, we find that many blocks remain unmodified for a long time and call them inactive blocks.

Based on the above update properties, we present an improved UMLE scheme (UP-MLE), in which updates of frequently updated active blocks go through a faster update path. Specifically, we set up a dynamic tree in which active blocks climb up along the tree path in each update, and even an enhanced dynamic tree in which the updated blocks are raised to the root node height in one step.

In addition, we discover the efficiency problem of updating the file tag of UMLE19 [22], which generates the file tag with $\mathsf{IH.Hash}$ of the Incremental Hash (IH). When updating a single block of a 1 GB file, its update algorithm $\mathsf{IH.Update}$ takes about ten milliseconds [22] for updating the file tag, which seems acceptable. However, the time cost of $\mathsf{IH.Update}$ is proportional to the number of updated blocks. Given our investigation into the real-world datasets, which show that the number of updated blocks at a time can be in thousands, UMLE19 [22] will take up to several minutes to update a file tag. The time cost is unacceptable when considering one server for the update calculation. We adopt a Merkle tree hash of the file ciphertext as the file tag to solve the problem. There are two advantages: 1) Block tags are ready-by as the leaf nodes of the Merkle tree, so that saves much computation overhead; 2) The update cost of the Merkle tree hash is a constant much smaller than that of $\mathsf{IH.Update}$.

We make three main **contributions**:

- We find some block-level update properties of the FSL datasets, including: 1) Most modified blocks will be modified again next time; 2) Modified blocks are always clustered in files; 3) More than half of the file blocks always remain unmodified. These properties may be of independent interest.
- We propose an efficiently updatable block-level MLE scheme (UP-MLE) based on the above update properties. It improves the update algorithms by introducing a dynamic tree. In terms of file tags, we adopt a Merkle

[2] In this paper, we term the UMLE scheme proposed in [22] as UMLE19.

tree to generate file tags supporting efficient updates. Besides, we provide security proofs for our scheme in the random oracle model.
– We implement the prototype of UP-MLE and experiment on the real-world datasets providing comprehensive performance comparisons with UMLE19 [22]. The experimental results show that our update algorithms are more efficient than that of UMLE19 [22].

2 Preliminary

Notations. For $i, x, y \in \mathbb{N}$, $[x, y]$ denotes the set of all i, $x \leqslant i \leqslant y$. For $m, n \in \mathbb{N}$, $i \in [1, n]$, and the message $A = A_1 \| A_2 \| \cdots \| A_n$, A_i denotes the *i-th* block of A, $\{A_i\}$ denotes the set of all A_i, $\{A_i\}_{n'-n}$ denotes the set of A_i, $i \in [n+1, n']$, and $A_i \| A_{i+1}$ denotes the concatenation of A_i and $A_i + 1$. If S is a finite set, $|S|$ denotes the number of elements contained in S, and $e \leftarrow \$S$ means to randomly select an element from S and assign it to e. For $l \in \mathbb{N}$, by $y \leftarrow Alg(x_1, x_2, \cdots, x_l)$, we mean to run the algorithm Alg on inputs $x_1, x_2, \cdots, x_l$, with the output of y. λ is the security parameter.

Locality. Locality describes the tendency of repeated access to local data over a short period. It mainly includes temporal and spatial locality, which widely exist in the data access patterns [14,16,19]. The temporal locality refers to repeated access to specific data for a relatively short duration. Spatial locality refers to the access of data within relatively fixed locations. Studying a target system's temporal locality and spatial locality can provide guidelines for appropriate data management solutions. For example, iDudep [16] designed an efficient inline data deduplication scheme, which took advantage of the temporal and spatial localities of duplicated data. RapidCDC [14] dramatically reduced the chunking time by uniquely leveraging duplication locality and access history. We discover that block updates in deduplication systems are with locality by analyzing real-world datasets. Based on the update locality, we improve the UMLE scheme.

Merkle Tree Hash. A Merkle tree can generate a compact hash (the root hash) for a large file. Each leaf node of a Merkle tree is a cryptographic hash of a file block, and each non-leaf node is a cryptographic hash of its children. The calculation of a Merkle tree starts from the leaf nodes to the root node, and the calculation process is irreversible. The Merkle tree always generates an identical hash value for the data with the same contents. It can be used to ensure the integrity and correctness of cloud outsourced data [13] and provide efficient authentication [7]. We generate file tags based on the Merkle tree instead of the incremental hash of UMLE19 [22] with ready-made block tags as leaf nodes. The correctness of the Merkle hash as the file tag is obvious, and its security of the tag consistency is proved in Section 4.5.

3 Analysis of the FSL Homes Datasets

The FSL Homes datasets [1] are widely utilized in the field of deduplication research [9,8,10,11,20]. In most cases, the FSL dataset participates in the deduplication research as experimental data; on the other hand, there are studies [17,18]

on its properties. For example, Tarasov *et al.* [18] studied the file-level update properties. They found that most files remain unchanged, and the probability of file re-modification is about 50%. Their analysis is at the file level, whereas our analysis is block-level and informative for block-level update schemes.

To the best of our knowledge, this is *the first analysis of block-level update properties for deduplication.*

Data Description. On a daily basis, FSL collects snapshots of the home directories of 33 people in the file-sharing system for 21 months. The dataset chunks files at seven chunking sizes (2, 4, 8, 16, 32, 128 KB) and records block information, including block metadata and md5-based hash values. The reasons why we choose the FSL dataset to explore block-level update properties are:

- It is a widely used dataset in deduplication. Our analysis based on the FSL dataset can provide more general guidance for other deduplication studies.
- It is collected for a sufficiently long period to meet the time-continuity requirement for exploring update properties.
- It is collected under various chunking sizes, which helps to explore update properties at different block sizes and facilitates spatial locality analysis.

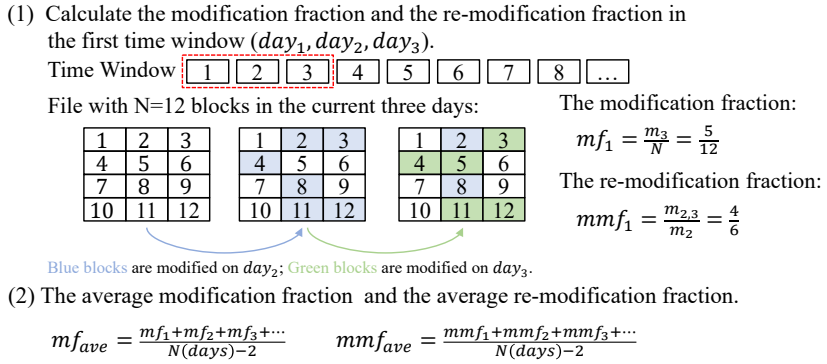


Fig. 2: Example of the calculation of the re-modification fraction (mmf) with the three-day time window.

3.1 Methodology and Analysis Results

Before the update property analysis, we need to filter the FSL datasets. There are three criteria for filtering: large file, the existence of updates, and time continuity. Firstly, since only the block update for large files significantly impact the performance of the UMLE schemes, we select large files with a size larger than 1 MB from the FSL datasets. Secondly, since not all large files collected by the datasets have updates, we further filter the large files with updates. Finally, some daily snapshots in the FSL datasets were not collected daily, hindering our analysis since discontinuous data cannot directly reflect updates in each time unit. Therefore, we remove these intermittent snapshots as well. Through the above three-step filter, we obtain the datasets shown in Table 1.

Table 1: Features of the FSL Homes datasets for our analysis.

Item	Number/Size
Users	18
Snapshots	1448
Files	1.5×10^6
Unique chunks	3.9×10^9
Total size	130.33 TB

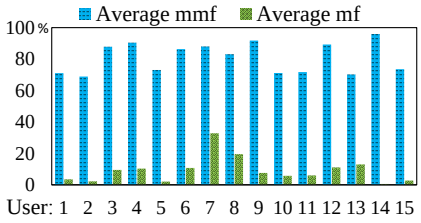


Fig. 3: Fractions of re-modification and modification of 15 users.

Second, we perform block-level deduplication for each file to ensure that our summarised update properties are valid for the deduplicated data. Block-level deduplication of the FSL datasets is straightforward. We can deduplicate files by comparing the block hashes and deleting duplicates with identical hashes. The deduplication ratio of the filtered dataset is 14.09%[3].

Property 1. Re-modification Fraction. The modification fraction (mf) defines the ratio of to-be-modified blocks to total blocks. And the re-modification fraction (mmf) defines the ratio of to-be-re-modified blocks to modified blocks; Namely, mmf establishes the mf in the set of modified blocks. Fig. 2 contains an example of calculating them. In detail, we set a three-day time window (day_i, day_{i+1}, and day_{i+2}), which is the smallest window for observing re-modifications and under which the window search is most efficient. We count the number of blocks modified[4] on day_{i+1} as m_{i+1} and count the number of blocks modified both on day_{i+1} and day_{i+2} as $m_{i+1,i+2}$. The re-modification fraction in the i-th time window (mmf_i) is calculated by

$$mmf_i = (m_{i+1,i+2}/m_{i+1}) \times 100\%. \tag{1}$$

Fig. 3 shows the re-modification fractions for different users, and their average re-modification fraction is 80.82%. The re-modification fraction is exceptionally high, being 9.40 times the average modification fraction (8.60%), which indicates that block modification is time-local. In other words, the modified blocks are more likely to be modified than other blocks.

Property 2. Spatial Distribution of Modified Blocks. We calculate the average modification fractions of datasets collected in various block sizes. We observe that with an increase in block size, the increase in modification fraction is slighter than the linear increase. This implies that modified blocks are clustered, indicating their distribution in file space is localized. With a similar method, iDudep [16] found that spatial locality exists in duplicated primary data.

Property 3. The Proportion of inactive blocks. We find that the modified blocks only account for a small part of the total blocks, between 4.50% and

[3] The deduplication ratio of the FSL Home datasets actually is 61% [17]; in contrast, that of our filtered dataset is low because we filter out both small files (< 1 MB) with big deduplication ratios and inactive files occupying an immense total size.

[4] If the hash of a block in day_{i+1} is different from the hash of the block in day_i, we say that the block is modified on day_{i+1}.

13.10%. Moreover, we find that the distinction between active and inactive blocks is clear. Some blocks are frequently modified, while others remain unchanged for a long time. The average proportion of file blocks never been modified is 53.41%.

Summary. By analyzing the block updates of large files continuously recorded for a period in the FSL datasets, we reveal three update properties:

- Temporal locality. Modified blocks are more likely to be modified again than others. The fraction of to-be-modified blocks in modified blocks is 9.4 times that of to-be-modified blocks in all blocks.
- Spatial locality. Modified blocks in a file are always clustered.
- Inactive blocks constitute a significant percentage (53.41%), i.e., most blocks remain unchanged over an extended period.

4 Updatable Block-Level MLE Scheme

We briefly outline the UMLE at a high level in terms of encryption, decryption, and update. Firstly, a large file is divided into blocks, and these blocks are encrypted with MLE, with numerous block keys (B_KeyGen and B_Enc). Then, block keys are encrypted with the master key derived from the whole file (M_KeyGen and K_Enc). The encrypted keys are uploaded to the server along with encrypted blocks to facilitate block key management. The server generates a block tag for each block ciphertext and a file tag for all block ciphertexts (B_TagGen and F_TagGen), to support block-level and file-level deduplication. Secondly, when the file owner with the master key accesses the outsourced file, the encrypted data is downloaded from the server. The owner first uses the master key to decrypt the key ciphertexts to obtain all MLE block keys (K_Dec), and then uses decrypted block keys to decrypt the file block ciphertexts (B_Dec). Thirdly, when the file owner wants to modify a block of the file, the related block ciphertext and key ciphertexts are downloaded. After decrypting, modifying, and re-encrypting, the owner uploads the updated ciphertexts. And the server re-generates the block tag of the modified block and a new file tag.

We focus on improving the update algorithm of UMLE. An update is essentially a transition from one state to another. In UMLE, each state contains a master key, block ciphertexts, block key ciphertexts, block tags, and a file tag. The updates of the block ciphertext, the block key, and the block tag are fixed in an MLE-based encryption scheme. Therefore, *we focus on improving the update performance of the master key, block key ciphertext, and file tag.* To this end, we introduce a new UMLE scheme, termed UP-MLE, based on the update properties embodied in the real-world datasets and the idea of hierarchical processing. Specifically, UP-MLE shortens the update paths of active blocks (frequently updated blocks) by dynamically adjusting the tree structure of the UMLE scheme [22] to enhance update efficiency.

4.1 Definition

We follow the UMLE definitions [22], revisited in Appendix A, except for the Update. Instead, we re-divide the update algorithm to separate those parts of the ir-

reducible algorithms and those that can be improved to facilitate scheme descriptions and performance analysis. Our Update algorithm is defined as $\{k_{mas}{}', C', T'\} \leftarrow Update(i, M_i', k_{mas}, C)$. It contains five sub-algorithms:

- $k_i \leftarrow K\_Return(k_{mas}, C_{key}, i)$. The block key decryption algorithm takes the master key k_{mas}, key ciphertexts C_{key} and a to-be-updated block index i as inputs and returns the block key k_i of the to-be-updated file block.
- $\{k_i', C_i'\} \leftarrow B\_Update(k_i, C_i, M_i')$. The block key and block ciphertext update algorithm takes the block key k_i, the block ciphertext C_i and a new file block M_i' as inputs and returns a new block key k_i' and a new block ciphertext C_i'.
- $\{k_{mas}{}', C_{key}{}'\} \leftarrow K\_Update(k_{mas}, k_i', \{k_i\}, i)$. The master key and key ciphertext update algorithm takes the old master key k_{mas}, the new block key k_i', block keys $\{k_i\}$, and the index i as inputs and returns a new master key $k_{mas}{}'$ and new key ciphertexts $C_{key}{}'$.
- $T_i' \leftarrow B\_UpdateTag(C_i')$. The block tag update algorithm returns a new block tag T_i', on input a new block ciphertext C_i'.
- $T_f{}' \leftarrow F\_UpdateTag(C')$. The file tag update algorithm takes new ciphertexts C' as inputs and returns a new file tag $T_f{}'$.

4.2 Partial Constructions of UP-MLE

Based on the definitions of UMLE and our Update definitions, we describes the details of partial constructions for our UP-MLE scheme directly:

- Setup: On input the security parameter λ, it returns $P = \{SE(Enc, Dec), H\}$, where SE is a symmetric encryption scheme, and H is a collision-resistant hash function, $H : \{0,1\}^{k(\lambda)} \leftarrow \{0,1\}^*$, where $k(\lambda)$ is the key length of the SE.
- KeyGen: On input $M = \{M_i\}_n$, it runs as follows and returns $K = \{\{k_i\}_n, k_{mas}\}$.
  - B_KeyGen: For each $i \in [1, n]$, $k_i = H(M_i)$;
  * M_KeyGen: Generate k_{mas} with $\{k_i\}_n$.
- Enc: On inputs K and M, it runs as follows and returns $C = \{\{C_i\}_n, C_{key}\}$.
  - B_Enc: For each $i \in [1, n]$, $C_i = SE.Enc(k_i, M_i)$;
  * K_Enc: Encrypt $\{k_i\}_n$ with k_{mas} and return C_{key}.
- Dec: On inputs K and C, it runs as follows and returns $M = \{M_i\}_n$.
  * K_Dec: Decrypt C_{key} with k_{mas} and return $\{k_i\}_n$;
  - B_Dec: For each $i \in [1, n]$, $M_i = SE.Dec(k_i, C_i)$.
- TagGen: On input C, it runs as follows and returns $T = \{\{T_i\}_{n'}, T_f\}$.
  - B_TagGen: For each $i \in [1, n']$, $T_i = H(C_i)$, where $\{C_i\}$ $(i \in [n+1, n'])$ are blocks of the C_{key};
  * F_TagGen: Generate T_f with C.
- Update: On inputs i, M_i', k_{mas}, and C, it runs as follows and returns k'_{mas}, $C' = \{C_i', C'_{key}\}$, and $T' = \{T_i', T_f'\}$.
  * K_Return: Decrypt C_{key} with k_{mas} and return k_i;
  - B_Update: $M_i = SE.Dec(k_i, C_i)$, modify M_i to M_i', $k_i' = H(M_i')$, and $C_i' = SE.Enc(k_i', M_i')$;
  * K_Update: Generate k'_{mas} with $\{k_i\}_n$ and k_i' and encrypt $\{k_i\}_n$ and k_i' with k'_{mas} as C'_{key};

- B_UpdateTag: $T'_i = H(C'_i)$;
- $*$ F_UpdateTag: Generate T'_f with $\{C_i\}_n$ and C'_i.

Besides, some algorithm constructions marked with "$*$", such as M_KeyGen, K_Enc, K_Dec, F_TagGen, K_Return, K_Update, and F_UpdateTag, are at the core of our improvements, described in Section 4.3.

4.3 Designs of the Update Algorithms of UP-MLE

We find that updated blocks will be quickly updated again. In contrast, other blocks will never be updated for a long time. Based on the update property, we adopts a dynamic tree for the master key generation and update. *In the dynamic tree, the frequently updated block keys approach to the tree root in updates by replacing the parent node.* As shown in the left tree in Fig. 4, the updated block key k'_1 moves one step along the leaf-to-root path and replaces the original k_{1-2} at its parent node. The new master key k'_{mas} is generated according to the dynamically adjusted tree by iteratively hashing children nodes as the parent node (i.e., $H(k_1{}'||k_2||k_{3-4}) \rightarrow k_{1-4}{}', H(k_{1-4}{}'||k_{5-6}) \rightarrow k_{1-8}{}'$). In addition, the children nodes always are encrypted with their parent node. Since the closer the location to the root, the fewer re-encrypted blocks in the path during updates, the less expensive the updates are. In a future update, $k_1{}'$ intuitively will have a shorter update path than other block keys. Additionally, the block key k_2 in the same block with k_1 shortens the update path as well, considering the spatial locality. The dynamic tree *differentiates between handling frequently and infrequently updated blocks, providing a faster update approach for frequent ones.*

However, unlike the static tree structure, which sets the master key generation algorithm by default, the dynamic tree leads to the possibility that users generate different master keys for identical files with different tree structures. For this, we mark the current height of each key block and help users learn the current tree structure to generate the correct master key. We call the space *the height space*, in which we store the key block heights.

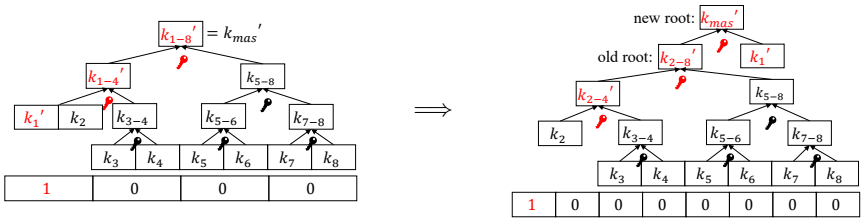


Fig. 4: The generation and update of the master key and block key ciphertexts of our UP-MLE scheme. For illustration, we assume an eight-block file, of which the first block will be updated. The key icons represent encryption and are in the middle of the keys in the same block. Child nodes are encrypted block-wise with their parent nodes. We assume that each key block contains two keys for the sake of brevity. Red nodes indicate the value to be recalculated.

Considering that there is a clear boundary between active and inactive blocks, if a block is updated, it will be updated frequently and will reach the root node in a short duration. In the final UP-MLE scheme, we *elevate the updated block key outside the tree directly in one update*, and its concatenation with the root hash is encrypted with MLE, which is equivalent to adding a tree layer. The generated MLE key will be the new master key and the new root.

As shown in the right tree in Fig. 4, the block key k_1 is removed from the leaf node, and other block keys are iteratively hashed to generate the new root key ${k_{2-8}}'$ in the old tree. Then, its concatenation $({k_{2-8}}'\|{k_1}')$ with the updated block key ${k_1}'$ is encrypted with MLE, and $H({k_{2-8}}'\|{k_1}')$ is used as the new master key. In a future update of the first block, UP-MLE can decrypt and re-encrypt k_1' directly and efficiently. It is more efficient for UP-MLE to elevate the block directly to the old root height than to approach the root layer by layer. Besides, by moving updated blocks outside the tree, all nodes in the old tree will be stabilized, thereby avoiding recalculation of nodes in the old tree in future updates. Moreover, we allocate 1 bit height space for each key block ciphertext to mark whether the key block is raised outside the old tree.

New Generation and Update Algorithms of the File Tag. We generate a file tag with the Merkle tree with block tags as leaf nodes. As shown in equation (2), MT.Hash computes the hash of the concatenation of ready-made block tags as file tags. The advantage of the Merkle tree hash as a file tag is that when some blocks of the file are modified, we do not need to recompute the hash of the whole ciphertext, just that of block tags as MT.Update in equation (3). For good measure, the update cost of the file tag is independent of the number of updated blocks and is constant for files with a constant block number. The file tag generation and update functions are defined by

$$MT.Hash\left(C_1,\cdots,C_m\right)=H(H(C_1)\|\cdots\|H(C_m))=H\left(T_1\|\cdots\|T_m\right), \quad (2)$$

$$MT.Update\left({C_1}',\cdots,C_m\right)=H(H({C_1}')\|\cdots\|H(C_m))=H\left({T_1}'\|\cdots\|T_m\right). \quad (3)$$

When any blocks are updated, the server only recalculates the hash of the concatenation of block tags as a new file tag with MT.Update.

4.4 Correctness and Efficiency

The correctness of the file tag is straightforwardly inherited from the correctness of the Merkle tree hash. The correctness of UP-MLE requires the correctness of symmetric encryption, which is straightforward, and the consistency of master keys, which is guaranteed by consistency of the tree structure. The correspondence of tree structure can be proved by the one-to-one correspondence of tree structure and leaf node heights, recorded in the height space.

With regard to efficiency, the dynamic tree structure may cause efficiency concerns, such as whether adjusting the tree structure will introduce an additional burden. However, it does not raise another cost because the impact of updates on the dynamic tree is limited to the leaf-to-root paths of the updated

blocks. In UMLE19 [22], the blocks in the update path would all need to be re-encrypted, which gives us an upper bound. In addition, we note that the update cost of UP-MLE increases instead when the block keys moved outside the tree are more than the keys in the original update path. We effectively solve this problem by organizing keys outside the tree in a tree structure as well.

UP-MLE logs the current height for each key block to help users generate a consistent master key. Thus, the server needs to send these additional data to users involved in the deduplication of the same file. Practically speaking, the delivery of auxiliary data does not increase the communication rounds of the deduplication protocol because these data can be returned to the user together with the duplicate detection results. If the protocol contains ownership authentication, it can be returned together with the authentication result.

4.5 Security Definitions and Proofs

Privacy. Any MLE schemes can not achieve semantic security, nor can the MLE-based schemes [3], such as BL-MLE and UMLE. Indeed, they ask for semantic security when messages are unpredictable, with high min-entropy. Bellare *et al.* [3] formalized PRV$-CDA notion where the encryption of an unpredictable message must be indistinguishable from a random string of the same length. Based on PRV$-CDA, Zhao *et al.* [22] defined PRV$-CDA-B* for their updatable MLE (UMLE) scheme. Since our UMLE scheme introduces additional items (i.e., heights) compared with [22], we follow its spirit and define a new privacy model, denoted by PRV$-CDA-B**. We say that our UMLE scheme is secure under chosen distribution attacks if no probabilistic polynomial-time non-adaptive adversary $\mathcal{A}$ has a non-negligible advantage in the following PRV$-CDA-B** game:

- **Setup**: The adversary $\mathcal{A}$ sends the description of an unpredictable message source $\mathcal{M}$ to the Challenger. The Challenger generates the system parameter P and sends it to $\mathcal{A}$.
- **Challenge**: The Challenger selects randomly $b \leftarrow \$\{0,1\}$. If $b = 0$, it samples $M^{\{0\}}$ from $\mathcal{M}$. If $b = 1$, the Challenger chooses uniformly at random from $\{0,1\}^{|M^{\{0\}}|}$ as $M^{\{1\}}$. Then, set $M = M^{\{b\}}$ and suppose $n(\lambda)$ (n simply) is the block numbers of M. For each $i \in [1, n]$, the Challenger runs as follows for the file block encryption:
  - Compute $\hat{k_i} \leftarrow B\_KeyGen(M_i)$ as the block key of the *i-th* block of the message M.
  - Compute $\hat{C_i} \leftarrow B\_Enc(k_i, M_i)$ as the block ciphertext of M_i.

  Also run as follows for the key block encryption:
  - Compute $\hat{k_{mas}} \leftarrow M\_KeyGen(\{\hat{k_i}\}_n)$ as the master key of M and compute $\{\hat{C_{n+i}}\}_{n'-n} \leftarrow K\_Enc(\{\hat{k_i}\}_n)$ as the ciphertext of all block keys.

  And then run as follows for the tag generation:
  - Compute $\hat{T_i} \leftarrow B\_TagGen(C_i)$ as the block tag of the *i-th* ciphertext, for each $i \in [1, n']$.
  - Compute $\hat{T_f} \leftarrow F\_TagGen(\{\hat{C_i}\}_{n'})$ as the file tag.

  Then, the Challenger randomly picks $S \leftarrow \$\{\{\{0,1\}^{\lceil \log(n(\lambda)) \rceil}\}_{\frac{n(\lambda)k(\lambda)}{B}}\}$.

- Compute $\{\{C_i\},\{T_i\},T_f,k_{mas}\}\leftarrow Update(\{\hat{C}_i\},\{\hat{T}_i\},\hat{T}_f,\hat{k_{mas}},S)$ as the updated data.

Denote $\{C_i\}$ as C, $\{T_i\}$ and T_f as T, and the additional items as S. Finally, Challenger sends $\{C,T,S\}$ to $\mathcal{A}$.

– **Output**: After receiving $\{C,T,S\}$, $\mathcal{A}$ guesses b' on b and outputs b'. If $b'=b$, the adversary $\mathcal{A}$ wins. Otherwise, the Challenger wins.

The above n' is the number of ciphertext blocks, and it can be calculated by

$$n' \leqslant n + \frac{((B/k(\lambda))^{\lceil \log_{B/k(\lambda)} n \rceil} - 1) \times k(\lambda)}{B - k(\lambda)}, \tag{4}$$

where $k(\lambda)$ is the key length, and B is the block size.

We define the advantage of the adversary $\mathcal{A}$ as:

$$Adv^{\mathcal{A},\mathcal{M}}_{\mathrm{PRV\$-CDA-B^{**}}} = |Pr[b=b'] - \frac{1}{2}|. \tag{5}$$

Definition 1. *We say that our scheme is PRV-CDA-B^{**}-secure if for any unpredictable source $\mathcal{M}$ and any probabilistic polynomial-time non-adaptive adversary $\mathcal{A}$, $Adv^{\mathcal{A},\mathcal{M}}_{PRV\$-CDA-B^{**}}$ is negligible.*

Theorem 1. *Let hash function H be a random oracle and let $SE=\{Enc,Dec\}$ is a symmetric encryption scheme with key length $k(\lambda)$. Suppose SE is both key recovery secure (KR-secure) and one-time real-or-random secure (ROR-secure). If there exists adversaries $\mathcal{B}'$ and $\mathcal{D}'$, for any adversary $\mathcal{A}$, his advantage is:*

$$\begin{aligned} Adv^{\mathcal{A},\mathcal{M}}_{\mathrm{PRV\$-CDA-B^{**}}} &\leqslant \frac{n^2}{2^{B-3}} + 2qn' Adv^{\mathcal{B}'}_{KR}(\lambda) + Adv^{\mathcal{D}'}_{ROR}(\lambda) + \frac{qn}{2^{\mu}} \\ &\leqslant \mathcal{O}(qn) Adv^{\mathcal{B}'}_{KR}(\lambda) + Adv^{\mathcal{D}'}_{ROR}(\lambda) + \frac{n^2}{2^{B-3}} + \frac{qn}{2^{\mu}}, \end{aligned} \tag{6}$$

where $\mu(\lambda)$ is the min-entropy of unpredictable source $\mathcal{M}$, $n(\lambda)$ (abbreviated as n) is the message block number, n' is the ciphertext block number, $q(\lambda)$ is the number of queries to the random oracle H by $\mathcal{A}$, $\mathcal{B}'$ and $\mathcal{D}'$ is adversaries in KR-game[5] and ROR-game[6] [3].

Proof. We prove the **Theorem** 1 by following the privacy proof of [22]. The proof introduces five games with the hidden bit b transition from 0 to 1 and proves that each transition is indistinguishable.

– **Game 0:** It has the hidden bit being 0.
– **Game 1:** It is identical to Game 0 except that the Challenger records query history to the random oracle H. If a random oracle query is a history query,

[5] The adversary $\mathcal{B}'$ guesses the random key K used by the oracle SE.Enc after obtaining the output C of the oracle SE.Enc on his specified input M.

[6] In ROR-game, the adversary $\mathcal{D}'$ needs to distinguish between a ciphertext generated by the oracle SE.Enc and a random string.

the Challenger aborts. Otherwise, he does not abort, and Game 1 is identical to Game 0.
All blocks of the deduplicated file are not identical. Since each block of message and each block of block key are hashed at most twice in KeyGen, Enc, and Update, the total number of random oracle queries is capped at twice the ciphertext block number, which can be calculated by equations (4). We conservatively assume $B > 2k(\lambda)$, and then $n \leqslant n' \leqslant (3n-3)$. Thus, Pr[Challenger aborting in Game 1] $< 2\sum_{i=n}^{3n-3} i/2^B < 2(4n^2-7n+3)/2^B < n^2/2^{B-3}$. (We stress that $B/k(\lambda) \gg 2$ in practice.)

- **Game 2:** The adversary makes a history query, and the Challenger aborts. In addition, the adversary $\mathcal{A}$ breaks the KR-security with the negligible advantage of $Adv_{KR}^{\mathcal{A}}(\lambda)$.
  A block hash is as the key of the block in encryption. Suppose an adversary $\mathcal{B}$ of KR-game who makes q history queries with non-negligible probability. So we can build an adversary $\mathcal{B}'$ who can break the KR-security. In each block encryption, $\mathcal{B}'$ guesses the key (hash query). Thus, Pr[History hash query in Game 2] $\leqslant 2qn' Adv_{KR}^{\mathcal{B}'}(\lambda)$.
- **Game 3:** It replaces the ciphertexts of message M_i by ciphertexts of random messages with the same length. The adversary $\mathcal{A}$ breaks the KR-security with the negligible advantage of $Adv_{ROR}^{\mathcal{A}}(\lambda)$.
  Suppose an adversary $\mathcal{D}$ of ROR-game who can differentiate Game 3 from Game 2. So we can build an adversary $\mathcal{D}'$ who can break the ROR-security. $\mathcal{D}'$ queries the encryption oracle of the ROR-game and outputs the $\mathcal{D}$'s outputs. Thus, Pr[History hash query with differentiating random M in Game 3] $\leqslant$ $Adv_{ROR}^{\mathcal{D}'}(\lambda)$.
- **Game 4:** The adversary queries $H(M_i)$ for *i-th* encryption, and the Challenger aborts. In Game 4, the hidden bit is 1 so that all returned ciphertexts are the ciphertexts of random messages. Each query is equivalent to randomly guessing M_i from a unpredictable message source $\mathcal{M}$. Thus, Pr[History hash query of completely random M in Game 4] $\leqslant qn/2^{\mu}$.

Tag Consistency. Our block tag generation algorithm is the same as MLE and has the same strong tag consistency (STC). As for the file tag, $T_f = MT.Hash\,(C)$. It is known that *deterministic schemes are STC-secure when the tags are collision-resistant hashes of the ciphertext* [3], meaning an adversary cannot erase a user's file by creating (M, C). In addition, the collision resistance of Merkle trees [6] shows that MT.Hash is collision-resistant when $H(\cdot)$ is a collision-resistant hash. Thus, we state the **Theorem** 2. Its proof is obvious.

Theorem 2. *Suppose $SE = \{Enc, Dec\}$ is a symmetric encryption scheme and H is a collision-resistant hash, our schemes are STC-secure.*

Context Hiding. Context hiding ensures that privacy is not degraded during the update, defined by [22]. Zhao *et al.* proved that context hiding of a deterministic UMLE scheme only requires that the updated ciphertexts are indistinguishable from fresh ciphertexts. Our $SE = \{Enc, Dec\}$ is a deterministic symmetric encryption scheme, and H is a deterministic hash function. Therefore,

the indistinguishability of ciphertexts, whether obtained by initial encryption or updated encryption, is not difficult to derive from the ROR-security. Thus, our scheme is context hiding.

Other Security Analysis. As proved in **Privacy**, with any height space as the returned parameter S by the Challenger in PRV-CDA-B**-game, our scheme are PRV-CDA-B**-secure. Thus, height space does not affect the privacy.

5 Evaluation and Comparison

We implement UP-MLE with the C Programming Language and adopt the SHA256 as the cryptographic hash function and AES as the symmetric encryption function. The basic cryptographic functions are implemented based on the OpenSSL Library (Version 1.1.1h). We test the update performance on the real-world datasets, including the FSL datasets of user000 [1] and the Kernels datasets, which contains unpacked Linux kernel sources from Linux v5.0 to v5.2.8 [2]. All experiment results are the averages of 10 independent executions.

5.1 Update Performance on the Real-World Datasets

We collect the indexes of daily update blocks for the user000's files in the FSL datasets at block size of 4 KB. We use update block indexes as the input of the update algorithms to test the time cost required to complete all block updates. In addition, we treat each of the 100 Kernel sources as a file, and consider the changes between successive versions as updates, and then perform the same tests as the FSL dataset. The execution time of the update algorithms is shown in Table 2. We can see that our scheme has better performance than UMLE19. In the FSL test, UP-MLE's update cost is only 83.40% of the time cost of UMLE19, and in the Kernels test, it is only 66.89%.

Table 2: Execution time (ms) of the *Update* sub-algorithms.

		UMLE19[22]	UP-MLE			UMLE19[22]	UP-MLE
FSL	K_Return	6.84	4.45	Kernels	K_Return	69.18	28.51
	K_Update	20.40	18.27		K_Update	201.49	152.55
	Sum	27.24	22.72		Sum	270.67	181.06
	(%)	100.00	83.40		(%)	100.00	66.89

5.2 Performance of the Generation and Update of the File Tag

We can take the runtime of SHA256 as a reference to measure the performance of file tag algorithms. As shown in Table 3, the time cost of the IH.Hash is about 50 times that of the SHA256. However, the time cost of our file tag generation algorithm (MT.Hash) is 0.18%-0.24% as that of the SHA256. From Table 4, we can see that when updating a block, the time cost of the file tag update algorithm (IH.Update) of UMLE19 [22] is 20.09%-39.29% as that of the SHA256. The time cost of our file tag update algorithm (MT.Update) is 0.04%-0.68% as that of the SHA256. Therefore, our file tag generation and update algorithms are

Table 3: F_TagGen execution time (ms) for various file sizes at 16 KB block size.

	File size(MB)	4	8	16	32	64
UMLE19[22]	IH.Hash	568.3	1138.5	2271.3	4545.2	9074.8
	SHA256†	11.2	22.4	45	89.9	179.6
	%‡	5074	5083	5047	5056	5053
Ours	MT.Hash	0.07	0.13	0.16	0.33	0.63
	SHA256	28.61	59.44	86.90	147.98	262.70
	%‡	0.23	0.21	0.18	0.22	0.24

†Execution time of SHA256 is from the supplementary material of [22].
‡Percentage of execution time compared to SHA256.

Table 4: F_TagUpdate execution time (ms) for 8 MB files at different block sizes.

	Block size(KB)	4	8	16	32	64
UMLE19[22]	IH.Update	8.8	8.8	4.5	4.6	4.7
	%†	39.29	39.29	20.09	20.54	20.98
Ours	MT.Update	0.40	0.26	0.12	0.06	0.02
	%‡	0.68	0.43	0.20	0.10	0.04

†Percentage compared to UMLE's SHA256, which costs 22.4 ms.
‡Percentage compared to our SHA256, which costs 59.44 ms.

significantly better than UMLE19's, and the time cost of our algorithms is reduced by several orders of magnitude. Additionally, the incremental hash update algorithm used by UMLE19 is positively correlated with the number of updated blocks. In contrast, our Merkle tree-based update algorithm is independent of the number of updated blocks. We conclude that the more the number of update blocks, the greater the performance advantage of our file tag update algorithm.

5.3 Storage Efficiency

Table 5: Size of ciphertext expansion at different block sizes, for files of 1 GB.

Block size(KB)	2	4	8	16	32	64	128
UMLE19[22]	16644.06	8256.50	4112.06	2052.00	1025.00	512.50	256.25
UP-MLE	16708.06	8288.50	4128.06	2060.00	1029.00	514.50	257.25

Considering a large file with a size of 1 GB, we calculate the ciphertext size when chunking with different block sizes in Table. 5. We can see that the ciphertext expansion size of our scheme does not exceed 0.39% compared with UMLE19 [22]. It shows that the auxiliary data (i.e., height space) do not lead to an unacceptable storage expansion.

6 Conclusion

We study the block-level update properties of the FSL Homes datasets such as the temporal locality and the spatial locality. Considering these properties, we design and implement an efficient and practical updatable block-level MLE scheme. Our scheme optimizes the update algorithm through hierarchical processing frequently and infrequently updated blocks in the dynamic tree. We test our update algorithm on two real-world datasets. The experimental results show that our scheme has better update performance than the existing scheme. In

the future work, we will quantify the update localities and analyze the update performance of our scheme on datasets with various locality strengths, in order to provide general guidance for other workloads to adopt our scheme.

Other secure deduplication studies can migrate our dynamic tree design to help them expand the function of efficient updates, such as the password-protected deduplication scheme [21]. It uses the password secret to encrypt block keys. Our designs can be directly plugged into the deduplication scheme by replacing the key encryption method to support efficient updates. Furthermore, our update properties summarized from the real datasets have reference value for future deduplication research.

References

1. Fsl traces and snapshots public archive. `http://tracer.filesystems.org/` (2014)
2. The linux kernel archives. `https://www.kernel.org` (2021)
3. Bellare, M., Keelveedhi, S., Ristenpart, T.: Message-locked encryption and secure deduplication. In: Proc. EUROCRYPT. pp. 296–312 (2013)
4. Chen, R., Mu, Y., Yang, G., Guo, F.: Bl-mle: block-level message-locked encryption for secure large file deduplication. IEEE Trans. Inf. Forensics Secur. **10**(12), 2643–2652 (2015)
5. Douceur, J.R., Adya, A., Bolosky, W.J., Simon, P., Theimer, M.: Reclaiming space from duplicate files in a serverless distributed file system. In: Proc. ICDCS. pp. 617–624 (2002)
6. Dowling, B., Günther, F., Herath, U., Stebila, D.: Secure logging schemes and certificate transparency. In: European Symposium on Research in Computer Security. pp. 140–158 (2016)
7. Li, H., Lu, R., Zhou, L., Yang, B., Shen, X.: An efficient Merkle-tree-based authentication scheme for smart grid. IEEE Systems Journal **8**(2), 655–663 (2013)
8. Li, J., Lee, P.P., Ren, Y., Zhang, X.: Metadedup: Deduplicating metadata in encrypted deduplication via indirection. In: Proc. MSST. pp. 269–281 (2019)
9. Li, J., Qin, C., Lee, P.P., Li, J.: Rekeying for encrypted deduplication storage. In: Proc. IEEE/IFIP DSN. pp. 618–629 (2016)
10. Li, M., Qin, C., Lee, P.P.: Cdstore: Toward reliable, secure, and cost-efficient cloud storage via convergent dispersal. In: Proc. USENIX ATC. pp. 111–124 (2015)
11. Lin, X., Douglis, F., Li, J., Li, X., Ricci, R., Smaldone, S., Wallace, G.: Metadata considered harmful... to deduplication. In: Proc. HotStorage (2015)
12. Liu, M., Yang, C., Jiang, Q., Chen, X., Ma, J., Ren, J.: Updatable block-level deduplication with dynamic ownership management on encrypted data. In: Proc. IEEE ICC. pp. 1–7 (2018)
13. Mao, J., Zhang, Y., Li, P., Li, T., Wu, Q., Liu, J.: A position-aware Merkle tree for dynamic cloud data integrity verification. Soft Computing **21**(8) (2017)
14. Ni, F., Jiang, S.: Rapidcdc: Leveraging duplicate locality to accelerate chunking in cdc-based deduplication systems. In: Proc. Cloud Computing. pp. 220–232 (2019)
15. Shin, Y., Koo, D., Hur, J.: A survey of secure data deduplication schemes for cloud storage systems. ACM CSUR **49**(4), 1–38 (2017)
16. Srinivasan, K., Bisson, T., Goodson, G.R., Voruganti, K.: idedup: latency-aware, inline data deduplication for primary storage. In: Proc. USENIX FAST (2012)
17. Sun, Z., Kuenning, G., Mandal, S., Shilane, P., Tarasov, V., Xiao, N., et al.: A long-term user-centric analysis of deduplication patterns. In: Proc. MSST (2016)

18. Tarasov, V., Mudrankit, A., Buik, W., Shilane, P., et al.: Generating realistic datasets for deduplication analysis. In: Proc. USENIX ATC. pp. 261–272 (2012)
19. Weinberg, J., McCracken, M.O., Strohmaier, E., Snavely, A.: Quantifying locality in the memory access patterns of hpc applications. In: Proc. ACM/IEEE Conference on Supercomputing. pp. 50–50 (2005)
20. Yuan, H., Chen, X., Li, J., Jiang, T., Wang, J., Deng, R.: Secure cloud data deduplication with efficient re-encryption. IEEE Trans. Services Computing (2019)
21. Zhang, Y., Xu, C., Cheng, N., Shen, X.S.: Secure password-protected encryption key for deduplicated cloud storage systems. IEEE Trans. Depend. Secure Comput. (2021). https://doi.org/10.1109/TDSC.2021.3074146
22. Zhao, Y., Chow, S.S.: Updatable block-level message-locked encryption. IEEE Trans. Depend. Secure Comput. **18**(4), 1620–1631 (2021)
23. Zhou, Y., Feng, D., Xia, W., Fu, M., Huang, F., Zhang, Y., Li, C.: Secdep: A user-aware efficient fine-grained secure deduplication scheme with multi-level key management. In: Proc. MSST. pp. 1–14 (2015)

A Complete Definition of the UMLE

Zhao *et al.* [22] initiated the study of the updatable block-level message-locked encryption (UMLE) and defined it based on MLE [3] and BL-MLE [4]. We follow the UMLE definitions [22]:

- $P \leftarrow Setup(1^\lambda)$: It takes the security parameter λ as input and returns the system parameter P, which is public to all entities in the system.
- $K \leftarrow KeyGen(M)$: It takes a file M as input and returns a master key and all block keys of the file. It contains two sub-algorithms:
  - $k_{mas} \leftarrow M\_KeyGen(M)$: It takes M as input and returns a master key k_{mas}.
  - $k_i \leftarrow B\_KeyGen(M_i)$: It takes a file block M_i as input and returns a block key k_i.
- $C \leftarrow Enc(K, M)$: It takes a file, a master key, and block keys as inputs and returns ciphertexts. It contains two sub-algorithms:
  - $C_i \leftarrow B\_Enc(k_i, M_i)$: It takes a file block M_i and a block key k_i as inputs and returns a ciphertext C_i of the file block.
  - $C_{n+j} \leftarrow K\_Enc(k_{mas}, \{k_i\}_b)$: It takes a master key k_{mas} and all block key $\{k_i\}$ as inputs and returns ciphertexts $\{C_{n+j}\}_{n_k}$, where $j \in [1, n_k]$, n_k is the number of key blocks, and $C_{key} = \{C_{n+j}\}_{n_k}$.
- $M \leftarrow Dec(k_{mas}, C)$: It takes a master key and ciphertexts as inputs and returns the file in plaintext. It contains two sub-algorithms:
  - $\{k_i\} \leftarrow K\_Dec(k_{mas}, C_{key})$: It takes a master key k_{mas} and key ciphertexts C_{key} as inputs and returns block keys $\{k_i\}$.
  - $M_i \leftarrow B\_Dec(k_i, C_i)$: It takes a block key k_i and a ciphertext C_i as inputs and returns the file block M_i.
- $T \leftarrow TagGen(C)$: It takes ciphertexts as inputs and returns tags. It contains two sub-algorithms:
  - $T_i \leftarrow B\_TagGen(C_i)$: It takes a ciphertext C_i of a block as input and returns a block tag T_i.
  - $T_f \leftarrow F\_TagGen(C)$: It takes all ciphertexts C as inputs and returns a file tag T_f.

There are no Update algorithms, but we redefine them in Section 4.1.