Updatable Block-Level Message-Locked Encryption

Yongjun Zhao
Department of Information Engineering
The Chinese University of Hong Kong
Shatin, N.T., Hong Kong
zy113@ie.cuhk.edu.hk

Sherman S.M. Chow
Department of Information Engineering
The Chinese University of Hong Kong
Shatin, N.T., Hong Kong
sherman@ie.cuhk.edu.hk

ABSTRACT

Deduplication is a widely used technique for reducing storage space of cloud service providers. Yet, it is unclear how to support deduplication of encrypted data securely until the study of Bellare *et al.* on message-locked encryption (Eurocrypt 2013). Since then, there are many improvements such as strengthening its security, reducing client storage, *etc.*

While updating a (shared) file is common, there is little attention on how to efficiently update large encrypted files in a remote storage with deduplication. To modify even a single bit, existing solutions require the trivial and expensive way of downloading and decrypting the large ciphertext.

We initiate the study of updatable block-level message-locked encryption. We propose a provably secure construction that is efficiently updatable with $O(\log|F|)$ computational cost, where $|F|$ is the file size. It also supports proof-of-ownership, a nice feature which protects storage providers from being abused as a free content distribution network.

CCS Concepts

•**Security and privacy** → **Symmetric cryptography and hash functions;** *Key management; Management and querying of encrypted data;*

Keywords

incremental cryptography; message-locked encryption; Merkle-hash tree; random oracle model

1. INTRODUCTION

Deduplication, which eliminates redundant copies in user-provided data, has been widely used to improve storage utilization and reduce communication cost. The saving is significant for cloud storage service provider which stores data from many customers. Typically, storage systems require seeing the client data in plaintext to perform deduplication, which incurs obvious privacy threat. On the other hand, if every user encrypts their own file using a user-specific key, the effectiveness of deduplication is drastically reduced since the probabilistic nature of a secure encryption intuitively produces random-looking ciphertexts. As far as we know, the first attempt to resolve the seemingly contradicting requirements (of achieving file confidentiality and file deduplication simultaneously) was *convergent encryption* [22]. It works by mapping each file into a key deterministically by hashing the file and then encrypting each file using such file-derived key. This ingenious solution guarantees that identical files are encrypted to identical ciphertexts, enabling efficient deduplication. Convergent encryption (and its variants) has been used in numerous applications [2,3,6,20,38].

However, the precise security guarantee provided by convergent encryption is unknown until the recent formulation of message-locked encryption (MLE) by Bellare *et al.* [13]. MLE is a symmetric encryption scheme in which the message is the lock, *i.e.*, it uses the message to derive the key for encryption and decryption. Their study provided several definitions that capture different aspects of MLE security and analyzed some prior schemes under their framework.

While earlier MLE [1,12,13] focus on *file-level deduplication*, a study of practical deduplication [32] has shown that *block-level deduplication* can be more space-efficient. Naturally, recent research extended file-level deduplication on encrypted data to the block-level setting [19,29,34]. The most straightforward solution is to apply file-level MLE on each block independently. But this approach introduces a large number (linear in the file size $|F|$) of block keys. Some researchers [29,34] thus introduced an additional key management server to handle those block keys. As another approach, Chen *et al.* [19] formalized the notion of *block-level message-locked encryption* (BL-MLE) and proposed a relatively efficient construction using the techniques in compact proofs-of-retrievability [37]. Their solution only requires the user to remember a single master key, but involves computationally expensive pairing operations.

Unfortunately, all existing works focus on *static files*. None of them supports efficient file update (even with the help of a key management server). To modify a single bit, the file owner has to download the whole encrypted file, decrypt, update, re-encrypt, and then upload the new ciphertext to the cloud. The computation and communication costs of all these operations are linear in the file size, *i.e.*, $O(|F|)$, which are certainly too expensive for large files. Naturally, we thus ask the following practically motivated question:

"Can the update cost of MLE be sublinear?"

If the file is unencrypted, or if it is block-wise encrypted using standard randomized encryption scheme, efficient up-

ASIA CCS '17, April 4–6, 2017, Abu Dhabi, United Arab Emirates.
© 2017 ACM. ISBN 978-1-4503-4944-4/17/04...$15.00.
DOI: http://dx.doi.org/10.1145/3052973.3053012

date is trivial because updating certain block is independent of the rest of the file. Yet, update becomes nontrivial in the MLE context. The difficulty arises because a typical MLE scheme derives the encryption key from the file deterministically. A single bit flip will lead to a totally different encryption key, and thus the whole file needs to be re-encrypted. One may try to remedy such inefficiency by independently applying MLE on each block. Unfortunately, this approach increases the number of keys for file F from $O(1)$ to $O(|F|/B)$ for block size B, which is also unsatisfactory. Bellare *et al.* proposed *interactive MLE* (iMLE) [12], which extends the original notion of MLE by introducing interaction. Their solution is the first secure deduplication system supporting incremental updates. However, their scheme only deduplicates at the file level. In a nutshell, updatable block-level MLE remains an interesting open question.

1.1 Incremental Cloud Cryptography

Bellare *et al.* initiated the research of incremental cryptography [10, 11, 14], which aims to update the result of a cryptographically processed document faster than redoing the whole computation from scratch. Early works focus on signing and hashing [10, 14], while the follow-ups studied encryption [11, 17, 33, 36]. Recently, we witness growing interest in cloud cryptography, such as searchable encryption, message-locked encryption, and proof of retrievability. Previous research studied dynamic searchable encryption [27, 28] such that searchable ciphertext can be updated without building it from scratch. This paper furthers this research line for cloud cryptography. Our paper initiates the study of incremental cryptography for secure (block-level) deduplication, which is of practical necessity due to the recent wide application of cloud-enabled technologies.

1.2 Our Contribution

This paper makes the following contributions:

- We initiate the study of efficiently updatable block-level message-locked encryption (UMLE)[1]. We formalize the definition and security notion for UMLE.

- We propose an efficient UMLE construction in the random oracle model. For a file of size $|F|$, the update cost of our UMLE construction is only $O(\log |F|)$ where the base of the logarithm is B/λ, B is the block size, and λ is the key size. Typically, B/λ can be a relatively large constant.

- Borrowing ideas from the database community, we propose an enhanced construction that additionally supports block insertion and deletion, which achieves an amortized complexity of $O(\log |F|)$.

1.3 Paper Organization

The rest of this paper is organized as follows. Section 2 introduces necessary notions and technical preliminaries for the rest of the paper. Section 3 defines the interfaces of UMLE and its security definitions. Our proposed construction is described in Section 4 with detailed security analysis in Section 5. Section 6 discusses how to extend our construction to support insertion and deletion. We survey related works in Section 7 and conclude our paper with future research direction in Section 8.

[1]We did not name it incremental MLE following incremental hash of Bellare *et al.* [10] to avoid clashing with the acronym iMLE used by Bellare *et al.* [12] for interactive MLE.

2. PRELIMINARY

2.1 Notations

We use ϵ to denote an empty string. For $i \in \mathbb{N}$ we let $[i] = \{1, \ldots, i\}$. If $\mathbf{x}$ is a vector then $|\mathbf{x}|$ denotes the dimension of $\mathbf{x}$, $\mathbf{x}[i]$ denotes the i-th component, and $\mathbf{x}[i, j] = \mathbf{x}[i] \ldots \mathbf{x}[j]$ for $1 \leq i \leq j \leq |\mathbf{x}|$. If S is a finite set, then $|S|$ denotes its size and $s \overset{\$}{\leftarrow} S$ denotes picking an element uniformly in S and assigning it to s. By $y \overset{\$}{\leftarrow} A(x_1, \ldots)$, we denote the operation of running algorithm A on inputs $x_1, \ldots$ with random coins. The guessing probability $\mathbf{GP}(X)$ and min-entropy $\mathbf{H}_\infty(X)$ of a random variable X are defined by $\max_x \Pr[X = x] = 2^{-\mathbf{H}_\infty(X)}$. The conditional guessing probability $\mathbf{GP}(X|Y)$ and conditional min-entropy $\mathbf{H}_\infty(X|Y)$ of a random variable X given a random variable Y are defined by $\sum_y \Pr[Y = y] \cdot \max_x \Pr[X = x|Y = y] = 2^{-\mathbf{H}_\infty(X|Y)}$. By $\Delta(X; Y)$ we mean the statistical distance between random variable X and Y.

2.2 Unpredictable Sources

A source is a polynomial-time algorithm $\mathcal{M}$ that on input 1^λ returns $(\mathbf{M}, Z)$ where $\mathbf{M}$ is a message vector in $\{0,1\}^*$ and $Z \in \{0,1\}^*$ denotes some auxiliary information. Let $n(\lambda)$ be the vector length. In our context, it is the number of blocks. For all $i \in [1, n(\lambda)]$, $\mathbf{M}[i]$ represents the i-th block of the message $\mathbf{M}$. In this paper, since we focus on block-level deduplication with fixed block size, we restrict our attention to sources that output message vector over $\{0,1\}^B$, where B is the block size. Therefore $\mathbf{M}[i] \in \{0,1\}^B$ for all i. Similar to previous works [8, 13], we require that $\mathbf{M}[i_1] \neq \mathbf{M}[i_2]$ for all distinct $i_1, i_2 \in [m(\lambda)]$ to bar against trivial adversary. We say that the source $\mathcal{M}$ is unpredictable if $\mathbf{GP}_\mathcal{M} = \max_i \{\mathbf{GP}(\mathbf{M}[i]|Z)$ is negligible.

2.3 Message-Locked Encryption

MLE consists of five algorithms [13]. The parameter generation algorithm Setup, given an input of a security parameter 1^λ, returns a public parameter P. We assume that P (implicitly includes 1^λ) is the default input for the rest of the algorithms, and thus will be ignored for notational simplicity. On input a message $\mathbf{M}$, the key-generation algorithm KeyGen returns a message-derived key $K \overset{\$}{\leftarrow} \mathsf{KeyGen}(\mathbf{M})$. On inputs $K, \mathbf{M}$, the encryption algorithm Enc returns a ciphertext $\mathbf{C} \overset{\$}{\leftarrow} \mathsf{Enc}(K, \mathbf{M})$. On inputs $K, \mathbf{C}$, the decryption algorithm returns $\mathsf{Dec}(K, \mathbf{C}) \in \{0,1\}^* \cup \{\perp\}$, where $\perp$ denotes invalid. On inputs $\mathbf{C}$, the tag generation algorithm returns a tag $\mathbf{T} \overset{\$}{\leftarrow} \mathsf{TagGen}(\mathbf{C})$. All algorithms except Dec can be probabilistic. If both KeyGen and Enc are deterministic, we say that such MLE is deterministic.

The message $\mathbf{M}$ comes from a message space defined by $\lambda \in \mathbb{N}$. Additionally, we require that the length of a ciphertext only depends on the length of the message apart from the security parameter. That is to say, for all $\lambda \in \mathbb{N}$, all $P \leftarrow \mathsf{Setup}(1^\lambda)$ and all $\mathbf{M} \in \{0,1\}^*$, the length of $\mathsf{Enc}(\mathsf{KeyGen}(\mathbf{M}), \mathbf{M})$ is some function $\mathsf{Len}(P, \lambda, |\mathbf{M}|)$.

2.4 Proof of Ownership

Proof-of-ownership (PoW) [25] is an interactive protocol between a prover (file owner) and a verifier (cloud storage server). By executing the protocol, the prover convinces the verifier that s/he is the owner of a file stored by the verifier. The motivation is to protect both storage servers and cloud users from various attacks. Consider a simple mechanism which just uses the hash of the file for "deduplication" and "ownership-proof". A malicious user can then "claim ownership" of a confidential file uploaded by another user to the cloud if she only obtained its short hash value somehow. Another attack is to publicize a short hash value of a huge file on the cloud (say a movie) so that everyone can easily claim its ownership and hence download it, effectively making the cloud as a cheap content distribution center.

Here, we briefly describe the core idea behind three existing PoW protocols [25] with different trade-offs between security and performance. All three require both the user and the server to build a Merkle hash tree [31] on a buffer derived from a pre-processed file. The server only keeps the root of the hash tree and challenges the client to present valid sibling paths for a subset of leaves of the tree. Therefore, the bandwidth cost would be a super-logarithmic number of sibling paths of the Merkle tree.

2.5 Deterministic Symmetric-Key Encryption

A deterministic symmetric-key encryption scheme consists of the following algorithms ($\mathsf{SKE.KeyGen}$, $\mathsf{SKE.Enc}$, $\mathsf{SKE.Dec}$), of which only $\mathsf{SKE.KeyGen}$ is randomized. We require it to provide both key recovery (KR) security and one-time real-or-random (ROR) security [9, 13, 35]. A blockcipher in counter mode with fixed IV satisfies these requirements.

The key recovery game is defined as follows: on input 1^λ, the challenger chooses a randomly chosen key K. The adversary can query at most once to an encryption oracle a message M. The challenger replies with $c = \mathsf{SKE.Enc}(K, M)$. The adversary wins the game if it can output $K' = K$. We say that a symmetric key encryption scheme is KR-secure if the advantage $\mathbf{Adv}^{\mathcal{A}}_{\mathsf{KR}}(\lambda)$ of any adversary in winning the above game is negligible.

The real-or-random game is defined as follows: on input 1^λ, the challenger chooses a random bit b. The adversary can make multiple queries to an encryption oracle. Each query is a message $M \in \{0,1\}^*$. If $b = 1$, the challenger chooses a random key K and returns $c = \mathsf{SKE.Enc}(K, M)$. If $b = 0$, the challenger chooses a random message $R\{0,1\}^{|M|}$ and returns $c = \mathsf{SKE.Enc}(K, R)$. Note that a random key is selected for each query. The adversary wins the game if it can output $b' = b$. We say that a symmetric key encryption scheme is ROR-secure if the advantage $\mathbf{Adv}^{\mathcal{A}}_{\mathsf{ROR}}(\lambda)$ of any adversary in winning the above game is negligible.

2.6 Hash Functions

In general, cryptographic hash functions are computationally efficient functions that take arbitrary-length strings and map them into fixed-length strings. A family of hash functions is a pair of polynomial time algorithms: key generation algorithm $\mathsf{HashGen}$ and hashing algorithm H. The key generation algorithm takes a security parameter 1^λ as input and outputs a key s. The hashing algorithm takes the key s and a message m, and returns a string $H_s(m)$.

For cryptographic use, we often require the hash functions to have *collision resistance*. A collision in a hash function H_s is a pair of distinct inputs m and m' such that $H_s(m) = H_s(m')$. Informally, it is infeasible for any computationally bounded adversary to find a collision in H_s where s is generated using the key generation algorithm. For the rest of the paper, we assume that s is part of the description of the hash function, which is chosen by some trusted setup algorithm. We will omit s when referring to hash functions.

Incremental hashing has been proposed [10, 14] to allow efficient update of a hash value. Notably, Bellare *et al.* [14] constructed collision-resistant *incremental* hash function in the random oracle model. Looking ahead, we will use their incremental hash in the file tag generation algorithm in our basic construction. For completeness, we present their construction ($\mathsf{Mu.HASH}$, $\mathsf{Mu.Update}$) below. Yet, we still need a special way to build the ciphertext and the file tag for efficient insertion and deletion in our updatable block-level MLE.

Let $\mathbb{G}$ be a multiplicative group such that discrete logarithm problem is hard, and let $\mathsf{H} : \{0,1\}^* \to \mathbb{G}$ be a hash function mapping an arbitrary bit string into a group element. Let $\langle i \rangle$ be the binary representation of integer index i. For an n-block input $\mathbf{M} = \mathbf{M}[1]||\mathbf{M}[2]|| \cdots ||\mathbf{M}[n]$, the incremental hash function of Bellare *et al.* [14], which is provably collision-resistant in the random oracle model is defined by

$$\mathsf{Mu.HASH}(\mathbf{M}[1]|| \cdots ||\mathbf{M}[n]) = \prod_{i=1}^{n} \mathsf{H}(\langle i \rangle || M[i]).$$

The product is taken in $\mathbb{G}$ over which we are working in.

The update algorithm takes as input a block index i, the original block $\mathbf{M}[i]$, the updated version $\mathbf{M}'[i]$, and the original hash value y for the whole message $\mathbf{M}$. It works in the straightforward way:

$$\mathsf{Mu.Update}(i, \mathbf{M}[i], \mathbf{M}'[i], y) = y \cdot \mathsf{H}(\langle i \rangle || M[i])^{-1} \cdot \mathsf{H}(\langle i \rangle || M'[i]),$$

where $(\cdot)^{-1}$ denote the inverse operation of the group $\mathbb{G}$.

2.7 The Random Oracle Model

Requiring hash functions to be collision-resistant is often not enough to allow security proof for higher cryptographic systems. Ideally, hash functions are expected to behave like random functions. Such strong assumption is captured by modeling them as a *random oracle* in the security proof. A random oracle [15] is a game procedure H that maintains a table $\mathsf{H}[\cdot, \cdot]$, initially everywhere $\perp$. Given a query (x, k) with $x \in \{0,1\}^*$ and $k \in \mathbb{N}$, it executes: If $\mathsf{H}[x, k] = \perp$ then $\mathsf{H}[x, k] \overset{\$}{\leftarrow} \{0,1\}^k$. It then returns $\mathsf{H}[x, k]$. If output length is clear, the second input is often omitted.

If a system is proven to be secure by replacing hash functions with random oracles, the system is said to be secure in the random oracle model. Such proof provides evidence that the construction does not have "inherent security flaws" and usually random oracle constructions are very efficient.

3. UPDATABLE BLOCK-LEVEL MLE

We define the notion of UMLE, for "Efficiently Updatable Block-Level Message-Locked Encryption". While block-level MLE exists [19] and our definitions extend those for file-level MLE [1, 13], there are key differences to be explained shortly.

3.1 Syntax

A UMLE scheme is specified by the following algorithms.

- Setup: takes a security parameter 1^λ and returns the system parameter P. We assume that P implicitly contains 1^λ, and it will be the default input for the rest of the algorithms.

- KeyGen: takes a file message $\mathbf{M} = \mathbf{M}[1]||\dots||\mathbf{M}[n]$ as input, returns block keys $k_1, \dots, k_n$ and a master key k_{mas} respectively via the following two sub-algorithms:
  - $\mathsf{B\text{-}KeyGen}$: takes $\mathbf{M}[i]$ as input, returns the block key k_i;
  - $\mathsf{M\text{-}KeyGen}$: takes $\mathbf{M}$ as input, returns the master key k_{mas}.
- Enc: takes a file $\mathbf{M}$ (and block keys $k_1, \dots, k_n$ which can be generated from $\mathsf{KeyGen}(\mathbf{M})$) as input, returns the ciphertext $\mathbf{C}$. It consists of the following two sub-algorithms:
  - $\mathsf{B\text{-}Enc}$: takes a block message $\mathbf{M}[i]$ and the corresponding block key k_i as input ($1 \le i \le n$), returns the block ciphertext $\mathbf{C}[i]$;
  - $\mathsf{BK\text{-}Enc}$: takes the block keys $k_1, \dots k_n$ as input only, outputs the encrypted block keys as $\mathbf{C}[n+1]||\dots||\mathbf{C}[n']$, where $n' \in O(n)$.

  After completion of the two sub-algorithms $\mathsf{B\text{-}Enc}$ and $\mathsf{BK\text{-}Enc}$, Enc returns encrypted file as $\mathbf{C}$ where $\mathbf{C} = \mathbf{C}[1]||\dots||\mathbf{C}[n]||\mathbf{C}[n+1]||\dots||\mathbf{C}[n']$.
- Dec: takes the encrypted file $\mathbf{C} = \mathbf{C}[1]||\dots||\mathbf{C}[n]|| \mathbf{C}[n+1]||\dots||\mathbf{C}[n']$ and the master key k_{mas} as input, returns $\mathbf{M} = \mathbf{M}[1]||\dots||\mathbf{M}[n]$. Again it consists of two sub-algorithms:
  - $\mathsf{BK\text{-}Dec}$: takes the master key k_{mas} and the encrypted block keys $\mathbf{C}[n+1]||\dots||\mathbf{C}[n']$ as input, outputs the set of block keys $k_1, \dots, k_n$;
  - $\mathsf{B\text{-}Dec}$: takes the block key k_i and the corresponding block ciphertext $\mathbf{C}[i]$ as input ($1 \le i \le n$), outputs the file block $\mathbf{M}[i]$.
- TagGen: takes an encrypted file $\mathbf{C}$ as input, returns block tags $T_1, \dots, T_{n'}$ and the file tag T_0 generated using the following two sub-algorithms respectively:
  - $\mathsf{B\text{-}TagGen}$: takes $\mathbf{C}[i]$ as input, returns the block tag T_i ($1 \le i \le n'$);
  - $\mathsf{M\text{-}TagGen}$: takes $\mathbf{C}$ as input, returns the file tag T_0.
- Update: this is an interactive protocol between the user (file owner) and the server. The user takes as inputs k_{mas}, the to-be-updated block index i, a plaintext block $\mathbf{M}[i]^*$; while the server takes $\mathbf{C}$ as input. At the end of the protocol, the user outputs an updated master key k^*_{mas} and the server outputs updated ciphertext $\mathbf{C}^*$. Using the notation of two-party computation, the syntax of Update is $(k^*_{\mathsf{mas}}, \mathbf{C}^*) \leftarrow \mathsf{Update}((k_{\mathsf{mas}}, i, \mathbf{M}[i]^*), \mathbf{C})$.
- $\mathsf{UpdateTag}$: takes the original file tag T_0 and block tags $T_1, \dots, T_{n'}$, the original ciphertext $\mathbf{C}$, and the updated version $\mathbf{C}'$ as inputs, returns updated file tag T'_0 and block tags $T'_1, \dots, T'_{n'}$.

  Note that the above definition is general. One can construct the $\mathsf{UpdateTag}$ algorithm which only touches the tags corresponding to the modified ciphertext, and the computational complexity can be only linear in the difference between $\mathbf{C}$ and $\mathbf{C}^*$, *i.e.*, sublinear in $|\mathbf{C}|$.
- PoWPrf: takes a challenge Q and a file $\mathbf{M}$ as inputs, returns a response $\mathcal{P}$.
- PoWVer: takes a challenge Q, the file tag T_0, the block tags $T_1, \dots, T_{n'}$, and the response $\mathcal{P}$ as inputs, returns true or false.

Compared with the old definitions [1,13,19], we make the following major changes:

1. We add a new interactive protocol Update between a user and the server that updates the ciphertext, and a $\mathsf{UpdateTag}$ algorithm. To exclude inefficient solutions requiring the user to download the original ciphertext, we require that (1) the user does not need to have $\mathbf{C}$ as an input; (2) the overall communication and computational complexities of Update and $\mathsf{UpdateTag}$ are sublinear in the file size, *i.e.*, $o(|F|)$. (Also see efficiency requirement below.)
2. The Enc algorithm takes M as input only without k_{mas}. Dropping k_{mas} does not affect correctness, nor does it conflict with existing notions, because in all existing MLE schemes that we are aware of, k_{mas} is derived from M anyway. Furthermore, in our proposed solution, the $\mathsf{M\text{-}KeyGen}$ and the $\mathsf{B\text{-}Enc}$ algorithms are basically the same algorithm with different outputs. (See Algorithm 1 for details.) Therefore, k_{mas} is only generated at the end of Enc algorithm (thus it is not available at the time when Enc is just executed). Hence it is important that we drop k_{mas} from the interface.
3. We split the Enc, Dec algorithms into two phases (in addition to KeyGen and TagGen [19]) for blocks and block keys respectively. The $\mathsf{B\text{-}Enc}$/$\mathsf{B\text{-}Dec}$ algorithm encrypts/decrypts each block $\mathbf{M}[i]/\mathbf{C}[i]$ using block keys k_i, while the $\mathsf{BK\text{-}Enc}$/$\mathsf{BK\text{-}Dec}$ algorithm generates/decrypts encrypted block keys $\mathbf{C}[n+1]||\dots||\mathbf{C}[n']$.
4. In some literatures (say [19]), it is the user who executes the TagGen algorithm which takes *plaintext* as input. To detect maliciously generated tags by users, Chen *et al.* [19] introduce a **ConTest** algorithm to check the consistency of the tags. In contrast, we assume that the tags (both file tag and block tags) are derived directly from *ciphertexts* by the honest server. So the **ConTest** algorithm [19] is not necessary.
5. Randomized tags are considered in some literature [1, 19], which mandate the need for a specialized **EqTest** algorithm for equality test. For deterministic tags generation as in our definition, **EqTest** is just the normal "=" operation, so we remove **EqTest** for simplicity[2].

3.2 Correctness

For UMLE, in additional to the file space $\mathrm{MsgSp}(\lambda)$, we also define the block space $\mathrm{BlSp}(\lambda)$ for any $\lambda \in \mathbb{N}$. For all $\lambda \in \mathbb{N}, P \leftarrow \mathsf{Setup}(1^\lambda)$ and all $\mathbf{M} \in \mathrm{MsgSp}(\lambda)$, we require the following notions of correctness.

- **Decryption Correctness.** For any block message $\mathbf{M}[i] \in \mathrm{BlSp}(\lambda)$, block key $k_i \leftarrow \mathsf{B\text{-}KeyGen}(\mathbf{M}[i])$, and block ciphertext $\mathbf{C}[i] \leftarrow \mathsf{B\text{-}Enc}(k_i, \mathbf{M}[i])$, decryption correctness requires $\mathsf{B\text{-}Dec}(k_i, \mathbf{C}[i]) = \mathbf{M}[i]$;

[2]Algorithms which are removed from our framework (**ConTest** and **EqTest**) are in bold-face.

- **Block Key Retrieval Correctness.** For all messages $\mathbf{M} \in \text{MsgSp}(\lambda)$, master key $k_{\mathsf{mas}} \leftarrow \mathsf{M\text{-}KeyGen}(\mathbf{M})$, and encrypted block keys $\mathbf{C}[n+1]||\ldots||\mathbf{C}[n']$, we have $\mathsf{BK\text{-}Dec}(k_{\mathsf{mas}}, \mathbf{C}[n+1]||\ldots||\mathbf{C}[n']) = (\mathsf{B\text{-}KeyGen}(\mathbf{M}[1]), \ldots, \mathsf{B\text{-}KeyGen}(\mathbf{M}[n]))$.

- **Tag Correctness**. For any two block messages $\mathbf{M}[i]$, $\mathbf{M}'[j] \in \text{MsgSp}(\lambda)$ s.t. $\mathbf{M}[i] = \mathbf{M}'[j]$, then for block tags $T_i \leftarrow \mathsf{B\text{-}TagGen}(\mathsf{B\text{-}Enc}(\mathsf{B\text{-}KeyGen}(\mathbf{M}[i]), \mathbf{M}[i]))$, $T'_j \leftarrow \mathsf{B\text{-}TagGen}(\mathsf{B\text{-}Enc}(\mathsf{B\text{-}KeyGen}(\mathbf{M}'[j]), \mathbf{M}'[j]))$, we have $T_i = T'_j$; furthermore, if it holds that $\mathbf{M} = \mathbf{M}'$, we require that the encrypted block keys (and thus their tags) are equal. Namely $\mathsf{Enc}(\mathbf{M}) = \mathsf{Enc}(\mathbf{M}')$ and $T_i = T'_i$ holds $\forall 0 \leq i \leq n'$.

- **Update Correctness**. For any $\mathbf{M} = \mathbf{M}[1]||\ldots||\mathbf{M}[n] \in \text{MsgSp}(\lambda)$, and any file block $\mathbf{M}[i]^* \in \text{BlSp}(\lambda)$, let $(k^*_{\mathsf{mas}}, \mathbf{C}^*) \leftarrow \mathsf{Update}((k_{\mathsf{mas}}, i, \mathbf{M}[i]^*), \mathbf{C})$ where $k_{\mathsf{mas}} = \mathsf{M\text{-}KeyGen}(\mathbf{M})$, $\mathbf{C} = \mathsf{Enc}(\mathbf{M})$, we have $\mathsf{Dec}(\mathbf{C}^*) = \mathbf{M}[1]||\ldots||\mathbf{M}[i]^*||\ldots||\mathbf{M}[n]$.

- **PoW Correctness**. For all tags $\mathbf{T} \leftarrow \mathsf{TagGen}(\mathbf{M})$, any challenge $Q, \mathcal{P} \leftarrow \mathsf{PoWPrf}(\mathbf{M}, Q)$, we have the probability $\Pr[\mathsf{PoWVer}(\mathbf{T}, \mathcal{P}) = \mathsf{true}] = 1$.

3.3 Security Definition for UMLE

We formalize the security definitions for UMLE. We consider two traditional notions for MLE: (strong) tag consistency and privacy. Furthermore, we design a new notion called "context hiding" for the Update protocol we introduce.

3.3.1 Strong Tag Consistency

We follow [13] to define a tag consistency notion for UMLE. We say that a UMLE scheme achieves strong tag consistency if no polynomial-time adversary $\mathcal{A}$ has a non-negligible advantage in the following strong tag consistency game:

- Setup: The challenger generates and sends $\mathcal{A}$ all the system parameter P.

- Output: Eventually, $\mathcal{A}$ outputs $(\mathbf{M}, C')$. Let $T'_0 = \mathsf{M\text{-}TagGen}(C')$, $T_0 = \mathsf{M\text{-}TagGen}(\mathsf{Enc}(\mathbf{M}))$, and $\mathbf{M}' = \mathsf{Dec}(\mathsf{M\text{-}KeyGen}(\mathbf{M}), C')$. If $T_0 = T'_0$, and $\mathbf{M} \neq \mathbf{M}'$, outputs 1. Outputs 0 otherwise.

We refer such an adversary $\mathcal{A}$ as an STC adversary and define $\mathcal{A}$'s advantage $\mathbf{Adv}^{\mathcal{A}}_{\mathsf{STC}}(\lambda)$ as the probability that the above game outputs 1.

Definition 1 (Tag Consistency). *A* UMLE *scheme is* STC*-secure if, for any unpredictable block source $\mathcal{M}$ and any polynomial-time* STC *adversary $\mathcal{A}$, the advantage in the strong tag consistency game, $\boldsymbol{Adv}^{\mathcal{A},\mathcal{M}}_{\mathsf{STC}}$, is negligible.*

3.3.2 Privacy

No MLE scheme can achieve the *de facto* standard of indistinguishability against chosen-plaintext attack (semantic security) [13], which the adversary has complete freedom in choosing the plaintext messages. What we could achieve, is a set of weakened privacy notions defined by Bellare *et al.* [13] for *unpredictable sources* (described in Section 2.2). In particular, the PRV-CDA notion [13] guarantees that an encryption of unpredictable message must be indistinguishable from an encryption of a random string of the same length.

We follow the spirit in existing works [13, 19] to define the privacy model for our UMLE, denoted by PRV-CDA-B*. Again, note that the syntax of our UMLE formulation is quite different from Chen *et al.*'s [19], and so does the privacy notion. (Details below.) We say that a BL-MLE scheme is secure under chosen distribution attacks if no polynomial-time adversary $\mathcal{A}$ has a non-negligible advantage in the following chosen distribution attack game PRV-CDA-B*:

- Setup: The adversary $\mathcal{A}$ sends the description of an unpredictable source $\mathcal{M}$ to the challenger. The challenger then generates and sends $\mathcal{A}$ the system parameter P.

- Challenge: The challenger randomly picks $b \leftarrow \{0,1\}$. If $b = 0$, it runs $(\mathbf{M}_0, Z) \leftarrow \mathcal{M}(\lambda)$. Otherwise, if $b = 1$, it chooses $\mathbf{M}_1$ uniformly at random from $\{0,1\}^{|\mathbf{M}_0|}$. Set $\mathbf{M} = \mathbf{M}_b$.

  Let n be the number of blocks. For each $i \in [1, n]$, the challenger computes

  1. the block keys $k_i \leftarrow \mathsf{B\text{-}KeyGen}(\mathbf{M}[i])$,
  2. the ciphertext $\mathbf{C}[i] \leftarrow \mathsf{B\text{-}Enc}(k_i, \mathbf{M}[i])$, and
  3. the encryption of k_i's: $\mathbf{C}[n+1]||\ldots||\mathbf{C}[n'] \leftarrow \mathsf{BK\text{-}Enc}(k_1, \ldots k_n)$.

  Denote $\mathbf{C} = \mathbf{C}[1]||\ldots||\mathbf{C}[n]||\mathbf{C}[n+1]||\ldots||\mathbf{C}[n']$. The challenger also computes the file tag $T_0 \leftarrow \mathsf{M\text{-}TagGen}(\mathbf{C})$ and block tags $T_i \leftarrow \mathsf{B\text{-}TagGen}(\mathbf{C}[i])$ for each $i \in [1, n]$. Set $\mathbf{T} = \{T_0, T_1, \ldots, T_{n'}\}$.

  Finally, the challenger gives auxiliary information Z, tags $\mathbf{T}$, and the ciphertext $\mathbf{C}$ to the adversary.

- Output: After receiving $(\mathbf{C}, \mathbf{T}, Z)$, the adversary outputs his guess b' and wins the game if $b' = b$.

We refer such an adversary $\mathcal{A}$ as a PRV-CDA-B* adversary and define $\mathcal{A}$'s advantage by

$$\mathbf{Adv}^{\mathcal{A},\mathcal{M}}_{\mathsf{PRV\text{-}CDA\text{-}B^*}}(\lambda) = |\Pr[b = b'] - \frac{1}{2}|.$$

Definition 2 (Privacy). *A* UMLE *scheme attains privacy or is* PRV-CDA-B**-secure if for any unpredictable block source $\mathcal{M}$ and any polynomial-time* PRV-CDA-B* *adversary $\mathcal{A}$, $\boldsymbol{Adv}^{\mathcal{A},\mathcal{M}}_{\mathsf{PRV\text{-}CDA\text{-}B^*}}$ is negligible.*

Our notion differs from that of Chen *et al.*'s [19] due to the different algorithm interface. For example, our tag generation algorithms (M-TagGen, B-TagGen) takes $\mathbf{C}$ as input instead of $\mathbf{M}$, our Enc algorithm is split into two phases, *etc.*

3.3.3 Context Hiding

We define a new security notion called "*Context hiding*", which is specific to UMLE. Roughly speaking, context hiding requires that the updated ciphertext is indistinguishable from a ciphertext encrypted from scratch. It is to ensure that the update process does not reduce the level of privacy. Consider the following motivating scenario: Alice uploaded a file F under UMLE to the storage server. She performed some updates to sanitize some sensitive information of the file before sharing it to Bob. Bob then uploads the received sanitized version F' to the same storage server. By the requirement of deduplication system, Bob will be able to download Alice's encryption version of F' from the storage server later on. If UMLE is not context hiding, it is possible that Bob can infer some information of the original file F.

We say a UMLE scheme is context hiding if no adversary $\mathcal{A}$ has a non-negligible advantage in the CXH game below:

- Setup: The challenger generates and sends $\mathcal{A}$ the system parameter P.

- Challenge: The adversary $\mathcal{A}$ chooses two messages $\mathbf{M}_0$ and $\mathbf{M}_1$ that differs in one arbitrary single block (say the i-th block) of his choosing. Upon receiving $\mathbf{M}_0, \mathbf{M}_1$, the challenger examines whether $\mathbf{M}_0, \mathbf{M}_1 \in \text{MsgSp}(\lambda)$ and whether $\mathbf{M}_0, \mathbf{M}_1$ differs in one block. If not, then the challenger aborts. Otherwise, it computes $\mathbf{C}_0 \leftarrow \mathsf{Enc}(\mathbf{M}_0)$ and $\mathbf{C}_1 \leftarrow \mathsf{Enc}(\mathbf{M}_1)$. Then it simulates an honest execution of the Update protocol. The simulated client uses $k_{\mathsf{mas}} \leftarrow \mathsf{M\text{-}KeyGen}(\mathbf{M}_1)$, i, and $\mathbf{M}_0[i]$ as inputs while the simulated server uses $\mathbf{C}_1$ as input. Denote the result by $\mathbf{C}_1'$. The simulator chooses a random bit b, and sets the challenge ciphertext $\mathbf{C}^* = \mathbf{C}_0$ if $b = 0$ or $\mathbf{C}^* = \mathbf{C}_1'$ otherwise.

- Output: Upon receiving $\mathbf{C}^*$, the adversary outputs its guess b'. The adversary wins the game if $b' = b$.

We refer such an adversary $\mathcal{A}$ as a CXH adversary and define $\mathcal{A}$'s advantage by

$$\mathbf{Adv}_{\mathsf{CXH}}^{\mathcal{A},\mathcal{M}}(\lambda) = |\Pr[b = b'] - \frac{1}{2}|.$$

Definition 3 (CONTEXT HIDING). *We say that a* UMLE *scheme is* CXH *if, for any* CXH *adversary $\mathcal{A}$, the advantage in the context hiding game, $\boldsymbol{Adv}_{\mathsf{CXH}}^{\mathcal{A},\mathcal{M}}$, is negligible.*

Note that we do not aim to hide the update pattern against the storage server since it is always revealed by the functional requirement of deduplication.

Statistical Indistinguishability. Our definition does not restrict the adversary to be polynomial time. This implies that the updated ciphertext must be statistically indistinguishable from one generated using Enc directly. If UMLE is deterministic (which is the case for our construction), this further implies that these two ciphertexts are identical.

The above definition requires that the challenge messages $\mathbf{M}_0, \mathbf{M}_1$ differ by only 1 block. This restriction is merely for definitional simplicity. By transitivity of statistical indistinguishability, it is immediate to see that the ciphertext indistinguishability remains after polynomially many updates.

3.3.4 PoW Security

As for the security of proof-of-ownership, similar to the existing definition [23], we consider the probability that an adversary can convince the server that it owns the entire file f while it only knows some partial information of f. Based on the bounded retrieval modal [21, 23] (*cf.* [41]), we assume that the adversary only knows a bounded number of blocks of f. We say that a UMLE scheme is secure against an uncheatable chosen distribution attack if no polynomially bounded adversary $\mathcal{A}$ has a non-negligible advantage in the following UNC-CDA game:

- Setup: The challenger generates and sends $\mathcal{A}$ the system parameter P.

- Challenge: $\mathcal{A}$ sends the challenger a valid source $\mathcal{M}$. The challenger runs $(\mathbf{M}, Z) \leftarrow \mathcal{M}(\lambda)$ and sends the proof query $Q = i$ and the auxiliary information Z to $\mathcal{A}$.

- Finally, $\mathcal{A}$ outputs $\mathcal{P}^*$ where $\mathsf{PoWPrf}(\mathsf{M\text{-}TagGen}(\mathbf{C}), \mathcal{P}^*, Q) \rightarrow \mathsf{true}$, *i.e.*, the proof $\mathcal{P}^*$ passes the verification. Let $\mathcal{P}$ be the expected honest response, *i.e.*, $\mathsf{PoWPrf}(\mathbf{M}, Q) \rightarrow \mathcal{P}$. If $\mathcal{P}^* \neq \mathcal{P}$, the challenger outputs 1; otherwise, outputs 0.

We refer such an adversary $\mathcal{A}$ as an UNC-CDA adversary and define $\mathcal{A}$'s advantage $\mathbf{Adv}_{\mathsf{UNC\text{-}CDA}}^{\mathcal{A},\mathcal{M}}(\lambda)$ as the probability that the game outputs 1.

Definition 4. *A* UMLE *scheme is* UNC-CDA *secure if, for any unpredictable block-source $\mathcal{M}$ and any polynomial time adversary $\mathcal{A}$, the advantage $\boldsymbol{Adv}_{\mathsf{UNC\text{-}CDA}}^{\mathcal{A},\mathcal{M}}(\lambda)$ is negligible.*

3.4 Efficiency Requirements

As mentioned in the introduction, we mandate the following efficiency requirements to exclude some degenerated solutions of UMLE.

1. For decrypting and updating arbitrarily long ciphertext, the local storage size ($|k_{\mathsf{mas}}|$) is $O(1)$.

2. The ciphertext size only inflates by a multiplicative constant: $|\mathsf{Enc}(\mathbf{M})| \in O(|\mathbf{M}|)$.

3. The computational and communicational complexities of the Update protocol and UpdateTag are sublinear, *i.e.*, $o(|\mathbf{M}|)$.

4. PROPOSED CONSTRUCTION

4.1 Intuition

We first describe the high-level idea of our construction. As far as we know, all existing MLE constructions follow the original paradigm of convergent encryption [13, 22]: derive the encryption key k_{mas} by hashing the *whole* file $\mathbf{M}$. As a consequence, a single bit-flip in $\mathbf{M}$ is likely to result in a totally different master key k'_{mas}, and hence completely different ciphertext. It seems that such paradigm inherently forbids efficient update. We bypass this obstacle by recursively deriving the master key k_{mas} *during* encryption.

In more details, we firstly derive the block keys for each block independently and encrypt each block. Then, we treat the concatenation of these block keys as a new plaintext message $\mathbf{M}'$, and apply MLE to encrypt these new blocks of messages. This process is repeated until one single block can eventually accommodate the block keys generated in the previous step. The MLE key for this last block becomes the master key k_{mas}, which is only used by the Dec algorithm and the Update protocol. Also, as a result, the M-KeyGen algorithm is almost the same as the BK-Enc algorithm except that the outputs are different. We present the detailed steps of M-KeyGen and BK-Enc together in Algorithm 1 with differences highlighted in boxes.

Figure 1 shows an example illustrating how the ciphertext looks like when encrypting an 8-block message with block size B being twice as large as the key size λ. The dashed lines indicate the implicit relation between ciphertext blocks.

With our specific design, to change one plaintext block, the Update protocol in Figure 3 only needs to change $\log(n)$

ciphertext blocks. (Also see Figure 2 for a concrete example.) Updating the block tags is straightforward: recompute directly from updated ciphertext blocks. On the other hand, updating file tag requires invoking Mu.Update of the incremental hash by Bellare *et al.* [14] $\log(n)$ times.

4.2 Detailed Construction

Here we describe the details of our proposed construction (Setup, KeyGen, Enc, Dec, TagGen). For the ease of presentation, in the following, we assume that the block size B is larger than, and divisible by the key length $|k_{\mathsf{mas}}| = \lambda$. We further assume that the number of blocks n is a power of B/λ. These restrictions are not essential (see Section 4.2.2).

- Setup: on input 1^λ, the algorithm chooses a hash function $H : \{0,1\}^* \to \{0,1\}^\lambda$. The system parameter P is the description of the hash function H.
- KeyGen: on input $\mathbf{M} = \mathbf{M}[1]||\ldots||\mathbf{M}[n]$, run the following two sub-algorithms:
  - B-KeyGen: on input $\mathbf{M}[i]$, output $k_i = H(\mathbf{M}[i])$.
  - M-KeyGen: almost identical to BK-Enc (see Algorithm 1) except a few differences shown boxed.
- Enc: on input $\mathbf{M}$, run the following two sub-algorithms:
  - B-Enc: on input block message $\mathbf{M}[i]$ and the block key k_i, $1 \le i \le n$, return $\mathbf{C}[i] = \mathsf{SKE.Enc}(k_i, \mathbf{M}[i])$.
  - BK-Enc: return $\mathbf{C}[n+1], \ldots, \mathbf{C}[n']$, see Algorithm 1.
- Dec: on input $\mathbf{C}$ and k_{mas}, run the following two sub-algorithms:
  - BK-Dec: return $k_1, \ldots, k_n$, see Algorithm 2.
  - B-Dec: on input block ciphertext $\mathbf{C}[i]$ and block key k_i, $1 \le i \le n$, return $\mathbf{M}[i] = \mathsf{SKE.Dec}(k_i, \mathbf{C}[i])$.
- TagGen: on input ciphertext $\mathbf{C}$, run the following two algorithms to return file tag T_0 and block tags $T_1, \ldots, T_{n'}$.
  - B-TagGen: return $T_i = H(\mathbf{C}[i])$ for $1 \le i \le n'$.
  - M-TagGen: return $T_0 = \mathsf{Mu.HASH}(\mathbf{C})$.

The above describes the detailed algorithms which are common in all MLE scheme. In addition to these, we also provide PoW related algorithms as [19]. This protocol will be executed when a user shows a file tag T_0 that is already stored in the server's database. The server challenges this user to make sure that s/he indeed owns this file.

- PoWPrf: Given a challenge query $Q = \{i\}$, compute the encrypted block $\mathbf{C}[i]$ for $i \in Q$. Output the proof $\Pi = \{\mathbf{C}[i]\}$.
- PoWVer: Given $\Pi = \{\mathbf{C}[i]\}$, check if they are the same as the records in the database. If so, output 1; otherwise output 0.

Finally, we present our constructions for the Update protocol that updates the ciphertext stored at the server as well as the UpdateTag algorithm. Figure 3 describes the Update protocol which is the core part of our UMLE.

- Update: The client takes as inputs k_{mas}, the to-be-updated block index i, a plaintext block $\mathbf{M}[i]^*$; the server takes $\mathbf{C}$ as input.

Algorithm 1 BK-Enc $\boxed{\text{M-KeyGen}}$

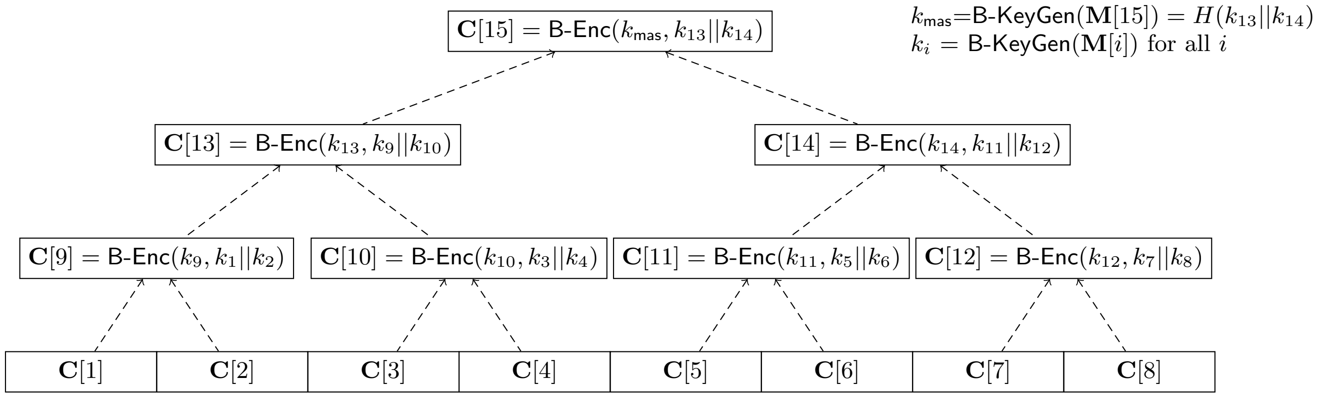
```
Input: $k_1, \ldots, k_n$ $\boxed{M}$
Output: $\mathbf{C}[n+1], \ldots, \mathbf{C}[n']$ $\boxed{k_{\mathsf{mas}}}$
 1: procedure
 2:     Set $len = n, \ell = \log_{B/\lambda} n$ // namely, $n = (B/\lambda)^\ell$
 3:     $\boxed{\text{For M-KeyGen, run B-KeyGen to get } k_1, \ldots, k_n}$
 4:     Set $\mathbf{M}' = k_1||\ldots||k_n$, $idx = n+1$
 5:     for $i \leftarrow \{1, \ldots, \ell\}$ do
 6:         $len = len \cdot \lambda/B$
 7:         Parse $\mathbf{M}'$ as blocks $\mathbf{M}'[1]||\ldots||\mathbf{M}'[len]$
 8:         for $j \leftarrow \{1, \ldots, len\}$ do
 9:             $k'_j = H(\mathbf{M}'[j])$
10:             $\mathbf{C}[idx] = \mathsf{SKE.Enc}(k'_j, \mathbf{M}'[j])$
11:             $idx = idx + 1$
12:         end for
13:         $\mathbf{M}' = k'_1||\ldots||k'_{len}$
14:         $k_{\mathsf{mas}} = H(\mathbf{M}')$
15:     end for
16:     return $\mathbf{C}[n+1], \ldots, \mathbf{C}[n']$ $\boxed{k_{\mathsf{mas}}}$
17: end procedure
```

Algorithm 2 BK-Dec

```
Input: $k_{\mathsf{mas}}, \mathbf{C}[n+1], \ldots, \mathbf{C}[n']$
Output: $k_1, \ldots, k_n$
 1: procedure
 2:     // Decrypt starting from the last block $\mathbf{C}[n']$ by $k_{\mathsf{mas}}$
 3:     Set $start = end = n', \ell = \log_{B/\lambda} n$
 4:     Set $len = 1$, $k_1 = k_{\mathsf{mas}}$
 5:     for $i \leftarrow \{1, \ldots, \ell\}$ do
 6:         $len = len \cdot B/\lambda$, $idx = 0$
 7:         // Decrypt $\mathbf{C}[start, end]$ to get $k_1, \ldots, k_{len}$
 8:         for $j \leftarrow \{start, \ldots, end\}$ do
 9:             $idx = idx + 1$
10:             $\mathbf{M}'[idx] = \mathsf{SKE.Dec}(k_{idx}, \mathbf{C}[j])$
11:         end for
12:         Parse $\mathbf{M}'[1]||\ldots||\mathbf{M}'[idx]$ as $k_1, \ldots, k_{len}$
13:         $end = start - 1$, $start = start - len$
14:     end for
15:     return $k_1, \ldots, k_n$
16: end procedure
```

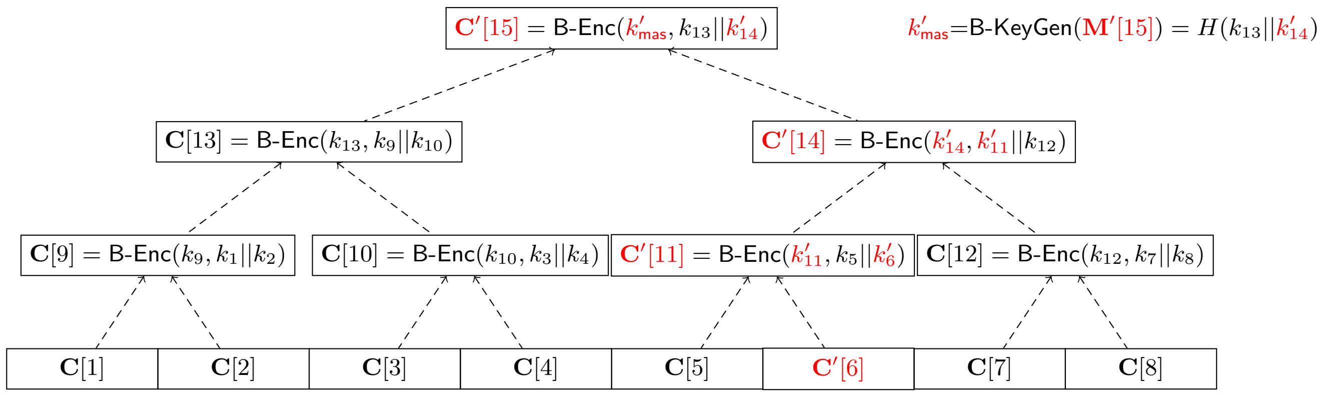
The main idea of Update is to perform a partial decryption on the ciphertext $\mathbf{C}$. Note that the ciphertexts corresponding to the actual plaintexts are placed at the leaf level of the tree (Figure 2). Changing one leaf affects all the nodes along the root-to-leaf path. The server returns all the ciphertext blocks along the path to the client. With the master key k_{mas}, the client can decrypt the blocks one by one by first decrypting the root ciphertext, retrieving the key k' for the next block, using k' to decrypt the next block, and so on. Then, the client generates a new key for the updated leaf, replaces the old key at its parent node with the new key, re-encrypts the parent node, and so on, until all the affected blocks along the root-to-leaf path are re-encrypted.

Within this process, a new master key k'_{mas} is also generated since our M-KeyGen and BK-Enc algorithms are almost the same. The above process is possible because the ciphertexts are generated in a hierarchical manner. Figure 2 shows



k_{mas}=B-KeyGen($\mathbf{M}[15]$) = $H(k_{13}||k_{14})$
k_i = B-KeyGen($\mathbf{M}[i]$) for all i
$\mathbf{C}[15]$ = B-Enc($k_{\text{mas}}, k_{13}||k_{14}$)
$\mathbf{C}[13]$ = B-Enc($k_{13}, k_9||k_{10}$)
$\mathbf{C}[14]$ = B-Enc($k_{14}, k_{11}||k_{12}$)
$\mathbf{C}[9]$ = B-Enc($k_9, k_1||k_2$)
$\mathbf{C}[10]$ = B-Enc($k_{10}, k_3||k_4$)
$\mathbf{C}[11]$ = B-Enc($k_{11}, k_5||k_6$)
$\mathbf{C}[12]$ = B-Enc($k_{12}, k_7||k_8$)
$\mathbf{C}[1]$
$\mathbf{C}[2]$
$\mathbf{C}[3]$
$\mathbf{C}[4]$
$\mathbf{C}[5]$
$\mathbf{C}[6]$
$\mathbf{C}[7]$
$\mathbf{C}[8]$

Figure 1: The ciphertext of an 8-block file $\mathbf{M} = \mathbf{M}[1]||\cdots||\mathbf{M}[8]$ when block size B is double of key size λ, $\mathbf{M}[9] = k_1||k_2$, $\mathbf{M}[10] = k_3||k_4$, $\mathbf{M}[11] = k_5||k_6$, $\mathbf{M}[12] = k_7||k_8$, $\mathbf{M}[13] = k_9||k_{10}$, $\mathbf{M}[14] = k_{11}||k_{12}$, $\mathbf{M}[15] = k_{13}||k_{14}$



k'_{mas}=B-KeyGen($\mathbf{M}'[15]$) = $H(k_{13}||k'_{14})$
$\mathbf{C}'[15]$ = B-Enc($k'_{\text{mas}}, k_{13}||k'_{14}$)
$\mathbf{C}[13]$ = B-Enc($k_{13}, k_9||k_{10}$)
$\mathbf{C}'[14]$ = B-Enc($k'_{14}, k'_{11}||k_{12}$)
$\mathbf{C}[9]$ = B-Enc($k_9, k_1||k_2$)
$\mathbf{C}[10]$ = B-Enc($k_{10}, k_3||k_4$)
$\mathbf{C}'[11]$ = B-Enc($k'_{11}, k_5||k'_6$)
$\mathbf{C}[12]$ = B-Enc($k_{12}, k_7||k_8$)
$\mathbf{C}[1]$
$\mathbf{C}[2]$
$\mathbf{C}[3]$
$\mathbf{C}[4]$
$\mathbf{C}[5]$
$\mathbf{C}'[6]$
$\mathbf{C}[7]$
$\mathbf{C}[8]$

Figure 2: Running Update to change $\mathbf{M}[6]$ to $\mathbf{M}'[6]$: only blocks $\mathbf{C}[6], \mathbf{C}[11], \mathbf{C}[14]$, and $\mathbf{C}[15]$, and k_{mas} are updated

a concrete example of how the Update protocol would change the master key (to k'_{mas}) and the ciphertext.

Finally, the following UpdateTag algorithm updates the file tag and block tags.

- UpdateTag: given two versions of ciphertext $\mathbf{C}$ and $\mathbf{C}'$, let I be the set of indices i such that $\mathbf{C}[i] \neq \mathbf{C}'[i]$, run B-TagGen($\mathbf{C}'[i]$) for all $i \in I$ to obtain new block tags T'_i. Furthermore, set $T_{\text{temp}} = T_0$, run $T_{\text{temp}} \leftarrow$ Mu.Update($i, \mathbf{C}[i], \mathbf{C}'[i], T_{\text{temp}}$) for all $i \in I$. Return the final updated file tag $T'_0 = T_{\text{temp}}$.

4.2.1 Saving time in B-Enc

The M-KeyGen and B-Enc are basically the same except the output is different. Therefore, for the first time a file is ever encrypted, or any other situation where M-KeyGen is executed right before B-Enc, we only execute M-KeyGen for once but also stores the output of $\mathbf{C}[n+1], \ldots, \mathbf{C}[n']$.

4.2.2 Flexible Number of Blocks

When the number of block n is not a power of B/λ, these three algorithms work roughly in the same way. The only difference appears when encrypting or decrypting the last block within the second for-loop. This very last block may not contain B/λ block keys. We can apply efficiently invertible padding to fill the space, so that the encryption and decryption algorithms can proceed normally.

4.3 Analysis

The correctness for *decryption*, *block key retrieval*, and *tag* are all straightforward. The former two come from the correctness of the symmetric key encryption while the last one comes from the fact that all tags are deterministic.

Recall the three efficiency requirements we defined in Section 3.4. The first one $|k_{\text{mas}}| \in O(1)$ is trivial. For the second one $|\text{Enc}(\mathbf{M})| \in O(|\mathbf{M}|)$, note that the variable *len* is initialized with n in the outer loop, and decreases by a factor of B/λ (assuming $B > \lambda$) for each iteration. Therefore, the total number of additional ciphertexts is $n \cdot \lambda/B + n \cdot (\lambda/B)^2 + \cdots + 1 = \frac{(n-1)\lambda}{B-\lambda} \in O(n)$. For the last requirement, it is straightforward to verify that the Update protocol only affects $\log(n)$ blocks due to the tree-like structure. (Also see Figure 2 for a concrete example.) The UpdateTag algorithm only involves recomputing block tag, and executing Mu.Update($\cdot$) for each updated ciphertext block.

5. SECURITY ANALYSIS

We prove the security of our scheme under the definition in Section 3. Some proofs hold in the random oracle model.

```
User($k_{\mathsf{mas}}, i, \mathbf{M}[i]^*$)                                    Server($\mathbf{C}$)

                                  $\xrightarrow{\quad i \quad}$
                                                    $start = 1,\ end = n,\ len = n$
                                                    $tmp = i,\ idx = 1,\ \ell = \log_{B/\lambda} n$
                                                    for $j \in \{1, \ldots, \ell\}$
                                                      $\mathbf{Buf}[idx] = \mathbf{C}[end + tmp]$
                                                      $tmp = \lceil tmp \cdot \lambda/B \rceil$
                                                      $len = len \cdot \lambda/B,\quad idx = idx + 1$
                                                      $start = end + 1,\quad end = start + len - 1$
                  $\xleftarrow{\mathbf{Buf}[1], \ldots, \mathbf{Buf}[\ell], \mathbf{C}[i]}$
$tmp = i$
for $j \in \{1, \ldots, \ell\}$
  $idx_j = tmp \bmod B/\lambda$
  if$(idx_j == 0)$ then $idx_j = B/\lambda$
  $tmp = \lceil tmp \cdot \lambda/B \rceil$
$varK = k_{\mathsf{mas}}$
for $j \in \{\ell, \ldots, 1\}$
  $\mathbf{K}[j]^* = \mathsf{SKE.Dec}(varK, \mathbf{Buf}[j])$
  Parse $\mathbf{K}[j]^*$ as $k_{j,1}||k_{j,2}||\ldots||k_{j,B/\lambda}$
  $varK = k_{j,idx_j}$
$varK = H(\mathbf{M}[i]^*),\ \mathbf{C}[i]^* = \mathsf{SKE.Enc}(varK, \mathbf{M}[i]^*)$
for $j \in \{1, \ldots, \ell\}$
  $\mathbf{K}[j]^* = k_{j,1}||\ldots||k_{j,idx_j-1}||varK||k_{j,idx_j+1}||\ldots||k_{j,\frac{B}{\lambda}}$
  $varK = H(\mathbf{K}[j]^*),\ \mathbf{Buf}[j]^* = \mathsf{SKE.Enc}(varK, \mathbf{K}[j]^*)$
                  $\xrightarrow{\mathbf{Buf}^*[1], \ldots, \mathbf{Buf}^*[\ell], \mathbf{C}[i]^*}$
Output $k^*_{\mathsf{mas}} = varK$                               Replace $\mathbf{Buf}[j]$ with $\mathbf{Buf}[j]^*\ \forall\ j$
                                                    Replace $\mathbf{C}[i]^*$ with $\mathbf{C}[i]^*$
```

Figure 3: Update **Protocol between User and Server**

5.1 Privacy

Theorem 1. *Suppose* $\mathsf{SKE}.\{\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec}\}$ *is a symmetric encryption scheme with key length* λ *and* $H(\cdot)$ *is modeled as a random oracle. For any source* $\mathcal{M}$ *with min-entropy* μ*, with the number of blocks being* n*, and for any adversary* $\mathcal{A}$ *making* q *queries to* $H(\cdot)$*, there exist adversaries* $\mathcal{B}', \mathcal{D}'$ *such that*

$$\begin{aligned} & \mathbf{Adv}^{\mathcal{A},\mathcal{M}}_{\mathsf{PRV\text{-}CDA\text{-}B^*}}(\lambda) \\ \leq\ & O(qn) \cdot \mathbf{Adv}^{\mathcal{B}'}_{\mathsf{KR}}(\lambda) + \mathbf{Adv}^{\mathcal{D}'}_{\mathsf{KR}}(\lambda) + \frac{2n^2}{2^B} + \frac{qn}{2^\mu} \end{aligned}$$

Proof. We prove that our construction satisfies privacy by introducing a sequence of games transiting from the world where the hidden bit is 0, to the world where the bit is 1. We show that each transition is indistinguishable by the security of the underlying primitive.

Game 0: This game has the hidden bit being 0.

Game 1: This game is identical to the previous one except that the challenger keeps a table tracking the random oracle queries when encrypting the file $\mathbf{M}$.

If there exists a query X such that $H(X)$ has been defined due to an earlier query, the challenger aborts. The total number of random oracle queries within calls to $\mathsf{BK\text{-}Enc}$ is bounded by the number of additional ciphertexts. From Section 4.3 we know that this number is $\frac{(n-1)\lambda}{B-\lambda} < n$. (We can safely assume that $B > 2\lambda$.) By the requirement that all $\mathbf{M}[i]$ are distinct, there are exactly n slots which have been defined before executing $\mathsf{BK\text{-}Enc}$. By union bound, we conclude that collision happens with probability at most $\frac{2n^2}{2^B}$.

Game 2: Now the challenger aborts if, in the challenge phase, the adversary makes "bad" query of $H(X)$ that has been defined in the encryption process. If the adversary does not make such bad queries, this game is the same as the last one. We argue that this happens with negligible probability due to KR-security of the symmetric-key encryption scheme.

Recall that the hash values defined during encryption serve as the (symmetric) encryption key, and the ciphertexts are sent to the adversary. Suppose there exists an adversary $\mathcal{B}$ who makes such bad queries with non-negligible probability, we can build an adversary $\mathcal{B}'$ which breaks the KR-security.

$\mathcal{B}'$ just guesses hash query j^* and the encryption index i^*. $\mathcal{B}'$ plants its own key-recovery challenge c^* in the i^*-th encryption and outputs $\mathcal{B}$'s j^*-th hash query. Let q be the number of random oracle queries. By a hybrid argument, we have that Pr[bad hash query in **Game 2**] $\leq qn' \cdot \mathbf{Adv}^{\mathcal{A}}_{\mathsf{KR}}(\lambda)$.

Game 3: We make a further transition where all encryptions of message $\mathbf{M}[i]$ or $\mathbf{M}'[j]$ are replaced by encryptions of random messages of the same length. This is possible due to ROR-security of the symmetric-key encryption scheme. Suppose there exists an adversary $\mathcal{D}$ who can differentiate the current game from the previous one, we can then build an adversary $\mathcal{D}'$ which breaks the ROR-security as below.

$\mathcal{D}'$ simply queries its own encryption oracle in the ROR game and uses the response to compute the ciphertext for $\mathcal{D}$. $\mathcal{D}'$ outputs whatever $\mathcal{D}$ outputs.

Game 4: The challenger aborts if the adversary queries $H(M[i])$ for some i. Note that in **Game 3**, all the ciphertexts that the adversary sees are independent of the true ciphertext $\mathbf{C}$. Thus we can apply the min-entropy of $\mathcal{M}$ to bound the above probability. By a union bound, we have Pr[bad hash query in **Game 4**] $\leq \frac{qn}{2^{\mu}}$. Moreover, this game implements exactly the case where $b = 1$. □

5.2 Tag Consistency

As observed [13], any deterministic scheme is STC secure when the tags are collision-resistant hashes of the ciphertext. It is not hard to verify that our scheme is deterministic. So we only state the theorem below and omit the obvious proof.

Theorem 2. *Suppose* SE = SKE.{KeyGen, Enc, Dec} *is a one-time secure symmetric encryption scheme and $H(\cdot)$ is a collision resistant hash function. Our proposed scheme in Section 4 is* STC*-secure.*

5.3 PoW Security

We follow the idea in [19] to show that our PoW protocol is UNC-CDA secure.

Theorem 3. *Suppose $H(\cdot)$ is a hash function modeled as a random oracle. The advantage $\mathbf{Adv}^{\mathcal{A},\mathcal{M}}_{\mathsf{UNC\text{-}CDA}}(\lambda)$ of any adversary $\mathcal{A}$ in the* UNC-CDA *game is bounded by*

$$(\frac{\ell(\lambda)}{n(\lambda)})^{q(\lambda)} + \frac{1}{2^{\mu(\lambda)}} \cdot (1 - (\frac{\ell(\lambda) - q(\lambda) + 1}{n(\lambda) - q(\lambda) + 1}))^{q(\lambda)}$$

where $n(\lambda)$ is the total number of blocks in the challenge file, $\ell(\lambda)$ is the number of blocks known by the adversary given the auxiliary information, $q(\lambda)$ is the number of queried blocks, and $\mu(\lambda)$ is the min-entropy of source $\mathcal{M}$.

Proof. This proof can be done using a simple probability argument [19][Theorem 7]. We thus omit the details. □

5.4 Context Hiding

As discussed in Section 3, when UMLE is deterministic, context hiding means the updated ciphertext must be identical to a ciphertext generated using Enc algorithm from scratch. In such case, the adversary's advantage $\mathbf{Adv}^{\mathcal{A},\mathcal{M}}_{\mathsf{CXH}}(\lambda)$ in winning the CXH game is zero. The above observation simplifies our analysis because the security will be a direct consequence of deterministic nature and correctness of our scheme. The former is trivial, and the latter comes from the correctness of the one-time secure symmetric encryption.

Theorem 4. *Suppose $H(\cdot)$ is a deterministic hash function. The adversary's advantage $\mathbf{Adv}^{\mathcal{A},\mathcal{M}}_{\mathsf{CXH}}(\lambda)$ in the* CXH *game against our scheme is* 0.

6. SUPPORTING INSERTION/DELETION

In our basic scheme, the update operation is restricted to replacing an existing block with a new one. Here, we describe an extension that also allows insertion and deletion.

Towards this end, we replace Bellare *et al.*'s incremental hash with a tailor-made hashing algorithm. To be concrete, we leverage the tree-like ciphertext structure in the basic construction to build a variant of Merkle-hash-tree [31] as the file tag. Some care must be taken because the ciphertext $\mathbf{C}$ is already implicitly organized as a tree. If we naïvely apply Merkle-hash-tree directly on the ciphertext $\mathbf{C} = \mathbf{C}[1]|| \cdots ||\mathbf{C}[n']$ ignoring the internal structure, updating a single ciphertext block will affect $O(\log n') = O(\log n)$ nodes in the Merkle tree. Now that our Update protocol changes $O(\log n)$ ciphertext blocks for each update operation, the overall complexity of UpdateTag will be $O(\log^2 n)$.

To further reduce the cost, the Merkle-tree should fit with the internal structure of the ciphertext blocks. The new M-TagGen* algorithm is presented as Algorithm 3, which outputs a complete B/λ-ary tree $\mathbf{T} = (T_0, T_1, \ldots, T_{n'})$ as the file tag. Now, when we update one ciphertext block $\mathbf{C}[i]$ for $i \in [n]$, all the $O(\log n)$ ciphertext blocks along the root-to-leaf path will be affected. The exact same root-to-leaf path in the Merkle-hash tree needs to be updated. Therefore, the overall complexity remains $O(\log n)$ only.

Algorithm 3 M-TagGen*

```
Input: $\mathbf{C} = \mathbf{C}[1]||\ldots||\mathbf{C}[n]||\mathbf{C}[n+1]||\ldots||\mathbf{C}[n']$
Output: $\mathbf{T} = (T_0, T_1, \ldots, T_{n'})$
 1: procedure
 2:     for $i \leftarrow \{1, n\}$ do
 3:         $T_i = H(\mathbf{M}[i])$
 4:     end for
 5:     Set $len = n, \ell = \log_{B/\lambda} n$ // namely, $n = (B/\lambda)^{\ell}$
 6:     Set $\mathbf{M}' = T_1||\ldots||T_n$, $idx = n + 1$
 7:     for $i \leftarrow \{1, \ell\}$ do
 8:         $len = len \cdot \lambda/B$
 9:         Parse $\mathbf{M}'$ as blocks $\mathbf{M}'[1]||\ldots||\mathbf{M}'[len]$
10:         for $j \leftarrow \{1, len\}$ do
11:             $T_{idx} = H(\mathbf{C}[idx]||\mathbf{M}'[j])$
12:             $idx = idx + 1$
13:         end for
14:         $\mathbf{M}' = T_{idx-len}||\ldots||T_{idx-1}$
15:     end for
16:     $T_0 = H(\mathbf{M}'||n')$
17:     return $\mathbf{T} = (T_0, T_1, \ldots, T_{n'})$
18: end procedure
```

Now we describe how to support insertion and deletion without degrading performance. We leverage the techniques of weight balancing B-tree [4], an efficient data structure that keeps data in order. Insertion and deletion of leaves in such a B-tree can be done in time logarithmic in the size of the tree. B-tree is parameterized by a branching parameter p. Each internal node of a weight balancing B-tree has a fan-out within the range $[p/4, p]$. If the fan-out of a node becomes more (less) than p ($p/4$) due to insertion (deletion), this node will be split into two (merged with a neighboring node). Merge and split are called "re-balancing" operations.

Our basic construction is modified as follows: let $p = B/\lambda$, instead of accommodating p keys in each block in encryption, we place only $p/2$ keys within a block initially. We impose the following restrictions after insertion/deletion of a certain block: the number of keys in each block must be within the range $[p/4, p]$. Violation will cause a block to split ($> p$) or merge with a neighboring block ($< p/4$). In case a merge causes an immediate overflow, we split this block evenly. The exact split and merge operations can be found in [4].

The complexity of our modified scheme matches that of weight balancing B-tree. We omit the repetitive details.

7. RELATED WORK

In this section, we will examine a few MLE related primitives. They are closely related to MLE because of their similarities in functionalities or security definition.

7.1 Secure Deduplication and MLE

To deal with a large number of block keys when block-level deduplication is employed, Li *et al.* [29] and Puzio *et al.* [34] introduce a set of key-management servers for storing the secret shares of the block keys. Jiang *et al.* [26] reduce the number of server-side comparison during deduplication.

Armknecht *et al.* [5] consider a new interesting research direction: how to make the customers of cloud storage service also economically benefit from deduplication technique. Liu *et al.* [30] connects password-based authenticated key exchange with secure deduplication.

After the first formulation of MLE [13], its security notion of MLE has been strengthened [1,12]. Abadi *et al.* [1] additionally consider plaintext distributions which may depend on the public parameter of the scheme. Unfortunately, their construction is not known to be secure for correlated input distribution. Bellare and Keelveedhi [12] achieve that in the standard model by allowing interactive upload/download protocols plus some heavy cryptographic machinery like fully homomorphic encryption [24] and composable distributional indistinguishable point-function obfuscation [16]. They further propose the first efficient secure deduplication system supporting incremental updates in the random oracle model, but their scheme only deduplicates at the file level.

7.2 Plaintext-Check in Encryption/Encoding

Plaintext-checkable encryption [18]/encoding [40] allows anyone to test if a ciphertext/encoding c corresponds to a given plaintext m. The difference between encoding and encryption is that encoding does not require decryption functionality. Their security guarantees are both called *unlinkability*. Essentially an encryption/encoding scheme is unlinkable if no polynomial-time adversary can distinguish between encryptions/encodings of two different messages and two encryptions/encodings of a single message, where the two messages are drawn from a high min-entropy space. Such notion cannot be achieved by deterministic encryption, but is strictly weaker than the standard semantic security.

All deterministic/randomized MLE is plaintext-checkable: holding m and c, one could derive a tag from c, re-encrypt m and generate a new tag to check if c is also derived from m. But there is no unlinkable MLE as MLE requires the ability to detect duplicated ciphertexts. Compared with MLE, plaintext-checkable encryption/encoding is slightly less versatile, hence allows a slightly stronger security notion.

7.3 Equality Test in Public-key Encryption

Public-key encryption with equality test (PKET) [39] is a (probabilistic) public-key encryption scheme which allows anyone to test whether two ciphertexts contain the same message. More formally, given two ciphertexts C_1 and C_2 generated under possibly different public-keys PK and PK' respectively, one can test if C_1 and C_2 are encryptions of the same message, without decryption. The security notion considered is one-way adaptive chosen ciphertext attack [39].

PKET is similar to MLE except that MLE is symmetric-key while PKET is public-key. As PKET allows plaintext check across different public-keys, its functionality seems to be slightly stronger than MLE (encryptions of the same message using MLE with different public parameters cannot be identified). Interestingly, the tag construction algorithm of an existing MLE scheme supporting lock-dependent message [1] is very similar to the test algorithm of the first PKET scheme [39]. PKET only considers one-wayness for security. It is interesting to formulate and achieve stronger notions.

7.4 Encryption with Weak/No Randomness

Hedged public-key encryption [8] achieves indistinguishability against chosen-plaintext attack (CPA) when the randomness is of high quality. Even when the randomness is bad, it can still achieve a weaker security notion called indistinguishability under a chosen distribution attack.

Deterministic public key encryption [7] is directly applicable to deduplicate ciphertexts, but it seems impossible to deduplicate encryption of the same message under different public keys. With no randomization in encryption, the "best possible" security notions under chosen plaintext attack or chosen ciphertext attack are formulated [7].

8. CONCLUSION AND FUTURE WORK

We initiate the study of efficiently updatable block-level message-locked encryption. We provide an efficient construction in the random oracle model, with update cost logarithmic in the file size. Notably, our construction is pairing-free, in contrast to existing non-updatable block-level MLE [19]. For a stronger foundation, a future direction is to propose new constructions in the standard model. Moreover, it is important to further reduce the computation and communication cost of the update.

Acknowledgements

Sherman Chow is supported in part by the Early Career Scheme and the Early Career Award (CUHK 439713), and General Research Funds (CUHK 14201914) of the Research Grants Council, University Grant Committee of Hong Kong.

9. REFERENCES

[1] M. Abadi, D. Boneh, I. Mironov, A. Raghunathan, and G. Segev. Message-locked encryption for lock-dependent messages. In *CRYPTO*, 2013.

[2] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. Wattenhofer. FARSITE: federated, available, and reliable storage for an incompletely trusted environment. In *OSDI*, 2002.

[3] P. Anderson and L. Zhang. Fast and secure laptop backups with encrypted de-duplication. In *LISA*, 2010.

[4] L. Arge and J. S. Vitter. Optimal external memory interval management. *SIAM J. Comput.*, 32(6):1488–1508, 2003.

[5] F. Armknecht, J. Bohli, G. O. Karame, and F. Youssef. Transparent data deduplication in the cloud. In *CCS*, 2015.

[6] C. Batten, K. Barr, A. Saraf, and S. Trepetin. pStore: A secure peer-to-peer backup system. *Unpublished report, MIT Laboratory for Computer Science*, 2001.

[7] M. Bellare, A. Boldyreva, and A. O'Neill. Deterministic and efficiently searchable encryption. In *CRYPTO*, 2007.

[8] M. Bellare, Z. Brakerski, M. Naor, T. Ristenpart, G. Segev, H. Shacham, and S. Yilek. Hedged public-key encryption: How to protect against bad randomness. In *ASIACRYPT*, 2009.

[9] M. Bellare, A. Desai, E. Jokipii, and P. Rogaway. A concrete security treatment of symmetric encryption. In *FOCS*, 1997.

[10] M. Bellare, O. Goldreich, and S. Goldwasser. Incremental cryptography: The case of hashing and signing. In *CRYPTO*, 1994.

[11] M. Bellare, O. Goldreich, and S. Goldwasser. Incremental cryptography and application to virus protection. In *STOC*, 1995.

[12] M. Bellare and S. Keelveedhi. Interactive message-locked encryption and secure deduplication. In *PKC*, 2015.

[13] M. Bellare, S. Keelveedhi, and T. Ristenpart. Message-locked encryption and secure deduplication. In *EUROCRYPT*, 2013.

[14] M. Bellare and D. Micciancio. A new paradigm for collision-free hashing: Incrementality at reduced cost. In *EUROCRYPT*, 1997.

[15] M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *CCS*, 1993.

[16] N. Bitansky and R. Canetti. On strong simulation and composable point obfuscation. *J. Cryptology*, 27(2):317–357, 2014.

[17] E. Buonanno, J. Katz, and M. Yung. Incremental unforgeable encryption. In *FSE*, 2001.

[18] S. Canard, G. Fuchsbauer, A. Gouget, and F. Laguillaumie. Plaintext-checkable encryption. In *CT-RSA*, 2012.

[19] R. Chen, Y. Mu, G. Yang, and F. Guo. BL-MLE: block-level message-locked encryption for secure large file deduplication. *IEEE Trans. Information Forensics and Security*, 10(12), 2015.

[20] L. P. Cox, C. D. Murray, and B. D. Noble. Pastiche: Making backup cheap and easy. In *OSDI*, 2002.

[21] G. D. Crescenzo, R. J. Lipton, and S. Walfish. Perfectly secure password protocols in the bounded retrieval model. In *TCC*, 2006.

[22] J. R. Douceur, A. Adya, W. J. Bolosky, D. Simon, and M. Theimer. Reclaiming space from duplicate files in a serverless distributed file system. In *ICDCS*, 2002.

[23] S. Dziembowski. Intrusion-resilience via the bounded-storage model. In *TCC*, 2006.

[24] C. Gentry. Fully homomorphic encryption using ideal lattices. In *STOC*, 2009.

[25] S. Halevi, D. Harnik, B. Pinkas, and A. Shulman-Peleg. Proofs of ownership in remote storage systems. In *CCS*, 2011.

[26] T. Jiang, X. Chen, Q. Wu, J. Ma, W. Susilo, and W. Lou. Towards efficient fully randomized message-locked encryption. In *ACISP*, 2016.

[27] S. Kamara, C. Papamanthou, and T. Roeder. Dynamic searchable symmetric encryption. In *CCS*, 2012.

[28] R. W. F. Lai and S. S. M. Chow. Parallel and dynamic structured encryption. In *SecureComm*, 2016. To appear.

[29] J. Li, X. Chen, M. Li, J. Li, P. P. C. Lee, and W. Lou. Secure deduplication with efficient and reliable convergent key management. *IEEE Trans. Parallel Distrib. Syst.*, 25(6), 2014.

[30] J. Liu, N. Asokan, and B. Pinkas. Secure deduplication of encrypted data without additional independent servers. In *CCS*, 2015.

[31] R. C. Merkle. A digital signature based on a conventional encryption function. In *CRYPTO*, 1987.

[32] D. T. Meyer and W. J. Bolosky. A study of practical deduplication. *TOS*, 7(4), 2012.

[33] I. Mironov, O. Pandey, O. Reingold, and G. Segev. Incremental deterministic public-key encryption. In *EUROCRYPT*, 2012.

[34] P. Puzio, R. Molva, M. Önen, and S. Loureiro. Cloudedup: Secure deduplication with encrypted data for cloud storage. In *IEEE CloudCom*, 2013.

[35] P. Rogaway, M. Bellare, and J. Black. OCB: A block-cipher mode of operation for efficient authenticated encryption. *ACM Trans. Inf. Syst. Secur.*, 6(3), 2003.

[36] Y. Sasaki and K. Yasuda. A new mode of operation for incremental authenticated encryption with associated data. In *SAC*, 2015.

[37] H. Shacham and B. Waters. Compact proofs of retrievability. *J. Cryptology*, 26(3):442–483, 2013.

[38] Z. Wilcox-O'Hearn and B. Warner. Tahoe: the least-authority filesystem. In *StorageSS*, 2008.

[39] G. Yang, C. H. Tan, Q. Huang, and D. S. Wong. Probabilistic public key encryption with equality test. In *CT-RSA*, 2010.

[40] Y. Zhao and S. S. M. Chow. Privacy preserving collaborative filtering from asymmetric randomized encoding. In *FC*, 2015.

[41] Y. Zhao and S. S. M. Chow. Towards proofs of ownership beyond bounded leakage. In *Provable Security*, pages 340–350, 2016.