Leakage-Resilient Client-side Deduplication of Encrypted Data in Cloud Storage*

Jia Xu[1], Ee-Chien Chang[2], and Jianying Zhou[1]

[1] Institute for Infocomm Research
{xuj, jyzhou}@i2r.a-star.edu.sg
[2] National University of Singapore
changec@comp.nus.edu.sg

Abstract. Cloud storage service is gaining popularity in recent years. Client-side deduplication is widely adopted by cloud storage services like Dropbox, MozyHome and Wuala, to save bandwidth and storage. Security flaws, which may lead to private data leakage, in the current client-side deduplication mechanism are found recently by Harnik *et al.* (S&P Magazine, '10) and Halevi *et al.* (CCS '11). Halevi *et al.* identified an important security issue in client side deduplication which leads to leakage of private users' files to outside attackers, and addressed this issue by constructing schemes which they called *proofs of ownership* (PoW). In a proof of ownership scheme, any owner of the same file F can prove to the cloud storage that he/she owns file F in a robust and efficient way, even if a certain amount of arbitrary information about file F is leaked. In this paper, we make two main contributions:

- We construct an efficient hash function $\mathsf{H}_k : \{0,1\}^M \rightarrow \{0,1\}^L$ with complexity in $\mathcal{O}(M + L)$, which is provably pairwise-independent in the random oracle model. We apply the constructed hash function to obtain a proof of ownership scheme, which is provably secure w.r.t. *any* distribution of input file with sufficient min-entropy, in the random oracle model. In contrast, the PoW scheme (the last and the most practical construction) in Halevi *et al.* is provably secure w.r.t. only *a particular type* of distribution (they call it a generalization of "block-fixing" distribution) of input file with sufficient min-entropy, in the random oracle model.
- We propose the first (to the best of our knowledge) solution to support cross-user client side deduplication over encrypted data in the *leakage-resilient* model, where a certain amount of arbitrary information about users' files are leaked. Particularly, we address another important security issue in client side deduplication— confidentiality of users' sensitive files against the honest-but-curious cloud storage server, by proposing a method to distribute a randomly chosen per-file encryption key to all owners of the same file, in an efficient and secure way. This key distribution method will be seamlessly incorporated into the process of client side deduplication. We emphasize that "convergent encryption", which encrypts a file F using hash value $h(F)$ as encryption key, is not leakage-resilient and is thus insecure in the setting of PoW. Therefore, the direct combination of a PoW scheme and convergent encryption is not a solution for client side deduplication over encrypted data.

Keywords: Cloud Storage, Client-side Deduplication, Zero Knowledge Proofs of Ownership, Privacy, Pairwise Independent Hash

1 Introduction

Cloud storage service is gaining popularity. To reduce resource consumption in bandwidth and storage, many cloud storage services including Dropbox [2][3] and Wuala [3] employs client-side deduplication [4].

* The original title of an early version of this work is "Secure Cloud Storage with Encrypted Data using File-Based Authentication".

[3] In Feb 2012, we noticed that Dropbox disabled the deduplication across different users, in face to rent vulnerabilities discovered in their original cross-user client-side deduplication method. This also indicates the importance and urgency in the study of security in client side deduplication.

That is, when a user tries to upload a file to the server, the server checks whether this particular file is already in the cloud (uploaded by some user previously), and saves the uploading process if it is. In this way, every single file will have only one copy in the cloud (Single Instance Storage). SNIA white paper [5] reported that the deduplication technique can save up to 90 % storage, dependent on applications.

According to Halevi *et al.* [6][7], *previous* implementation of client side deduplication of Dropbox can be described as below: A user Alice tries to upload a file F to the cloud storage of Dropbox. The client software of Dropbox installed on Alice's computer will compute and send the hash value $\texttt{SHA256}(F)$ to the Dropbox server. The Dropbox server maintains a database of hash values of all received files, and looks up the value $\texttt{SHA256}(F)$ in this database. If there is no match found, then file F does not exist in Dropbox's cloud storage yet. Alice's Dropbox client software will be required to upload F to Dropbox's cloud storage, and the hash value $\texttt{SHA256}(F)$ will be added into the look-up database. If there is a match found, then file F is already in Dropbox's cloud storage, uploaded by other users or even by the same user Alice before. In this case, uploading of file F from Alice's computer to Dropbox's cloud storage is saved and Dropbox will allow Alice to access the file F in its cloud storage. We may refer to the above client side deduplicaiton method as "hash-as-a-proof" method. Note that in this method, the hash value $\texttt{SHA256}(F)$ serves two purposes: (1) it is an index of file F, used by Dropbox to locate information of F among a huge number of files; (2) it is treated as a "proof" that Alice owns file F.

1.1 Security Concerns

Despite its significant benefits in saving resource, client side deduplication may bring in new security vulnerability and lead to leakage of users' private data, especially when partial information of users' files have already been leaked.

1.1.1 Data Privacy against Third Party Attackers Recently, an attack on hash-as-a-proof method in popular cloud storage service like Dropbox and MozyHome is proposed [8,6]: If the adversary *somehow* has the hash value of a file stored in the cloud storage, he could fool the cloud server that he has the file by presenting only the hash value as "proof" in the client-side deduplication process, and thus gain access to that file in the cloud. This attack is practical and does not require the adversary to find a collision of the hash function, since client software of cloud service can be easily bypassed. For example, an adversary may develop his/her own version of client software using public API[4], and manipulate the computation result of hash function.

Halevi *et al.* [6] proposed a primitive called *Proofs of Ownership* (PoW) to allow an owner of a file to prove to the server that he/she indeed has that file in a robust and efficient way, even if some arbitrary information about the file is leaked. Informally, in a Proofs of Ownership scheme, if an adversary *somehow* has only *limited* amount of arbitrary information of a file, he cannot convince the cloud that he/she owns the file.

1.1.2 Data Privacy against Cloud Storage Server Confidentiality of users' sensitive data against the cloud storage server itself is another security concern. As long as it is possible, prudent users hope to ensure that the cloud storage server is technically unable to access their data. Dropbox claims they protect users data with AES encryption. However, the encryption keys are chosen and kept by Dropbox itself. It is reported that, Dropbox mistakenly kept all user data open to public for almost 4 hours, due to a new bug in their software [9]. If users' data are encrypted on client side and the encryption keys are kept away from Dropbox, then there will be no such single point failure of

[4] Dorpbox provides public API. Furthermore, such attacks can not be eliminated just by hidding API, since the adversary could perform reverse-engineering attack to guess the communication protocol between client and server of the cloud service.

privacy protection of all users's data, even if Dropbox made such mistakes or was hacked in. We also notice that very recently, a bug in Twitter's client software which allows adversary to access users' private data, is discovered [10].

However, straightforward client side encryption with randomly chosen encryption key will stop deduplication [8]: Encryptions of the same file by different users with different encryption keys will result in different ciphertexts! Traditional solution to this problem is *Convergent encryption* [11,12], which encrypts an input file with the hash value of the input file as AES encryption key. Thus identical files will be always encrypted into identical ciphertext, assuming the same initial value `IV` is used in the AES encryption. Unfortunately, convergent encryption is insecure in the setting of PoW, where the hash value of the file to be encrypted may be leaked.

1.1.3 A New Attack—Poison attack We found that a third party attacker might be able to corrupt users' data file indirectly via the cloud storage service, if client side deduplication is not implemented properly. Take hash-as-a-proof method as an example. Suppose user Alice has a file F, and an adversary Bob somehow obtains the hash value $h(F)$ but does not have file F. Bob finds an arbitrary file F', and tells the cloud storage server that he wants to upload a file with hash value equal to $h(F)$. Suppose $h(F)$ is not in the cloud storage server's lookup database yet. Then the cloud storage server will put $h(F)$ in the lookup database and transmit the file F' provided by Bob to its data center. Next, Alice tells the cloud storage server that she wants to upload a file with hash value $h(F)$. The cloud storage server finds this hash value is already in the lookup database—a match occurs and client side deduplication applies. As a result, Alice will be informed that her file F is already in the cloud and uploading of F to the cloud is saved. Alice might delete the file F from her local storage, since she believes that her file is already in the cloud storage. Later, when Alice wants to access file F, she may try to retrieve file F from the cloud. However, what she can download from the cloud will be a poisoned version F', instead of the original file F.

The above attack does not require that $h(F') = h(F)$. The root cause of this attack is that, the cloud server only computes the hash value of each file uploaded by users on client side, in order to save computation resource on server side, and the client software of cloud storage can be bypassed. In case of Dropbox, users' files will be uploaded directly to Amazon's S3 server from Dropbox's client software.

We call the above attack as "poison attack". It can be prevented if the cloud storage server re-compute the hash value of each file on the server side. However, if users' files are encrypted such that the cloud storage server does not have the encryption key, the cloud storage server *alone* may not be able to defend such attack: If the cloud uses hash value $h(F)$ to index file F, then the cloud is not able to re-compute $h(F)$ with only a ciphertext C_F of F; If the cloud uses hash value $h(C_F)$ of the ciphertext C_F to index file F, the a subsequent user who wants to upload file F to the cloud, is not able to compute the hash value $h(C_F)$, since at this moment he/she does not know the encryption key which encrypts F into C_F.

1.1.4 Plausible Approaches Intuitively, convergent encryption [11,12], which encrypts a file F with the hash value $h(F)$ as key, together with PoW might provide a solution for client side deduplication over encrypted files. That is, encrypted file F to generate ciphertext C_F with hash value $h(F)$ as encryption key and then apply PoW scheme over C_F. Indeed, Wuala [3] adotps convergent encryption to encrypt users files. However, the threat models of PoW and convergent encryption are incompatible. In the setting of PoW, some arbitrary information about the file F can be leaked. But convergent encryption is insecure if the short value $h(F)$ is leaked. Roughly speaking, convergent encryption is as insecure as using hash value $h(F)$ as a proof of ownership of file F.

Another approach is that each cloud user chooses his/her own per-user encryption file, and all files uploaded to the cloud by the same user will be encrypted under this user's encryption key. If a single user uploads the same file more than once to the cloud, the subsequent upload will be saved. This approach only allows deduplication of files that belong to the same user, which will severely whittle

down the effect of deduplication. In this paper, we are interesting in the deduplicaiton cross different users, that is, identical duplicated files from different uses will be detected and removed safely.

1.2 Our results and contribution

1.2.1 Overview of proposed scheme We briefly describe the proposed client side deduplication scheme over encrypted files as below.

First Upload of File F. Suppose Alice is the first user who tries to upload a sensitive file F to the cloud storage. She will independently choose a random AES key τ, and produces two ciphertexts: The first ciphertext C_F is generated by encrypting file F with encryption key τ using AES method; the second ciphertext C_τ is generated by encrypting the short AES key τ with file F as the encryption key using some *custom encryption method*. Next, Alice will compute a long (e.g. 64 megabytes) digest $\mathsf{H}_k(F)$ with public random key k using our custom designed pairwise independent hash function H_k, and build a Merkle Hash Tree over $\mathsf{H}_k(F)$. Let r_F denote the value at the root of Merkle Hash Tree. Finally, Alice will send two hash values $h(F)$ and $h(C_F)$, two ciphertexts C_F and C_τ, and value r_F at the root of Merkle Hash Tree to the cloud storage server, where C_F is the encryption of user file F indexed with $h(F)$, and $(h(C_F), C_\tau, r_F)$ is the short meta data indexed with $h(F)$. The cloud storage server will add an entry ($\texttt{key} = h(F)$; $\texttt{value} = (h(C_F), C_\tau, r_F)$) into its lookup database.

Subsequent Upload of File F. Suppose another user Carol tries to upload the same file F into the cloud after Alice has already uploaded F. Carol sends hash value $h(F)$ to the cloud, and the cloud finds a match of $h(F)$ in its lookup database. The cloud also finds the corresponding meta data—$(h(C_F), C_\tau, r_F)$. Carol first proves to the cloud that she indeed owns file F without reveal useful information of F, using a *privacy-preserving proof of ownership* scheme: The cloud asks for the value associated to a randomly chosen leaf node (say the i-th leaf node) in the Merkle Hash Tree associated with file F. Carol re-computes the long digest $\mathsf{H}_k(F)$ with the same public random key k and re-builds the Merkle Hash Tree over $\mathsf{H}_k(F)$, and finds the value v_i associated to the queried leaf node. Then, Carol can prove to the cloud that v_i is the correct value for i-th leaf using the Merkle Hash Tree, and the cloud can verify Carol's proof against the root value r_F provided by Alice.

If the cloud is convinced, then it will send the short ciphertext C_τ to Carol. From the received short ciphertext C_τ, Carol can decrypt it using file F as decryption key to obtain the AES key τ. Carol can encrypt her file F with AES key τ to generate C_F and send the hash value $h(C_F)$ to the cloud. The cloud will check whether Carol's version of hash value $h(C_F)$ is the same as the one provided by Alice. If the two hash values are different, then a poison attack on file F is detected.

After this, assuming that there is no poison attack, Carol could remove the local copy of file F if she likes and keep the AES key τ safely in local storage. Carol can always recover file F by download the ciphertext C_F from the cloud and decrypt it with key τ.

1.2.2 Our Contributions In this paper, we focus on client side deduplication over users' sensitive data files, and protect data privacy from both outside attackers and the honest-but-curious cloud storage server. Our contributions in this paper can be summarized as below:

- We design a hash function $\mathsf{H}_k : \{0,1\}^M \to \{0,1\}^{\rho\ell}$ with large output size in Section 3.3, based on an underlying hash function h_k with output size equal to a constant ρ (e.g. $\rho = 256$). We prove in Theorem 2 that H_k is pairwise independent, if h_k is 4ℓ-independent. If the hash function h_k is instantiated as $h_k(x) = \texttt{SHA256}(k\|x)$, then the computation complexity of H_k is in $\mathcal{O}(M + \ell)$. The high efficiency of the constructed hash function is achieved based on our observation: *Computation of* $\texttt{SHA256}(x\|y)$ *can be computed in time* $\mathcal{O}(|y|)$ *if some short intermediate result in computation of* $\texttt{SHA256}(x)$ *is available, due to the underlying Merkle-Damgård construction of* SHA256 *hash function.*

- We construct a proof of ownership scheme, by incorporating the newly constructed pairwise independent hash function H and the generic framework of Halevi *et al.* [6]. The new proof of ownership scheme reveals only two hash value $h(F)$ and $\mathsf{H}_k(F)$ to the verifier during the proof process, where F is the input file.
- We propose in Section 4 the first client side deduplication scheme over encrypted files in the leakage resilient model. In this scheme, each sensitive file is encrypted on the client side using a randomly chosen encryption key and this key is kept securely against the honest-but-curious cloud storage server. Cross-user client side deduplication of the encrypted files are supported. We prove in Theorem 6 that the proposed scheme is secure under formulation in Section 2, even if a certain amount of arbitrary information of the sensitive file is leaked, in the random oracle model. This scheme consists two main components: one is the newly constructed proof of ownership scheme, and the other is a custom designed method which distributes a secret per-file encryption key to each owner of the corresponding file in an efficient and secure way. This key distribution method requires the help of the cloud storage server, but does not reveal the per-file encryption key to the cloud storage server.
- We also identify a new attack (called "poison attack") in cloud storage service. In this attack, a third party adversary is able to corrupt cloud users' files indirectly, if the client side deduplicaiton is implemented in an improper way. The proposed client side deduplication scheme in this paper can defeat this attack.

2 Security Model: Leakage-Resilient Client-side Deduplication

Although PoW [6] solves an important security issue in client side deduplicaiton, it does not address the problem of protecting data privacy against the honest-but-curious cloud storage server. Furthermore, PoW formulation does not consider user authentication system and has to be revised, in front of user authentication system: One one side, PoW requires there should be no short "proxy" (or "representation") of a file, such that with only this short proxy, an adversary can prove to the cloud storage server that he/she owns the file; on the other side, the user account of some owner of a file can effectively serve as a short proxy of the file. If an adversary obtains such user account, then he/she can download the file from the cloud storage server and then prove to the cloud that he/she owns the file. Even worse, the root account of the cloud storage server can server as a short proxy of all users' files.

In this paper, we revise the leakage model of PoW, by introducing restriction on information leakage of the file of interest to the adversary.

- In PoW, the adversary can collude with an accomplice who has file F and has already been recognized as an owner of file F by the cloud. So any short secret states that the user establishes with the cloud is known to the adversary. There are two constraints: (1) the number of bits that adversary can receive from the accomplice is limited; (2) the accomplice cannot help the adversary, *during* the process that the adversary attempts to prove the ownership of the file F to the cloud.
- In our security formulation, the adversary can only collude with an accomplice who has file F but yet has not been recognized as an owner of file F by the cloud. The same two constraints applies.

2.1 Client side Deduplication: System Model and Trust Model

2.1.1 Cloud Storage Server. Cloud Storage Server (Cloud Server or Cloud for short) is the entity who provides cloud storage service to various users. Cloud Storage Server has a small and fast primary storage and a large but slow secondary storage. Although the computation power (CPU, I/O, network, etc) of Cloud Storage Server is much stronger than a single average user, the computation power per each online user is usually very limited. Cloud Storage Server is considered as honest but curious: It will maintain the integrity of users' data and the availability of cloud service. However, it is curious

to access users' sensitive data. We assume that the small and fast primary storage is well-protected from outside attackers, and the large but slow secondary storage is visible to outside attackers.

An example of cloud storage server is Dropbox [2]. Users' data uploaded to Dropbox are actually stored in Amazon's S3 data center (We call this as Dropbox's secondary storage) and Dropbox only runs relative small server to manage meta data (We call this as Dropbox's primary storage). Another cloud storage service provider Wuala [3] stored large users data in P2P network (the secondary stoarge) in the early stage of the company. Although the computation power of a cloud storage server could be much larger than each single user, the average computation power per each online user could be much smaller than a single user, due to large number of online users.

2.1.2 Cloud Users. There are many Cloud Users who register user accounts (i.e. a pair of short id and password) from a Cloud Storage Server. Each cloud user could login the cloud storage server with their user accounts, and then upload their files to the cloud or retrieve files from cloud.

Some of users' data files stored in the cloud could be sensitive, including those digital asset protected by copyright. Prudent users may intend to protect the confidentiality of their sensitive data against not only the outside attackers but also the curious cloud storage server. In this paper, we focus on users' sensitive data files only. We emphasize that different users may share identical sensitive files due to many reasons, including that a group of users who might not know each other, receive a classified or copyright-protected files directly or indirectly from the same information source, via a legal channel. Furthermore, users may have their rights and personal opinions to decide what files are sensitive, for example, any file that they do not want to share with public, could be considered as sensitive.

2.1.3 Adversaries We consider two types of adversaries: Malicious third party adversary and honest-but-curious Cloud server.

Third Party Adversary. The third party adversary is also called as outside adversary. This adversary may obtain some knowledge (e.g. a hash value) of the file of interest via some channels. For example, they might receive a file which is similar to the file of interest. PoW [6] gave an example that different employees of the same company may get a long profile documents from the company, where such documents may only differ in small places like personal particulars and salaries. Similar to PoW, in our formulation, the third party adversary is allowed to access a certain amount of arbitrary information of the file of interest.

Honest-but-Curious Cloud Server. This honest but curious cloud storage server will maintain the integrity of users' files and availability of the cloud service, but is curious about users' sensitive files. This could capture at least the following cases in real world applications:

1. Some technical employee or even the owner of the cloud tries to access user data due to some reason, e.g. pure curiosity or for commercial benefits.
2. The company, which provide the cloud storage service, made careless technical mistake and leak users' private data, e.g. introducing a software bug. It is reported that Dropbox [9] made users' data open to public for almost 4 hours due a new software bug. Very recently, a bug is discovered in one of Twitter's official client software, which allows attackers to access users' accounts [10].

2.2 Client-side Deduplication: Syntax Definition

A Client-side Deduplication (called $\mathcal{CSD}$ for short) scheme $(\mathcal{E}, \mathcal{D}, \mathcal{P}, \mathcal{V})$ consists of four algorithms $\mathcal{E}$, $\mathcal{D}$, $\mathcal{P}$ and $\mathcal{V}$, which are explained as below:

– $\mathcal{E}(F, 1^\lambda) \rightarrow (\tau, C_\tau, C_F)$: The probabilistic encoding algorithm $\mathcal{E}$ takes as input a data file F and a security parameter λ, and outputs a short secret encryption key τ, a ciphertext C_τ of τ, and a ciphertext C_F of file F.

- $\mathcal{D}(\tau, C_F) \to F$: The deterministic decoding algorithm takes as input an encryption key τ and a ciphertext C_F, and outputs a file F.
- $\langle \mathcal{P}(F), \mathcal{V}(C_\tau) \rangle \to (y_0;\ y_1, y_2)$: The prover algorithm $\mathcal{P}$, which takes a file F as input, interacts with the verifier algorithm $\mathcal{V}$, which takes a ciphertext C_τ as input. At the end of interaction, the prover algorithm $\mathcal{P}$ gets output $y_0 \in \{\tau, \perp\}$ and the verifier algorithm $\mathcal{V}$ gets output (y_1, y_2) where $y_1 \in \{\texttt{Accept}, \texttt{Reject}\}$ and $y_2 \in \{h(C_F), \perp\}$.

Definition 1 (Correctness) *We say a $\mathcal{CSD}$ scheme $(\mathcal{E}, \mathcal{D}, \mathcal{P}, \mathcal{V})$ is* correct*, if the following conditions hold with overwhelming high probability (i.e. $1 - negl$): For any data file $F \in \{0,1\}^*$ and any positive integer λ, and $(\tau, C_\tau, C_F) := \mathcal{E}(F, 1^\lambda)$,*

- $\mathcal{D}(\tau, C_F) = F$.
- $\langle \mathcal{P}(F), \mathcal{V}(C_\tau) \rangle = (\tau;\ \texttt{Accept}, h(C_F))$.

Here the hash value $h(C_F)$ is required, in order to defend poison attack. In case that the cloud does not have plaintext of file F, the cloud storage server *alone* is not able to decide whether a given tuple $(h(F), C_\tau, C_F)$ is consistent or inconsistent (i.e. poisoned).

2.3 Client-side Deduplication: Security Definition

Informally, in a $\mathcal{CSD}$ security game, the adversary first plays the role of verifier and talks to owners of file F to learn information, and then plays the role of prover, attempting to convince an honest verifier to accept the adversary as an owner of F, or try to find the value of F.

The $\mathcal{CSD}$ security game $\mathsf{G}_{\mathcal{A}}^{\mathcal{CSD}}(\lambda, \zeta, \xi)$ between a PPT adversary $\mathcal{A}$ and a challenger w.r.t. $\mathcal{CSD}$ scheme $(\mathcal{E}, \mathcal{D}, \mathcal{P}, \mathcal{V})$ is defined as below.

Commit. The description of $(\mathcal{E}, \mathcal{D}, \mathcal{P}, \mathcal{V})$ is made public. Let F be any file from the domain $\{0,1\}^*$, which has min-entropy $\geq \zeta$ to the view fo the adversary.
Setup. The challenger runs the encoding algorithm to obtain $(\tau, C_\tau, C_F) := \mathcal{E}(F, 1^\lambda)$, where C_τ contains a hash value $h(F)$ of F as a part. The challenger sends C_F to the adversary.
Learning. The adversary $\mathcal{A}$ can adaptively make queries to the challenger, where each query is in one of the following forms:

- Verify-Query: The challenger, running the prover algorithm $\mathcal{P}$ with input F, interacts with adversary $\mathcal{A}$ which replaces the verifier algorithm $\mathcal{V}$, to obtain $(y_0; y_1, y_2) := \langle \mathcal{P}(F), \mathcal{A} \rangle$. The adversary knows the values of y_1 and y_2, and can make polynomially many queries in this type.
- Prove-Query: The challenger, running the verifier algorithm $\mathcal{V}$ with input C_τ, interacts with the adversary $\mathcal{A}$ which replaces the prover algorithm $\mathcal{P}$, to obtain $(y_0; y_1, y_2) := \langle \mathcal{A}, \mathcal{V}(C_\tau) \rangle$. The adversary knows the value of y_0 and can make polynomially many queries in this type.
- Leak-Query(TM, x): The leak query (TM, x) consists of a description of PPT turing machine TM and a string $x \in \{0,1\}^*$. The challenger responses this query by computing $y := \mathsf{TM}(F, x)$ and sending y to the adversary. The adversary can make polynomially many queries in this type, with a constraint that sum of bit-lengths of all responses y's is not greater than ξ. Therefore, at most ξ bits information about file F will be leaked to the adversary via this Leak-Query.

Attack. The challenger sends C_τ to the adversary.
Note: This is to capture the case that the adversary is the honest-but-curious cloud server.

- **Guess**: The adversary $\mathcal{A}$ outputs a guess F' and wins if $F' = F$.
- **Forge**: The challenger, running the verifier algorithm $\mathcal{V}$ with input C_τ, and interacts with the adversary $\mathcal{A}$, which replaces the prover algorithm $\mathcal{P}$, to obtain result $(y_0'; y_1', y_2') := \langle \mathcal{A}, \mathcal{V}(C_\tau) \rangle$, where $y_1' \in \{\texttt{Accept}, \texttt{Reject}\}$. The adversary wins if $y_1' = \texttt{Accept}$.

The adversary in the above security game can capture both third party adversary and honest-but-curious cloud storage server. We give the following remarks on the above security game:

- We allow the adversary to make VERIFY-QUERY in the above security game, to capture the case that (1) a third party adversary impersonates the cloud storage server (e.g. phishing attack); or (2) the adversary is the honest-but-curious cloud storage server.
- We allow the adversary to obtain the ciphertext C_F of users' data file F in the above security game, for the following reasons. (1) Data file F could be very large, so is the ciphertext C_F. It is more difficult to protect large amount of data than short random secret keys. (2) In real world examples, Dropbox [2] stores users' data file in Amazon S3 server and Wuala [3] stored users' data files in a P2P network in the early stage of the company; (3) the adversary could be the honest-but-curious cloud storage server.
- We allow the adversary to know the ciphertext C_τ of the encryption key at the beginning of **Attack** phase and after the **Learning** phase, because (1) the adversary could be the honest-but-curious cloud storage server, who has information of C_τ; (2) all LEAK-QUERYs have to be made before the adversary knows the value C_τ, otherwise, the adversary can obtain the encryption key τ by making a LEAK-QUERY$(\mathsf{TM}_{\mathcal{P},\mathcal{V}}, C_\tau)$, where the turing machine $\mathsf{TM}_{\mathcal{P},\mathcal{V}}(F, C_\tau) = \langle \mathcal{P}(F), \mathcal{V}(C_\tau) \rangle$. Therefore, no secure $\mathcal{CSD}$ scheme exists, if the adversary is given the ciphertext C_τ by the challenger before he/she can make a LEAK-QUERY.
- If the ciphertext C_F is obtained by encrypting data file F using the convergent encryption [11,12], i.e. encrypting the data file F under AES method with the hash value $h(F)$ as encryption key, then the adversary will have both ciphertext C_F and decryption key $h(F)$ and can obtain the file F, where
  - C_F is given by the challenger in the security game.
  - $h(F)$ can be obtained by making a LEAK-QUERY$(\mathsf{TM}_h, \mathtt{NULL})$ where turing machine $\mathsf{TM}_h(F) = h(F)$. Therefore, convergent encryption is insecure in our security model.

Definition 2 *Let integer λ be the security parameter, $\zeta - \xi \geq \lambda$, and $\delta \in [0,1]$. We say a $\mathcal{CSD}$ system $(\mathcal{E}, \mathcal{D}, \mathcal{P}, \mathcal{V})$ is $(\lambda, \zeta, \xi, \delta)$-Secure, if for any PPT adversary $\mathcal{A}$, the following conditions hold: Let F' and y'_1 be as in the* **Guess** *phase and* **Forge** *phase in game $\mathsf{G}_{\mathcal{A}}^{\mathcal{CSD}}(\lambda, \zeta, \xi)$,*

- *(ζ, ξ, δ)-privacy-preserving:* $\mathsf{Pr}[F' = F] \leq \delta$.
- *(ζ, ξ, δ)-unforgeable:* $\mathsf{Pr}[y'_1 = \mathtt{Accept}] \leq \delta$.

3 Privacy-Preserving PoW

Before constructing a client side deduplication scheme, we will construct a "privacy preserving PoW", which will serve as a subroutine in our main construction in the next section.

3.1 Definitions

Halevi *et al.* [6] proposed the formulation of proofs of ownership. In this section, we briefly review their definitions and formulate the notion of *privacy-preserving proofs of ownership*. Readers can find more details in Halevi *et al.* [6].

Definition 3 (Proofs of Ownership [6]) *A proof of ownership scheme consists of a probabilistic algorithm S and a pair of interactive algorithm $\langle \mathsf{P}, \mathsf{V} \rangle$, where are described as below:*

- *$\mathsf{S}(F, 1^\lambda) \to \pi$: The randomized summary function S takes a file F and the security parameter λ as input, and outputs a short summary value π where the size of π is in $\mathcal{O}(\lambda)$.*
- *$\langle \mathsf{P}(F, 1^\lambda), \mathsf{V}(\pi) \rangle \to \{\mathtt{Accept}, \mathtt{Reject}\}$: The prover algorithm P which takes as input a file F and the security parameter λ, interacts with the verifier algorithm V which takes as input a short summary value π, and outputs* `Accept` *or* `Reject`.

We point out, the efficiency requirement excludes some straightforward secure methods, for example, both prover and verifier have access to the file F and compute a key-ed hash value over F with a randomly chosen hash key.

Definition 4 (Correctness of PoW [6]) *A PoW scheme* $(\mathsf{S}, \langle \mathsf{P}, \mathsf{V} \rangle)$ *is correct, if for all positive integer* λ *and for any file* $F \in \{0,1\}^{poly(\lambda)}$*, it holds that*

$$\langle \mathsf{P}(F, 1^\lambda), \mathsf{V}(\mathsf{S}(F, 1^\lambda)) \rightarrow \texttt{Accept}.$$

Definition 5 (Security of PoW, Definition 2 in [6]) *Let* $\mathbb{N}$ *denote the set of all positive integers. Fix parameters* $s, T \in \mathbb{N}, \epsilon \in [0,1]$*. A scheme is a proof of ownership with leakage threshold* T*, slackness* s*, and soundness error* ϵ*, if for any* $t \in \mathbb{N}$*, any input distribution* $\mathcal{D}$ *with* t *bits of min entropy, and any PPT adversary that receives less than* $\min\{T - s, t - s\}$ *bits from the accomplice who has the file of interest, the verifier rejects the proof with all but negligibly more than* ϵ *probability.*

3.2 Construction of Privacy-Preserving PoW

Halevi *et al.* [6] proposed a framework to construct PoW scheme: Given a file F of large size, apply a pairwise independent hash function H on the file to generate a hash value of L bits. Then apply an error erasure code (e.g. Reed-Solomon code) on the T bits hash value to obtain, say $2L$ bits encoded value V. At last, the standard Merkle Hash Tree based PoW scheme is applied over the $2L$ bits value V. Intuitively, any efficient adversary who knows at most $(L - \lambda)$ bits information of the file F will not be able to present the value of a randomly chosen leaf node in the Merkle Hash Tree over V with probability larger than $1/2$.

Halevi *et al.* [6] proved in Theorem 2 that if the hash function H is pairwise independent, then the above construction is a secure PoW under Definition 5.

In this paper, we will propose a PoW following the above framework, with the below differences:

- We use our custom designed pairwise independent hash function with large output size. The construction will be presented in Table 1.
- We present an alternative way to avoid the use of expensive error correcting code. Instead of generating a hash value of size L bits and encoding into a value of size $2L$ bits using error erasure coding scheme, we directly generate a hash value of size $2L$ bits. The alternative way will achieve the same effect: An adversary with at most $(T - \lambda)$ bits information about the input file of size $\geq 2T$ bits will fail to present a randomly chosen Merkle Hash Tree leaf with probability at least $1/2$.

3.3 Construction of Hash Function of Output Size

In this subsection, we construct a hash function H with large output size (say $\rho\ell$ bits), using an underlying hash function h with small output size (say $\rho\ell$ bits). We expect the constructed hash function H can behave like a pairwise independent hash if the underlying hash function h is 4ℓ-independent. The details of the construction is in Table 1. Intuitively and not identically, our construction can be described in the following way:

1. The input is a file F. Assume the bit-length of F is, say ℓm, a multiple of ℓ. Treat F as a sequence of ℓ number of m-bits-words.
2. Find all of ℓ prefixes[5] of the sequence F of m-bits-words, and hash each prefix using hash function h from the shortest prefix to the longest prefix. Concatenate all of ℓ hash values to produce a string Y of $\rho\ell$ bits.

[5] Here we do not consider the empty string as a prefix or suffix.

3. Reverse the order of m-bits-words in the sequence F to produce $\overline{F}$ Find all of ℓ prefixes of the sequence $\overline{F}$ of m-bits-words, and hash each prefix using hash function h from the shortest prefix to the longest prefix. Concatenate all of ℓ hash values to produce a string Z of $\rho\ell$ bits.
4. Reverse the order of bits in the string Z to obtain $\overline{Z}$. The final output is $Y \oplus \overline{Z}$, where $\oplus$ denotes the XOR operator.

Our design follows this guideline: *If a single bit in the input file is changed, then every block of ρ bits in the output will be changed with high probability*[6]. Suppose the complexity of computation of $h(F)$ is in $\mathcal{O}(|F|)$. Then it is straightforward to find that the complexity of our construction is in $\mathcal{O}(\ell|F|)$, which could be very slow for large file F. We realize that the efficiency of our construction can be greatly improved if the underlying hash function h is instantiated as `SHA256` function. Due to the underlying Merkle-Damgård construction of `SHA256` hash function, computation of $\texttt{SHA256}(x\|y)$ can be computed in time $\mathcal{O}(|y|)$ if some short intermediate result in computation of $\texttt{SHA256}(x)$ is available. Thus, hashing all of ℓ prefixes of file F can be almost as efficient as hashing the file F for once. Therefore, the complexity of our construction is improved into $\mathcal{O}(|F| + \ell)$.

Lemma 1 *Let $\mathsf{H}_k : \{0,1\}^{\geq \rho\ell} \to \{0,1\}^{\rho\ell}$ be the hash function constructed in Table 1 using the underlying hash function $h_k : \{0,1\}^* \to \{0,1\}^{\rho}$. Given an input file $F \in \{0,1\}^{|F|}$ with bit-length $|F|$, we have the following conclusions on the complexity of $\mathcal{H}_k(F)$:*

- *Suppose the complexity of computation of $h_k(F)$ is in $\mathcal{O}(|F|)$. Then the complexity of $\mathcal{H}_k(F)$ is in $\mathcal{O}(|F| \times \ell)$.*
- *Suppose $h_k(x) = \texttt{SHA256}(k\|x)$ and $\rho = 256$. Then the hash function H_k constructed in Table 1 can be computed in time $\mathcal{O}(|F| + \ell)$.*

Alternatively, $h_k(x)$ can be defined as $h_k(x) = \texttt{SHA512}(k\|x)$ and $\rho = 512$. Gueron *et al.* [13] reported that, in 64 bit system, `SHA512` is much faster than `SHA256`.

Theorem 2 *Let $\mathsf{H}_k : \{0,1\}^{\geq \rho\ell} \to \{0,1\}^{\rho\ell}$ be the hash function constructed in Table 1 using the underlying hash function $h_k : \{0,1\}^* \to \{0,1\}^{\rho}$. If h is 4ℓ-independent hash, then H is 2-independent (i.e. pairwise independent).*

Notice that the assumption that h is 4ℓ-independent, is weaker than the assumption that h is a random oracle. The proof of the above Theorem 2 is given in Appendix A.

For completeness, we repeat the Theorem 2 of Halevi *et al.* [6] as below.

Theorem 3 (Theorem 2 in [6]) *A protocol where the input file is first hashed to an L-bit value using pairwise-independent hash function and then we run the Merkle Hash Tree protocol on the resulting L-bit value is a proof-of-ownership as per Definition 5, with leakage threshold $L \cdot (\frac{1}{3} - \frac{1}{2\rho})$. Here ρ is the block size.*

From Theorem 2 and Theorem 3, we have the following conclusion:

Corollary 4 *A protocol where the input file is first hashed to an $\rho\ell$-bit value using pairwise-independent hash function $\mathcal{H}$ constructed in Table 1, and then we run the Merkle Hash Tree protocol on the resulting $\rho\ell$-bit value without error erasure encoding, is a proof-of-ownership as per Definition 5, with leakage threshold $\frac{1}{2}\rho\ell \cdot (\frac{1}{3} - \frac{1}{2\rho})$. Here ρ is the block size.*

Theorem 5 *If the hash function h is replaced by a random oracle, then the protocol described in Corollary 4 is a* Zero Knowledge Proof.

[6] In contrast, the construction of hash function of large output size in Table 1 in PoW [6], if a single bit in the input file is changed, then about 15/16 fraction of blocks in the output will be changed with high probability.

Table 1. Construction of a keyed-hash function H with output size equal to $\rho\ell$ bits, using an underlying keyed-hash function h with output size equal to ρ bits.

Input: The input is a file $F \in \{0,1\}^{|F|}$ with bit-length $|F| \geq \rho\ell$, where integers ρ and ℓ are system parameters.

1. Define a subroutine $\mathcal{H}_k$ as below:
   (a) The input is a bit-string $w_1 \| w_2 \| \dots \| w_\ell$, where each substring w_i has equal bit-length, and "$\|$" denotes the string concatenation operator.
   (b) For each $i \in [1, \ell]$, compute $u_i := h_k(w_1 \| w_2 \| \dots \| w_i) \in \{0,1\}^\rho$.
   Note: If $h_k(w) = \texttt{SHA256}(k \| w)$, then the computation of u_{i+1} can be done in time $\mathcal{O}(|w_{i+1}|)$ by taking advantage of the intermediate result in computation of u_i, due to the underlying Merkle-Damgård construction of hash function SHA256*. Consequently, the computation of ℓ hash values $u_1, \dots, u_\ell$ is essentially as efficient as the computation of a single hash value u_ℓ.*
   (c) The output is $u_1 \| u_2 \| \dots \| u_\ell \in \{0,1\}^{\rho\ell}$.
2. Let $\texttt{Len} \in \{0,1\}^{64}$ be the 64 bits big endian integer representation of the bit-length of file F. Pad file F into F^* as below:
$$F^* = \text{`0'} \| \texttt{Len} \| F \| \text{`1'} \| \underbrace{\text{`0'} \dots \text{`0'}}_{R \text{ number of `0'}} \| \texttt{Len} \| \text{`1'},$$
   where R is the smallest nonnegative integer such that the bit-length of F^* is a multiple of ℓ, i.e. $1 + 64 + |F| + 1 + R + 64 + 1 \mod \ell = 0$.
   Note:
   – *If $\ell = 512$, then $F \| \text{`1'} \| \underbrace{\text{`0'} \dots \text{`0'}}_{R \text{ number of `0'}} \| \texttt{Len}$ is identical to the padding method in* SHA256 *algorithm.*
   – *We pad the extra bit '0' at the beginning and the extra bit '1' at the end of F^*, so that F^* is definitely* not *a palindrome.*
   – *We pad length* Len *in F^* twice, so that for any two files F_0 and F_1 with different lengths, their corresponding padded files F_0^* and F_1^* will be different in both the first 65 bits and the last 65 bits.*
3. Compute $y_1 \| y_2 \| \dots \| y_\ell := \mathcal{H}_k(F^*)$, where each $y_i \in \{0,1\}^\rho$.
4. Compute $z_1 \| z_2 \| \dots \| z_\ell := \mathcal{H}_k(\overline{F^*})$, where each $z_i \in \{0,1\}^\rho$ and $\overline{F^*}$ is the bit-string obtained by reversing the order of bits in F^*.

Output: The output is
$$\Big(y_1 \| y_2 \| \dots \| y_\ell\Big) \oplus \Big(z_\ell \| \dots \| z_2 \| z_1\Big),$$
where $\oplus$ denotes the XOR operator.
Note: The above construction is conceptually simple, but reversing the order of all bits in a file could be expressive. A more practical and equally secure construction is as below: Everything is the same except the computation of z_i's. Break F^* into ℓ equal length substring x_i's, that is, $F^* = x_1 \| x_2 \| \dots \| x_{\ell-1} \| x_\ell$, where all bit-string x_i's have equal lengths ≥ 65. Reverse the order of bits in x_ℓ to produce the bit-string $\overline{x_\ell}$. Compute $z_1 \| z_2 \| \dots \| z_\ell := \mathcal{H}_k(\overline{x_\ell} \| x_{\ell-1} \| \dots \| x_2 \| x_1)$, where each $z_i \in \{0,1\}^\rho$. That is, instead of reversing the order of bits in file F^*, we just reverse the order of blocks x_i's in F^* and reverse the order of bits in a single block x_ℓ.

Proof (sketch proof of Theorem 5). In the protocol described in Corollary 4, all information of file F that the verifier can obtain is $(k_1, h_{k_1}(F), k_2, \mathsf{H}_{k_2}(F)$, where k_1 and k_2 are independent random hash keys. If $h_k(\cdot)$ is replaced by a random oracle, both hash values $h_{k_1}(F)$ and $\mathsf{H}_{k_2}(F)$ will be uniformly random variables independent on k_1 and k_2. Therefore, a simulator can choose four independent random variables (r_0, r_1, r_2, r_3), such that $|r_0| = |k_1|, |r_1| = |h_{k_1}(F)|, |r_2| = |k_2|$ and $|r_3| = |\mathsf{H}_{k_2}(F)|$, and simulate the interaction between the prover and the verifier. Since (r_0, r_1, r_2, r_3) is indistinguishable from $(k_1, h_{k_1}(F), k_2, \mathsf{H}_{k_2}(F)$, the simulated interaction is indistinguishable from the real case. □

4 Main Construction

Let $\mathsf{PoW} = (\mathsf{S}, \langle \mathsf{P}, \mathsf{V} \rangle)$ be the privacy-preserving proof of ownership scheme constructed in previous Section 3. Let $\mathsf{E} = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ be a deterministic symmetric encryption scheme with ρ bits long key length (e.g. AES-256[7]). Let h be a full domain secure hash function with output size ρ bits (e.g $\mathtt{SHA256}$) and $\{H_s\}$ be a keyed-hash family (e.g. $H_s(x) = \mathtt{SHA256}(s\|x)$). Let $\oplus$ denote the XOR operator between two equal length bit-strings. We give the main construction of client side deduplication scheme $(\mathcal{E}, \mathcal{D}, \langle \mathcal{P}, \mathcal{V} \rangle)$ as below.

$\boldsymbol{\mathcal{E}(F, 1^\lambda)}$	$\boldsymbol{\mathcal{D}(\tau, C_F)}$
1. $\tau := \mathsf{E.KeyGen}(1^\lambda) \in \{0,1\}^\lambda$.	1. $F' := \mathsf{E.Dec}_\tau(C_F)$
2. $s \xleftarrow{\$} \{0,1\}^\lambda$.	2. Output F'.
3. $C_F := \mathsf{E.Enc}_\tau(F)$.	
4. $\pi := \mathsf{PoW.S}(F, 1^\lambda)$.	
5. $C_\tau := (s, H_s(F) \oplus \tau, \pi, h(F))$.	
6. Output (τ, C_τ, C_F).	

$\langle \boldsymbol{\mathcal{P}(F')}, \ \boldsymbol{\mathcal{V}(C_\tau)} \rangle$

V1: Parse C_τ as $(s, H_s(F) \oplus \tau, \pi, h(F))$.

V↔P: Run the Privacy-Preserving PoW scheme between the prover, who executes algorithm $\mathsf{PoW.P}(F')$, and the verifier, who executes algorithm $\mathsf{PoW.V}(\pi)$. If the verifier rejects the prover at the end of PoW scheme, then abort and output $(\perp; \mathtt{Reject}, \perp)$.

V2: Send $(s, H_s(F) \oplus \tau)$ to the prover.

P1: Compute the secret key τ' as below

$$\tau' := H_s(F') \ \oplus \ \big(H_s(F) \oplus \tau\big).$$

Encrypt[8] F' with key τ' to generate ciphertext $C_{F'}$ and compute the hash value $h(C_{F'})$ of the ciphertext. Send this hash value $h(C_{F'})$ to verifier.

At the end of this procedure, the prover gets output $y_0 := \tau'$ and the verifier gets output $y'_1 := \mathtt{Accept}$ and $y'_2 := h(C_{F'})$.

We remark that in the application of the above client side deduplication scheme, some owner of file F runs the encoding scheme $\mathcal{E}$ to choose a random encryption key τ, and the verifier (i.e. the cloud storage server) helps to distribute the encryption key τ to all owners of file F in a robust and secure way, such that (1) any owner of file F is able to obtain the encryption key τ efficiently from the verifier; (2) an adversary with limit amount of information of F is not able to obtain τ from the verifier; (3) the verifier does not know the secret key τ.

[7] AES encryption with random initial value can be used, as long as the random initial value apears as a part of the ciphertext.

[8] As mentioned in the overview in Section 1, this encryption is required to compute the hash $h(C_{F'})$, which will help the verifier (i.e. cloud storage server) to detect poison attack.

Theorem 6 (The Main Construction is Secure) *Suppose the PoW scheme constructed in Section 3 is secure with threshold T. Suppose hash functions h_s and H_s are random oracles, and encryption scheme $\mathsf{E} = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ is (ϵ, δ)-secure such that given the tuple $(h(F), \mathsf{H}(F), \mathsf{E.Enc}_k(F), M)$ where $F \in \{0,1\}^M$ has at least ϵ min-entropy and k is a randomly chosen encryption key, any PPT adversary cannot find the value of F with probability larger than $\delta \in [0,1]$. Then the client side deduplciation scheme $(\mathcal{E}, \mathcal{D}, \langle \mathcal{P}, \mathcal{V} \rangle)$ constructed above is (ζ, ξ, δ)-privacy-preserving and $(\zeta, \xi - \lambda, \delta)$-unforgeable, where $\xi = T$.*

5 Conclusion

In this paper, we addressed two important security concerns in cross-user client side deduplication of encrypted files in the cloud storage: privacy against both outside attackers and the honest-but-curious cloud storage server, and poison attack in which an outside attacker might be able to corrupt users files via the client side deduplicaiton service.

On technique aspect, we constructed a new hash function and proposed a novel key distribution method to distribute per-file encryption key to each owner of the file. We designed a novel efficient hash function with large output size from `SHA256`. The new hash function is provably pairwise independent, if `SHA256` is replaced by a random oracle. By incorporating this new hash function with the generic framework of PoW construction by Halevi *et al.* [6], we obtained a new PoW which only reveals at most two hash values about the input file during the process of proof. It is worthy to point out that, we gave an alternative design to avoid the use of expensive error correcting code. Our new key distribution method together with AES encryption method can replace traditional convergent encryption to support deduplicaiton of encrypted files, with better security property, especially when some information of the files are leaked.

Design of provable PoW scheme in the standard model remains an open problem.

References

1. Xu, J., Chang, E.C., Zhou, J.: Secure cloud storage with encrypted data using file-based authentication. Cryptology ePrint Archive, Report 2011/538 (2011) `http://eprint.iacr.org/`.
2. Dropbox: Dropbox `http://www.dropbox.com/`.
3. Wuala: Wuala `http://www.wuala.com/`.
4. Wikipedia: Comparison of online backup services `http://en.wikipedia.org/wiki/Comparison_of_online_backup_services`.
5. SNIA: Understanding Data De-duplication Ratios. white paper `http://www.google.com/url?q=http://www.snia.org/sites/default/files/Understanding_Data_Deduplication_Ratios-20080718.pdf&sa=U&ei=aGFUT8ndGK7KmQX6sK2lCg&ved=0CAQQFjAA&client=internal-uds-cse&usg=AFQjCNEq7nm_nMWRbOkNnghVwew9MlgWlA`.
6. Halevi, S., Harnik, D., Pinkas, B., Shulman-Peleg, A.: Proofs of ownership in remote storage systems. In: CCS '11: ACM conference on Computer and communications security. (2011) 491–500
7. Dropship: Dropbox api utilities (April 2011) `https://github.com/driverdan/dropship`.
8. Harnik D., Pinkas B., S.P.A.: Side Channels in Cloud Services: Deduplication in Cloud Storage. IEEE Security and Privacy Magazine, special issue of Cloud Security **8**(6) (2010)
9. : Dropbox left user accounts unlocked for 4 hours sunday `http://www.wired.com/threatlevel/2011/06/dropbox/`.
10. Twitter: Tweetdeck `http://money.cnn.com/2012/03/30/technology/tweetdeck-bug-twitter/`.
11. Douceur, J.R., Douceur, J.R., Adya, A., Adya, A., Bolosky, W.J., Bolosky, W.J., Simon, D., Simon, D., Theimer, M., Theimer, M.: Reclaiming space from duplicate files in a serverless distributed file system. In: In Proceedings of 22nd International Conference on Distributed Computing Systems (ICDCS). (2002)
12. Douceur, J.R., Bolosky, W.J., Theimer, M.M.: Encryption systems and methods for identifying and coalescing identical objects encrypted with different keys (09 2007)
13. Shay Gueron, Simon Johnson, J.W.: Sha-512/256. Cryptology ePrint Archive, Report 2010/548 (2010) `http://eprint.iacr.org/`.

A Proof of Theorem 2

By definition, a hash function $\mathsf{H}_k : \{0,1\}^{\geq \rho\ell} \rightarrow \{0,1\}^{\rho\ell}$ is pairwise independent, if for any two distinct inputs $F_0 \neq F_1$ in the domain of H_k, for any two values a and b in the range of H_k,

$$\mathsf{Pr}_k\left[\mathsf{H}_k(F_0) = a \wedge \mathsf{H}_k(F_1) = b\right] = \frac{1}{2^{2\rho\ell}},$$

where the probability is over randomly chosen hash key k.

For any bit-string x, let us denote with $\overline{x}$ the bit-string obtained by reserving the order of bits in x. It is not difficult ot drive the following claim from the description of H in Table 1.

Claim 1 *Let F be a file of size $\geq \rho\ell$ bits and F^* be the padded version of F. Break F^* into ℓ equal length bit-strings w_i's, i.e. $F^* = w_1 \| \ldots \| w_\ell$. Let y_i, z_i, $i \in [1,\ell]$, as in Table 1. We have*

$$y_i = h_k(w_1 \| w_2 \| \ldots \| w_i); \tag{1}$$
$$z_i = h_k(\overline{w_\ell} \| \overline{w_{\ell-1}} \| \ldots \| \overline{w_{\ell-i+1}}); \tag{2}$$
$$z_{\ell-i+1} = h_k(\overline{w_\ell} \| \overline{w_{\ell-1}} \| \ldots \| \overline{w_i}). \tag{3}$$

The above claim implies that every substring $w_j, j \in [1,\ell]$, is involved in the computation of y_i or $z_{\ell-i+1}$, for each $i \in [1,\ell]$. Therefore, if a single bit in the input of H is changed, then each block $y_i \oplus z_{\ell-i+1}$ in the output of H will be influenced.

Claim 2 *Let $F_0, F_1 \in \{0,1\}^{\geq \rho\ell}$ be two files of different lengths. For any two values a and b in the range of H_k,*

$$\mathsf{Pr}_k\left[\mathsf{H}_k(F_0) = a \wedge \mathsf{H}_k(F_1) = b\right] = \frac{1}{2^{2\rho\ell}},$$

where the probability is over randomly chosen hash key k.

Proof (Proof of Claim 2). Since the underlying hash function h_k is 4ℓ-independent, the above Claim 2 can be implied directly by this statement: "*If files F_0 and F_1 have different lengths, then in the computations of $\mathsf{H}_k(F_0)$ and $\mathsf{H}_k(F_1)$, every invocation of the hash function h_k has distinct input.*" Let $w_i^{(\iota)}, y_i^{(\iota)}$ and $z_i^{(\iota)}$ denote the corresponding variables w.r.t. file F_ι, $\iota \in \{0,1\}$. The above statement can be proved straightforwardly from the below facts:

- For any $\iota \in \{0,1\}$ and $i \neq j \in [1,\ell]$, $y_i^{(\iota)}$ and $y_j^{(\iota)}$ are computed from inputs of different lengths under hash function h_k, during the computation of $\mathsf{H}_k(F_\iota)$. Similar statement holds for $z_i^{(\iota)}$'s.
- For any $\iota \in \{0,1\}$ and $i,j \in [1,\ell]$, $y_i^{(\iota)}$ (respectively, $z_i^{(\iota)}$) is computed from an input with first 65 bits equal to " '0' $\|\mathtt{Len}_0$" (respectively, " '1' $\|\overline{\mathtt{Len}_1}$") under hash function h_k, during the computation of $\mathsf{H}_k(F_\iota)$, where $\mathtt{Len}_0$ (respectively, $\mathtt{Len}_1$) is the 64 bit integer representation of bit-length of file F_0 (respectively, F_1), and $\mathtt{Len}_0 \neq \mathtt{Len}_1$.

□

Claim 3 *Let $F_0, F_1 \in \{0,1\}^{\geq \rho\ell}$ be two files of equal length. For any two values a and b in the range of H_k,*

$$\mathsf{Pr}_k\left[\mathsf{H}_k(F_0) = a \wedge \mathsf{H}_k(F_1) = b\right] = \frac{1}{2^{2\rho\ell}},$$

where the probability is over randomly chosen hash key k.

Proof (of Claim 3). Pairwisely comparing all inputs to the underlying hash function h_k, the bit-length or the initial bit can distinguish all of them with two *exceptions*: (1) inputs associated to $y_i^{(0)}$ and $y_i^{(1)}$, $i \in [1,\ell]$; (2) inputs associated to $z_i^{(0)}$ and $z_i^{(1)}$, $i \in [1,\ell]$.

By proof of contradiction, we suppose that $\exists i \in [1, \ell]$, the inputs associated to $y_i^{(0)}$ and $y_i^{(1)}$ are equal, and the inputs associated to $z_{\ell-i+1}^{(0)}$ and $z_{\ell-i+1}^{(1)}$ are equal. Applying Claim 1, we can conclude that $F_0 = F_1$, which is a **contradiction!**

Define the state variable Δ as the set of all inputs which are fed into the hash function h_k so far. Initially, Δ is empty set. Every time when hash function h_k is called, new input will be added into the set Δ if any. We look at the computation of hash function $\mathsf{H}_k(F_0)$ and $\mathsf{H}_k(F_1)$ in this order: (1) in first phase, at step $i = 1, 2, \ldots, \ell$, compute the i-th block $y_i^{(0)} \oplus z_{\ell-i+1}^{(0)}$ of $\mathsf{H}_k(F_0)$; (2) in the second phase, at step $i = 1, 2, \ldots, \ell$, compute the i-th block $y_i^{(1)} \oplus z_{\ell-i+1}^{(1)}$ of $\mathsf{H}_k(F_1)$. Note that each step, the underlying hash function h_k is called for once, and the above steps include all 4ℓ calls to function h_k.

It is not difficult ot draw the following conclusions:

- In the first phase, just before each step $i \in [1, \ell]$, both the input associated to $y_i^{(0)}$ and input associated to $z_{\ell-i+1}^{(0)}$, are not inside the current set Δ. Since h_k is 4ℓ-independent, the block $y_i^{(0)} \oplus z_{\ell-i+1}^{(0)}$ will be a random variable w.r.t. hash key k, independent on all blocks computed in previous steps.
- In the second phase, just before each step $i \in [1, \ell]$, either the input associated to $y_i^{(1)}$ or the input associated to $z_{\ell-i+1}^{(1)}$, is not inside the current set Δ. Since h_k is 4ℓ-independent, the block $y_i^{(1)} \oplus z_{\ell-i+1}^{(1)}$ will be a random variable w.r.t. hash key k, independent on all blocks computed in previous steps in both phases.

Therefore, for *any* fixed files F_0 and F_1 with equal length, $\mathsf{H}_k(F_0)$ and $\mathsf{H}_k(F_1)$ are two independent random variables when the hash key k is chosen randomly from its domain. □