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

Jia Xu
Institute for Infocomm Research
1 Fusionopolis Way
Singapore, 138632
xuj@i2r.a-star.edu.sg

Ee-Chien Chang
National University of Singapore
21 Lower Kent Ridge Road
Singapore, 119077
changec@comp.nus.edu.sg

Jianying Zhou
Institute for Infocomm Research
1 Fusionopolis Way
Singapore, 138632
jyzhou@i2r.a-star.edu.sg

Abstract

Recently, Halevi *et al.* (CCS '11) proposed a cryptographic primitive called *proofs of ownership* (PoW) to enhance security of client-side deduplication in cloud storage. 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, in the *bounded leakage* setting where a certain amount of efficiently-extractable information about file F is leaked. Following this work, we propose a secure client-side deduplication scheme, with the following advantages:

- our scheme protects data confidentiality (and *some* partial information) against both outside adversaries and honest-but-curious cloud storage server, while Halevi *et al.* trusts cloud storage server in data confidentiality;
- our scheme is proved secure w.r.t. any distribution with sufficient min-entropy, while Halevi *et al.* (the last and the most practical construction) is particular to a specific type of distribution (a generalization of "block-fixing" distribution) of input files.

The cost of our improvements is that we adopt a weaker leakage setting: We allow a bounded amount *one-time* leakage of a target file *before* our scheme starts to execute, while Halevi *et al.* allows a bounded amount *multi-time* leakage of the target file *before and after* their scheme starts to execute. To the best of our knowledge, previous works on client-side deduplication prior Halevi *et al.* do not consider any leakage setting.

[*] Funded by A*Star project SecDC-112172014.

Categories and Subject Descriptors H.3.5 [*Information Systems*]: Information storge and retrieval—On-line information services; E.3 [*Data Encryption*]

General Terms Algorithms, Security

Keywords Cloud Storage, Client-side Deduplication, Proofs of Ownership, Privacy, Leakage-Resilient, Universal Hash

1. Introduction

Cloud storage service is gaining popularity in recent years. To reduce resource consumption in both network bandwidth and storage, many cloud storage services including Dropbox [1] and Wuala [2] employs client-side deduplication [21, 39]. 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 already in the cloud storage. In this way, every single file will have only one copy in the cloud (i.e. Single Instance Storage). SNIA white paper [34] reported that the deduplication technique can save up to 90% storage, dependent on applications.

According to Halevi *et al.* [20] and Dropship [17], an existing implementation of client-side deduplication is as below: Cloud user Alice tries to upload a file F to the cloud storage. The client software of the cloud storage service installed on Alice's computer, will compute and send the hash value $\mathsf{hash}(F)$ to the cloud server. The cloud server maintains a database of hash values of all received files, and looks up the value $\mathsf{hash}(F)$ in this database. If there is no match found, then file F is not in the cloud storage yet. Alice's client software will be required to upload F to the cloud storage, and the hash value $\mathsf{hash}(F)$ will be added into the look-up database. If there is a match found, then file F is already in the 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 the cloud storage is saved, and the cloud server will allow Alice to

[1] http://www.dropbox.com/

[2] http://www.wuala.com/

access the file F in its cloud storage. We may refer to the above client-side deduplication method as "hash-as-a-proof" method. In this method, the hash value $\mathsf{hash}(F)$ serves two purposes: (1) it is an index of file F, used by the cloud server to locate information of F among a huge number of files; (2) it is treated as a "proof" that Alice owns file F. Previously, Dropbox[3] applied the above "hash-as-a-proof" method on block-level cross-users deduplication [20][17]. If the client software of the cloud storage service is trusted and the hash function $\mathsf{hash}(\cdot)$ is collision-resistant, then the "hash-as-a-proof" method is sufficiently secure. However, malicious users may develop their own version of client software using public API[4] of the cloud service, so that they can send any manipulated messages (e.g. manipulated hash output) to the cloud server. Therefore, a more sophisticated solution without trusting the client software is required.

1.1 Security Concerns

Different users may possess some identical sensitive files for many reasons, even if they have no knowledge on each other. For example they may receive a classified or copyright-protected file directly or indirectly from the same source. Partial information of these sensitive files could be leaked via various channels [20, 21] by some owners intentionally or unintentionally. Despite its significant benefits in saving resource, client-side deduplication may bring in new security vulnerability and lead to leakage of users' sensitive files, especially when a certain amount of partial information of these files have already been leaked.

1.1.1 Data Privacy against Outside Adversaries

Recently, an attack on "hash-as-a-proof" method in popular cloud storage service like Dropbox and MozyHome is proposed [17, 20]: If the adversary *somehow* has the short hash value of a file stored in the cloud storage, he/she 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 via 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 could be bypassed.

1.1.2 Data Privacy against Inside Adversaries (Cloud Storage Servers)

Confidentiality of users' sensitive data against the cloud storage server itself is another important security concern that is not addressed by Halevi *et al.* [20]. 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 accounts unlocked for almost 4 hours, due to a new bug in their software [40]. 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 of failure of privacy protection of all users' data, even if Dropbox made such mistakes or was hacked in. Very recently, a bug in Twitter's client software is discovered [37], which allows adversary to access users' private data.

It is worth noting that, cloud storage service providers, including Amazon (S3), Apple (iCloud), Dropbox, Google (Drive) and Microsoft (SkyDrive), explicitly or implicitly declare that they reserve rights to access users' files, in their official statements of privacy policy [2, 10, 16, 19, 22, 25].

1.1.3 Divide and Conquer Attack

Let us consider an example: A classified document consists of many pages. Although the whole document has sufficient min-entropy to the view of adversaries, the first page has very low min-entropy, say 1 bit min-entropy which indicates "Acceptance" or "Rejection". Suppose this classified document is stored in a cloud storage, which supports block-level cross-user deduplication. Then the adversary could recover the 1 bit unknown information in the first page, through the block-level deduplication[5]. This is because: (1) deduplication inevitably provides adversaries a way to do brute force search for unknown information, and (2) block-level deduplication that divides a file into blocks and applies deduplication on each block, will isolate min-entropy of each block, and allow adversaries to do brute force search in a much smaller search space. It is not unusual that a file with high min-entropy contains some part, which has very low min-entropy compared to its bit-length. Deterministic encryption scheme also need resolve this issue [26]. We emphasize that block-level cross-user client-side deduplication should not be applied over sensitive files.

Very recently, Ng *et al.* [28] proposed a scheme, which aimed to support PoW over encrypted data. However, their work [28] did not address the above issue and consequently is secure only if every block of the file of interest has sufficient min-entropy. We will discuss this work later in the related work section.

1.1.4 Poison Attack

When a file F is encrypted on client side with randomly chosen encryption key, the cloud server may not be able to verify consistency between the ciphertext and meta-data of file F

[3] In Feb 2012, we noticed that Dropbox disabled the deduplication across different users, probably due to recent 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.

[4] Dropbox provides public API. Furthermore, this issue can not be eliminated just by hidding API, since the adversary could perform reverse-engineering attack to guess the communication protocol of the cloud service. Note the effect of obfuscating is limited [6].

[5] Users can find whether deduplication occurs by timing the uploading time or monitoring communication packet between the cloud client software and cloud server, or develop a custom cloud client software using public API. Here we assume each block contains only one page.

uploaded by a user. For example, given a tuple (H_F, C_F), the cloud server is unable to verify whether there exists a file F and an encryption key k such that $H_F = \mathsf{hash}(F)$ and $C_F = \mathsf{Enc}_k(F)$. A malicious user may substitute the valid ciphertext C_F with an equal size poisoned file before uploading it to the cloud. Suppose a subsequent user Carol tries to upload the same file F to the cloud. She will be told that F is already in cloud and uploading of F is saved. She may delete her local copy of F to save local storage, and will retrieve file F from the cloud when required in the future. However, what she can retrieve from the cloud is a poisoned file—her file F is lost! This attack is also known as *Target Collision attack* [36]. We remark that this attack does not require to find a collision of the underlying hash function, and could be practical if the client-side deduplication over encrypted data is not properly implemented. For example, the "hash-as-a-proof" method suffers from such poison attack, if the cloud server does not verify the hash computation in order to save computation burden.

1.1.5 Plausible Approaches

Convergent Encryption. Intuitively, convergent encryption [14, 15] together with PoW might provide a solution for client-side deduplication of encrypted files: Encrypt file F to generate ciphertext C_F with hash value $\mathsf{hash}(F)$ as encryption key and then apply PoW scheme over C_F. Indeed, cloud storage service provider Wuala adopts convergent encryption to encrypt users files on client side and supports cross-user deduplication. However, the threat models of PoW and convergent encryption are incompatible. In the setting of PoW where a bounded amount of efficiently-extractable information about the file F can be leaked, convergent encryption is insecure, since its short encryption key is generated from the input file in a deterministic way and could be leaked. Roughly speaking, convergent encryption is as insecure as "hash-as-a-proof" method (i.e using hash value $\mathsf{hash}(F)$ as a proof of ownership of file F), in the presence of leakage. Therefore, all existing works on applying convergent encryption method to implement deduplication of encrypted data (e.g. [3, 24, 36]) are insecure in the bounded leakage setting.

Per-User Encryption Key. Another approach is that each cloud user chooses his/her own per-user encryption key, and all files uploaded to the cloud by the same user will be deterministically 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 interested in the deduplication cross different users, that is, identical duplicated files from different uses will be detected and removed safely.

1.1.6 Current states of various Cloud Storage Services

We collect some technique information about various cloud storage services (Dropbox, SpiderOak [6] and Wuala) in Table 1. All information comes from public official blogs, white papers, private communication with these cloud storage service providers, or through simple experiments with their public service. We notice that Microsoft SkyDrive and Google Drive do not provide client-side deduplication function, even within a single user account. We conjecture that all cloud storage service with simple web access support (i.e. without requiring special browser plug-in) either do not encrypt users' data or encrypt users' data on server side only.

Table 1. Comparison of various cloud storage services.

Name	Deduplication	Cross-User	Encryption
Dropbox	Yes	No (See footnote 1)	Server side enc
SpiderOak	Yes	No [35]	Client side enc
Wuala	Yes	Yes	Convergent enc

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 uploads a sensitive file F to the cloud storage. She will independently choose a random AES key τ, and produces two ciphertexts as below (See Figure 1): The first ciphertext C_F is generated by encrypting file F with encryption key τ using AES method, and the size of C_F is almost equal to the size of F; 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*, and the size of C_τ is in $\mathcal{O}(|\tau|)$ which is very small.

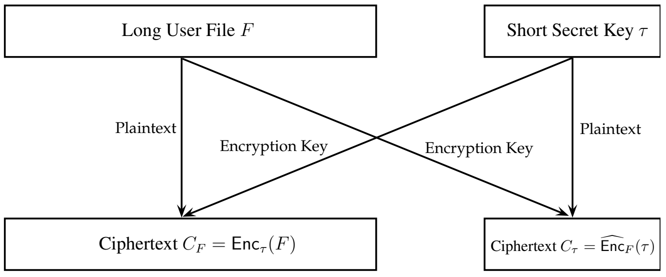
Long User File F
Short Secret Key τ
Plaintext
Encryption Key
Encryption Key
Plaintext
Ciphertext $C_F = \mathsf{Enc}_\tau(F)$
Ciphertext $C_\tau = \widehat{\mathsf{Enc}}_F(\tau)$

Figure 1. The generation of large ciphertext C_F and short ciphertext C_τ.

Finally, Alice will send a hash value $\mathsf{hash}(F)$ and two ciphertexts (C_F, C_τ) to the cloud storage server. The cloud storage server will compute the hash value $\mathsf{hash}(C_F)$, insert a *short* entry $(\mathtt{key} = \mathsf{hash}(F);\ \mathtt{value} = (\mathsf{hash}(C_F), C_\tau))$ into its lookup database, and store the potentially *large* ciphertext C_F separately.

[6] https://spideroak.com/

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 $\mathsf{hash}(F)$ to the cloud storage server, and the cloud storage server finds a matched entry in its lookup database: $(\mathtt{key} = \mathsf{hash}(F);\ \mathtt{value} = (\mathsf{hash}(C_F), C_\tau))$. Next, the cloud storage server will send the short ciphertext C_τ to Carol. Carol can decrypt C_τ using file F as decryption key and obtain the secret AES key τ. Carol can encrypt her file F with AES key τ to generate C_F and send the newly computed hash value $\mathsf{hash}(C_F)$ to the cloud storage server. The cloud will compare Carol's version of hash value $\mathsf{hash}(C_F)$ with the one computed by itself. If the two hash values are equal, then Carol is allowed to download C_F from the cloud storage from now on. If the two hash values are different, then with overwhelming high probability[7], either Alice has launched a poison attack w.r.t. file F, or Carol is cheating, or both. If Alice is honest, she can recover file F from the cloud, and present file F as a proof; if Carol is honest, she can present her local copy of F as a proof.

After this, assuming that both Alice and Carol are honest, Carol may remove the local copy of file F if she likes and keeps the AES key τ safely in local storage. Carol can always recover file F by downloading the ciphertext C_F from the cloud and decrypting it with key τ.

1.2.2 Our Contributions

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

- In Section 3, we propose a formulation for client-side deduplication of encrypted files. Our formulation protects confidentiality of users' sensitive files against both malicious outside adversaries and honest-but-curious inside adversaries. Furthermore, our formulation also protects an important type of partial information (particularly, any physical bit $F[i]$ at position i of file F) of users' sensitive files, although the nature of deduplication implies that semantic-security is unachievable. In contrast, the recent work by Ng *et al.* [28] does not protect partial information and suffers from divide and conquer attack.
- In Section 4, we propose the first secure (Definition 2) client-side deduplication scheme of encrypted files in the bounded leakage setting. We prove its security against malicious outside adversaries and honest-but-curious inside adversaries and with respect to *any* distribution of user file in Theorem 1. In contrast, the PoW schemes by Halevi *et al.* [20] only deals with outside adversaries, and their most practical construction is only proved secure against a particular type of distribution of user file and their proof (in random oracle) relies on an untypical assumption (More details are given later in Section 2).

[7] Except the negligibly rare case that a collision of the hash function is found.

Our scheme can be applied for deduplication of sensitive files that have very low min-entropy due to a one-time leakage. It is worth pointing out our leakage setting is weaker than Halevi *et al.* [20], so that our scheme achieves a weaker goal in this aspect and allows exposure of the whole file by leaking a short string (rather than account id/password) in the strong leakage setting of Halevi *et al.* This weakness has been resolved in our full paper [42].

The next Section 2 briefs the background and discusses related works. Experiment result is reported in Section 5. Section 6 concludes this paper.

2. Related works

2.1 Pairwise-Independent Hash Family

A hash family $\{\mathsf{H}_k : \mathbb{M} \to \{0,1\}^L\}$ is *pairwise-independent* (or say *universal hash* [8, 38]), if for any two distinct inputs $x_1, x_2 \in \mathbb{M}$, $\mathsf{Pr}_k[\mathsf{H}_k(x_1) = \mathsf{H}_k(x_2)] = 2^{-L}$.

2.2 Works on Secure Deduplication before PoW

Deduplication of encrypted files have been studied since the design of convergent encryption by Douceur *et al.* [14]. To the best of our knowledge, all existing works (e.g. [3, 14, 24, 36]) before Halevi *et al.* [20] do not consider leakage setting and most of them focus on server-side deduplication.

2.3 Proofs of Ownership

To prevent private data leakage to outside adversary, Halevi *et al.* [20] proposed a notion of "proofs of ownership" (PoW). In a PoW scheme, any owner of a file F, without necessarily knowing other owners of F, can efficiently prove to the cloud storage server that he/she owns the file F; any outside adversary cannot prove that he/she has the file F with probability larger than a predefined threshold, even if a certain amount of efficiently-extractable information of file F is leaked to the adversary. Such leakage may occur at any time except during the course of interactive proof between the cloud storage server and the cloud user.

Halevi *et al.* [20] proposed three constructions. The first construction encodes a file using some error erasure code, and then applies the standard Merkle Hash Tree proof method over the encoded file. The second construction is a generic framework. Let $\mathsf{H}_k : \{0,1\}^M \to \{0,1\}^L$ be *any* pairwise independent hash family. Given a file F of M bits long, the second construction computes the hash value $\mathsf{H}_k(F)$ with a public randomness k as hash key and applies the standard Merkle Hash Tree proof method over the L bits value $\mathsf{H}_k(F)$. The third construction is the most practical one. It designs an efficient hash family $\mathsf{H}'_k : \{0,1\}^M \to \{0,1\}^L$ and applies the standard Merkle Hash Tree proof method over $\mathsf{H}'_k(F)$. However, their construction of H'_k is not pairwise-independent (even if in the random oracle model). Consequently, the generic framework in the second construction cannot apply and a new security proof is required. As the authors explicitly mentioned, Halevi *et al.* [20]'s security proof for their third construction has some limitations: (1) the proof assumes that the file F is sampled from a *particular* type of distribution (a generalization of "block-fixing distribution"); (2) the proof is given "*under the unproven assumption that their scheme will generate a good code*" (See text around Theorem 3 in their paper [20]); (3) the proof is given in random oracle model, where SHA256 is treated as a random function.

2.4 Extremely Efficient but Less Secure "PoW"

Very recently, Pietro and Sorniotti [32] proposed an efficient "PoW" scheme: They use the projection of the file F onto K [8] randomly selected bit-position $i_1, \ldots, i_K$ as the "proof" of ownership of the file F, that is, the knowledge of bit-string $F[i_1]\| \ldots \|F[i_K]$ is a "proof" of ownership of file F.

This scheme is extremely efficient. However, this work [32] has at least these limitations: (1) it does not protect privacy against honest-but-curious cloud storage server; (2) it is secure only if the min-entropy of file F to the view of adversaries is close to the bit-length of file F, after the leakage occurs.

2.5 Existing Plausible Attempt for Privacy-Preserving PoW

Recently, Ng *et al.* [28] made an attempt to support PoW over encrypted files. Their method encrypted files on client side and shared the encryption key among a group of users who know each other. Their method applies existing scheme [12] to do key management within the group, and focus on formulating and devising proofs of ownership scheme in a privacy preserving manner.

Here we brief their PoW scheme as below: A file is divided into many blocks x_i's, and a commitment c_i is computed from each x_i under a secret key. Then the standard Merkle Hash Tree method applies over the commitments $(c_1, c_2, \ldots)$. After the completion of Merkle Hash Tree proof protocol, the verifier knows some commitment value c_i, and the prover has to show that he/she has the knowledge of some secret value x_i whose commitment is c_i, without revealing information on x_i to the verifier.

We observe that their proof of knowledge of x_i against c_i is similar to the generalized Okamoto-Identification scheme [30], given by Alwen *et al.* [1]. This proof of identification scheme allows the verifier to efficiently decide whether the secret value x_i is equal to any given candidate value x, thus allows brute-force search of x.

In summary, Ng *et al.* [28]'s PoW scheme has the following limitations: (1) it is very slow in computation: in every execution of the proof protocol, to generate all commitment values c_i's, $|F|/1024$ number of exponentiations[9] in a modulo group of size $\approx 2^{1024}$ are required where $|F|$ denotes the bit-length of file F; (2) the encryption key is shared among a group of "friends", which is not suitable for client-side deduplication over encrypted files, since in current typical client-side duplication setting, owners of the same file will be anonymous to each other; (3) it suffers from "divide and conquer attack" mentioned previously in Section 1.1.3, although (i) it is provably privacy-preserving under their formulation [28] and (ii) it is applied in file-level instead of block-level.

In an extreme example, a large file F with $L(\geq \lambda)$ bits min-entropy to the view of the curious cloud server is divided into L blocks x_i's where each x_i has exactly 1 bit min-entropy to the view of the curious cloud server. If Ng *et al.* [28]'s method is applied over such a file F, then the honest-but-curious cloud server could learn everything of the file efficiently in time $\mathcal{O}(L)$ instead of $\mathcal{O}(2^L)$ during the proof process, by brute-force searching of the value of each x_i independently. In contrast, if the proposed scheme in this paper applies over the same file F, any efficient curious cloud server or outside adversary cannot recover the file and cannot obtain the bit value $F[i]$ at any bit-position i in file F, assuming $F[i]$ was unknown to adversaries before the execution of the proof protocol.

Another recent work [43] combines proofs of storage (i.e POR [9, 23, 33, 41] and PDP [4]) with proofs of ownership. However, this work [43] fails to fulfill the efficiency requirement on server side given by Halevi *et al.* [20], not to mention privacy protection of user data against the curious server.

It is worth noting that, after our work (the full version [42] of this paper), a very recent independent work by Bellare *et al.* [7] contains a scheme called "Randomized Convergent Encryption" (RCE), which has encryption part essentially identical to the encryption part of our scheme. However, Bellare *et al.* does not consider any leakage setting.

3. Security Model

In this section, we propose a security formulation for client-side deduplication of encrypted files.

3.1 System Model and Trust Model

3.1.1 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 bandwidth, etc) of Cloud Storage Server is much stronger than a single average

[8] K is a system parameter. In their experiment [32], K takes values in the range $[100, 2000]$.

[9] One such group exponentiation requires more than 3 milliseconds in a model PC, which means that it requires more than 3000 seconds to generate all commitments for a file of size 1 giga-bits. Such expensive operation will be executed every time when a user tries to upload the same file to the cloud.

user, the average computation power per each online user is usually very limited. We assume that the small and fast primary storage is well-protected from outside adversaries, and the large but slow secondary storage could be visible to outside adversaries.

An example of cloud storage server is Dropbox. Users' files uploaded to Dropbox are actually stored in Amazon's S3 data center (i.e. Dropbox's secondary storage) and Dropbox only runs relatively small server to manage meta data (i.e. Dropbox's primary storage). Another cloud storage service provider Wuala stored users' files in P2P network (the secondary storage) in the early stage of the company.

3.1.2 Cloud Users

Many cloud users may upload their files to the cloud storage and possibly remove their local copies. These users may download files, which are uploaded by themselves, from cloud storage. File sharing among users is not the focus of this paper, although it can be achieved along with our solution for encrypted data.

3.1.3 Adversaries

We consider two types of adversaries: Malicious outside adversary and honest-but-curious Cloud server.

Malicious Outside Adversary. The outside adversary may obtain some knowledge (e.g. a hash value) of the file of interest via some channels, and plays a role of cloud user to interact with the cloud server.

Semi-honest Inside Adversary (Honest-but-Curious Cloud Server). This honest but curious cloud storage server (also known as inside adversary) 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.
2. The company, which provide the cloud storage service, made careless technical mistakes which may leak users' private data, e.g. introducing a software bug. It is reported that Dropbox [40] 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 [37].
3. The cloud storage server is hacked in.

3.2 Syntax Definition

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

- $\mathcal{E}(F, 1^\lambda) \rightarrow (\tau, C_0, C_1)$: The probabilistic encoding algorithm $\mathcal{E}$ takes as input a data file F and a security parameter λ, and outputs a short secret per-file encryption key τ, a short encoding C_0 which contains $\mathsf{hash}(F)$ as a part, and a long encoding C_1. C_0 will be stored in cloud server's small and secure primary storage and C_1 will be stored in cloud server's large but potentially insecure secondary storage. The lengths of τ and C_0 should be both in $\mathcal{O}(\lambda)$, and the length of C_1 should be in $|F| + \mathcal{O}(poly(\lambda))$.
- $\mathcal{D}(\tau, C_1) \rightarrow F$: The deterministic decoding algorithm takes as input a secret key τ and the long encoding C_1, and outputs a file F.
- $\langle \mathcal{P}(F), \mathcal{V}(C_0) \rangle \rightarrow (y_0;\ y_1, y_2)$: The prover algorithm $\mathcal{P}$, which takes a file F as input, interacts with the verifier algorithm $\mathcal{V}$, which takes a short encoding C_0 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 $y_2 \in \{\mathsf{hash}(C_1), \perp\}$.

We point out, the efficiency requirement in the above description follows Halevi *et al.* [20]. Such efficiency requirement excludes some straightforward secure methods: For example, both prover and verifier have access to the file F during the interactive proof and compute a key-ed hash value over F with a randomly chosen fresh nonce as hash key per each proof session. Another negative example is Zheng *et al.* [43] which turns a proof of storage scheme into a proof of ownership scheme.

Definition 1 (Correctness)**.** *We say a $\mathcal{CSD}$ scheme $(\mathcal{E}, \mathcal{D}, \langle \mathcal{P}, \mathcal{V} \rangle)$ is* correct*, if the following conditions hold with overwhelming high probability (i.e. $1 - negl(\lambda)$): For any data file $F \in \{0,1\}^*$ and any positive integer λ, and $(\tau, C_0, C_1) := \mathcal{E}(F, 1^\lambda)$,*

- *$\mathcal{D}(\tau, C_1) = F$.*
- *$\langle \mathcal{P}(F), \mathcal{V}(C_0) \rangle = (\tau;\ \texttt{Accept}, \mathsf{hash}(C_1))$.*

Here the hash value $\mathsf{hash}(C_1)$ 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 $(\mathsf{hash}(F), C_0, C_1)$ is consistent or inconsistent (i.e. poisoned).

3.3 Security Definition

In this subsection, we will propose a security formulation for client-side deduplication. Our formulation will address the protection of *useful* partial information (particularly any physical bit $F[i]$ in the sensitive file F) from the malicious outside adversary or the honest-but-curious cloud server: Roughly speaking, PPT outside/inside adversary cannot learn any *new* information on any physical bit $F[i]$ of file F from client-side deduplication process beyond the side channel leakage.

The $\mathcal{CSD}$ security game $\mathsf{G}_{\mathcal{A}}^{\mathcal{CSD}}(\xi_0, \xi_1)$ between a PPT adversary $\mathcal{A}$ and a challenger w.r.t. $\mathcal{CSD}$ scheme $(\mathcal{E}, \mathcal{D}, \langle \mathcal{P}, \mathcal{V} \rangle)$ is defined as below, where $\xi_0 > \xi_1 \geq \lambda$. Here ξ_0 is the lower bound of min-entropy of the challenged file F at the be-

ginning of the game, and the adversary is allowed to learn at most $(\xi_0 - \xi_1)$ bits information of file F from the challenger.
Setup. The description of $(\mathcal{E}, \mathcal{D}, \langle \mathcal{P}, \mathcal{V} \rangle)$ is made public. Let F be sampled from any distribution over $\{0,1\}^M$ with min-entropy $\geq \xi_0$, where the public integer parameter $M \geq \xi_0$ is polynomially bounded in λ. The challenger sends $\mathsf{hash}(F)$ to the adversary $\mathcal{A}$.
Learning-I. The adversary $\mathcal{A}$ can make a LEAK-QUERY in the following form to the challenger:

- LEAK-QUERY(Func): This query consists of a PPT-computable function Func. The challenger responses this query by computing $y := \mathsf{Func}(F)$ and sending y to the adversary, where the bit-length of output y is required to be smaller than $(\xi_0 - \xi_1)$.

Commit. The adversary $\mathcal{A}$ chooses a subset of v indices $i_1, \ldots, i_v$ from $[1, |F|]$, where $v \geq 1$ and $v + |y| \leq \xi_0 - \xi_1$. The challenger finds the subsequence $\alpha \in \{0,1\}^v$ of F, such that, for each $j \in [1, v], \alpha[j] = F[i_j]$. The challenger chooses a random bit $b \in \{0,1\}$ and sets $\alpha_b := \alpha$ and $\alpha_{1-b} \xleftarrow{\$} \{0,1\}^v$. The challenger sends (α_0, α_1) to the adversary $\mathcal{A}$.

Guess-I. Let $\mathsf{View}_{\mathcal{A}}^{\mathtt{Commit}}$ denote the view of the adversary $\mathcal{A}$ at this moment. Given $\mathsf{View}_{\mathcal{A}}^{\mathtt{Commit}}$ as input, another PPT algorithm (called "extractor") $\mathcal{A}^*$ outputs a guess $b_{\mathcal{A}^*} \in \{0,1\}$ of value b.

Learning-II. The adversary $\mathcal{A}$ can adaptively make queries to the challenger, where concurrent queries are not allowed[10] and each query is in one of the following forms:

- ENCODE-QUERY: The challenger responses the ENCODE-QUERY by running the probabilistic encoding algorithm on F to generate $(\tau, C_0, C_1) := \mathcal{E}(F, 1^\lambda)$ and sending (C_0, C_1) to the adversary. The adversary can make exactly one query in this type.
- 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_0, interacts with the adversary $\mathcal{A}$ which replaces the prover algorithm $\mathcal{P}$, to obtain $(y_0; y_1, y_2) := \langle \mathcal{A}, \mathcal{V}(C_0) \rangle$. The adversary $\mathcal{A}$ knows the value of y_0, and can make polynomially many queries in this type.

Guess-II. The adversary $\mathcal{A}$ outputs a guess $b_{\mathcal{A}} \in \{0,1\}$ of value b.

Definition 2 (Secure $\mathcal{CSD}$ against inside/outside attack)**.** *Let integer λ be the security parameter and $\xi_0 > \xi_1 \geq \lambda$. We say a $\mathcal{CSD}$ scheme $(\mathcal{E}, \mathcal{D}, \langle \mathcal{P}, \mathcal{V} \rangle)$ is (ξ_0, ξ_1)-secure, if for any PPT (inside or outside) adversary $\mathcal{A}$, there exists some PPT extractor algorithm $\mathcal{A}^*$, such that in the security game $\mathsf{G}_{\mathcal{A}}^{\mathcal{CSD}}(\xi_0, \xi_1)$,*

$$\begin{aligned} &\Pr[\mathcal{A} \textit{ finds } b \textit{ in } \textbf{\textit{Guess-II}} \textit{ phase}] \\ \leq &\Pr[\mathcal{A}^* \textit{ finds } b \textit{ in } \textbf{\textit{Guess-I}} \textit{ phase}] + negl(\lambda). \end{aligned} \quad (1)$$

Equivalently, the above Equation (1) *can be written as*

$$\Pr[b_{\mathcal{A}} = b] \leq \Pr[b_{\mathcal{A}^*} = b] + negl(\lambda). \quad (2)$$

Remarks on the security formulation.

- Our formulation (particularly, Equation (1) and (2) in Definition 2) requires that $\Pr[b_{\mathcal{A}} = b] \leq \Pr[b_{\mathcal{A}^*} = b] + negl(\lambda)$, which means the adversary $\mathcal{A}$ essentially cannot learn any *new* information on physical bits $F[i_1] \ldots F[i_v]$ in file F during **Learning-II** phase. We emphasize that it is important to ask *some* extractor $\mathcal{A}^*$ instead of the adversary $\mathcal{A}$ to make a guess $b_{\mathcal{A}^*}$ before **Learning-II**, to exclude a trivial plausible attack: Adversary $\mathcal{A}$ intentionally outputs a random guess of b before **Learning-II**, and outputs its maximum-likelihood of b after **Learning-II**, in order to increase the difference between success probability in **Guess-I** and **Guess-II**. Note that this requirement follows the style of original definition of semantic security (Definition 5.2.1 in Goldreich [18]).
- The adversary is allowed to obtain the long encoding C_1 of users' data file F in the above security game, since in real applications, C_1 is typically stored in the large but potentially insecure secondary storage, as mentioned in Section 3.1.1.
- A $\mathcal{CSD}$ scheme does not have any master secret key. Therefore, the adversary $\mathcal{A}$ himself/herself can find answers to any queries (i.e. ENCODE-QUERY, VERIFY-QUERY, PROVE-QUERY, and LEAK-QUERY, etc) w.r.t any input file F' that is owned by $\mathcal{A}$, without help of the challenger.
- If the long encoding C_1 is obtained by encrypting file F using the convergent encryption [14, 15], i.e. encrypting the file F under AES method with some hash value $\mathsf{hash}'(F)$ as encryption key, then the adversary (i.e. the curious cloud server) will have both ciphertext C_1 and decryption key $\mathsf{hash}'(F)$, and thus obtain the file F, where
  - the ciphertext C_1 is given by the challenger in the security game;

[10] Similar to Halevi *et al.* [20], concurrent PROVE-QUERY and VERIFY-QUERY will allow the adversary to replay messages back and forth between these two queries, and eliminate the possibility of any secure and efficient solution to client-side deduplication. Therefore, both this work and Halevi *et al.* [20] do not allow concurrent queries of different types in the security formulation. We clarify that, concurrent queries of the same type can be supported. Thus, in the real application, the cloud storage server (verifier) can safely interact with multiple cloud users (prover) w.r.t. the same file concurrently.

- the convergent encryption key $\mathsf{hash}'(F)$ can be obtained by making a LEAK-QUERY in **Learning-I** phase.

Therefore, convergent encryption is insecure in our security game due to the bounded leakage setting. One may argue that this was not a fair setting for convergent encryption scheme, since leakage of encryption key will render any encryption scheme insecure. However, if the encryption key is chosen independent on the plaintext (i.e. the file F), then LEAK-QUERY in our setting will be unable to reveal the encryption key. Thus the distinctive feature of convergent encryption—deriving encryption key from plaintext—becomes a two-bladed-sword: on one side, it makes deduplication of encrypted data possible; on the other side, it becomes the security vulnerability in the higher level of security formulation. We will give a more detailed comparison between convergent encryption and our approach later in Section 4.3.

- For similar reasons as above, the "hash-as-a-proof" method is also insecure in our leakage setting. Therefore, when the owner of file F does not understand well what information of F has been leaked by other owners of the same file, our scheme will be much more preferable than the efficient "hash-as-a-proof" method.

3.4 Comparison of two formulations: Proofs of Ownership and Client-side Deduplication

In this subsection, we compare the formulation of PoW [20] and our formulation of $\mathcal{CSD}$. Recall that more details on PoW is given in Section 2.

Both PoW [20] and $\mathcal{CSD}$ formulation aim to secure deduplication mechanism in cloud storage service and protect the confidentiality of users' data in the setting that a bounded amount of information of users' data has been leaked. However, the two formulations differ in both breadth and depth of protection of confidentiality of users' data and also differ in the leakage setting:

- **Breadth**: PoW only formulates the protection of confidentiality of users' data from outside adversaries; $\mathcal{CSD}$ formulation protects the confidentiality of users' data from both outside adversaries and inside adversaries (i.e. the honest-but-curious cloud storage server). In other words, PoW formulation trusts the cloud storage server in data confidentiality, but $\mathcal{CSD}$ formulation doesn't.
- **Depth**: PoW formulation only prevents attackers from recovering the whole user file F and *potentially* allows attackers to recover some partial unknown information of F; $\mathcal{CSD}$ gives a stronger formulation and prevents attackers from recovering any unknown bits $F[i]$, which implies that the adversary cannot recover F. Here, we make two clarifications: (1) The nature of deduplication problem inevitably provides adversaries a way to do brute search for unknown information by observing whether client-side deduplication occurs[11], and thus make semantic security impossible. Therefore, any solution to client-side deduplication cannot achieve semantic security and has to reveal some partial information. Our formulation will protect an important type of partial information: the physical bits $F[i]$'s in file F. (2) Although PoW formulation does not address protection of partial information, the constructions in Halevi *et al.* [20] indeed protects the privacy of physical bits $F[i]$'s against outside adversaries (Of course, this is not true for inside adversaries).
- **Leakage**: PoW allows a stronger leakage setting than $\mathcal{CSD}$. PoW allows *multi-time* leakage at any time except during the course of the interactive proof between the verifier and prover, while $\mathcal{CSD}$ allows only *one-time* leakage before $\mathcal{CSD}$ scheme starts to execute. Putting the PoW formulation in our context, the security game of PoW consists of multiple interleaved executions of **Learning-I** and **Learning-II** phases, i.e. (**Learning-I**, **Learning-II**, **Learning-I**, **Learning-II**, ...), subject to an additional constraint that the adversary cannot make the LEAK-QUERY and PROVE-QUERY concurrently. On the other side, our $\mathcal{CSD}$ allows larger amount of leakage than the PoW formulation (precisely Definition 2 and 3 in PoW [20]).

4. Secure Client-side Deduplication

4.1 Construction

We present the construction of a $\mathcal{CSD}$ scheme $(\mathcal{E}, \mathcal{D}, \langle\mathcal{P}, \mathcal{V}\rangle)$ in Figure 2 on page 9. Since this construction relies on universal hash function to achieve leakage-resilience, we call the constructed $\mathcal{CSD}$ scheme as UH-CSD. Suppose Alice is the first user who uploads file F. She will execute algorithm $\mathcal{E}$ with file F and security parameter 1^λ as input and obtain a short secret encryption key τ, a short encoding $C_\tau \in \{0,1\}^{3\lambda}$ and a long encoding C_F. Alice will send both C_τ and C_F to the cloud storage server Bob. Bob will compute the hash value $\mathsf{hash}(C_F)$, put C_τ in secure and small primary storage, and put C_F in the potentially insecure but large secondary storage. At the last, Bob will add $(\mathtt{key} = \mathsf{hash}(F), \mathtt{value} = (\mathsf{hash}(C_F), C_\tau)$ into his lookup database. Suppose Carol is another user who tries to upload the same file F after Alice. Carol will send $\mathsf{hash}(F)$ to the cloud storage server Bob. Bob finds that $\mathsf{hash}(F)$ is already in his lookup database. Then Bob, who is running algorithm $\mathcal{V}$ with C_τ as input, interacts with Carol, who is running algorithm $\mathcal{P}$ with F as input. At the end of interaction, Carol will learn τ and Bob will compare the hash value $\mathsf{hash}(C_F)$ provided by Carol with the one computed by himself. Later, Carol is allowed to download C_F from Bob at any

[11] Probabilistic deduplication that saves the duplicated copy only with a certain probability (say 0.5) still allows brute force attack, since the success probability of deduplication can be amplified close to 1 via repetition.

time and decrypt it to obtain the file F by running algorithm $\mathcal{D}(\tau, C_F)$.

Figure 2. The construction of a $\mathcal{CSD}$ scheme, denoted as UH-CSD. Let $\mathsf{E} = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ be a symmetric encryption scheme with λ $(= \rho)$ bits long key length and $\mathsf{h}_k : \{0,1\}^* \to \{0,1\}^\rho$ be a key-ed hash function. Notice that the random coin of Enc will be put in the generated ciphertext.

```
E(F, 1^λ)
1. $\tau := \mathsf{KeyGen}(1^\lambda) \in \{0,1\}^\lambda$.
2. $s \xleftarrow{\$} \{0,1\}^\lambda$.
3. $C_F := \mathsf{Enc}_\tau(F)$.
4. $C_\tau := (s, \mathsf{h}_s(F) \oplus \tau, \mathsf{hash}(F))$.
5. Output $(\tau, C_\tau, C_F)$.

D(τ, C_F)
1. $F' := \mathsf{Dec}_\tau(C_F)$
2. Output $F'$.
```

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

V1: Parse C_τ as $(s, \mathsf{h}_s(F) \oplus \tau, \mathsf{hash}(F))$. Send $(s, \mathsf{h}_s(F) \oplus \tau)$ to the prover.

P1: Compute the secret key y_0 as below

$$y_0 := \mathsf{h}_s(F') \ \oplus\ \big(\mathsf{h}_s(F) \oplus \tau\big), \text{ where } \oplus \text{ refers to XOR.}$$

Encrypt[a] F' with key y_0 to generate ciphertext $C_{F'}$ and compute the hash value $y_2 := \mathsf{hash}(C_{F'})$ of the ciphertext. Send y_2 to verifier.

V2: Let $H_{C_F} := \mathsf{hash}(C_F)$ be computed previously when receiving ciphertext C_F. If $y_2 = H_{C_F}$, set $y_1 := \texttt{accept}$, otherwise $y_1 := \texttt{reject}$.

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

4.2 Security Analysis

Theorem 1. *Let $\xi_0 > \xi_1 = 2\lambda + \Omega(\lambda)$. Let* hash *be a collision-resistant full domain hash function. Suppose the encryption scheme* E *is private-key semantic secure (Definition 5.2.1 in Goldreich [18]) and the hash function family $\{\mathsf{h}_k\}$ is a universal hash. Then the* UH-CSD *scheme in Figure 2 is (ξ_0, ξ_1)-secure. (Proof is in Appendix A)*

The proof of correctness of UH-CSD scheme under Definition 1 is straightforward. We save the details. Notice that the *leakage rate* (i.e. the ratio of the amount of leakage to the entropy of the sensitive file) of our scheme UH-CSD is $1 - \xi_1/\xi_0$, which could be close to 1 for suitable ξ_0 and ξ_1.

4.3 Comparison with Convergent Encryption

4.3.1 Our custom encryption method is derived from convergent encryption

Convergent encryption encrypts a file F as $\mathsf{Enc}_{\mathsf{hash}(F)}(F)$. In our scheme UH-CSD given in Figure 2, the encryption of file F is $(s, \mathsf{h}_s(F) \oplus \tau, \mathsf{Enc}_\tau(F))$. This encryption method can be treated as a natural extension of convergent encryption which overcomes the below shortcomings of convergent encryption.

Revocation of encryption key. It is very difficult, if not impossible, to revoke the encryption key of convergent encryption, when the current encryption key is compromised. Suppose a user tries to encrypt file F using $\mathsf{hash}(F)$ as AES encryption key, and finds that the value of $\mathsf{hash}(F)$ has already been revealed to Internet by some other owner of file F. He may switch to use $\mathsf{hash}'(F)$ as encryption key where $\mathsf{hash}'(\cdot)$ is another secure hash function. Meanwhile, the user has to broadcast this switch of hash function to all future users. This approach will face two issues: (1) The number of different secure hash function is very limited. (2) Users may abuse the above hash-revoking functionality. A natural fixes to the above two issues are: (1) Use a secure key-ed hash function and revoke the hash key if necessary. (2) It is not necessary that every user adopts the same hash function (i.e. the same keyed-hash function and hash key) to generate the AES encryption key. Every user can independently choose a new hash key without notifying others. As a result, a user can encrypt a file F in this way: Randomly choose a hash key s and generate the ciphertext $(s, \mathsf{AES}_{\mathsf{h}_s(F)}(F))$.

Can any hash value be a valid encryption key? It is a coincidence that the range of hash function (e.g. `SHA256`) is consistent with the key space of encryption method (e.g. AES). Many other encryption schemes have special key generating algorithm and the generated key should have a particular structure, for example, some public key encryption schemes. Therefore, convergent encryption cannot generalize to generic encryption scheme. Our proposed encryption method overcomes this weakness, by invoking the key generating algorithm of the underlying encryption method to generate an encryption key and protect this generated encryption key using a one-time pad. Let $(\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ be the underlying encryption method. The ciphertext of F will be $(s, \mathsf{h}_s(F) \oplus \tau, \mathsf{Enc}_\tau(F))$, where the hash key s is randomly generated and the underlying encryption key τ is generated by algorithm KeyGen.

Leakage Resilient. More importantly, convergent encryption is insecure if a bounded amount of efficiently-extractable information of the plaintext F is leaked *before* encryption (i.e. the leakage setting of $\mathcal{CSD}$ in this paper). Our encryption method is resilient to such bounded leakage of the plaintext F, in the random oracle (assuming h is a random oracle) or in the standard model (assuming h is pairwise-independent hash function). It is worth pointing out that no encryption schemes could be secure in the stronger leakage setting model (e.g. POW [20]) where leakage may occur *before* and *after* the encryption process.

4.3.2 Advantage of Convergent Encryption

Convergent encryption can be used for both client-side and server-side deduplication. In contrast, our encryption method can be used only for client-side deduplication, since the one round interaction in the client-side deduplication is essential for our solution to synchronize the hash key. Unsurprisingly, both convergent encryption and our custom encryption method are not semantically secure [18].

5. Performance

We have implemented a prototype of the proposed scheme UH-CSD with SHA256 as the collision-resistant full domain hash function $\mathsf{hash}(\cdot)$, and with $\mathtt{SHA256}(k\|x)$ as the keyed-hash[12] $\mathsf{h}_k(x)$, and AES encryption[13] as the semantic-secure symmetric cipher E. The hash function SHA256 [29] and the symmetric cipher AES [11] are provided in OpenSSL [31] library (version 1.0.0g). The whole program is written in C language and compiled with GCC 4.4.5. It runs in a single process. Our implementation is not optimized and further performance improvements can be expected.

The test machine is a laptop computer, which is equipped with a 2.5GHz Intel Core 2 Duo mobile CPU (model T9300), a 3GB PC2700-800MHZ RAM and a 7200RPM hard disk. The test machine runs 32 bits version of Gentoo Linux OS with kernel 3.1.10. The file system is EXT4 with 4KB page size.

We run the proposed client-side deduplication scheme UH-CSD over files[14] of size 128MB, 256MB, 512MB, and 1024MB, respectively. The running time of the proof protocol (i.e. interactive algorithm $\langle\mathcal{P},\mathcal{V}\rangle$) in UH-CSD is reported in Figure 3, compared with network transfer time of test files without encryption or deduplication. The running time of encoding algorithm $\mathcal{E}$ is very close to (and smaller than) the interactive algorithm $\langle\mathcal{P},\mathcal{V}\rangle$. The decoding algorithm $\mathcal{D}$ is just the AES decryption algorithm, which is more efficient than $\mathcal{E}$. Here we save the actual running time for $\mathcal{E}$ and $\mathcal{D}$. All measurement represents the mean of 5 trails. Since the variants are very small, we do not report it.

We observe that, for small files, the saving in uploading time is small if the network *upload* speed is as fast as 5Mbps or even 20Mbps, but saving in server storage still matters to the cloud storage server and is independent on network speed. We remark that, leakage resilient server-side deduplication over encrypted files remains an open problem.

6. Conclusion and Future work

In this paper, we addressed an important security concern in cross-user client-side deduplication of encrypted files in the cloud storage: confidentiality of users' sensitive files against both outside adversaries and the honest-but-curious cloud storage server in the bounded leakage model. On technique aspect, we enhanced and generalized the convergent encryption method, and the resulting encryption scheme could support client-side deduplication of encrypted file in the bounded leakage model.

We clarify that this paper adopts a weaker leakage setting than Halevi *et al.* [20]. Our unpublished full paper [42] with stronger result adopts the same leakage setting as Halevi *et al.* w.r.t. outside adversaries. Furthermore, construction of secure client-side deduplication scheme in the strong leakage setting w.r.t. both outside adversaries and honest-but-curious cloud storage server is in our future work.

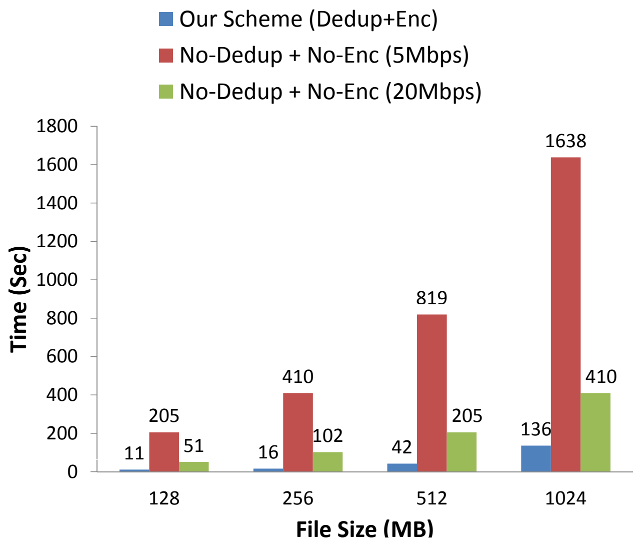

Figure 3. Comparison between the running time of the proof protocol (i.e. interactive algorithm $\langle\mathcal{P},\mathcal{V}\rangle$) of our client-side deduplication scheme UH-CSD and the network transfer time of files without encryption.

[12] We treat SHA256 as a random oracle and choose $\mathtt{SHA256}(k\|x)$ to simulate the universal hash function $\mathsf{h}_k(x)$ in order to achieve high performance. Software performance of universal hash is reported by Nevelsteen and Preneel [27].

[13] AES encryption in CBC mode with fresh random IV, where IV will be a part of the ciphertext.

[14] Our test files are generated by encrypting a large file using AES method with a random encryption key, so could be considered as random files.

References

[1] Joël Alwen, Yevgeniy Dodis, and Daniel Wichs. Leakage-Resilient Public-Key Cryptography in the Bounded-Retrieval Model. In *CRYPTO '09: Annual International Cryptology Conference on Advances in Cryptology*, pages 36–54, 2009.

[2] Amazon. AWS Customer Agreement. `http://aws.amazon.com/agreement/`.

[3] Paul Anderson and Le Zhang. Fast and secure laptop backups with encrypted de-duplication. In *Proceedings of the 24th international conference on Large installation system administration*, LISA'10, pages 1–8, 2010.

[4] Giuseppe Ateniese, Randal Burns, Reza Curtmola, Joseph Herring, Lea Kissner, Zachary Peterson, and Dawn Song. Provable data possession at untrusted stores. In *CCS '07: ACM conference on Computer and communications security*, pages 598–609, 2007.

[5] Boaz Barak, Yevgeniy Dodis, Hugo Krawczyk, Olivier Pereira, Krzysztof Pietrzak, François-Xavier Standaert, and Yu Yu. Leftover Hash Lemma, Revisited. In *CRYPTO*, pages 1–20, 2011.

[6] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. *J. ACM*, 59(2):6:1–6:48, 2012.

[7] Mihir Bellare, Sriram Keelveedhi, and Thomas Ristenpart. Message-locked encryption and secure deduplication. Cryptology ePrint Archive, Report 2012/631, 2012. `http://eprint.iacr.org/`.

[8] Lawrence Carter and Mark Wegman. Universal classes of hash functions (Extended Abstract). In *STOC '77: ACM symposium on Theory of computing*, pages 106–112, 1977.

[9] Ee-Chien Chang and Jia Xu. Remote Integrity Check with Dishonest Storage Server. In *ESORICS '08: European Symposium on Research in Computer Security: Computer Security*, pages 223–237, 2008. `http://eprint.iacr.org/2008/346`.

[10] CNET. Who owns your files on Google Drive? `http://news.cnet.com/8301-1023_3-57420551-93/who-owns-your-files-on-google-drive/`.

[11] Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. 2002.

[12] Ernesto Damiani, S. De Capitani di Vimercati, Sara Foresti, Sushil Jajodia, Stefano Paraboschi, and Pierangela Samarati. Key management for multi-user encrypted databases. In *StorageSS '05: Proceedings of ACM workshop on Storage security and survivability*, pages 74–83, 2005.

[13] Yevgeniy Dodis, Rafail Ostrovsky, Leonid Reyzin, and Adam Smith. Fuzzy Extractors: How to Generate Strong Keys from Biometrics and Other Noisy Data. *SIAM J. Comput.*, 38(1):97–139, 2008.

[14] John Douceur, Atul Adya, William Bolosky, Dan Simon, , and Marvin Theimer. Reclaiming space from duplicate files in a serverless distributed file system. In *ICDCS '02: International Conference on Distributed Computing Systems*, 2002.

[15] John Douceur, William Bolosky, and Marvin Theimer. US Patent 7266689: Encryption systems and methods for identifying and coalescing identical objects encrypted with different keys, 2007.

[16] Dropbox. Dropbox Privacy Policy. `https://www.dropbox.com/privacy`.

[17] Dropship. Dropbox api utilities, April 2011. `https://github.com/driverdan/dropship`.

[18] Oded Goldreich. *Foundations of Cryptography: Volume 2, Basic Applications*. 2004.

[19] Google. Google Terms of Service. `http://www.google.com/policies/terms/`.

[20] Shai Halevi, Danny Harnik, Benny Pinkas, and Alexandra Shulman-Peleg. Proofs of ownership in remote storage systems. In *CCS '11: ACM conference on Computer and communications security*, pages 491–500, 2011. `http://eprint.iacr.org/2011/207`.

[21] Shulman-Peleg A. Harnik D., Pinkas B. Side Channels in Cloud Services: Deduplication in Cloud Storage. *IEEE Security and Privacy Magazine, special issue of Cloud Security*, 8(6), 2010.

[22] Apple Inc. Apple Privacy Policy (Covering iCloud). `http://www.apple.com/privacy/`.

[23] Ari Juels and Burton S. Kaliski, Jr. Pors: proofs of retrievability for large files. In *CCS '07: ACM conference on Computer and communications security*, pages 584–597, 2007.

[24] Luis Marques and Carlos Costa. Secure deduplication on mobile devices. In *OSDOC '11: Workshop on Open Source and Design of Communication*, pages 19–26, 2011.

[25] Microsoft. Microsoft Services Agreement. `http://windows.microsoft.com/en-US/windows-live/microsoft-service-agreement`.

[26] Ilya Mironov, Omkant Pandey, Omer Reingold, and Gil Segev. Incremental deterministic public-key encryption. In *EUROCRYPT '12: Proceedings of the 31st Annual international conference on Theory and Applications of Cryptographic Techniques*, pages 628–644, 2012.

[27] Wim Nevelsteen and Bart Preneel. Software Performance of Universal Hash Functions. In *EUROCRYPT '99, International Conference on the Theory and Application of Cryptographic Techniques*, pages 24–41.

[28] Wee Keong Ng, Yonggang Wen, and Huafei Zhu. Private data deduplication protocols in cloud storage. In *SAC '12: Proceedings of the 27th Annual ACM Symposium on Applied Computing*, pages 441–446, 2012.

[29] NIST. National Institute of Standards and Technology. Secure hash standard (SHS). FIPS 180-2, August 2002.

[30] Tatsuaki Okamoto. Provably Secure and Practical Identification Schemes and Corresponding Signature Schemes. In *CRYPTO '92: Annual International Cryptology Conference on Advances in Cryptology*, pages 31–53, 1993.

[31] OpenSSL. OpenSSL Project. `http://www.openssl.org/`.

[32] Roberto Di Pietro and Alessandro Sorniotti. Boosting Efciency and Security in Proof of Ownership for Deduplication. In *ASIACCS '12: ACM Symposium on Information, Computer and Communications Security (Full Paper)*, 2012.

[33] Hovav Shacham and Brent Waters. Compact Proofs of Retrievability. In *ASIACRYPT '08*, pages 90–107, 2008.

[34] SNIA. Understanding Data De-duplication Ratios. white paper.

[35] SpiderOak-Blog. Why SpiderOak doesn't de-duplicate data across users.

[36] Mark Storer, Kevin Greenan, Darrell Long, and Ethan Miller. Secure Data Deduplication. In *StorageSS '08: ACM international workshop on Storage security and survivability*, pages 1–10, 2008.

[37] Twitter. Tweetdeck. `http://money.cnn.com/2012/03/30/technology/tweetdeck-bug-twitter/`.

[38] Mark Wegman and Larry Carter. New Hash Functions and Their Use in Authentication and Set Equality. *J. Comput. Syst. Sci.*, pages 265–279, 1981.

[39] Wikipedia. Comparison of online backup services. `http://en.wikipedia.org/wiki/Comparison_of_online_backup_services`.

[40] wired.com. Dropbox Left User Accounts Unlocked for 4 Hours Sunday. `http://www.wired.com/threatlevel/2011/06/dropbox/`; `http://blog.dropbox.com/?p=821`.

[41] Jia Xu and Ee-Chien Chang. Towards efficient proof of retrievability in cloud storage. In *ASIACCS '12: Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security (Full Paper)*, 2011.

[42] Jia Xu, Ee-Chien Chang, and Jianying Zhou. Leakage-resilient client-side deduplication of encrypted data in cloud storage. Cryptology ePrint Archive, Report 2011/538, 2011. `http://eprint.iacr.org/`.

[43] Qingji Zheng and Shouhuai Xu. Secure and efficient proof of storage with deduplication. In *CODASPY '12: ACM conference on Data and Application Security and Privacy*, pages 1–12, 2012.

A. Proof of Theorem 1

Proof. For any PPT adversary $\mathcal{A}_{\mathcal{CSD}}$ against the UH-CSD scheme in Figure 2, we construct a PPT adversary $\mathcal{A}_{\mathsf{E}}$ against the underlying private-key semantic secure encryption scheme E. We emphasize that we adopt the equivalent alternative definition of "private-key semantic secure encryption" given by Goldreich [18].

Construction of $\mathcal{A}_{\mathsf{E}}$: The adversary $\mathcal{A}_{\mathsf{E}}$ is given a ciphertext $C_F = \mathsf{E.Enc}_\tau(F)$ where the encryption key τ and the input file F are unknown and F has at least ξ_0 bits min-entropy. $\mathcal{A}_{\mathsf{E}}$ is allowed to learn any output of $\mathsf{Func}(F)$ from the oracle $\mathcal{O}^F$, where the PPT-computable function Func is chosen by $\mathcal{A}_{\mathsf{E}}$.

$\mathcal{A}_{\mathsf{E}}$ can simulate a security game $\mathsf{G}^{\mathtt{Sim}}$ as below, where $\mathcal{A}_{\mathsf{E}}$ plays the role of challenger and $\mathcal{A}_{\mathcal{CSD}}$ plays the role of adversary:

Setup. $\mathcal{A}_{\mathsf{E}}$ learns the hash value $\mathsf{hash}(F)$ from the oracle $\mathcal{O}^F$ and sends $\mathsf{hash}(F)$ to $\mathcal{A}_{\mathcal{CSD}}$.

Learning-I. $\mathcal{A}_{\mathsf{E}}$ simply forwards LEAKQUERY made by $\mathcal{A}_{\mathcal{CSD}}$ to the oracle $\mathcal{O}^F$ and forwards the response given by the oracle to $\mathcal{A}_{\mathcal{CSD}}$.

Commit. $\mathcal{A}_{\mathsf{E}}$ learns the value of the challenged subsequence $\alpha = F[i_1]\|\ldots\|F[i_v]$ from the oracle $\mathcal{O}^F$ and then exactly follows the rest part of **Commit** phase in the real game $\mathsf{G}_{\mathcal{A}}^{\mathcal{CSD}}$.

Guess-I. Denote the output of the extractor $\mathcal{A}^*_{\mathcal{CSD}}$ as $b^{\mathtt{Sim}}_{\mathcal{A}^*_{\mathcal{CSD}}} \in \{0,1\}$.

Learning-II. Challenger $\mathcal{A}_{\mathsf{E}}$ answers the following queries made by $\mathcal{A}_{\mathcal{CSD}}$:

- ENCODE-QUERY: In response to the encode query, the challenger $\mathcal{A}_{\mathsf{E}}$ independently and randomly chooses $\hat{\tau} \xleftarrow{\$} \mathsf{KeyGen}(1^\lambda)$ and $s_1, s_2 \xleftarrow{\$} \{0,1\}^\lambda$, and set $C_{\hat{\tau}} := (s_1, s_2, \mathsf{hash}(F))$. Let $(C_0, C_1) = (C_{\hat{\tau}}, C_F)$ and sends (C_0, C_1) to $\mathcal{A}_{\mathcal{CSD}}$. Recall that $\mathcal{A}_{\mathsf{E}}$ is given the ciphertext C_F, and $\mathsf{hash}(F)$ is obtained from the oracle $\mathcal{O}^F$ in the **Setup** phase.

- VERIFY-QUERY: $\mathcal{A}_{\mathsf{E}}$ runs the prover algorithm and $\mathcal{A}_{\mathcal{CSD}}$ replaces the verifier algorithm. Denote with (u_1, u_2) the message received from $\mathcal{A}_{\mathcal{CSD}}$. If $(u_1, u_2) = (s_1, s_2)$, then send $\mathsf{hash}(C_F)$ to $\mathcal{A}_{\mathcal{CSD}}$; otherwise, send a random value $H \xleftarrow{\$} \{0,1\}^\lambda$ to $\mathcal{A}_{\mathcal{CSD}}$.
- PROVE-QUERY: $\mathcal{A}_{\mathsf{E}}$ runs $\mathcal{V}(C_0)$ to interact with adversary $\mathcal{A}_{\mathcal{CSD}}$ which replaces the prover algorithm, following the description in game $\mathsf{G}_{\mathcal{A}}^{\mathcal{CSD}}$ exactly.

Guess-II. The adversary $\mathcal{A}_{\mathcal{CSD}}$ outputs a guess $b^{\mathtt{Sim}}_{\mathcal{A}_{\mathcal{CSD}}} \in \{0,1\}$ of b. The game $\mathsf{G}^{\mathtt{Sim}}$ simulated by $\mathcal{A}_{\mathsf{E}}$ completes.

At the end, $\mathcal{A}_{\mathsf{E}}$ outputs $\alpha_{b^{\mathtt{Sim}}_{\mathcal{A}_{\mathcal{CSD}}}} \in \{\alpha_0, \alpha_1\}$ and wins the semantic-security game w.r.t. encryption scheme E if $\alpha_{b^{\mathtt{Sim}}_{\mathcal{A}_{\mathcal{CSD}}}} = \alpha = F[i_1]\|\ldots\|F[i_v]$.

So far, $\mathcal{A}_{\mathsf{E}}$ has received at most $(\lambda + \xi_0 - \xi_1)$ bits (in term of length) message about the unknown file F from the oracle $\mathcal{O}^F$. Thus, after leakage from the oracle, the unknown file F should have at least $(\xi_1 - \lambda) = \lambda + \Omega(\lambda)$ bits min-entropy, according to Lemma 2.2 in Dodis *et al.* [13].

Claim 1. *Suppose* E *is private key ciphertext-indistinguishable and* $\{\mathsf{h}_k(\cdot)\}$ *be a universal hash family. The simulated game* $\mathsf{G}^{\mathtt{Sim}}$ *is computationally indistinguishable with the real game* $\mathsf{G}^{\mathtt{Real}} = \mathsf{G}^{\mathcal{CSD}}_{\mathcal{A}_{\mathcal{CSD}}}(\xi_0, \xi_1)$*, to the view of adversary* $\mathcal{A}_{\mathcal{CSD}}$*.*

Proof of Claim 1. In game $\mathsf{G}^{\mathtt{Real}}$ all messages that the adversary $\mathcal{A}_{\mathcal{CSD}}$ obtain from the challenger are (derived from) $\big(\mathfrak{S}^{\mathtt{Real}}, \mathsf{hash}(F), s, \mathsf{h}_s(F) \oplus \tau, C_F\big)$, where $\mathfrak{S}^{\mathtt{Real}} = (y, \alpha)$ and y is the output of LEAK-QUERY and α is computed in **Commit** phase. Similarly, in game $\mathsf{G}^{\mathtt{Sim}}$, the counterpart is $\big(\mathfrak{S}^{\mathtt{Sim}}, \mathsf{hash}(F), s_1, s_2, C_F\big)$. For the same adversary $\mathcal{A}_{\mathcal{CSD}}$ with the same random coin, $\mathfrak{S}^{\mathtt{Real}} = \mathfrak{S}^{\mathtt{Sim}}$. So we just write them as $\mathfrak{S}$ for simplicity.

Sample a file F' from $\{0,1\}^{|F|}$ following the same distribution from which F is sampled. Generate a key $\tau' = \mathsf{E.KeyGen}(1^\lambda)$ and encrypt F' with key τ' to obtain ciphertext $C_{F'} = \mathsf{E.Enc}_{\tau'}(F')$. Let $X \approx_c Y$ denote that random variable X and Y are computationally-indistinguishable. We have

$$\begin{aligned} &\big(\mathfrak{S}, \mathsf{hash}(F), s, \mathsf{h}_s(F) \oplus \tau, C_F\big) && (3)\\ \approx_c &\big(\mathfrak{S}, \mathsf{hash}(F), s, \mathsf{h}_s(F) \oplus \tau, C_{F'}\big) && (4)\\ \approx_c &\big(\mathfrak{S}, \mathsf{hash}(F), s_1, s_2, C_{F'}\big) && (5)\\ \approx_c &\big(\mathfrak{S}, \mathsf{hash}(F), s_1, s_2, C_F\big). && (6)\end{aligned}$$

We explain the above equations as below. Eq (3) $\approx_c$ Eq (4) is because E is private key ciphertext-indistinguishable: Given information $(\mathfrak{S}, \mathsf{hash}(F), s, \mathsf{h}_s(F) \oplus \tau)$ about F, the unknown file F still has at least $\Omega(\lambda)$ entropy, hence its encryption C_F is computationally indistinguishable from an encryption $C_{F'}$ of a random file $F' \in \{0,1\}^{|F|}$ under a random encryption key τ', where F' is sampled following the same distribution from which F is sampled.

Eq (4) $\approx_c$ Eq (5) is followed directly from the leftover hash lemma [5] which applies to the universal hash $\{\mathsf{h}_k\}$. Note that $C_{F'}$ is independent on other terms in these two equations.

Eq (5) $\approx_c$ Eq (6) is again implied by the ciphertext-indistinguishability property of the encryption scheme E. Note that s_1, s_2 are independent on the other terms in these two equations.

Therefore, Claim 1 is proved.

□

Claim 2. *There exists some PPT extractor* $\mathcal{A}^*_{\mathcal{CSD}}$*, such that* $\Pr[b^{\mathtt{Sim}}_{\mathcal{A}_{\mathcal{CSD}}} = b^{\mathtt{Sim}}] \le \Pr[b^{\mathtt{Sim}}_{\mathcal{A}^*_{\mathcal{CSD}}} = b^{\mathtt{Sim}}] + negl(\lambda)$*.*

Proof. In **Learning-II** phase of $\mathsf{G}^{\mathtt{Sim}}$, the challenger $\mathcal{A}_{\mathsf{E}}$ does not make any new queries to $\mathcal{O}^F$, and all responses that $\mathcal{A}_{\mathsf{E}}$ provided to $\mathcal{A}_{\mathcal{CSD}}$ are computed from randomly sampled values and information that $\mathcal{A}_{\mathcal{CSD}}$ has already known before **Learning-II** (i.e. the hash value $\mathsf{hash}(F)$), except the ciphertext C_F.

It is straightforward that

$$\Pr[\mathcal{A}_{\mathsf{E}}^{\mathcal{O}^F}(C_F, |F|) = \alpha] = \Pr[b^{\mathtt{Sim}}_{\mathcal{A}_{\mathcal{CSD}}} = b^{\mathtt{Sim}}].$$

Since the underlying encryption scheme E is semantic secure (See the definition in Exercise 18 of Chapter 5 in Goldreich [18]), there exists a PPT algorithm $\mathcal{B}$, such that

$$\Pr[\mathcal{A}_{\mathsf{E}}^{\mathcal{O}^F}(C_F, |F|) = \alpha] \le \Pr[\mathcal{B}^{\mathcal{O}^F}(|F|) = \alpha] + negl(\lambda).$$

We construct the extractor $\mathcal{A}^*_{\mathcal{CSD}}$ based on algorithm $\mathcal{B}$: Let $\alpha_{\mathcal{B}}$ be the output of $\mathcal{B}$. If $\alpha_{\mathcal{B}} = \alpha_{\hat{b}} \in \{\alpha_0, \alpha_1\}$ for some $\hat{b} \in \{0,1\}$, then $\mathcal{A}^*_{\mathcal{CSD}}$ outputs $b^{\mathtt{Sim}}_{\mathcal{A}^*_{\mathcal{CSD}}} := \hat{b}$; otherwise $\mathcal{A}^*_{\mathcal{CSD}}$ outputs a random bit $b^{\mathtt{Sim}}_{\mathcal{A}^*_{\mathcal{CSD}}} \xleftarrow{\$} \{0,1\}$. We have

$$\begin{aligned}\Pr[b^{\mathtt{Sim}}_{\mathcal{A}_{\mathcal{CSD}}} = b^{\mathtt{Sim}}] =&\Pr[\mathcal{A}_{\mathsf{E}}^{\mathcal{O}^F}(C_F, |F|) = \alpha]\\ \le&\Pr[\mathcal{B}^{\mathcal{O}^F}(|F|) = \alpha] + negl(\lambda)\\ \le&\Pr[b^{\mathtt{Sim}}_{\mathcal{A}^*_{\mathcal{CSD}}} = b^{\mathtt{Sim}}] + negl(\lambda). \qquad (7)\end{aligned}$$

Therefore, there exists some PPT extractor $\mathcal{A}^*_{\mathcal{CSD}}$, such that $\Pr[b^{\mathtt{Sim}}_{\mathcal{A}_{\mathcal{CSD}}} = b^{\mathtt{Sim}}] \le \Pr[b^{\mathtt{Sim}}_{\mathcal{A}^*_{\mathcal{CSD}}} = b^{\mathtt{Sim}}] + negl(\lambda)$.

Furthermore, Claim 1 implies that

$$|\Pr[b^{\mathtt{Sim}}_{\mathcal{A}_{\mathcal{CSD}}} = b^{\mathtt{Sim}}] - \Pr[b^{\mathtt{Real}}_{\mathcal{A}_{\mathcal{CSD}}} = b^{\mathtt{Real}}]| \le negl(\lambda) \quad (8)$$
$$|\Pr[b^{\mathtt{Sim}}_{\mathcal{A}^*_{\mathcal{CSD}}} = b^{\mathtt{Sim}}] - \Pr[b^{\mathtt{Real}}_{\mathcal{A}^*_{\mathcal{CSD}}} = b^{\mathtt{Real}}]| \le negl(\lambda). \quad (9)$$

Combine Eq (7), Eq (8) and Eq (9), we have

$$\Pr[b^{\mathtt{Real}}_{\mathcal{A}_{\mathcal{CSD}}} = b^{\mathtt{Real}}] \le \Pr[b^{\mathtt{Real}}_{\mathcal{A}^*_{\mathcal{CSD}}} = b^{\mathtt{Real}}] + negl(\lambda).$$

□

Therefore, the client-side deduplication scheme UH-CSD is (ξ_0, ξ_1)-secure according to Definition 2.

□