XDedup: Efficient Provably-Secure Cross-User Chunk-Level Client-Side Deduplicated Cloud Storage of Encrypted Data

Chia-Mu Yu
Department of Computer Science and Engineering, National Chung Hsing University, Taichung, Taiwan

***Abstract*—Data deduplication, aiming to eliminate duplicate data, has been widely used in cloud storage to reduce the amount of storage space and save bandwidth. Unfortunately, as an increasing number of sensitive data are stored remotely, the encryption, the simplest way for data privacy, is not compatible with data deduplication. Though many research efforts have been devoted to securing deduplication, they all are subject to performance, security, and applicability limitations. Here, we propose two encrypted deduplication schemes, SDedup and XDedup, both based on Merkle puzzle. To the best of our knowledge, XDedup is the first brute-force resilient encrypted deduplication with only symmetrically cryptographic two-party interactions. The analysis and numerical simulations are conducted to demonstrate the performance and practicality of SDedup and XDedup.**

I. Introduction

A. Data Deduplication

Data deduplication, aiming to avoid storing the identical file twice, is an essential technique widely used in cloud storage providers (CSPs) (*e.g.*, Dropbox). The use of data deduplication may achieve up to 90% storage savings [9]. In particular, client-side data deduplication, the form of deduplication that takes place at the user side*, can further achieve the same level of network bandwidth savings. Due to the potential similarity of popular files, public cloud storage services are in favor of cross-user client-side deduplication, where deduplication is carried out on the data even from distinct users. Moreover, deduplication can also be performed at different granularities (e.g., file-level and chunk-level). As a result, according to the extent of the deduplication effectiveness and overhead reduction, cross-user chunk-level client-side deduplication acts as the most aggressive technique in eliminating the redundant transmitted and stored data. Throughout the paper, *data deduplication* refers to cross-user chunk-level client-side data deduplication, unless stated otherwise.

The implementation of data deduplication is, in fact, straightforward; for a chunk f to be uploaded, the user first calculates and sends the hash $h(f)$ to the cloud, where $h(\cdot)$ denotes the cryptographic hash function (e.g., SHA256). Once the cloud finds a copy of $h(f)$ in the memory (i.e., file existence), the user has no need to upload f again. Otherwise, the user simply uploads f and the cloud keeps $h(f)$ in the memory for duplicate checks in the future.

B. Security and Privacy Concerns

Unfortunately, the benefits of data deduplication come with the security and privacy threats. For example, the unauthorized access of the files in the cloud storage due to the system implementation flaw [22] and the design nature of deduplication [10], [13] has been found to be a potential security threat. Side channel [13] and covert channel [12] in the cloud storage have also proven feasible. Several solutions [10], [12], [22] have been presented to defend against the security threats on the deduplicated cloud storage.

In addition, since more and more sensitive data are uploaded to the cloud storage, one may have a privacy concern that the cloud will be benefited by looking at the user's private data. Encrypting the data before uploading it might be a solution for the privacy leakage, but the encryptions of the identical file from independent users result in different ciphertexts, losing the storage and bandwidth advantages of data deduplication. As a consequence, in this paper, we put the particular emphasis on the development of a cloud storage with the reconciliation of the encryption and data deduplication [27].

C. Related Work

In what follows, we briefly review three categories of techniques for deduplication of encrypted data.

1) Convergent Encryption: Convergent encryption (CE) [2], [8], [28], [32], [33] is the simplest way for tackling the privacy concern without compromising the deduplication effectiveness. Examples of storage systems with CE include Farsite [8] and Tahoe-LAFS [30]. In particular, with the hash $h(f)$ as *convergent key*, the user calculates and uploads $\mathcal{E}_{h(f)}(f)$, where $\mathcal{E}_k(\cdot)$ denotes the symmetric encryption with key k. Since the users with f are all able to derive the same $h(f)$ and $\mathcal{E}_{h(f)}(f)$, the deduplication still takes place on $\mathcal{E}_{h(f)}(f)$. From the theory point of view, CE can be generalized as message-locked encryption (MLE) [5], which achieves the PRV\$-CDA security and remarkable speedup in the encryption calculation. The follow-up studies [1], [4] further examine message correlation and parameter dependency of MLE. From the system point of view, CE is incompatible with the setting of different user privileges in the current corporate environment. Thus, a private cloud is introduced to manage files with differential privileges [17]. As the number of convergent keys is linearly increased with the number of uploaded pieces, Li et al [16] introduce Dekey, where a quorum of key management servers are used to manage convergent keys. CE with dynamic data ownership is studied in [11].

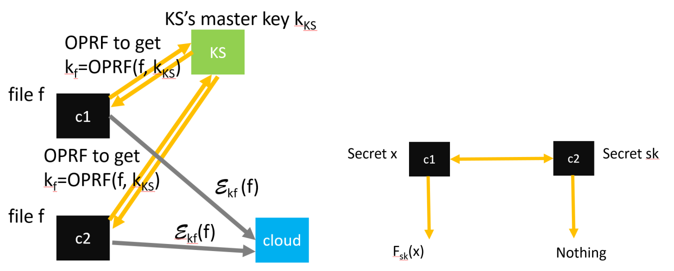
2) Encrypted Deduplication with Independent Servers: Despite their simplicity, CE and MLE suffer from the brute-force attack, particularly in the case of low min-entropy files (i.e., predictable files). In real world, the files usually have low min-entropy and therefore potential predictability, because of

*Cross-user deduplication is also called inter-user deduplication or global deduplication in the literature. In addition, throughout the paper, the terms *user* and *client*, and the terms *cloud* and *server* are used interchangeably.

the prior knowledge such as document format. Since the key space in CE is identical to the plaintext space, the low min-entropy characteristic leads to the possibility of brute-force. Consider an extreme case, where f has only two possibilities. Even only with the access to $\mathcal{E}_{h(f)}(f)$, the adversary can still easily infer f through two encryption calculations.

To counter against the brute-force attack, based on the idea of additional randomness, Bellare and Keelveedhi present DupLESS [3], where an additional key server KS is introduced to assist the key generation. More specifically, as shown in Fig. 1a, before uploading f, the user c_i and KS jointly compute a content-dependent key k_f for f by using oblivious pseudorandom function (OPRF) [23] (see Fig. 1b and will be described in more details in Sec. III-A) with the guarantee that no one, except for c_i, can derive k_f. After that, k_f is used for the calculation of the deduplicatable $\mathcal{E}_{k_f}(f)$. DupLESS has been formally proven D-IND\$-CPA secure [7]. However, though it is not the limit from the design nature, DupLESS can only be file-level deduplication; otherwise, the OPRF calculations lead to the dramatic performance degradation in both users and the key server.

SecDep [36] improves the performance of DupLESS by first performing cross-user file-level deduplication. If not succeeded, single-user chunk-level deduplication is then executed to look for the opportunity of fine-grained deduplicability. Similar to SecDep, a deduplication proxy sits in the middle between users and the cloud in [21], where the proxy and cloud perform cross-user deduplication but the user and proxy perform single-user deduplication. ClouDedup [25] works in a similar way. Threshold CE (tCE) [29] and PerfectDedup [26] perform the chunk-level deduplication by taking advantage of chunk popularity. Their idea is that the popular files will not contain sensitive information and therefore needs less privacy protection.



(a) DupLESS [3] (b) OPRF [23]

Fig. 1: DupLESS [3] and its cryptographic component.

3) Encrypted Deduplication without Independent Servers: Though the key server helps generate k_f for users in an oblivious manner, DupLESS and the follow-up solutions still find useless in the real world because the independent server has to be run by the third party. Encrypt-with-Signature (EwS) [7] claims to eliminate the need for a key server and retain the security guarantee by using threshold signatures. Nonetheless, Zheng et al. [37] argued that the dealers in EwS serve as the similar role of the key server in DupLESS. Very recently, Liu et al. [14] propose PAKEDedup (see Figs. 2a and 2b) based on a client-as-a-key-server (CaS) framework to dedeuplicate encrypted data by using password authenticated key agreement (PAKE)[†] [6] (see Fig. 2c) and partially homomorphic encryption (PHE) [24]. In essence, the independent server with additional secret is still necessary in PAKEDedup for generating k_f's; however, all of the users in CaS framework are potential key servers that check the chunk hash consistency via PAKE and exchange the chunk key via PHE.

4) Encrypted Deduplication in Other Contexts: In fact, the encrypted deduplication can apply to different contexts. For example, the aforementioned solutions inherently assume a single cloud, whereas CDStore [18] works in a cloud-of-clouds environment. Attribute-based encryption (ABE) and Proxy Re-Encryption (PRE) are considered in [31] and [35], respectively, to offer access control. Moreover, SVCDedup, enabling the encrypted cloud media center, is presented in [37]. In essence, SVCDedup considers the use of DupLESS in the context of scalable video coding (SVC) in the bounded leakage setting [33], [37]. With the consideration of key compromise, the rekeying issue for server-side deduplication is studied in [19].

D. Challenges in Designing Encrypted Deduplication

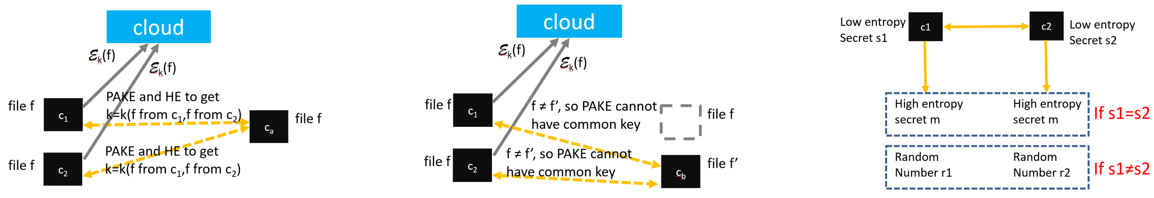
Many solutions have been propopsed to address the issue of encrypted data deduplication. Unfortunately, they are all subject to certain performance, security, and applicability limitations. Here, the limitations are summarized as follows.

1) Brute-Force Resiliency (L1): CE, MLE, and their variants all involve the uploading of $h(f)$ and $\mathcal{E}_{h(f)}(f)$. Nevertheless, as mentioned in Sec. I-C1, real world applications usually generate low min-entropy contents f, making the *offline brute-force attack* (see Sec. II-A2) easy to find out sensitive information.

2) Independent Server Assumption (L2): In spite of the differences in their functionalities, all of the solutions in Sec. I-C2 assume the use of independent servers. Note that the compromise of those independent servers usually will not lead to the crash down of the system, but unfortunately the security guarantee will degrade to the level of CE. Furthermore, the adversary is able to launch *online brute-force attack* (see Sec. II-A), aiming to recover the matching ciphertext by repeatedly querying servers. Lastly, the most critical weakness of independent servers is the impracticality of running independent servers without business justification [14].

3) Complicated Computation and Architecture (L3): Some of the current methods involve complicated arithmetics or architectures. More specifically, with the consideration of computationally expensive cryptographic techniques, tCE, DupLESS, PAKEDedup, and SVCDedup provide the data protection at the expense of the significantly degraded computation efficiency. The security of tCE also relies on a two-additional-server architecture, which is difficult to be deployed practically. The above performance drawback even has adverse impact on the deduplication granularity and effectiveness. In fact, the fine-grained chunk-level duplicate checks will bring too much computation burden at both user and server sides if intensive computing tasks need to be accomplished.

[†]PAKE is a two-party protocol; at the end of PAKE, both parties will agree with a common high-entropy secret if their initial low-entropy secrets are identical, and cannot make an agreement otherwise.



(a) PAKEDedup [14] (case 1) (b) PAKEDedup [14] (case 2) (c) PAKE [6]

Fig. 2: PAKEDedup [14] and its cryptographic component.

4) Heuristic Parameter Setting (L4): For some heuristic approaches, choosing an explicit threshold by using file popularity [26], [29], file sizes [13], and user privileges [17] to differentiate the data sensitivity would be difficult. Moreover, heuristic approaches also have restriction on application scenarios. For example, tCE has the preference on data sources such as VM-images and email attachments for multi-recipients [29].

5) Protocol Transparency (L5): CSPs may be reluctant to modify the internal configuration for the support of data privacy due to the lack of economic incentives. In this sense, some works (e.g., DupLESS and ClouDedup) work transparently with existing CSPs, but most of existing works are still waiting for the implementation and find little practical values currently.

6) Additional Privacy Leakage (L6): In the CaS framework, online user status needs to be exposed to either cloud or even the public, depending on protocol design. However, in either case, the privacy of user behavior is sacrificed for data privacy.

E. Contribution

We propose two encrypted data deduplication schemes, SDedup and XDedup. To the best of our knowledge, XDedup is the first provably-secure brute-force resilient encrypted deduplication with only lightweight cryptography and with only the interactions between the uploader and cloud. Moreover, XDedup also achieves perfect deduplication. In fact, our schemes possess the following distinguishing advantages.

- SDedup and XDedup only involve the symmetrically cryptographic operations, making our schemes computationally efficient and chunk-level deduplication feasible.
- SDedup can work transparently with any current cloud storage, without the need of CSP's engineering work at the backend.
- In XDedup, only the uploader and cloud participate in the uploading process, in contrast to the most of the current solutions in need of the third party's participation.
- SDedup and XDedup are formally proved secure, under the paradigm of simulation-based security [15].

The analysis and numerical simulations are used to demonstrate the practicality of our proposed methods.

A comparison among different schemes is shown in Table I. Table. I will be described in more detail in Sec. V-D.

TABLE I: COMPARISONS BETWEEN DIFFERENT ENCRYPTED DATA DEDUPLICATION

Methods	**(L1)**	**(L2)**	**(L3)**	**(L4)**	**(L5)**	**(L6)**
CE [8] and MLE [5]	–	✓	✓	✓	✓	✓
DupLESS [3]	✓	–	–	✓	✓	✓
ClouDedup [25]	✓	–	✓	✓	✓	✓
tCE [29]	✓	–	–	–	–	✓
EwS [7]	✓	–	–	✓	–	✓
PerfectDedup [26]	✓	✓	✓	–	–	✓
PAKEDedup [14]	✓	✓	–	✓	–	–
SVCDedup [37]	✓	–	–	✓	–	✓
SDedup (this paper)	✓	✓	✓	✓	✓	–
XDedup (this paper)	✓	✓	✓	✓	–	✓

II. SYSTEM MODEL

A. Storage and Threat Models

1) Storage Model: We consider a public CSP S and a number of independent users $\{c_j\}$. The cloud S performs cross-user chunk-level client-side data deduplication, attempting to reduce the cost of storage and network bandwidth. In this sense, all cloud users share a common disk storage. The user c_i possesses an individual key $\mathbf{k}_i$, which will not be shared with anyone else. The symmetric encryption $\mathcal{E}_k(\cdot)$ (e.g., AES), cryptographic hash function $h(\cdot)$ (e.g., SHA512), and message authentication code (MAC) $h_k(\cdot)$ (e.g., HMAC-SHA256) are ready for use.

Throughout the paper, the common setting is an uploader c_i attempting to upload a low min-entropy chunk f. This c_i has access to the full-length *chunk hash* $h(f)$ and truncated hash (also called *short hash*) $sh(f)$. Short hash $sh(f)$ can be implemented by keeping only partial bits of $h(f)$. Obviously, $sh(f)$ has high collision rate and thus the adversary cannot be confident that it is the specific f that implies $sh(f)$, mitigating the brute-force threat. c_i is aimed to calculate the *chunk key* k_f for encrypting f and to upload $\mathcal{E}_{k_f}(f)$.

2) Threat Model: We consider the adversaries in the malicious model, where users and the cloud can behave arbitrarily. There could also be a collusion of users and the cloud. The objective of adversaries is to recover the f or k_f owned by the honest user c_i.

Moreover, we consider *offline brute-force* and *online brute-force* attacks. Both share the same goal of recovering f. In the former, the adversary eavesdrops on a deterministic representation of low min-entropy content and then constructs deterministic representations of all candidate chunks to see whether a match can be found. In the latter, the adversary recovers f by querying S or independent servers with all candidate chunks to see whether the deduplication occurs or a match can be found.

B. Evaluation Metrics

We consider the following four metrics to evaluate the performance of data deduplication techniques. Throughout our evaluation, we consider the scenario, where c_i uploads a random distinct chunk to S for evaluating the expected performance. Such scenario addresses the cloud with uniformly distributed uploading requests.

1) Deduplication Percentage: A (normalized) metric for evaluating the effectiveness of data deduplication is *deduplication percentage* [9], defined as $1 - \frac{1}{\pi_o/\pi_i}$, where π_o and π_i denote the numbers of bytes input to and output from the deduplicated storage, respectively. The *perfect deduplication* will detect all duplicates and reaches the deduplication ratio of $1 - 1/n = (n-1)/n$ given n identical files uploaded.

2) Memory Overhead: An important step in client-side deduplication is hash matching. Thus, the chunk hash needs to be kept in the memory, so that S can perform a prompt duplicate check and sends back the matching result to users. However, as the amount of memory is usually limited, one has to consider the memory footprints incurred by the encrypted data deduplication. Moreover, one should also consider the memory/disk footprints in user devices, because of resource scarsity in certain cases.

3) Communication Overhead: The communication overhead is reversely proportional to deduplication percentage in client-side deduplication. However, depending on the protocol design, one may have subtle difference between different implementations. Such difference may affect user experience from the user point view and may aggregately cause performance bottleneck from the cloud point of view.

4) Computation Overhead: Both client-side and server-side computation overhead are our concern. The user devices could be resource-constrained (e.g., cellphones) and cannot afford computationally intensive tasks. On the other hand, though S is resource-abundant, it faces a huge number of upload/download requests from $\{c_j\}$. Thus, sophisticated computation tasks in the server-side should also be avoided in the design.

In the following, $\mathbb{D}_X$, $\mathbb{M}_X^S$ ($\mathbb{M}_X^c$), $\mathbb{T}_X$, and $\mathbb{C}_X$ denote the deduplication percentage under the first (second) scenario, server-side (user-side) memory overhead, computation overhead, communication overhead of scheme X, respectively.

III. Proposed Solutions

In this section, our deduplication solutions are described. Our study is conducted evolutionarily; first, we present a strawman protocol (SP) in Sec. III-A. Motivated by the security flaw of SP, we present our encrypted data deduplication with only symmetrically cryptographic operations (SDedup) in Sec. III-B. Nevertheless, CaS-based SDedup actually gives away the online user privacy. Hence, we present our encrypted data deduplication with extreme efficiency (XDedup) in Sec. III-C. Some minor implementation issues are described in Sec. III-D. Table II summarizes the notations frequently used in this paper.

TABLE II: Notation Table.

Notation	Description
S	the cloud storage.
c_i	the cloud user.
f	the chunk to be uploaded.
$\mathbf{k}_i$	the individual key possessed by c_i.
k_f	the chunk key of f.
$h(f)$	the chunk hash of f.
$sh(f)$	the short hash of f.
ℓ_f	the (encrypted) chunk size.
$\ell_k, \ell_r, \ell_h, \ell_{sh}$	the number of bits representing key, random number, hash, and short hash, respectively.
ℓ_{oprf}	the number of bits required in OPRF interactions.
$p_\bullet$	the probability that a specific chunk is in S.
p_{off}	the probability that a specific user is offline.
p_{own}	the probability that a user owns a specific chunk.
$\mathcal{E}_{k_f}(f) \in S$	the encrypted chunk $\mathcal{E}_{k_f}(f)$ is in the cloud storage.

A. Strawman Protocol (SP)

1) Basic Idea of SP: Our strawman protocol (SP) still bases its security on the CaS framework [14], where each user can assist the key generation. SP differs from PAKEDedup in that the online user contributes to the generation of chunk key for specific f. In addition, a large number of online users in PAKEDedup will participate in the calculations of PAKE and PHE. However, though still a large number of users in SP involve in the key generation, c_i can derive the chunk key by performing only OPRF, resulting in computation reduction. In fact, SP can be seen as a variant of DupLESS with all users as distributed key servers.

Before describing SP, we have a brief review of OPRF first. Oblivious pseudorandom function (OPRF) (see Fig. 1b) is a two party protocol involving a sender c_j and receiver c_i. OPRF $OPRF[x, sk]$ aims to compute a pseudorandom function (PRF) $\mathcal{F}_{sk}(x)$, where x is c_i's secret input and sk is a scret key owned by c_j. OPRF enforces that c_i obtains $\mathcal{F}_{sk}(x)$ but c_j learns nothing during the OPRF interactions. Basically, we inherently assume the use of RSA-OPRF built on RSA blind signatures in SP.

2) Detailed Description of SP: The protocol description of SP is shown in Fig. 3. Here, the communications among $\{c_j\}$ are through S; i.e., $\{c_j\}$ do not have direct communications. Furthermore, S maintains a publicly available online user list $\mathcal{U}$, which lists and updates all online cloud users in real time. This can be accomplished by making sure that the client software of cloud storage sends heartbeat messages to S periodically. Moreover, S also maintains a lookup table $\mathcal{L}$, where each record is of the form $[sh(f), c_i]$. The purpose of $\mathcal{L}$ is to associate users with their key servers (the other users) (explained below). Thus, there could be multiple records in $\mathcal{L}$ with the same $sh(f)$ but different c_i's.

In SP, S works like a broker who bridges c_i and the other users. In particular, if S, after receiving $sh(f)$ from c_i, cannot find a match in $\mathcal{L}$ (**case 1** of Fig. 3), c_i randomly picks another user (e.g., c_a) from $\mathcal{U}$ as its key server. Then, c_i with $h(f)$ and c_a with $\mathbf{k}_a$ jointly performs OPRF, so that c_i can calculate the chunk key $k_f^{i,a} = OPRF[h(f), \mathbf{k}_a]$. Note that the superscripts i, a of $k_f^{i,a}$ emphasize that this chunk key is generated jointly by c_i and c_a. Afterwards, S adds a record

```
Offline Setting:
$S$ maintains a lookup table $\mathcal{L}$ of records $[sh(f), c_i]$
$S$ maintains a online user list $\mathcal{U}$
Online Execution:
01 $C \to S : sh(f)$
02 if $\mathcal{L}(sh(f)) = \emptyset$ (case 1)
03     $c_i$ randomly picks $c_a$ from $\mathcal{U}$
04     $c_i$ gets $k_f^{i,a} = OPRF[h(f), \mathbf{k}_a]$
05     $c_i \to S : \mathcal{E}_{k_f^{i,a}}(f)$ and $\mathcal{E}_{\mathbf{k}_i}(k_f^{i,a})$
06     $S$ adds a record $[sh(f), c_a]$ to lookup table $\mathcal{L}$
07 else
08     if $\mathcal{L}(sh(f)) \cap \mathcal{U} \neq \emptyset$ (case 2a)
09         for each $c_a \in \mathcal{L}(sh(f)) \cap \mathcal{U}$
10             $c_i$ gets $k_f^{i,a} = OPRF[h(f), \mathbf{k}_a]$
11             $c_i \to S : h(\mathcal{E}_{k_f^{i,a}}(f))$
12                 if $h(\mathcal{E}_{k_f^{i,a}}(f))$ indicates dedup
13                     $c_i \to S : \mathcal{E}_{\mathbf{k}_i}(k_f^{i,a})$
14         if no $c_a \in \mathcal{L}(sh(f)) \cap \mathcal{U}$ causes dedup
15             $c_i$ randomly picks $c_b$ from $\mathcal{U}$
16             $c_i$ gets $k_f^{i,b} = OPRF[h(f), \mathbf{k}_b]$
17             $c_i \to S : \mathcal{E}_{k_f^{i,b}}(f)$ and $\mathcal{E}_{\mathbf{k}_i}(k_f^{i,b})$
18             $S$ adds a record $[sh(f), c_b]$ to lookup table $\mathcal{L}$
19     else (case 2b)
20         $c_i$ randomly picks $c_b$ from $\mathcal{U}$
21         $c_i$ gets $k_f^{i,b} = OPRF[h(f), \mathbf{k}_b]$
22         $c_i \to S : \mathcal{E}_{k_f^{i,b}}(f)$ and $\mathcal{E}_{\mathbf{k}_i}(k_f^{i,b})$
23         $S$ adds a record $[sh(f), c_b]$ to lookup table $\mathcal{L}$
```

Fig. 3: The protocol description of SP.

$[sh(f), c_a]$ to $\mathcal{L}$ for the association between $sh(f)$ and c_a. As a consequence, the user with f' such that $sh(f') = sh(f)$ may also derive a random key from $OPRF[h(f'), \mathbf{k}_a]$. The conceptual illustration of case 1 of SP is shown in Fig. 4a.

On the other hand, if S can find at least one online user in $\mathcal{L}(sh(f))$ (**case 2a** of Fig. 3), where $\mathcal{L}(sh(f))$ denotes a set of c_j's, these users might ever contribute to the generation of k_f. Thus, c_i performs OPRF with each $c_a \in \mathcal{L}(sh(f)) \cap \mathcal{U}$ and calculates $k_f^{i,a}$ for each $c_a \in \mathcal{L}(sh(f)) \cap \mathcal{U}$. Then, c_i uses the derived $k_f^{i,a}$'s and $h(\mathcal{E}_{k_f^{i,a}}(f))$'s to perform the duplicate check and data uploading. If all of the online users happen to have f' such that $sh(f') = sh(f)$, c_i will conduct similar procedures in case 1 to pick a random online user and calculate k_f.

The last case (**case 2b** of Fig. 3) that S, after receiving $sh(f)$, cannot find any online user in $\mathcal{L}(sh(f))$ is identical to case 1. Therefore, the operations in case 1 apply to case 2b. The conceptual illustration of case 2b is depicted in Fig. 4b.

3) Performance Evaluation of SP: We evaluate the performance of SP based on the metrics in Sec. II. Let $p_\bullet$ be the probability of a specific chunk in S. Let p_{off} be the probability that a user is offline. Let p_{own} be the probability that a user owns a particular chunk. The probabilities $p_\bullet$, p_{off}, and p_{own}, in fact, vary with time and users; nevertheless, to simplify the performance analysis, we abuse them slightly and assume that they are time- and user-independent.

Deduplication Percentage $\mathbb{D}_{\text{SP}}$. Since each uploading can be seen as an independent event, our strategy of computing $\mathbb{D}_{\text{SP}}$ is to first estimate the expected size ℓ_e of $\mathcal{E}_{k_f}(f)$ for each uploading request. After that, $\mathbb{D}_{\text{SP}}$ can be calculated as $1-(1/(x\ell_f/x\ell_e)) = 1-(1/(\ell_f/\ell_e))$, where ℓ_f is the number of bits for representing the (encrypted) chunk.

The occupied spaces for different uploading behaviors are shown in Fig. 5a. In particular, we can see from Figs. 3 and 5a that if $\mathcal{E}_{k_f}(f) \notin S$, c_i always needs to send $\mathcal{E}_{k_f}(f)$ to S. In contrast, given $\mathcal{E}_{k_f}(f) \in S$, whether c_i needs to send $\mathcal{E}_{k_f}(f)$ again depends on if c_i can find an online user who uploads $\mathcal{E}_{k_f}(f)$ previously. As a result, $\mathbb{D}_{\text{SP}}$ can be formulated as

$$1 - \frac{1}{\ell_f/((1-p_\bullet)(\ell_f+\ell_k) + p_\bullet p_{off}^{|\mathcal{L}(sh(f))|}(\ell_f+\ell_k))}, \quad (1)$$

where ℓ_k is the bit length of the (encrypted) chunk key.

Memory overhead $\mathbb{M}_{\text{SP}}$. Since $\mathcal{E}_{k_f}(f)$, $\mathcal{E}_{\mathbf{k}_i}(k_f)$, chunk hash, chunk key are necessary among all of the solutions, we do not consider them as overhead. In this sense, $\mathbb{M}_{\text{SP}}^{S}$ consists of $\mathcal{L}$ and $\mathcal{U}$. On the other hand, though users in SP serve as key servers, users are only required to present their individual keys in OPRF. Hence, SP does not in cur the memory overhead at the user side; i.e., $\mathbb{M}_{\text{SP}}^{c}$, is zero.

Communication overhead $\mathbb{T}_{\text{SP}}$. The numbers of bits required in the message exchanges of SP in different cases for the uploading behaviors are shown in Fig. 5b. One can see from Figs. 3 and 5b that if $sh(f)$ cannot be found in $\mathcal{L}$, then OPRF interactions and the uploading of $\mathcal{E}_{k_f}(f)$ and $\mathcal{E}_{\mathbf{k}_i}(k_f)$ are necessary. In fact, given that $sh(f)$ can be found in $\mathcal{L}$, unless all users in $\mathcal{L}(sh(f))$ are offline, $|\mathcal{L}(sh(f)) \cap \mathcal{U}|$ OPRF interactions and $|\mathcal{L}(sh(f)) \cap \mathcal{U}|$ duplicate checks are necessary. Here, to ease the calculation, we assume that the probability $Pr[\mathcal{L}(sh(f)) \neq \emptyset | f \notin S]$ of $\mathcal{L}(sh(f)) \neq \emptyset$ conditioned on the existence of f in S is 1. With such an assumption, the communication overhead $\mathbb{T}_{\text{SP}}$ of SP can be approximated as

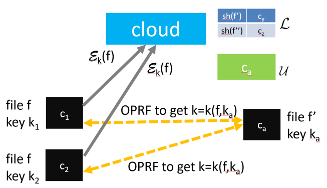
$$\begin{aligned}\mathbb{T}_{\text{SP}} &\approx \ell_{sh} + p_\bullet(1-p_{off})(|\mathcal{L}(sh(f)) \cap \mathcal{U}|(\ell_{oprf}+\ell_h)) \\ &+ (1-p_\bullet)(1-p_{off})(|\mathcal{L}(sh(f)) \cap \mathcal{U}|(\ell_{oprf}+\ell_h) + \ell_f + \ell_k) \\ &+ p_{off}(\ell_{oprf}+\ell_f+\ell_k), \end{aligned} \quad (2)$$

where ℓ_{sh} is the bit length of short hash and ℓ_{oprf} denotes the number of bits required in the OPRF interaction.

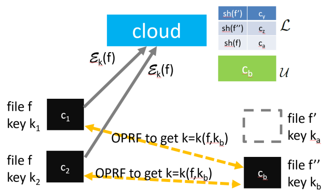
Computation overhead $\mathbb{C}_{\text{SP}}$. Though Fig. 5b is used to present different cases of $\mathbb{T}_{\text{SP}}$, it actually can also be used to show the computation overhead. In particular, since the communications are also associated with the corresponding computing tasks, the computation overhead $\mathbb{C}_{\text{SP}}$ can be, in a form similar to $\mathbb{T}_{\text{SP}}$, approximated as

$$\begin{aligned}\mathbb{C}_{\text{SP}} &\approx C_{sh} + p_\bullet(1-p_{off})(|\mathcal{L}(sh(f)) \cap \mathcal{U}|(C_{oprf}+C_h)) \\ &+ (1-p_\bullet)(1-p_{off})(|\mathcal{L}(sh(f)) \cap \mathcal{U}|(C_{oprf}+C_h) + C_f + C_k) \\ &+ p_{off}(C_{oprf}+C_f+C_k), \end{aligned} \quad (3)$$

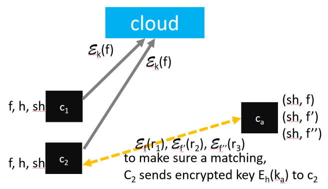
where C_h, C_{oprf}, C_f, and C_k are the computation overhead for the hash calculation, OPRF, chunk encryption, and key encryption, respectively.



(a) SP (**case 1** of Fig. 4a).

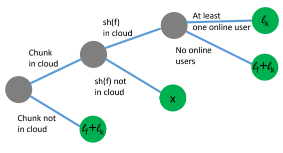


(b) SP (**case 2b** of Fig. 4a).



(c) SDedup.

Fig. 4: Our proposed XDedup solutions for data deduplication on encrypted data.



(a) $\mathbb{D}_{\text{SP}}$.

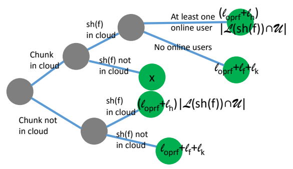

sh(f) in cloud
At least one online user
$(\ell_{oprf}+\ell_h)|\mathcal{L}(sh(f))\cap\mathcal{U}|$
Chunk in cloud
No online users
$\ell_{oprf}+\ell_f+\ell_k$
sh(f) not in cloud
x
Chunk not in cloud
sh(f) in cloud
$(\ell_{oprf}+\ell_h)|\mathcal{L}(sh(f))\cap\mathcal{U}|$
sh(f) not in cloud
$\ell_{oprf}+\ell_f+\ell_k$
(b) $\mathbb{T}_{\text{SP}}$.

Fig. 5: Overhead calculation in SP.

4) Protocol Transparency of SP: CSPs may be reluctant to modify the internal configuration of the cloud for providing stronger confidentiality, because of the engineering cost. Thus, the solutions for the encrypted deduplication only make little real impact. Nevertheless, with an additional software layer between $\{c_j\}$ and S, one can tackle this adaptiveness and applicability issues. Now, the difficulty in the design lies in how to leave S unchanged but still provide the deduplication functionality. Here, SP is inherently transparent in the sense that, S in fact only provides the query service on $\mathcal{L}$ and $\mathcal{U}$. As a consequence, we can move $\mathcal{L}$ and $\mathcal{U}$ to fully decentralized peer-to-peer (P2P) network (e.g., freenet) with the aim to have a public space for $\{c_j\}$ to query $\mathcal{L}$ and $\mathcal{U}$, achieving the protocol transparency.

5) Design Flaws in Performance and Security of SP: SP still suffers from the serious performance degradation, albeit the computing tasks are simplified. The root cause for the performance bottleneck lies in its extensive uses of OPRF. More specifically, in case 2a of Fig. 3, c_i has to perform OPRF with $|\mathcal{L}(sh(f)\cap\mathcal{U})|$ other users, causing considerable amount of computing tasks, in addition to the reliance on heavyweight cryptographic primitives.

Even worse, the more severe problem of SP is its security flaw. Since any user can be a member in $\mathcal{L}$, $\mathcal{L}$ can be manipulated maliciously, and all of the members in $\mathcal{L}$ are controlled by S (or S masquerades as users in $\mathcal{L}$), c_i communicates with the fake users, dramatically degrading the brute-force resiliency. Even if $\mathcal{L}$ and $\mathcal{U}$ are publicly available or maintained in a P2P fashion, the security can only be improved slightly because S can add a huge number of fake users in $\mathcal{L}$ and $\mathcal{U}$ to increase the probability for these fake users of being picked by c_i. The security goes worse because $\mathcal{L}$ is indexed by $sh(f)$. The fake user chosen by c_i actually affects not only the subsequent users who upload f but also all of users who upload f' with $sh(f') = sh(f)$. In the worst case, only $2^{\ell_{sh}}$ fake users are able to completely break the security of SP.

B. SDedup

1) Basic Idea of SDedup: Here, we present SDedup, the first brute-force resilient encrypted dedeuplication with the use of only symmetric cryptography, to the best of our knowledge. We make observations that PAKEDedup incurs significant performance degradation while SP is flawed in the design that every users c_a, even without the common $h(f)$, can participate in the key generation $OPRF[h(f), \mathbf{k}_a]$. Thus, the idea behind the design is to use Merkle puzzle [20] for checking whether two users share the same $h(f)$. In essence, Merkle puzzle, in place of asymmetrically cryptographic tools such as OPRF, PAKE and PHE, contributes to the performance speedup.

```
Offline Setting:
$S$ maintains a lookup table $\mathcal{L}$ of records $[sh(f), c_i]$
$S$ maintains a online user list $\mathcal{U}$
Online Execution:
01  $c_i \to S : sh(f)$
02  if $\mathcal{L}(sh(f)) = \emptyset$ (case 1)
03    $c_i$ picks a random key $k_f$
04    $c_i \to S : \mathcal{E}_{k_f}(f)$ and $\mathcal{E}_{\mathbf{k}_i}(k_f)$
05    $S$ adds a record $[sh(f), c_i]$ to $\mathcal{L}$
06  else
07    if $\mathcal{L}(sh(f)) \cap \mathcal{U} \neq \emptyset$ (case 2a)
08      for each $c_a \in \mathcal{L}(sh(f)) \cap \mathcal{U}$
09        for each $h(f_a^j)$ with $sh(f_a^j) = sh(f)$
10          $c_a$ picks a random number $r_a^j$
11          $c_a \to c_i : \mathcal{E}_{h(f_a^j)}(r_a^j)$
12          $c_i \to c_a : h_{h(f)}(\mathcal{D}_{h(f)}(\mathcal{E}_{h(f_a^j)}(r_a^j)))$
13          if $h_{h(f)}(r_a^j) = h_{h(f)}(\mathcal{D}_{h(f)}(\mathcal{E}_{h(f_a^j)}(r_a^j)))$
14            $c_a \to c_i : \mathcal{E}_{h(f)}(k_f)$
15            $c_i$ decrypts and derives $k_f$
16            $c_i \to S : \mathcal{E}_{\mathbf{k}_i}(k_f)$
17      if no $c_a \in \mathcal{L}(sh(f)) \cap \mathcal{U}$ causes dedup
18        $c_i$ picks a random key $k_f$
19        $c_i \to S : \mathcal{E}_{\mathbf{k}_i}(k_f)$ and $\mathcal{E}_{\mathbf{k}_i}(k_f)$
20        $S$ adds a record $[sh(f), c_i]$ to $\mathcal{L}$
21    else (case 2b)
22      $c_i$ picks a random key $k_f$
23      $c_i \to S : \mathcal{E}_{k_f}(f)$ and $\mathcal{E}_{\mathbf{k}_i}(k_f)$
24      $S$ adds a record $[sh(f), c_i]$ to $\mathcal{L}$
```

Fig. 6: The protocol description of SDedup.

2) Detailed Description of SDedup: The algorithmic description of SDedup is shown in Fig. 6. SDedup and SP share many settings; for example, users do not have direct communications. Furthermore, S maintains a publicly available online user list $\mathcal{U}$ and lookup table $\mathcal{L}$.

In SDedup, S still works like a broker who bridges c_i and the other users. In particular, if S receives $sh(f)$ from c_i but cannot find a match in $\mathcal{L}$ (**case 1** of Fig. 6), c_i simply picks a random chunk key k_f to encrypt f and uses $\mathbf{k}_i$ to encrypt k_f. Subsequently, c_i uploads $\mathcal{E}_{k_f}(f)$ and $\mathcal{E}_{\mathbf{k}_i}(k_f)$ to S, which then adds $[sh(f), c_i]$ to $\mathcal{L}$.

Consider the case, where S can find at least one online user c_a who possesses $sh(f)$ (**case 2a** of Fig. 6). There could be cases, where c_a does have $sh(f)$, but it just happens to have f' such that $sh(f') = sh(f)$ due to the high collision rate of short hash. Here, for each $h(f_a^j)$ in the memory with short hash $sh(f)$, each c_a sends *Merkle challenges* $\mathcal{E}_{h(f_a^j)}(r_a^j)$'s, where f_a^j is the jth file owned by c_a with $sh(f_a^j) = sh(f)$ and r_a^j is a random number picked by c_a for f_a^j, to c_i, which then replies the *Merkle responses* $h_{h(f)}(\mathcal{D}_{h(f)}(\mathcal{E}_{h(f_a^j)}(r_a^j)))$'s for each Merkle challenge. Once c_a finds the consistency between the received Merkle response $h_{h(f)}(\mathcal{D}_{h(f)}(\mathcal{E}_{h(f_a^j)}(r_a^j)))$ and hash result $h_{h(f)}(r_a^j)$ calculated by himself, c_a is confident that c_i is in possession of f. If so, c_a sends $\mathcal{E}_{h(f)}(k_f)$ to c_i, which then decrypts to derive k_f and uploads $\mathcal{E}_{k_f}(f)$ to S. Otherwise, since no online user can assist the key generation, this is similar to the case where no one uploads f before c_i. As a consequence, c_i proceeds the same actions as in case 1.

Consider the case, where S cannot find any online user in $\mathcal{L}(sh(f))$ (**case 2b** of Fig. 6). This is the same as case 1. Therefore, the procedures in case 1 also apply to case 2b. The conceptual illustration of case 2b is depicted in Fig. 4c.

3) Security of $\mathcal{L}$ and $\mathcal{U}$: Despite their similarity, our design in SDedup eliminates the vulnerability of SP. The client-side software is configured to send out heartbeat messages and check the existence of its user ID on $\mathcal{U}$ periodically, preventing it from being intentionally removed from $\mathcal{U}$. On the ther hand, though S can add fabricated users to $\mathcal{U}$, this finds harmless because, unless fake users happen to possess $h(f)$, S and fake users cannot know or manipulate k_f used by c_i. For example, consider case 2a of Fig. 4c, where c_i picks a fake user c_a without $h(f)$. c_a can always reports "duplicate found" to c_i, irrespective of the Merkle puzzles exchanged. So, c_a purposedly chooses a message z, claims $z = \mathcal{E}_{h(f)}(k_f)$ and sends z to c_i, which then performs the decryption $\mathcal{D}_{h(f)}(z)$. From the adversary point of view, since c_a is unaware of $h(f)$, even if z is chosen by c_a, $\mathcal{D}_{h(f)}(z)$ could be a random string for c_a. Hence, we claim that the fake users in $\mathcal{U}$ do not increase the adversary's knowledge gain. The above arguments can apply to $\mathcal{L}$ as well in a similar way. In short, the deletion of records in $\mathcal{L}$ only damages the possibility of deduplication, and the insertion of fake records in $\mathcal{L}$ has the same effect as in the insertion of fake users in $\mathcal{U}$. As a result, according to the above arguments, we ensure the security of $\mathcal{L}$ and $\mathcal{U}$.

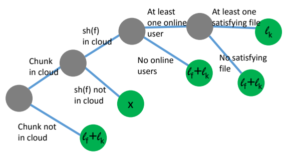
4) Protocol Transparency of SDedup: Due to the similarity to SP, SDedup is also inherently transparent in the sense that, S in fact only provides the query service on $\mathcal{L}$ and $\mathcal{U}$. As mentioned in Sec. III-B3, since the adversary cannot introduce arbitrary members in $\mathcal{L}$, the security of $\mathcal{L}$ and $\mathcal{U}$ is guaranteed. As a consequence, we can move $\mathcal{L}$ and $\mathcal{U}$ to fully decentralized peer-to-peer (P2P) network (e.g., freenet) with the aim to have a public space for $\{c_j\}$ to query $\mathcal{L}$ and $\mathcal{U}$, achieving the protocol transparency.

5) Performance Evaluation of SDedup: We follow the same strategy in Sec. III-A3 for calculating the overhead.

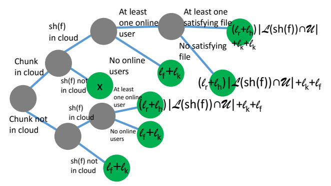
Deduplication Percentage $\mathbb{D}_{\text{SDedup}}$. The occupied spaces in different cases of uploading behaviors are shown in Fig. 7a. In particular, we can see from Figs. 6 and 7a that if $\mathcal{E}_{k_f}(f) \notin S$, c_i always needs to send $\mathcal{E}_{k_f}(f)$ to S. In contrast, given $\mathcal{E}_{k_f}(f) \in S$, whether c_i needs to send $\mathcal{E}_{k_f}(f)$ again depends on whether c_i can find an online *matching user* who uploads $\mathcal{E}_{k_f}(f)$ previously. Hence, $\mathbb{D}_{\text{SP}}$ can be formulated as

$$\mathbb{D}_{\text{SDedup}} = 1 - \frac{1}{\ell_f/(\alpha_1 + \alpha_2 + \alpha_3)}, \tag{4}$$

where $\alpha_1 = (1 - p_\bullet)(\ell_f + \ell_k)$, $\alpha_2 = p_\bullet p_{off}^{|\mathcal{L}(sh(f))|}(\ell_f + \ell_k)$, and $\alpha_3 = p_\bullet(1 - p_{off}^{|\mathcal{L}(sh(f))|})(1 - p_{own}^{|\mathcal{L}(sh(f)\cap\mathcal{U}|})$.



(a) $\mathbb{D}_{\text{SDedup}}$.



(b) $\mathbb{T}_{\text{SDedup}}$.

Fig. 7: Overhead calculation in SDedup.

Memory overhead $\mathbb{M}_{\text{SDedup}}$. $\mathbb{M}^S_{\text{SDedup}}$ is also formulated as $|\mathcal{L}| + |\mathcal{U}|$. On the other hand, users have to keep k_f's and $h(f)$'s, even after the chunk uploading. This can be attributed to the fact that the effectiveness of SDedup replies on the matching users performing Merkle puzzles on $\mathcal{E}_{h(f)}(r)$ and forwarding $\mathcal{E}_{h(f)}(k_f)$. Hence, we claim that $\mathbb{M}^c_{\text{SDedup}}$ consists of k_f's and $h(f)$'s for all of f's uploaded by herself.

Communication overhead $\mathbb{T}_{\text{SDedup}}$. The numbers of bits required in the message exchanges of SDedup are shown in Fig. 7b. One can see from Figs. 6 and 7b that if $sh(f)$ cannot be found in $\mathcal{L}$, then the uploading of $\mathcal{E}_{k_f}(f)$ and $\mathcal{E}_{\mathbf{k}_i}(k_f)$ is necessary. In fact, given that $sh(f)$ can be found in $\mathcal{L}$, if all of the matching users are offline, then the uploading of $\mathcal{E}_{k_f}(f)$ and $\mathcal{E}_{\mathbf{k}_i}(k_f)$ with k_f randomly picked by c_i is necessary. On the other hand, given that $sh(f)$ can be found in $\mathcal{L}$, if at least one matching user is online, then $|\mathcal{L}(sh(f) \cap \mathcal{U}|$ Merkle puzzles are exchanged. Here, we also assume $Pr[\mathcal{L}(sh(f)) \neq \emptyset | f \notin S] = 1$. The communication overhead $\mathbb{T}_{\text{SP}}$ of SP can be approximated as

$$\begin{aligned}\mathbb{T}_{\text{SP}} \approx\, & \ell_{sh} + p_\bullet(1 - p_{off})p_{own}(|\mathcal{L}(sh(f)) \cap \mathcal{U}|(\ell_r + \ell_h) + 2\ell_k) \\ & + p_\bullet(1 - p_{off})(1 - p_{own})(|\mathcal{L}(sh(f)) \cap \mathcal{U}|(\ell_r + \ell_h) + \ell_f + \ell_k) \\ & + (1 - p_\bullet)(1 - p_{off})(|\mathcal{L}(sh(f)) \cap \mathcal{U}|(\ell_r + \ell_h) + \ell_f + \ell_k) \\ & + p_\bullet p_{off}(\ell_f + \ell_k) + (1 - p_\bullet)p_{off}(\ell_f + \ell_k).\end{aligned} \tag{5}$$

Computation overhead $\mathbb{C}_{\text{SDedup}}$. Similarly, since the computing tasks are also associated with the corresponding communications, the computation overhead $\mathbb{C}_{\text{SP}}$ can be, in a form similar to $\mathbb{T}_{\mathrm{SP}}$, approximated as

$$\begin{aligned}\mathbb{C}_{\mathrm{SP}} \approx\ & C_{sh} + p_{\bullet}(1-p_{off})p_{own}(|\mathcal{L}(sh(f)) \cap \mathcal{U}|(C_r + C_h) + 2C_k) \\ & + p_{\bullet}(1-p_{off})(1-p_{own})(|\mathcal{L}(sh(f)) \cap \mathcal{U}|(C_r + C_h) + C_f + C_k) \\ & + (1-p_{\bullet})(1-p_{off})(|\mathcal{L}(sh(f)) \cap \mathcal{U}|(C_r + C_h) + C_f + C_k) \\ & + p_{\bullet}p_{off}(C_f + C_k) + (1-p_{\bullet})p_{off}(C_f + C_k). \qquad (6)\end{aligned}$$

C. XDedup

1) Basic Idea of XDedup: SDedup and PAKEDedup work under the CaS framework, which is helpful in eliminating the need of independent servers. Nevertheless, we find three common drawbacks shared by solutions under CaS framework. First, the online user status will be exposed due to the use of $\mathcal{U}$, reducing users' willingness in using the cloud storage services. Second, a tremendous amount of communications and computation efforts are required for the uploader and matching users to determine whether they share the same $h(f)$. Third, each user needs to keep $h(f)$ and k_f for each f uploaded by herself, imposing unnecessary overhead. In particular, the user who deletes $h(f)$ and k_f from the local memory either may be unable to perform duplicate check (e.g., PAKE in PAKEDedup and Merkle puzzle in SDedup) or incurs more communications and computations to retrieve them from S on demand.

Aiming to tackle these problems, we propose XDedup as the first brute-force resilient symmetrically encrypted data deduplication involving only the uploader and cloud. In particular, XDedup goes back to the simplest scenario, where the uploading and downloading of f rely solely on the interactions between c_i and S. Basically, the design of XDedup is similar to SDedup, except that the operations performed by matching users c_a's in SDedup are shifted to S.

2) Detailed Description of XDedup: The detailed description of XDedup is shown in Fig. 8. XDedup has its unique setting; S is assumed to maintain an extended lookup table $\mathcal{L}^+$. $\mathcal{L}^+$ in XDedup is similar to $\mathcal{L}$, is indexed by $sh(f)$, but contains more information (e.g., $h_{h(f)}(r)$, $\mathcal{E}_{h(f)}(r)$, $\mathcal{E}_{h(f)}(k_f)$) for duplicate checks via Merkle puzzle and key exchange. Nevertheless, S here does not need to maintain $\mathcal{U}$, keeping the user online status private.

In XDedup, after receiving $sh(f)$ from c_i, S looks for a match in $\mathcal{L}^+$ (**case 1** of Fig. 8). The case of $\mathcal{L}^+(sh(f)) = \emptyset$, where $\mathcal{L}^+(sh(f))$ returns a set of 3-tuples of the form $[h_{h(f)}(r), \mathcal{E}_{h(f)}(r), \mathcal{E}_{h(f)}(k_f)]$, and $\mathcal{L}^+(sh(f))[i]$ denotes the ith element of $\mathcal{L}^+(sh(f))$, implies that no $\mathcal{E}_{k_f}(f)$ corresponding to $sh(f)$ has been uploaded previously. Thus, c_i simply picks a random chunk key k_f to encrypt f and uses $\mathbf{k}_i$ to encrypt k_f. Subsequently, c_i uploads $\mathcal{E}_{k_f}(f)$ and $\mathcal{E}_{\mathbf{k}_i}(k_f)$, and the necessary materials such as $h_{h(f)}(r)$, $\mathcal{E}_{h(f)}(r)$, $\mathcal{E}_{h(f)}(k_f)$ to S. Here, the former two items $h_{h(f)}(r)$ and $\mathcal{E}_{h(f)}(r)$ are particularly for Merkle puzzle used to make sure whether c_i has the same f, while the last item $\mathcal{E}_{h(f)}(k_f)$ is used to make sure c_i with f can derive k_f. The conceptual illustration of case 1 of XDedup is shown in Fig. 9a.

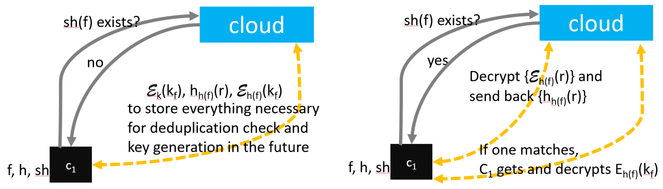
Consider the case of $\mathcal{L}^+(sh(f)) \neq \emptyset$, where S can find at least one match of $sh(f)$ in $\mathcal{L}^+$ (case 2 of Fig. 8). S extracts and sends all of the ciphertexts $\mathcal{E}_{h(f^j)}(r^j)$'s from $\mathcal{L}^+(sh(f))[2]$ to c_i, where f^j denotes the jth possibility of f with the same $sh(f)$ and r^j is a random number for f^j. For each received Merkle challenge, c_i performs the decryption and then hash calculation, both with $h(f)$ as the key. After that, c_i replies Merkle responses to S. Once S finds the consistency between the received Merkle response $h_{h(f)}(\mathcal{D}_{h(f)}(\mathcal{E}_{h(f^j)}(r^j)))$ and the Merkle response $h_{h(f^j)}(r^j)$ kept in the memory, S has confidence that the deduplication can take place on $\mathcal{E}_{k_f}(f)$. If so, S sends $\mathcal{E}_{h(f)}(k_f)$ to c_i, which then decrypts to derive k_f and uploads $\mathcal{E}_{k_f}(f)$ to S. Otherwise, this is equivalent to the case, where all of $\mathcal{E}_{k_f}(f)$'s in S happen to have short hash $sh(f)$ and no one uploads $\mathcal{E}_{k_f}(f)$ previously. Hence, c_i uploads $\mathcal{E}_{k_f}(f)$ as in case 1. The conceptual illustration of case 2 of XDedup is shown in Fig. 9b. It is worthy to note that XDedup can achieve perfect deduplication (see Sec. II-B) because c_i with f can always receive $\mathcal{E}_{h(f)}(r)$ and the duplicate, if any, can always be detected.

3) Performance Evaluation of XDedup: Here, we also conduct the same strategy in Sec. III-B to calculate the overhead.

Deduplication Percentage $\mathbb{D}_{\mathrm{XDedup}}$. The occupied spaces in different cases of the uploading behaviors in XDedup are

```
Offline Setting:
$S$ maintains a lookup table $\mathcal{L}^+$
Online Execution:
$c_i \rightarrow S : sh(f)$
if $\mathcal{L}^+(sh(f)) = \emptyset$ (case 1)
    $c_i$ picks a random key $k_f$ and a random value $r$
    $c_i \rightarrow S : h_{h(f)}(r)$, $\mathcal{E}_{h(f)}(r)$, and $\mathcal{E}_{h(f)}(k_f)$
    $\mathcal{L}^+ = \mathcal{L}^+ \cup [sh(f), \langle h_{h(f)}(r), \mathcal{E}_{h(f)}(r), \mathcal{E}_{h(f)}(k_f) \rangle]$
    $c_i \rightarrow S : \mathcal{E}_{k_f}(f)$ and $\mathcal{E}_{\mathbf{k}_i}(k_f)$
else (case 2)
    $S \rightarrow c_i : \{\mathcal{E}_{h(f^j)}(r^j)\}_{\mathcal{E}_{h(f^j)}(r^j) \in \mathcal{L}^+(sh(f))[2]}$
    $c_i \rightarrow S : \{h_{h(f)}(\mathcal{D}_{h(f)}(\mathcal{E}_{h(f^j)}(r^j)))\}$
    if $\exists \pi$ s.t. $h_{h(f^\pi)}(r^\pi) = h_{h(f)}(\mathcal{D}_{h(f)}(\mathcal{E}_{h(f^\pi)}(r^\pi)))$
        $S \rightarrow c_i : \mathcal{E}_{h(f^\pi)}(k_f^\pi)$
        $c_i$ obtains $k_f$ by calculating $\mathcal{D}_{h(f)}(\mathcal{E}_{h(f^j)}(k_f^\pi)))$
        $c_i \rightarrow S : \mathcal{E}_{\mathbf{k}_i}(k_f)$
    else
        $c_i$ picks a random key $k_f$ and a random value $r$
        $c_i \rightarrow S : h_{h(f)}(r)$, $\mathcal{E}_{h(f)}(r)$, and $\mathcal{E}_{h(f)}(k_f)$
        $\mathcal{L}^+ = \mathcal{L}^+ \cup [sh(f), h_{h(f)}(r), \mathcal{E}_{h(f)}(r), \mathcal{E}_{h(f)}(k_f)]$
        $c_i \rightarrow S : \mathcal{E}_{k_f}(f)$ and $\mathcal{E}_{\mathbf{k}_i}(k_f)$
```

Fig. 8: The protocol description of XDedup.

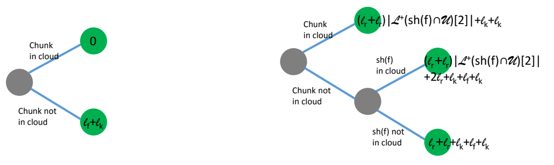

sh(f) exists?
cloud
no
$\mathcal{E}_k(k_f)$, $h_{h(f)}(r)$, $\mathcal{E}_{h(f)}(k_f)$ to store everything necessary for deduplication check and key generation in the future
f, h, sh
c_1
sh(f) exists?
cloud
yes
Decrypt $\{\mathcal{E}_{h(f)}(r)\}$ and send back $\{h_{h(f)}(r)\}$
If one matches, C_1 gets and decrypts $E_{h(f)}(k_f)$
f, h, sh
c_1

(a) XDedup (negative response). (b) XDedup (positive response).

Fig. 9: Our proposed XDedup solution.

shown in Fig. 10a. In particular, we can see from Figs. 8 and 10a that if $\mathcal{E}_{k_f}(f) \notin S$, c_i needs to send $\mathcal{E}_{k_f}(f)$ to S, while if $\mathcal{E}_{k_f}(f) \in S$, the uploading of $\mathcal{E}_{k_f}(f)$ can always be omitted. As a result, $\mathbb{D}_{\text{XDedup}}$ can be formulated as

$$\mathbb{D}_{\text{XDedup}} = 1 - \frac{1}{\ell_f/(((1-p_\bullet)(\ell_f+\ell_k)))}. \tag{7}$$


Chunk in cloud
0
Chunk not in cloud
$\ell_f+\ell_k$
Chunk in cloud
$(\ell_r+\ell_r)|\mathcal{L}^+(sh(f)\cap\mathcal{U})[2]|+\ell_k+\ell_k$
sh(f) in cloud
$(\ell_r+\ell_r)|\mathcal{L}^+(sh(f)\cap\mathcal{U})[2]|+2\ell_r+\ell_k+\ell_f+\ell_k$
Chunk not in cloud
sh(f) not in cloud
$\ell_r+\ell_r+\ell_k+\ell_f+\ell_k$

(a) $\mathbb{D}_{\text{XDedup}}$. (b) $\mathbb{T}_{\text{XDedup}}$.

Fig. 10: Overhead calculation in XDedup.

Memory overhead $\mathbb{M}_{\text{XDedup}}$. Since $\mathcal{E}_{k_f}(f)$ and $\mathcal{E}_{\mathbf{k}_i}(k_f)$ are necessary among all of the solutions, we do not consider them as overhead. In this sense, $\mathbb{M}^S_{\text{XDedup}} = |\mathcal{L}^+|$ only. On the other hand, since only c_i communicates with S, c_i does not keep information after finishing the uploading. Hence, we claim that the memory overhead at the user side, $\mathbb{M}^c_{\text{SP}}$, is zero.

Communication overhead $\mathbb{T}_{\text{XDedup}}$. The numbers of bits required in the message exchanges of XDedup are shown in Fig. 10b. One can see from Figs. 8 and 10b that in the case of $\mathcal{L}(sh(f)) = \emptyset$, c_i has to spend $(2\ell_r + 2\ell_k + \ell_f)$ bits to upload $\mathcal{E}_{k_f}(f)$, $\mathcal{E}_{\mathbf{k}_i}(k_f)$, and the required information in $\mathcal{L}^+$. In contrast, given $\mathcal{L}(sh(f)) \neq \emptyset$, as $|\mathcal{L}^+(sh(f))[2]|$ Merkle puzzles are inevitably needed, the exact number of bits required in the communication depends on whether the chunk is in S. Here, to ease the calculation, we still assume $Pr[\mathcal{L}(sh(f)) \neq \emptyset | f \notin S] = 1$ and $Pr[\mathcal{L}(sh(f)) = \emptyset | f \notin S] = 0$. The communication overhead $\mathbb{T}_{\text{SP}}$ of SP can be approximated as

$$\begin{aligned}&\mathbb{T}_{\text{XDedup}} = p_\bullet((\ell_r+\ell_r)|\mathcal{L}^+(sh(f))[2]| + \ell_k + \ell_k) \\ &+ (1-p_\bullet)((\ell_r+\ell_r)|\mathcal{L}^+(sh(f))[2]| + 2\ell_r + 2\ell_k + \ell_f).\end{aligned} \tag{8}$$

Computation overhead $\mathbb{C}_{\text{XDedup}}$. Similarly, the computation overhead $\mathbb{C}_{\text{SP}}$ can be approximated as

$$\begin{aligned}&\mathbb{C}_{\text{XDedup}} = p_\bullet((C_r + C_r)|\mathcal{L}^+(sh(f))[2]| + C_k + C_k) \\ &(1-p_\bullet)((C_r+C_r)|\mathcal{L}^+(sh(f))[2]| + 2C_r + C_k + C_f + C_k).\end{aligned} \tag{9}$$

D. Implementation Issues

We consider two minor implementation issues in this section. In particular, we examine the online brute-force resiliency in Sec. III-D1 and the overhead reduction based on data characteristics in Sec. III-D2.

1) Online Brute-Force Resiliency: The online brute-force vulnerability is due to the nature of the deduplicated storage in the sense that the adversary can always know the duplicate check result and then infer the sensitive content by repeatedly making queries on candidate chunks. Below we adopt two heuristic approaches to counteract the online brute-force attack.

Rate limiting. The rate limiting approach has been used by [3], [14] to resist online brute-force attack. In particular, we consider per-file rate limiting [14] and adapt it to be per-chunk rate limiting and fit in our context. The rationale behind the design is to ensure that the uncertainty of a predictable chunk is larger than the number of duplicate checks applied on the potential online users. Let RL_a, RL_i, and RL_S be the rate limits for $\{c_a\}$, c_i, and S, respectively. Let m and x be the min-entropy of f and the number of users who potentially possess f, respectively. The above notion cab be instantiated as the constraints $2^m > 2^{\ell_{sh}} x(RL_i + RL_a)$ in SDedup and $2^m > 2^{\ell_{sh}}(RL_i + RL_S)$ in XDedup. Despite its online brute-force resiliency, such a defense actually sacrifices the deduplication effectiveness. This can be attributed to the fact that the uploading of a highly popular chunk may easily exceed the rate limit, resulting in the un-deduplicatable chunk.

User Unawareness of duplicate check result. We find that the online brute-force stems from the duplicate result awareness of the uploader c_i. Thus, once the deduplication system is designed such that c_i is unaware of duplicate result result, one can avoid the online brute-force attack. We also find that the side channel prevention in deduplicated storage and our brute-force resiliency design actually share the same objective. Thus, the existing solutions in side channel prevention [13], where each chunk f is associated a random deduplication threshold t_f and a counter c_f that indicates the number of copies in S, can be used in SDedup and XDedup to enhance their online brute-force resiliency. Specifically, for the uploading request of f, a duplicate is detected if $c_f \geq t_f$ and is undetected otherwise. The use of t_f's, to some extent, obfuscates the duplicate result. Nevertheless, this approach shares similar downside with rate limiting; in the case of $c_f < t_f$, actually this approach resist the online brute-force by sacrificing dedeuplication effectiveness.

2) Overhead Reduction via Rate Limiting: The uploader c_i in PAKEDedup, in theory, needs to communicate with a large number of c_a's to derive k_f. Nevertheless, in practice, via simulations, Liu et al. [14] demonstrate that only a few (e.g., two) PAKE runs suffice to derive k_f with high probability. Thus, rate limiting constraint can be strict. The reason behind the surprising result is that the real world data usually follows power law distribution (a.k.a., Zipf distribution) such that most of the uploading requests for files that have already been uploaded can find a matched file within the rate limit. The performance of SDedup and XDedup can also be benefited by taking advantage of Zipf data distribution. In particular, in SDedup and XDedup, we inherently assume that all of Merkle challenges are sent to c_i at once. Now, c_a in SDedup and S in XDedup instead send Merkle challenges to c_i based on descending order of chunk popularity. Since chunk popularity is Zipf distributed, sending Merkle challenges in this way ensures that popular uploaded chunks have a much higher likelihood of being selected and thus deduplicated, achieving the same benefit of overhead reduction.

IV. Security Analysis

Inspired by [14], below we formally prove the security of SDedup and XDedup via simulation-based approach (or say, ideal/real paradigm) widely used in theoretical cryptography community [15]. The general strategy of ideal/real paradigm is to define ideal and real models, where ideal world leaks no privacy. If one can prove that these two models are computationally indistinguishable, then one may claim the security of real world. In the following, we follow the above general strategy to prove the security of SDedup and XDedup.

A. Security of SDedup

Recall that we consider malicious model. We define the ideal functionality $\mathcal{F}_{dedup[3^+]}$ of deduplicating encrypted data in a multiparty protocol in Fig. 11. The term "3^+" in $\mathcal{F}_{dedup[3^+]}$ stresses that there are three or more participants. Three types of roles, uploader c_i, cloud S, and matching users $\{c_a\}$, participate in the protocol. Here, if SDedup can implements $\mathcal{F}_{dedup[3^+]}$, we claim that SDedup leaks no information about f and k_f, and only S and c_i know duplicate check result. Theorem 1 shows that SDedup can implements $\mathcal{F}_{dedup[3^+]}$.

Input:
(1) The uploader c_i has a chunk f;
(2) Each online user c_a has inputs f_a and k_{f_a};
(3) S's input is empty.
Output:
(1) c_i leans chunk existence status.
c_i derives a chunk key k_f for f. If $f = f_a$ for some online user c_a, then $k_f = k_{f_a}$. Otherwise, c_i generates a random k_f;
(2) Each c_a knows whether $f = f_a$;
(3) S obtains $\mathcal{E}_{k_f}(f)$ if $\mathcal{E}_{k_f}(f) \notin S$ and obtains nothing otherwise. Moreover, S learns the chunk existence status of $\mathcal{E}_{k_f}(f)$; if $\mathcal{E}_{k_f}(f)$ already has a copy $\mathcal{E}_{k_{f_j}}(f_j)$ in S, S learns the index j.

Fig. 11: The ideal functionality $\mathcal{F}_{dedup[3^+]}$.

Theorem 1. *SDedup implements $\mathcal{F}_{dedup[3^+]}$ with security against malicious adversaries, if the encryption $\mathcal{E}_k(\cdot)$ is semantically secure, the hash function $h(\cdot)$ and MAC $h_k(\cdot)$ are modeled as a random oracle.*

Proof: (Sketch) Our objective is to show the execution of SDedup in the real model is computationally indistinguishable from the execution of $\mathcal{F}_{dedup[3^+]}$ in the ideal model. Our proof strategy is to construct a simulator in the ideal model that can obtain messages that the corrupt parties would generate or send in the real model. Such a simulator is aimed to generate a message transcript ($IDEAL$) during the ideal model execution and ensures that it is computationally indistinguishable from the ($REAL$) collected during the real model execution.

Corrupt c_i: We first construct a simulator for c_i given honest S and $\{c_a\}$. The simulator operates as follows. The simulator records the calls that c_i makes to hash functions and the recorded tuples are of the form $\{(f, h(f), sh(f))\}$. On receiving sh from c_i, the simulator chooses a set of users $U = \{c_{b_1}, \ldots, c_{b_{|U|}}\}$ with $c_i \notin U$, pretending to be chosen users to generate random numbers $r_{b_1}, \ldots, r_{b_{|U|}}$ and sending $A = \{\mathcal{E}_k(r_{b_1}), \ldots, \mathcal{E}_k(r_{b_{|U|}})\}$ with a random key k, to c_i. With the knowledge of $h(f)$ from c_i, the simulator sends $B = \{h_{h(f)}(r_{b_1}), \ldots, h_{h(f)}(r_{b_{|U|}})\}$ to $\{c_a\}$. Now, the simulator invokes $\mathcal{F}_{dedup[3^+]}$ with f. From $\mathcal{F}_{dedup[3^+]}$, the simulator either obtains k_f and knows "duplicated detected," or know "no duplicate." After that, the simulator prepares $\{h(\mathcal{E}_k(r_{b_1})), \ldots, h(\mathcal{E}_k(r_{b_{|U|}})\}$. Moreover, the simulator sends $C = \mathcal{E}_{h(f)}(k_f)$ to c_i if "duplicated detected" from $\mathcal{F}_{dedup[3^+]}$ and sends an empty string as C to c_i otherwise. After the above construction, the transcript $IDEAL_{c_i} = \langle A, B, C\rangle$ with corrupt c_i and $REAL_{c_i} = \langle\{\mathcal{E}_{h(f)}(r), h_{h(f)}(\mathcal{D}_{h(f)}(\mathcal{E}_{h(f)}(r)))\}, \mathcal{E}_{h(f)}(k_f)\rangle$ are identically distributed.

Corrupt S: The simulator chooses a set of users $U = \{c_{b_1}, \ldots, c_{b_{|U|}}\}$ with $c_i \notin U$ and pretends to be chosen users to generate and send random numbers $A = \{\mathcal{E}_k(r_{b_1}), \ldots, \mathcal{E}_k(r_{b_{|U|}})\}$ with a random key as k and random numbers as r_{b_i}'s, to c_i. The simulator sends $B = \{h(r_{b_1}), \ldots, h(r_{b_{|U|}})\}$ to $\{c_a\}$. Then, the simulator invokes $\mathcal{F}_{dedup[3^+]}$ and obtains index j if f has a copy in S and obtains $\mathcal{E}_{k_f}(f)$ otherwise. Then, the simulator sends a random string as C to c_i. It sends another random string of length ℓ_f as D to S if $c_j \notin U$ and sends an empty string as D if $c_j \in U$. After the above construction, $IDEAL_S = \langle A, B, C, D\rangle$ and $REAL_S = \langle\mathcal{E}_{h(f)}(r), h_{h(f)}(\mathcal{D}_{h(f)}(\mathcal{E}_{h(f)}(r))), \mathcal{E}_{h(f)}(k_f), \mathcal{E}_{k_f}(f)\rangle$ are identically distributed.

Corrupt $\{c_a\}$: The simulator, on behalf of $\{c_a\}$, generates and sends random numbers $A = \{\mathcal{E}_k(r_{b_1}), \ldots, \mathcal{E}_k(r_{b_{|U|}})\}$ with k as a random key and r_{b_i}'s are random numbers, to c_i. The simulator sends $B = \{h(r_{b_1}), \ldots, h(r_{b_{|U|}})\}$ to $\{c_a\}$. Then, the simulator invokes $\mathcal{F}_{dedup[3^+]}$ and obtains the information that either $f_a = f$ or $f_a = f$. If $f_a = f$, the simulator sends a random string as C to c_i. If $f_a \neq f$, $\{c_a\}$ does nothing. After the above construction, $IDEAL_{\{c_a\}} = \langle A, B, C\rangle$ and $REAL_{\{c_a\}} = \langle\mathcal{E}_{h(f)}(r), h_{h(f)}(\mathcal{D}_{h(f)}(\mathcal{E}_{h(f)}(r))).\mathcal{E}_{k_f}(f)\rangle$ are identically distributed.

A Collusion of Corrupt c_i and Corrupt S: The simulation is similar to the case of a corrupt S, except that S is able to know k_f after invoking $\mathcal{F}_{dedup[3^+]}$. Therefore, the simulation begins as the proof of a corrupt S and the simulator extracts c_i's input f. Then it invokes $\mathcal{F}_{dedup[3^+]}$ with input f. With the knowledge of f and k_f, the simulator can easily perform the same operations as in the case of corrupt c_i or S.

A Collusion of Corrupt S and Corrupt $\{c_a\}$: The simulation is similar to the case of a corrupted uploader, except that S might choose a subset of $\{c_a\}$. □

B. Security of XDedup

Recall that we consider malicious model. We define the ideal functionality $\mathcal{F}_{dedup[2]}$ of deduplicating encrypted data in a multiparty protocol in Fig. 12. Two types of roles, uploader c_i and cloud S participate in the protocol. Here, if XDedup can implements $\mathcal{F}_{dedup[2]}$, we claim that XDedup achieves the same security as $\mathcal{F}_{dedup[2]}$.

Theorem 2. *XDedup implements $\mathcal{F}_{dedup[2]}$ with security against malicious adversaries, if the encryption $\mathcal{E}_k(\cdot)$ is se-*

Input:

(1) The uploader c_i has a chunk f;

(2) S's input is empty.

Output:

(1) c_i learns chunk existence status.
c_i derives a chunk key k_f for f. If $f = f_a$, then $k_f = k_{f_a}$. Otherwise, c_i generates a random k_f;

(2) S obtains $\mathcal{E}_{k_f}(f)$ if $\mathcal{E}_{k_f}(f) \notin S$.
Moreover, S learns the chunk existence status of $\mathcal{E}_{k_f}(f)$; if $\mathcal{E}_{k_f}(f)$ already has a copy $\mathcal{E}_{k_{f_j}}(f_j)$ in S, S learns the index j.

Fig. 12: The ideal functionality $\mathcal{F}_{dedup[2]}$.

mantically secure, the hash function $h(\cdot)$ and MAC $h_k(\cdot)$ are modeled as a random oracle.

Proof: (Sketch) We are aimed to show the execution of XDedup in the real model is computationally indistinguishable from the execution of $\mathcal{F}_{dedup[2]}$ in the ideal model. Similarly, our proof strategy is to construct a simulator in the ideal model that can obtain messages that the corrupt parties would generate or send in the real model. Such a simulator is aimed to generate a message transcript ($IDEAL$) during the ideal model execution and ensures that it is computationally indistinguishable from the ($REAL$) collected during the real model execution.

Corrupt c_i: The simulator chooses a random number z and sends $A = \{\mathcal{E}_{r'_1}(r''_1), \ldots, \mathcal{E}_{r'_z}(r''_z)\}$ to c_i, where r'_i's and r''_i, $1 \leq i \leq z$ are random numbers. The simulator invokes $\mathcal{F}_{dedup[2]}$ with f. It either knows "dedeuplication occurs" and obtains k_f, or knows "dedeuplication not occurs." The simulator prepares and sends $B = \{h(\mathcal{E}_{r'_1}(r''_1)), \ldots, h(\mathcal{E}_{r'_z}(r''_z))\}$ to S. With $h(f)$ from corrupt c_i and k_f from $\mathcal{F}_{dedup[2]}$, the simulator sends $C = \mathcal{E}_{h(f)}(k_f)$ to c_i if dedeuplication occurs and sends nothing as C to c_i otherwise. Afterwards, the simulator sends $D = \mathcal{E}_{k_f}(f)$ to S if dedeuplication occurs and sends a random string as D to S otherwise. After the above construction, the transcript $IDEAL_{c_i} = \langle A, B, C, D\rangle$ with corrupt c_i and $REAL_{c_i} = \langle\{\mathcal{E}_{h(f)}(r), h_{h(f)}(\mathcal{D}_{h(f)}(\mathcal{E}_{h(f)}(r)))\}, \mathcal{E}_{h(f)}(k_f), \mathcal{E}_{k_f}(f)\rangle$ are identically distributed.

Corrupt S: The simulator sends a random string of length ℓ_{sh} to S. Then, the simulator chooses a random number z and sends $A = \{\mathcal{E}_{r'_1}(r''_1), \ldots, \mathcal{E}_{r'_z}(r''_z)\}$ to c_i, where r'_i's and r''_i, $1 \leq i \leq z$ are random numbers. The simulator prepares and sends $B = \{h(\mathcal{E}_{r'_1}(r''_1)), \ldots, h(\mathcal{E}_{r'_z}(r''_z))\}$ to S. The simulator invokes $\mathcal{F}_{dedup[2]}$. It either knows "dedeuplication occurs" and an index j, or "dedeuplication not occurs" and $\mathcal{E}_{k_f}(f)$. The simulator sends a random string of length ℓ_k as C to c_i if "dedeuplication occurs" and sends an empty string as C to c_i otherwise. The simulator sends $D = \mathcal{E}_{k_f}(f)$ obtained from $\mathcal{F}_{dedup[2]}$ to S if "dedeuplication not occurs" and an empty string as D to S otherwise. After the above construction, the transcript $IDEAL_{c_i} = \langle A, B, C, D\rangle$ with corrupt c_i and $REAL_{c_i} = \langle\{\mathcal{E}_{h(f)}(r), h_{h(f)}(\mathcal{D}_{h(f)}(\mathcal{E}_{h(f)}(r)))\}, \mathcal{E}_{h(f)}(k_f), \mathcal{E}_{k_f}(f)\rangle$ are identically distributed. □

V. Numerical Simulations

In this section, we conducted numerical simulations to evaluate the performance of our proposed SDedup and XDedup solutions, based on the deduplication percentage and communication cost described in Sec. II-B. The memory overhead and computation overhead are skipped, because the former is related to the protocol configuration in a straightforward manner while the computation shares the similar pattern to the communication.

During our numerical simulations, the number $|\mathcal{L}(sh(f))|$ of matching users in $\mathcal{L}$ is fixed to be 4. We inherently assume the use of AES as the only encryption in the encryption deduplication; no public key cryptography is used. In addition, SHA512 is used as the cryptographic hash function. As a consequence, we set $\ell_h = 512$. The length ℓ_{sh} of short hash is fixed to be 32. The lengths ℓ_k and ℓ_r of the (encrypted) chunk keys and (encrypted) random numbers are both configured to be 512. The finite field in OPRF calculation is assumed to be of length 1024 bits and therefore ℓ_{oprf} is set to be 2048 for simplicity. In what follows, we consider how parameters ℓ_f, p_{off}, and p_{own} have impact on the deduplication percentage and communication cost of the proposed deduplication schemes. Note that we show the communication cost in terms of bandwidth saving; the ratios $\mathbb{T}_X/\ell_f$ are shown in Figs. 14, 16, and 18 to put the emphasis on the communication reduction, compared to the case, where no deduplication is used. In this section, the overhead reduction via rate limiting in Sec. III-D2 is not considered.

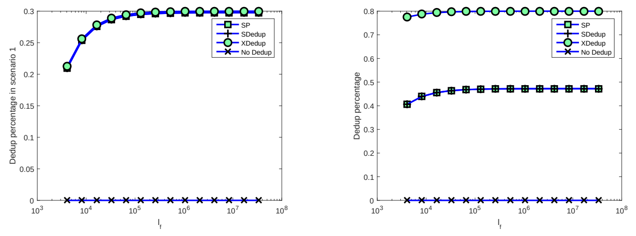
A. Impact of ℓ_f

In Figs. 13 and 14, we consider ℓ_f from 2^{12} to 2^{25} bits. One can see from Fig. 13 that lower ℓ_f has negative impact on the deduplication percentage. This can be attributed to the fact that all of the encrypted deduplication schemes involves storing encrypted chunk keys and other indexes (e.g., $\mathcal{L}$ and $\mathcal{L}^+$). Therefore, once ℓ_f is small, the additional effort for storing encrypted chunk keys and other indexes becomes burdensome, reducing the storage gain from deduplication.

The communication cost exhibits the similar behavior; one can easily see from Fig. 14 that lower ℓ_f has adverse impact on the communication cost. In addition, a closer look at Figs. 14a and 14b reveals the different communication gains from SP and XDedup. In Fig. 14a, where users are often online, since SP spend less additional exchanged bits, SP's communication gain outperforms XDedup's one. However, in the case where users are not always online, though XDedup needs to add more materials to $\mathcal{L}^+$, XDedup's communication gain turns to be superior to SP's communication gain.

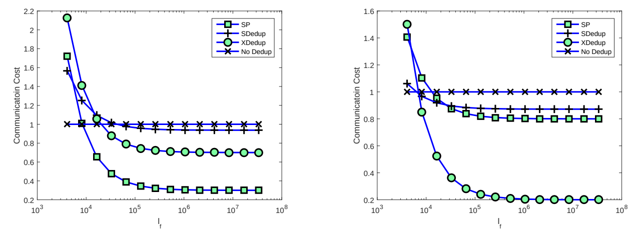
B. Impact of p_{off}

Figs. 15 and 16 show how p_{off} affects the deduplication percentage and communication cost. The result is, in fact, straightforward; since the deduplication effectiveness of SP and SDedup heavily relies on the help from online users, less



(a) $p_\bullet = p_{off} = p_{own} = 0.3$ (b) $p_\bullet = p_{off} = p_{own} = 0.8$

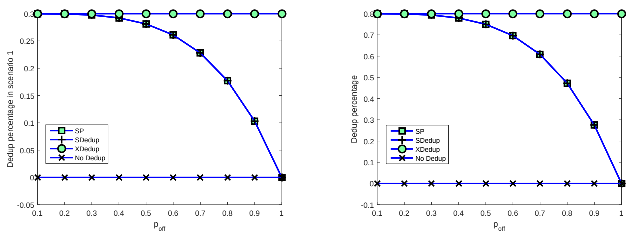
Fig. 13: Dedup percentage with varying ℓ_f.



(a) $p_\bullet = p_{off} = p_{own} = 0.3$ (b) $p_\bullet = p_{off} = p_{own} = 0.8$

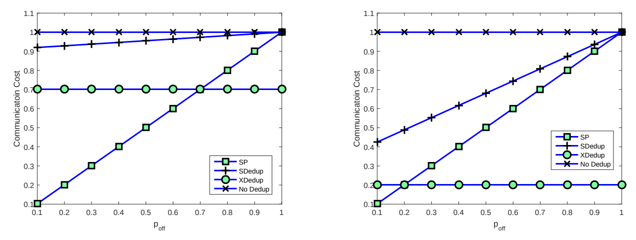
Fig. 14: Communication cost with varying ℓ_f.

online users can simply be translated to the worse deduplication percentage and communication cost. In particular, in the extreme case of no online user ($p_{off} = 1$), both Figs. 15 and 16 show that SP and SDedup degenerate to the case of no deduplication, in both the deduplication percentage and communication cost.



(a) $\ell_f = 2^{22}$, $p_\bullet = p_{own} = 0.3$ (b) $\ell_f = 2^{22}$, $p_\bullet = p_{own} = 0.8$

Fig. 15: Dedup percentage with varying p_{off}.



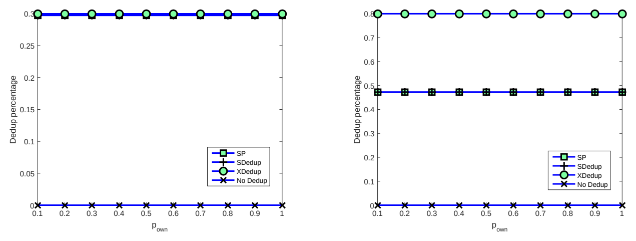
(a) $\ell_f = 2^{22}$, $p_\bullet = p_{own} = 0.3$ (b) $\ell_f = 2^{22}$, $p_\bullet = p_{own} = 0.8$

Fig. 16: Communication cost with varying p_{off}.

C. *Impact of* p_{own}

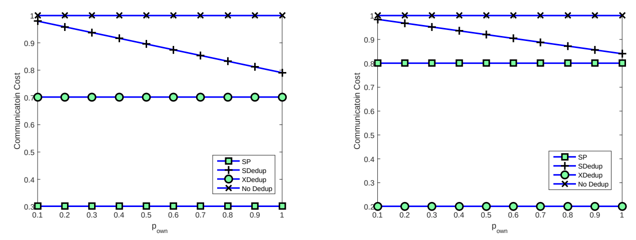
Obviously, only the performance of SDedup will be influenced by different p_{own}'s. Surprisingly, we observe from Fig. 17 that the $\mathbb{D}_{\text{SDedup}}$ remains seemingly stable with varying p_{own}'s. In fact, different p_{own}'s has impact on $\mathbb{D}_{\text{SDedup}}$ (see α_3 of Eq. 4); however, since $\mathbb{D}_{\text{SDedup}}$ is dominated by α_1 and α_2 of Eq. 4, the differences of deduplication percentages among different p_{own}'s become vague here.

On the contrary, different p_{own}'s contribute the considerable impact to $\mathbb{T}_{\text{SDedup}}$, shown in Fig. 18. One can see from Figs. 18a and 18b that the communication cost of SDedup is declining with the increased p_{own}.



(a) $\ell_f = 2^{22}$, $p_\bullet = p_{off} = 0.3$ (b) $\ell_f = 2^{22}$, $p_\bullet = p_{off} = 0.8$

Fig. 17: Dedup percentage with varying p_{own}.



(a) $\ell_f = 2^{22}$, $p_\bullet = p_{off} = 0.3$ (b) $\ell_f = 2^{22}$, $p_\bullet = p_{off} = 0.8$

Fig. 18: Communication cost with varying p_{own}.

D. *Discussion*

From the above numerical simulations, we find that XDedup serves as the most stable encrypted deduplication, in terms of both its security and performance guarantees under different system settings. In particular, if the chunk size is more than 2^{17} bits[‡], XDedup could be the choice with first priority. Moreover, XDedup could be the best choice when users in the underlying deduplication system are frequently disconnected.

Nevertheless, XDedup has its weakness in offering protocol transparency (see. Sec. I-D). As the cloud storage backend is required to be reconfigured in XDedup, we see from Table I that only few solutions (e.g., CE, MLE, DupLESS, and ClouDedup) satisfy the protocol transparency requirement. Since users can enjoy the storage gain without compromising the data privacy in encrypted deduplication with protocol transparency, despite its inferiority in additional privacy leakage and deduplication effectiveness, SDedup still proves its value in the transient state toward the full support of encrypted deduplication on cloud storages.

[‡]The chunk size in Dropbox client-side software is 4MB (2^{25} bits).

VI. Conclusion

This paper proposes SDedup and XDedup schemes. Particularly, XDedup serves as the first encrypted data deduplication with only symmetrically cryptographic two-party interactions. The security has been proved rigorously via ideal/real paradigm. We also demonstrate the great performance of the proposed solutions via the analysis and numerical simulations. All of the salient features prove the practicality of our proposed SDedup and XDedup solutions.

Acknowledgment

The preliminary version of this paper was accepted as a 3-page poster paper in *ACM Conference on Computer and Communications Security (CCS)*, 2016 [34]. The differences of [34] and this paper are : 1) more detailed descriptions of related work and design challenges in Sec. I, 2) the description of system model in Sec. II, 3) two more protocols in Sec. III-A and Sec. III-B, 4) two formal cryptographic security proofs in Sec. IV, and 5) a number of numerical simulations in Sec. V. This research was partly funded by the Ministry of Science and Technology of the R.O.C. under grants MOST 104-2628-E-155-001-MY2, 103-2221-E-197-018, 105-2218-E-155-010, 105-2923-E-002-014-MY3, and 101-2221-E-197-008-MY3. This research was also partly supported by CAS project 2015TW2GA0002 and NSFC project U1505251.

References

[1] M. Abadi, D. Boneh, I. Mironov, A. Raghunathan, and G. Se. Message-locked encryption for lock-dependent messages. *International Cryptology Conference (CRYPTO)*, 2013.

[2] P. Anderson and L. Zhang. Fast and secure laptop backups with encrypted de-duplication. *USENIX Large Installation Systems Administration Conference (LISA)*, 2010.

[3] M. Bellare and S. Keelveedhi. DupLESS server-aided encryption for deduplicated storage. *USENIX Security Symposium*, 2013.

[4] M. Bellare and S. Keelveedhi. Interactive message-locked encryption and secure deduplication. *Pubic-Key Cryptography (PKC)*, 2015.

[5] M. Bellare, S. Keelveedhi, and T. Ristenpart. Message-locked encryption and secure deduplication. *Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2013.

[6] S. M. Bellovin and M. Merritt. Encrypted key exchange: password-based protocols secure against dictionary attacks. *IEEE Symposium on Research in Security and Privacy*, 1992.

[7] Y. Duan. Distributed key generation for encrypted deduplication: Achieving the strongest privacy. *ACM Cloud Computing Security Workshop (CCSW)*, 2014.

[8] J. R. Douceur, A. Adya, W. J. Bolosky, D. Simon, M. Theimer. Reclaiming space from duplicate files in a serverless distributed file system. *IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2002.

[9] M. Dutch, *Understanding data de-duplication ratios*, SNIA, 2008.

[10] S. Halevi, D. Harnik, B. Pinkas, and A. Shulman-Peleg. Proofs of ownership in remote storage systems. *ACM conference on Computer and Communications Security (CCS)*, 2011.

[11] J. Hur, D. Koo, Y. Shin, and K. Kang. Secure data deduplication with dynamic ownership management in cloud storage. *IEEE Transactions on Knowledge and Data Engineering*. (to appear)

[12] H. Hovhannisyan, K. Lu, R. Yang, W. Qi, J. Wang, and M. Wen. A novel deduplication-based covert channel in cloud storage service. *IEEE Blobal Communications Conference (Globecom)*, 2015.

[13] D. Harnik, B. Pinkas, and A. Shulman-Peleg. Side channels in cloud services, the case of deduplication in cloud storage. *IEEE Security and Privacy*, vol. 8, no. 6, pp. 40-47, 2011.

[14] J. Liu, N. Asokan, and B. Pinkas. Secure deduplication of encrypted data without additional independent servers. *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[15] Y. Lindell. How to simulate it - A tutorial on the simulation proof technique. IACR Cryptology ePrint Archive 2016: 46.

[16] J. Li, X. Chen, M. Li, J. Li, P. P. C. Lee, and W. Lou. Secure deduplication with efficient and reliable convergent key management, *IEEE Transactions on Parallel and Distributed System*s, vol. 25, no. 6, pp. 1615-1625, 2014.

[17] J. Li, Y. K. Li, X. Chen, P. P. C. Lee, and W. Lou. A hybrid cloud approach for secure authorized deduplication, *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 55, pp. 1206-1216, 2015.

[18] M. Li, C. Qin, and P. P. C. Lee. CDStore: toward reliable, secure, and cost-efficient cloud storage via convergent dispersal. *USENIX Annual Technical Conference (ATC)*, 2015.

[19] J. Li, C. Qin, P. P. C. Lee, and J. Li. Rekeying for encrypted deduplication storage. *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2016.

[20] R. C. Merkle. Secure communications over insecure channels. *Communications of the ACM*, vol. 21, no. 4, pp. 294-299, 1978.

[21] P. Meye, P. Raipin, F. Tronel, and E. Anceaume. A secure two-phase data deduplication scheme. *International Symposium on Cyberspace Safety and Security (CSS)*, 2014.

[22] M. Mulazzani, S. Schrittwieser, M. Leithner, and M. Huber. Dark clouds on the horizon: using cloud storage as attack vector and online slack space. *USENIX Security Symposium*, 2012.

[23] M. Naor and O. Reingold. Number-theoretic constructions of efficient pseudo-random functions. *IEEE Symposium on Foundations of Computer Science (FOCS)*, 1997.

[24] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. *International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 1999.

[25] P. Puzio, R. Molva, M. Önen, S. Loureiro. ClouDedup: secure deduplication with encrypted data for cloud storage. *IEEE Conference on Cloud Computing Technology and Science (CloudCom)*, 2013.

[26] P. Puzio, R. Molva, M. Önen, S. Loureiro. PerfectDedup: secure data deduplication. *DPM International Workshop on Data Privacy Management (DPM)*, 2015.

[27] V. Rabotka and M. Mannan. An evaluation of recent secure deduplication proposals. *Journal of Information Security and Applications*, vol. 27, no. C, pp. 3-18, 2016.

[28] M. W. Storer, K. Greenan, D. D. E. Long, and E. L. Miller. Secure data deduplication. *ACM international workshop on Storage security and survivability (StorageSS)*, 2008.

[29] J. Stanek, A. Sorniotti, E. Androulaki, and L Kenel. A secure data deduplication scheme for cloud storage. *Financial Cryptography (FC)*, 2014.

[30] Tahoe-LAFS. `https://www.tahoe-lafs.org/trac/tahoe-lafs`.

[31] H. Tang, Y. Cui, C. Guan, J. Wu, J. Weng, K. Ren. Enabling ciphertext deduplication for secure cloud storage and access control. *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2016.

[32] M. Wen, K. Ota, H. Li, J. Lei, C. Gu, and Z. Su. Secure data deduplication with reliable key management for dynamic updates in cpss. *IEEE Transactions on Computational Social Systems*, 2015.

[33] J. Xia, E.-C. Chang, and J. Zhou. Weak leakage-resilient client-side deduplication of encrypted data in cloud storage. *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2013.

[34] C.-M. Yu. Efficient Cross-User Chunk-Level Client-Side Data Deduplication with Symmetrically Encrypted Two-Party Interactions. *ACM Symposium Computer and Communications Security (CCS)*, 2016. (poster)

[35] Z. Yan, W. Ding, X. Yu, H. Zhu, and R. H. Deng. Deduplication on encrypted big data in cloud. *IEEE Transactions on Big Data*, vol. 2, no. 2, pp. 138-150, 2016.

[36] Y. Zhou, D. Feng, W. Xia, M. Fu, F. Huang, Y. Zhang, C. Li. SecDep: a user-aware efficient fine-grained secure deduplication scheme with multi-level key management. *IEEE Conference on Mass Storage Systems and Technologies (MSST)*, 2015.

[37] Y. Zheng, X. Yuan, X. Wang, J. Jiang, C. Wang, and X. Gui. Enabling encrypted cloud media center with secure deduplication. *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2015.