POSTER: Efficient Cross-User Chunk-Level Client-Side Data Deduplication with Symmetrically Encrypted Two-Party Interactions

Chia-Mu Yu (chiamuyu@nchu.edu.tw)
Department of Computer Science and Engineering, National Chung Hsing University, Taiwan

ABSTRACT

Data deduplication 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. Here, we propose an encrypted deduplication scheme, XDedup, 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. XDedup also achieves perfect deduplication.

1. INTRODUCTION

1.1 Privacy Concerns in 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. In particular, 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.

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.

Unfortunately, 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.

CCS'16 October 24-28, 2016, Vienna, Austria

ACM ISBN 978-1-4503-4139-4/16/10.

DOI: http://dx.doi.org/10.1145/2976749.2989047

1.2 Related Work

Convergent Encryption. Convergent encryption (CE) is the simplest way for tackling the privacy concern without compromising the deduplication effectiveness. 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 users with f are all able to derive the same $h(f)$ and $\mathcal{E}_{h(f)}(f)$, the deduplication still takes place on $\mathcal{E}_{h(f)}(f)$. CE can be generalized as message-locked encryption (MLE), which achieves remarkable speedup in the encryption calculation.

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 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.

To counter against the brute-force attack, based on the idea of additional randomness, Bellare and Keelveedhi present DupLESS [1], where an additional key server KS is introduced to assist the key generation. More specifically, 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) 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 $\mathcal{E}_{k_f}(f)$.

Encrypted Deduplication w/o Independent Servers. Albeit key server helps generate k_f for users, 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) [2] claims to eliminate the need for a key server. Nonetheless, the dealers in EwS serve as the similar role of key server [4]. Very recently, Liu et al. [3] propose PAKEDedup based on a client-as-a-key-server (CaS) framework to dedeuplicate encrypted data by using password authenticated key agreement (PAKE) and partially homomorphic encryption (PHE). 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.

1.3 Design Challenges

Existing solutions are all subject to certain performance, security, and applicability limitations.

Brute-Force Resiliency (L1). CE, MLE, and their variants all involve the uploading of $h(f)$ and $\mathcal{E}_{h(f)}(f)$. Nevertheless, real world applications usually generate low min-entropy contents f, making the *offline brute-force attack* easy to find out f.

Table 1: COMPARISONS AMONG SOLUTIONS

Methods	(L1)	(L2)	(L3)	(L4)	(L5)
CE and MLE	–	✓	✓	✓	✓
DupLESS [1]	✓	–	–	✓	✓
EwS [2]	✓	–	–	✓	✓
PAKEDedup [3]	✓	✓	–	✓	–
XDedup (this paper)	✓	✓	✓	✓	✓

Independent Server Assumption (L2). In spite of the differences in their functionalities, most of previous solutions 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*, 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 [3].

Complicated Computation and Architecture (L3). Some of the current methods involve complicated arithmetics or architectures. The corresponding 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.

Heuristic Parameter Setting (L4). For some heuristic approaches, choosing an explicit threshold by using file popularity, file sizes, and user privileges to differentiate the data sensitivity would be difficult. Moreover, heuristic approaches also have restriction on application scenarios.

Additional Privacy Leakage (L5). 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.

2. XDEDUP

CaS framework is helpful in eliminating the need of independent servers. Nevertheless, we find three drawbacks under CaS framework. First, the online user status will be exposed, 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 *chunk hash* $h(f)$ and *chunk key* k_f for each f uploaded by herself, imposing unnecessary overhead.

Aiming to tackle these problems, we propose XDedup as the first brute-force resilient symmetrically encrypted data deduplication involving only the uploader c_i and cloud S. 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. A comparison among different schemes is shown in Table 1.

2.1 Detailed Description of XDedup

In XDedup (see Fig. 1), S is assumed to maintain an extended lookup table $\mathcal{L}^+$. $\mathcal{L}^+$ is indexed by $sh(f)$ and 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.

The setting is an uploader c_i attempting to upload a low min-entropy chunk f. This c_i has access to $h(f)$ and truncated hash

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 $\mathcal{E}_{\mathbf{k}_i}(k_f)$

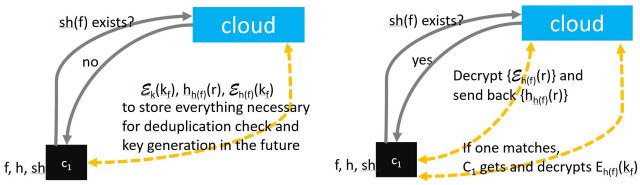
Figure 1: The protocol description of XDedup.

(also called *short hash*) $sh(f)$. Short hash $sh(f)$ can be implemented by keeping only partial bits of $h(f)$ and has high collision rate so that the adversary cannot be confident that it is the specific f that implies $sh(f)$, mitigating the brute-force threat.

In XDedup, after receiving $sh(f)$ from c_i, S looks for a match in $\mathcal{L}^+$ (**case 1** of Fig. 1). 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. 2a.



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

Figure 2: Our proposed XDedup solution.

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. 1). 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

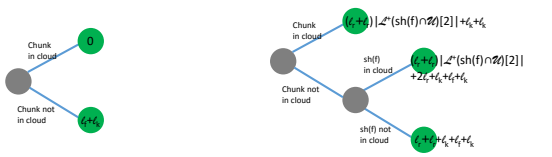
with $h(f)$ as the key. After that, c_i replies Merkle responses to S. Once S finds the consistency between the received Merkle response $h_{h(f)}(\mathcal{D}_{h(f)}(\mathcal{E}_{h(f^j)}(r^j)))$ and the Merkle response $h_{h(f^j)}(r^j)$ kept in the memory, S has confidence that the deduplication can 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. 2b. It is worthy to note that XDedup achieves perfect deduplication because c_i with f can always receive $\mathcal{E}_{h(f)}(r)$ and the duplicate can always be detected.

2.2 Performance Evaluation of XDedup

We consider the scenario, where c_i uploads a random distinct chunk to S for evaluating the expected performance.

Deduplication Percentage $\mathbb{D}_{\text{XDedup}}$. The occupied spaces in different cases of the uploading behaviors in XDedup are shown in Fig. 3a. In particular, we can see from Figs. 1 and 3a 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

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



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

Figure 3: 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, the memory overhead at the server side, $\mathbb{M}^S_{\text{XDedup}} = |\mathcal{L}^+|$ 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, $\mathbb{M}^{\mathcal{C}}_{\text{SP}}$, is zero.

Communication overhead $\mathbb{T}_{\text{XDedup}}$. The numbers of bits required in the message exchanges of XDedup are shown in Fig. 3b. One can see from Figs. 1 and 3b 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

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

2.3 Implementation Issues

2.3.1 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. We adopt two heuristic approaches to counteract the online brute-force attack.

Rate limiting. The rate limiting approach has been used by [1,3] to resist online brute-force attack. In particular, we consider per-file rate limiting [3] 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, 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 XDedup to enhance 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.

2.3.2 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. [3] 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 XDedup can also be benefited by taking advantage of Zipf data distribution. In particular, in XDedup, we inherently assume that all of Merkle challenges are sent to c_i at once. Now, S in XDedup instead sends 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.

3. REFERENCES

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

[2] Y. Duan. Distributed key generation for encrypted deduplication: Achieving the strongest privacy. *ACM CCSW*, 2014.

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

[4] Y. Zheng, X. Yuan, X. Wang, J. Jiang, C. Wang, and X. Gui. Enabling encrypted cloud media center with secure deduplication. *ACM ASIACCS*, 2015.