A Bandwidth-Efficient Middleware for Encrypted Deduplication

Helei Cui*†, Cong Wang*†, Yu Hua‡, Yuefeng Du*, and Xingliang Yuan§
*Department of Computer Science, City University of Hong Kong, Hong Kong, China
†City University of Hong Kong Shenzhen Research Institute, Shenzhen, 518057, China
‡School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China
§Faculty of Information Technology, Monash University, Melbourne, Australia
{helei.cui, yf.du}@my.cityu.edu.hk, congwang@cityu.edu.hk, csyhua@hust.edu.cn, xingliang.yuan@monash.edu

***Abstract*—Data deduplication is a vital component of current cloud storage systems for optimized space utilization. However, users cannot fairly enjoy the storage savings of deduplication. Uploading two identical files consumes twice the storage quota from a user's account, but the server may store one file copy only. In this paper, we design and implement a middleware system, namely UWare. It brings storage and bandwidth savings back to users, while preserving user data privacy. UWare starts from the message-locked encryption for efficient deduplication over encrypted data, and initiates the endeavor in leveraging the similarity characteristics of block-level deduplication to balance the effectiveness of secure deduplication and system efficiency. Also, UWare patches a practically feasible side-channel threat when deploying the proof-of-ownership protocol, i.e., hiding the existence of a target file during the protocol execution. We implement a prototype and use a real-world dataset to demonstrate that UWare can save about 30% storage and bandwidth cost for users, and reduce over 80% memory space consumption compared to the secure block-level deduplication.**

I. Introduction

Data deduplication, as an effective redundancy elimination technique, not only improves storage utilization but also alleviates network transmission [1]. While this technique has been widely adopted on the server side, most of the existing cloud storage services, such as Dropbox, OneDrive, and Google Drive, still charge users regardless of the number of duplicated copies to maximize their profits [2]. Hence in reality, users cannot fairly enjoy the benefits of server-side deduplication, either within their own folders or across each other's [3].

To eliminate this unfair cost on behalf of users, dedicated deduplication services, as a middleware deployed between clients and cloud storage servers, are desired to detect duplicated files before actually uploading them to the cloud. However, introducing this kind of middleware would bring more privacy concerns on users' data, as it has to scan the incoming files for deduplication purpose.

To further address such concerns, a mandatory requirement is to adopt *secure deduplication* techniques, which aims to ensure data privacy during deduplication and has been intensively studied in the literature [4], [5]. Apart from those studies on the primitive formulation, such as Message-Locked Encryption (MLE) [6], Block-Level MLE (BL-MLE) [7], and Updatable block-level MLE [8], there are various specialized system designs, of which the deduplication occurs at either server side (e.g., DupLESS [9]) or client side (e.g., ClearBox [3] and SecDep [10]). Compared with the server-side designs that can only save storage, in the client-side designs, the user sends a deterministically-derived tag (e.g., a hash value) of the file to the server for duplicate checking and later uploads the non-duplicated ones. As the transmission of redundant copies is avoided, both the storage and bandwidth can be saved. If such secure client-side deduplication is applied here, our envisioned deduplication middleware, as well as the backend storage server, would be able to process encrypted data, while bringing the benefits of storage and bandwidth savings to users.

978-1-5386-5541-2/18/$31.00 ©2018 IEEE

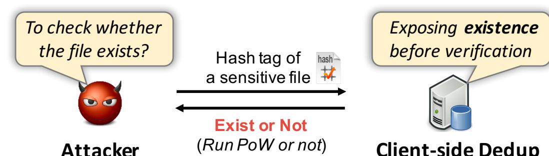


Fig. 1. An overlooked side-channel in prior secure client-side deduplication schemes with the adoption of proof-of-ownership protocol.

Despite promising, existing client-side designs are not directly applicable to build a secure deduplication middleware, as they are inherently vulnerable to the *ownership cheating attacks* [11], a.k.a., hash-only attacks [5]. That is, the client-side deduplication allows using a piece of short information (e.g., the hash tag) to determine and retrieve a duplicated file, which could be abused by an attacker who does not hold the actual file but attempts to cheat ownership of that file [11]. To prevent such attacks, it is essential to involve an additional step to verify that a user indeed holds a claimed file. Particularly, the proof-of-ownership (PoW) protocol, as one common approach, has been incorporated into many secure client-side deduplication systems, e.g., [3], [7], [12], [13].

An overlooked side-channel in client-side deduplication. It is recently recognized that the PoW can also be abused to turn the deduplication server into an oracle [5], allowing an attacker to learn the file existence by observing whether or not the PoW testing is performed. Therefore, this overlooked side-channel is vulnerable to the *existence-of-file attacks* [14] without holding the actual file, as shown in Fig. 1. However, to the best of our knowledge, no secure client-side dedupli-

cation systems considered this when integrating the PoW into their deduplication procedure. And such unauthorized privacy leakage must be patched in our middleware design.

Tradeoff among various deduplication modes. Most of prior secure deduplication systems [3], [9], [12], [15]–[17] perform file-level deduplication. The major downside of these systems is the rather low deduplication ratio. To achieve a higher ratio of deduplication, block-oriented deduplication is naturally favored. Yet this inevitably coerces additional overhead upon memory and bandwidth [7], [10], [18]. Our experiment also verifies that processing at the block level slows down the overall deduplication procedures by orders of magnitude compared with the file-level approach (see Table II in Section IV-B). The contradiction between file- and block-level approaches urges a solution that acquires an acceptable deduplication ratio while alleviating performance issues.

This work. We propose UWare, a secure client-side deduplication middleware system:

To enable efficient deduplication over encrypted data, UWare applies MLE [6] as its encryption scheme and slightly modifies the key generation procedure by involving some randomnesses, similar to [12], [13]. It preserves the functionality of data deduplication while protecting data confidentiality against UWare and the backend cloud storage server.

To patch the aforementioned side-channel issue, UWare makes the file existence oblivious throughout the PoW testing. Unlike prior systems [3], [7], [12], [13] that indicate the file existence before running the PoW protocol, UWare hides the existence information by always responding a random PoW challenge and delays the deduplication result till the completion of the PoW protocol. In this way, the user knows nothing about the file existence by using its hash only.

To balance the deduplication effectiveness and efficiency, UWare initiates an endeavor in leveraging the similarity characteristic of file blocks (i.e., a small group of sampled block tags can approximately represent the whole file [19]) in secure deduplication. The tailored similarity-based deduplication mode reduces the memory consumption significantly with a little but acceptable loss of deduplication ratio. This also minimizes the necessary leakage in the block-level deduplication, which does not reveal the equality among all the blocks [20]. That is, only the identical blocks of similar files are learned, while the identical blocks of dissimilar files are encrypted into totally different ciphertext blocks.

To summarize, UWare has the following features:

- **Enhanced security.** UWare enables cross-user client-side deduplication over encrypted data, so as to bring deduplication benefits of storage and bandwidth savings to the client. It patches a practically feasible side-channel overlooked in the deployment of the PoW protocol. A user who does not hold the file will not be aware of its existence in the server.
- **Synergized system.** UWare systematically incorporates MLE, PoW, and near-exact deduplication together. Especially, the tailored similarity-based deduplication mode is tunable to balance the effectiveness (in terms of deduplication ratio) and efficiency (in terms of required memory space and bandwidth).

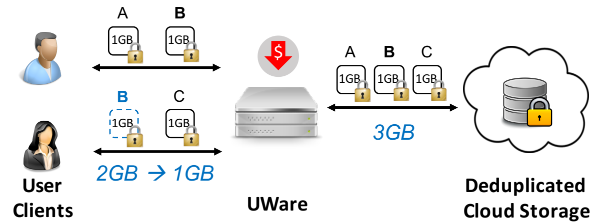


Fig. 2. Usage scenario of UWare system.

- **Balanced performance.** We implement a prototype, test it at AWS EC2, and publish the code on GitHub[1]. Extensive evaluations show that UWare saves over 30% bandwidth when uploading the whole dataset. Compared with the block-level mode, UWare shrinks over 80% of the index size, and shortens the average processing time from 25.9 ms to 14.7 ms for large files, with only a loss of 10% in the deduplication ratio.

II. System Overview

UWare is designed to be an imminent and easy-to-be-deployed solution to save cloud storage fees for users with enhanced defensive mechanisms against hash-only existence-of-file attacks. Compatible with popular cloud storage services, UWare will perform secure deduplication services at the gateway of enterprise-level networks.

Fig. 2 shows the usage scenario of UWare. Three parties are involved: user client (C), storage server (S), and secure deduplication middleware UWare. Specifically, C is installed at a local device, helping the user to encrypt and upload files to S for storage purpose. S stores all users' data, which is maintained by a public cloud storage service provider like AWS or Azure. As a middleware between C and S, UWare indexes short information (e.g., hash tags for duplicate checking and some random values for PoW testing) of encrypted files/blocks of all users for efficiency, and enables Cs to further save cloud-storage cost via deduplication within the enterprise-level networks. Moreover, UWare defends against threats directly caused by conveniently accessible short information for deduplication, such as cheating ownership [11], identifying file existence [14] via hash, and poisoning stored data [12].

A. Preliminaries

- **Message-locked encryption** (MLE) [6] is a cryptographic primitive formalized for ensuring data confidentiality in secure data deduplication, where the ciphertexts of unpredictable messages cannot be distinguished by an efficient attacker except with negligible probability. In MLE, the same data always result in identical tags for the use of duplicate checking, where the ciphertext could be randomized in some constructions, e.g., randomized convergent encryption (RCE) [6].
- **Proof-of-ownership** (PoW) is an essential component in client-side deduplication systems, where a verifier (i.e., UWare in our system) confirms that a prover (C) indeed holds a claimed file/block. In the literature, there are a line of studies [3], [7], [11]–[13] adopting PoW protocols under different threat models. In this work, we comply with a

[1] Source code: https://github.com/harrycui/UWare

strong model proposed by Halevi et al. [11], where multi-time leakage of data could exist before/after an execution of PoW protocol. A weaker model means that a bounded amount of information of data could be leaked only before PoW starts [12], [13]. As introduced later in Section III-A, UWare patches the PoW protocol and can address the aforementioned file-existence side-channel under both threat models.

B. Threats

We consider the two following adversaries in UWare:

• **A malicious user.** It refers to a user who attempts to launch the ownership cheating attacks [11] or the existence-of-file attacks [21] by using some short information about a file (e.g., hash tag) obtained via certain public channels [11]. Also, it refers to a user who utilizes a correct tag for duplicate checking but uploads a fake ciphertext (that is not consistent with the tag) to compromise the integrity of other users' files, so-called duplicate faking attacks [6] or poison attacks [12].

• **A compromised cloud storage server.** A stronger attacker may compromise S and attempt to steal and learn the underlying content of the stored file ciphertexts.

C. Assumptions

Like previous work (e.g., [3], [9], [13]), we assume that UWare and S are both *honest-but-curious*. They faithfully perform the designated secure deduplication and storage services. Particularly, UWare will not collude with S to compromise data confidentiality. This is practical in the deployment of existing deduplication services [17], [22], as UWare and S maintained by independent service providers are normally not co-located. We are aware that such attacks can be mitigated by either resorting to an additional independent party (for embedding another private key) [9] or involving a group of online users (for obliviously sharing the data encryption key) [15] (see more discussions in Section III-D).

UWare does not hide the equality of processed files or blocks, owing to the intrinsic essence of deduplication that it is unavoidable to know whether the target file/block is duplicated [9], [15]. However, the confidentiality of each uploaded file should always be protected with MLE. In the meanwhile, UWare tries to minimize the necessary leakage when leveraging file similarity characteristic for deduplication (see details in Section III-C).

It is worth mentioning that other side-channel attacks based on timing analysis (e.g., measuring the operation time (or the response time) made by UWare) and traffic analysis (e.g., observing whether deduplication occurs by using an actual file) as well as file-injection attacks (e.g., uploading two similar files) are orthogonal to our UWare design rationale. Note that some traffic obfuscation approaches can mitigate these attacks by randomizing the operation delays and traffic volumes but inevitably sacrifice deduplication performance [4], [23], [24].

III. Our Proposed Design

Adhering to the principle of benefiting users with storage and bandwidth savings, UWare offers various modes of secure data deduplication services.

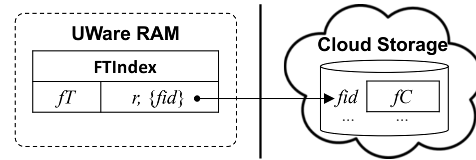


Fig. 3. Secure file-level dedup in UWare.

A. UWare Operation at File Level

For ease of exposition, we begin with the most straightforward mode, i.e., secure file-level deduplication. UWare is built on top of MLE because it is the most efficient cryptographic primitive known so far for deduplication over encrypted data [9]. Yet, only adopting MLE cannot address the threats in our target client-side deduplication middleware. A malicious user could use a piece of short information about a file (e.g., the file hash) to cheat UWare the ownership of that file [11], [12] or identify its existence [14].

The ownership cheating attack has been addressed by incorporating a PoW protocol in prior systems [3], [7], [12], [13]. However, as mentioned, the *existence-of-file* attack is still feasible due to the overlooked side-channel in the deployment of their PoW protocols. Namely, an attacker who has a file hash can know if it exists without having to complete the PoW testing. We emphasize that such attack is meaningful in practice. For instance, if the target file is highly sensitive or too huge to be carried out, it cannot be provided to the employee (or a third-party agency) to launch file-existence detection, e.g., mass surveillance via a hash database [25].

We emphasize that the reason for this side-channel leakage is that existing deduplication designs perform the PoW testing after the (initial) duplicate checking passes. The PoW protocol will be skipped if the duplicate checking fails, i.e., no file hash is presented. Therefore, to patch this, a promising solution is to make the file existence oblivious unless the user (who holds the actual file) passes the PoW testing. That means no matter the initial duplicate checking is successful or not, a PoW challenge will be always returned. And the user should not be able to infer the duplicate checking result based on the challenge message in PoW protocol.

Our modified deduplication procedure. As shown in Table I, UWare first checks whether the incoming file tag fT (computed via a cryptographic hash function $\texttt{Hash}(f)$, e.g., SHA256) exists in the index, called FTIndex (see Fig. 3). If fT is not present in FTIndex, UWare knows that the file f has not been uploaded, and returns two fresh randomnesses for r and r^*. Otherwise, UWare returns the previously used r in FTIndex together with a one-time randomness r^*. Here, r is used for the key generation, ensuring that the same file can always produce the same ciphertext, while r^* is for the proof generation, verifying that the user indeed holds the claimed file f even in a strong threat model [11], i.e., the previous proof fCT (file ciphertext tag) can be leaked.

Upon receiving the challenge (r, r^*), the C computes the file encryption key fK via $\texttt{Hash}(r, f)$. Next, it encrypts f via a deterministic symmetric encryption scheme (e.g., CTR mode using AES with a fixed IV [9]), i.e., $fC \leftarrow \texttt{Enc}(fK, f)$. Then

TABLE I

AN EXEMPLARY SERVICE FLOW OF UWARE IN FILE-LEVEL DEDUPLICATION MODE.

User Client		**UWare**
1. Generate file tag fT from a file f	$\xrightarrow{fT}$	2. Perform initial checking:
3. Generate file key fK by using f and r 4. Encrypt f with fK	$\xleftarrow{r, r^*}$	– If fT exists: return the previously used r and a fresh r^* – Else: return fresh randomness r and r^*
Our patch: no matter initial checking successes or not, the challenge message (r, r^*) is sent back to enforce PoW execution.		
5. Generate ciphertext tag fCT^* as the proof by jointly using the ciphertext fC and r^*	$\xrightarrow{fCT^*}$	6. Perform final verification: – If fT exists in Step 2 and fCT^* matches (i.e., passing PoW): return $True$ – Else: return $False$
7. Upload operation: – If $True$: skip upload – Else: upload ciphertext fC	$\xleftarrow{True/False}$ $\xrightarrow{fC}$	8. Update metadata, and store fC at cloud storage

it computes the file ciphertext tag $fCT^* \leftarrow \texttt{Hash}(r^*, fC)$, and sends back fCT^* as the proof to UWare.

At last, an additional verification is performed by UWare:

i. If the fT was found in FTIndex in initial checking, UWare checks whether the proof fCT^* is correct by computing it with the r^* and the corresponding fC[2]. If the proof matches, UWare returns the fid and $True$ to the C, and directly appends the C on the owner list of the file ciphertext fC.

ii. Otherwise, UWare assigns a new fid' to this incoming file, and returns the fid' and $False$. Later, UWare will update FTIndex with $[fT, (r, \{fid'\})]$.

Remark. Different from prior secure client-side schemes [3], [7], [12], [13], our design hides the file existence and delays the result notification (i.e., $True/False$) until the completion of PoW. The user who just holds a file hash is still requested to upload the corresponding file ciphertext fC as she cannot compute the correct proof. Thus, she does not know the existence of that file. Note that the randomness r^* can be removed, if one adopts a weaker threat model, i.e., allowing a bounded amount one-time leakage of a file (e.g., file hash) before running PoW. In this case, the file ciphertext fC is directly used for generating the proof, i.e., $fCT^* \leftarrow \texttt{Hash}(fC)$.

Addressing duplicate faking attack. Once a user (i.e., either the initial uploader or the subsequent uploader) launches the *duplicate faking attack* [6] to tamper with the file integrity, there would be a special case that the fT was found in FTIndex but the received proof (i.e., fCT^*) is incorrect. When this happens, UWare will perform as if the fT was not found in FTIndex, assign a new fid', and update FTIndex by adding fid' into the ID list of the fT. Since all unmatched ciphertexts are requested to upload, the correct ciphertexts will not be affected by those dummy ones.

B. UWare Operation at Block Level

To achieve more storage savings, dividing a file into blocks and exploiting data redundancy at the block level is a promising approach [1], [18]. Thus, we now consider how to support block-level deduplication in UWare. A common practice is to treat each block as an independent file (e.g., [15], [17], [26]). So the basic element in duplicate checking and data encryption is a block, where the block size could be either fixed or variable in different settings [18].

Performance issues on memory space and bandwidth. Although the deduplication ratio significantly increases compared with the file-level mode, e.g., from 5% to 40% in our experiment, directly adopting the above file-level design into the block-level mode still faces crucial performance issues. With the growing volume of the files, the amount of metadata for secure duplication (e.g., block tags) would be huge [7]. For example, given 100 TB unique files with 4 KB average block size (supposing no duplicated blocks), at least 800 GB of the block tags (e.g., via SHA256) would be kept in the index without considering other metadata (e.g., file/block IDs and the randomness r for key generation). Regarding the bandwidth, the number of returned challenge messages (i.e., $\{r, r^*\}$) after the initial checking step would increase linearly to the block numbers, which could a considerable cost when facing a large file. To alleviate these issues, we will elaborate a tunable design, aiming to balance the deduplication performance and system efficiency for secure deduplication.

C. UWare Operation at Dual Level with File Similarity

Among different deduplication strategies in the plaintext domain, near-exact deduplication achieves lower memory cost with comparable deduplication ratio to the original block-level deduplication [18]. Its success comes from the Broder's theorem [19], i.e., the similarity of the two randomly sampled subsets is an unbiased approximation of that of the two complete sets [18]. In other words, a small group of sampled block tags can approximately represent the entire block tags of the file. Thus, it is possible to leverage such similarity characteristics to narrow down the similar files for higher deduplication performance and reduce the memory cost [18].

Based on the above methodology, UWare can use the sampled RP ("RePresentative") tags to quickly narrow down the similar files [18], and then perform pairwise block-level checking between the incoming file and similar files. It also needs another similarity-based inverted index, namely RPTIndex, where the key is each RP tag rpT and the value is a list of file IDs $\{fid\}$ of which the files share the same rpT.

[2]For efficiency, UWare can pre-compute a number of "challenge-proof pairs", i.e., $\{r_i^*, fCT_i\}$. Once they are used up, UWare fetches the fC back from S and prepare other new pairs.

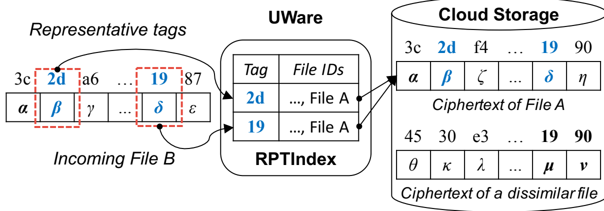


Fig. 4. Illustration of similarity-based deduplication mode. The identical blocks of two similar files (e.g., the stored *File A* and the new *File B*) will remain identical after encryption. But the identical blocks of two dissimilar files will be encrypted into totally different ciphertexts.

Here, the RP tags are sampled from all the block tags of a file, where different sampling methods can be used. For example, the *uniform* method always selects the first block tag in every R blocks, which is easy to implement and can still achieve a higher deduplication ratio as indicated by [18]. The tradeoff is that, a smaller *sampling ratio* R will lead to more RP tags, which increases the chance to locate similar files, but incurs more space and computational cost.

Potential performance issues. Directly applying the above approach requires the client C to send all block tags, including the RP tags, to UWare. Later, the same number of challenge messages $\{r, r^*\}$ will be returned for block key generation and proof generation. From another perspective, in order to locate the previously used randomness r of each duplicated block in initial duplicate checking, UWare requires either huge index space for storing all existing blocks' metadata (e.g., *per-block* randomness r) or longer processing time for loading the blocks' metadata of similar files from disk into memory.

Our per-file randomness design. To balance the performance and efficiency, we prefer a *per-file randomness* r design. That is, each file, regardless of how many blocks it has, will be assigned a single randomness r for the key generation. The same r will be shared with subsequent similar files. Particularly, the block-level duplicate checking will be performed by using the blocks of the most (Top-1) similar file only.

1) Pros and Cons: This per-file randomness design can accelerate the duplicate checking step by keeping all required metadata in RAM. One reason is that the number of randomnesses is much smaller than the above per-block randomness design. Also, the bandwidth cost can reduce accordingly, as each file requires only a pair of challenge randomness (r, r^*). From the perspective of security, exploiting data redundancy at the block level will inevitably reveal the similarity of files, explicitly the number of common blocks. But our per-file randomness design can minimize the necessary leakage only among the similar files. The reason is that only identical blocks of the similar files that share the same r can be detected, while the identical blocks of dissimilar files will be encrypted into totally different ciphertext blocks, as shown in Fig. 4.

On the other hand, as the detection range is limited to the blocks of the most similar file, the deduplication ratio will decrease accordingly, say 6% loss in our experiment.

2) The Detailed Deduplication Procedure: Fig. 5 illustrates the major data structures in this mode. Now, UWare requires three indexes: 1) FTIndex stores the metadata of each file, e.g., file tag fT and per-file randomness r; 2) RPTIndex maintains the similarity information of the previously appeared files; 3) FidIndex is used for quickly retrieving the previously used randomness r after locating the most similar file.

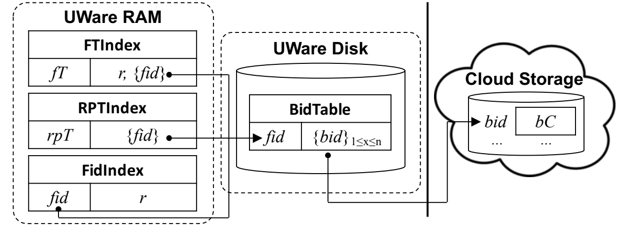


Fig. 5. Secure similarity-based dedup in UWare.

Moreover, to reduce the memory consumption, UWare moves some "less-urgent" metadata to the disk, e.g., the BidTable maintains the relationship between a file and its blocks. This kind of metadata is required in PoW verification (i.e., Step 4), and can be loaded dynamically after the initial duplicate checking (i.e., Step 2).

For practical purpose, UWare always conducts the file-level duplicate checking before performing the similarity-based block-level deduplication, so-called a *dual-level* design [7].

- **Step 1: Tag generation by user.** Suppose that a user wants to upload a file f, which is divided into multiple blocks $f[1]||\cdots||f[n]$ by the client C. Next, the C computes a file tag fT and a group of block tags $\{bT[x]\}_{1\leq x\leq n}$ via $\texttt{Hash}(\cdot)$; and then samples the RP tags $\{rpT[y]\}_{1\leq y\leq m}$ via a sampling method (e.g., the aforementioned *uniform* method that always selects the first block tag in every R blocks). Then the C sends $(fT, \{rpT[y]\}_{1\leq y\leq m})$ to UWare for duplicate checking.
- **Step 2: Initial checking by UWare.** Upon receiving the request, UWare runs Algorithm 1 for duplicate checking. Then it returns a challenge message (r, r^*).
- **Step 3: Data encryption and proof generation by user.** By using the randomness r, the C first generates a block key $bK[x]$ for each block $f[x]$ via $\texttt{Hash}(r, f[x])$. Then it encrypts the blocks $\{bC[x] \leftarrow \texttt{Enc}(bK[x], f[x])\}_{1\leq x\leq n}$, and computes the block ciphertext tags by using the other one-time randomness r^*, i.e., $\{bCT^*[x] \leftarrow \texttt{Hash}(r^*, bC[x])\}_{1\leq x\leq n}$. Next, the $\{bCT^*[x]\}_{1\leq x\leq n}$, as the PoW proof, is sent back to UWare for final verification.
- **Step 4: Final verification by UWare.** At last, UWare needs to return a file ID fid, a list of block IDs $\{bid\}$, and a signal vector σ_f, which marks the duplicated blocks, i.e., "$\sigma_f[x] = 1$" indicates the x-th block of file f is duplicated. UWare initializes each element of σ_f to 0, and changes them according to the following situations:

i. The file is duplicated. If the fT was found in FTIndex in Step 2, UWare will check whether the block ciphertext tags $\{bCT^*[x]\}_{1\leq x\leq n}$ are correct. Normally, all the tags would be matched with the duplicated file's blocks, i.e., passing the PoW testing. Thus, UWare modifies each element of σ_f to 1.

ii. Some blocks are duplicated. Otherwise, UWare will check whether each of the block ciphertext tags $\{bCT^*[x]\}_{1\leq x\leq n}$ matches one of the blocks of the most similar file. For those matched block ciphertext, UWare modifies

Algorithm 1: Initial duplicate checking

```
Input: File tag fT, RP tags {rpT[y]}_{1≤y≤m}.
Output: Challenge message (r, r*).
1  if FTIndex.contains(fT) then
2      r ← FTIndex.get(fT);
3  else
4      Initialize a HashMap Map;
5      for each y ∈ [1, m] do
6          if RPTIndex.contains(rpT[y]) then
7              {fid} ← RPTIndex.get(rpT[y]);
8              for each fid ∈ {fid} do
9                  if Map.contains(fid) then
10                     Map.put(fid, Map.get(fid) + 1);
11                 else
12                     Map.put(fid, 1);
13     if Map.size() > 0 then
14         Rank the Map based on its values, and get the most
           similar fid*;
15         r ← FidIndex.get(fid*);
16     else
17         Randomly generate a fresh r ← {0,1}*;
18 Randomly generate a fresh r* ← {0,1}*;
19 Send back (r, r*).
```

the corresponding elements of σ_f to 1, and updates all related indexes accordingly. In this case, a new fid' will be assigned, and the related data structures will be updated accordingly.

iii. The file is duplicated but verification fails. Similar to the file-level mode, there is a special case that the fT was found in FTIndex in Step 2, but the received proofs (i.e., $\{bCT^*[x]\}_{1\leq x\leq n}$) are not correct entirely. In this case, UWare will perform as if the fT was new, update FTIndex by adding a new fid' into the ID list of the fT, and request the C to upload all unmatched block ciphertexts. So the user can always retrieve the uploaded file, without being affected by others.

D. Security of UWare

Now, we show that UWare can effectively address the threats from a malicious user (C) and a compromised cloud storage server (S), as defined in Section II-B.

Security against a malicious C. The malicious C attempts to use a hash tag fT (or $\{bT[x]\}_{1\leq x\leq n}$) to cheat the ownership of a file that she does not have or identify its existence at S. With these tags, the C can send a deduplication request to UWare. However, UWare always returns a random challenge message (r, r^*), and asks the C to use the file f to compute the ciphertext tag(s) as the PoW proof. Particularly, without holding f, the C is unable to compute the correct encryption key fK (or $\{bK[x]\}_{1\leq x\leq n}$), and the correct proof fCT^* (or $\{bCT^*[x]\}_{1\leq x\leq n}$). So the PoW testing will fail and UWare will not be fooled to treat the C as a legitimate owner.

The C may also attempt to compromise the integrity of other users' data by uploading a fake ciphertext. However, this will cause a special case as mentioned in Section III-A and III-C, and UWare will treat it as a "new" file connected to the C.

Security against a compromised S. An attacker may compromise the backend cloud storage S to obtain the stored ciphertext. However, the ciphertext of files/blocks (i.e., fC or bC) is encrypted by the data owner via $\texttt{Enc}(fK, f)$ (or $\texttt{Enc}(bK[x], f[x])$), where the key is generated via $\texttt{Hash}(r, f)$ (or $\texttt{Hash}(r, f[x])$) after interacting with UWare. Note that introducing randomness r guarantees that even the attacker obtains the partial information of files or hashes, she cannot compromise the encryption key. Similar treatment is also used in [12] and [13]. Without the correct fK (or bK) to pass the PoW, the attacker is unable to decrypt the ciphertext.

Further mitigation against brute-force attacks. A compromised UWare may launch the *offline brute-force attack* by leveraging the stored tags (e.g., fT and bT). This is because the tags for duplicate checking are derived deterministically from the file/block itself [13]. To address this attack, one can involve an additional key server to securely embed another secret sk in the process of tag generation as in [9], [13]. Without sk, UWare is unable to compute the tags. Another potential brute-force attack is the *online brute-force attack*, where an attacker uploads all possible files and observes which one is deduplicated. To address this, standard rate-limiting strategies as used in [9], [15] can be adopted in UWare.

IV. Performance Evaluation

A. Experimental Setup

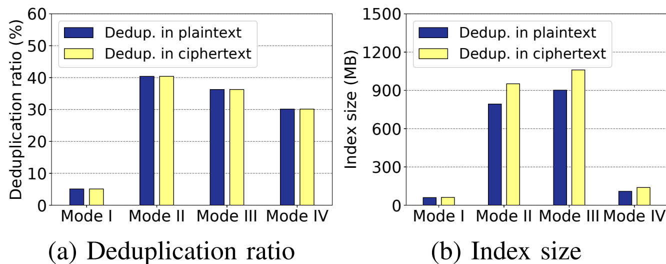
We implement a prototype with roughly 2,400 lines of Java code. We evaluate its performance on an AWS instance "r3.xlarge" with Intel Xeon E5-2670 v2 (4 vCPU @ 2.50 GHz) with 30.5 GiB of RAM in Linux (Ubuntu 16.04 LTS). Particularly, we use Java Cryptography Architecture to realize the symmetric encryption via AES-256 in CTR mode and the cryptographic hash function via SHA256.

In our experiment, we use a real dataset *Fslhomes Snapshots*[3], containing daily snapshots of students' home directories. Each snapshot file was collected with multiple average block sizes, ranging from 2 KB to 128 KB. It also contains a rich set of file system metadata, such as file names, file/block sizes, and block hashes. Here, we randomly select one snapshot file for each of the seven students in the directory of "2014", and use the collection whose the average block size is 16 KB as our dataset. Consequently, we have 2,400,000 files with 34,172,800 blocks, and the total size is about 564 GB.

For demonstration purpose, we keep all the required metadata of stored files/blocks in RAM. During the testing, we simulate a scenario that a C is uploading all the files in our dataset via UWare one by one. And we do not compare with prior secure deduplication systems, because none of them is designed as a deduplication middleware, nor considers the similarity characteristic of files. Regarding the sampling method in similarity-based deduplication, we adopt the *uniform* method[4] (i.e., selecting the first block tag in every R blocks), and set the sampling ratio R as 128, as suggested by [18].

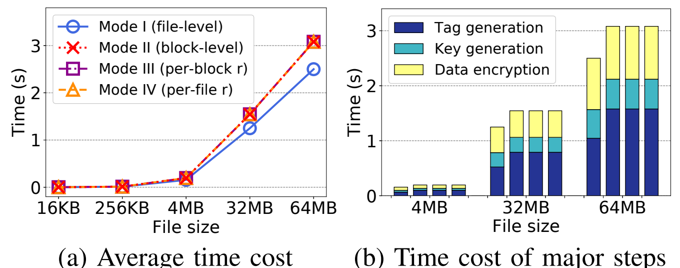
[3] FSL Traces and Snapshots Public Archive: http://tracer.filesystems.org/

[4] Other sampling methods (e.g., the *random* [18] and the *minimum* [19]) can also be adopted here, with (slightly) different performance.



(a) Deduplication ratio (b) Index size

Fig. 6. Comparison between plaintext design and UWare in four modes.



(a) Average time cost (b) Time cost of major steps

Fig. 7. Client side cost for secure deduplication (Note: the four bars in (b) represent the four modes).

TABLE II
AVERAGE OF UWARE PROCESSING TIME.

File size (MB)	File-level (ms)	Block-level (ms)	Similarity-based (ms) per-block r	Similarity-based (ms) per-file r (Top-1)
0∼64	0.00103	0.0026	1.86	0.00205
64∼2048	0.00091	25.9	457.8	14.7

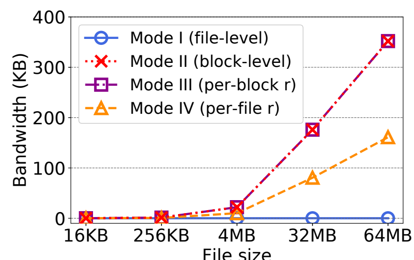


Fig. 8. Bandwidth cost for secure deduplication.

B. Evaluation

1) Deduplication Effectiveness: To demonstrate the effectiveness in terms of deduplication ratio, we test UWare in all modes (i.e., file-level (Mode I), block-level (Mode II), similarity-based dual-level with per-block randomness in all similar files (Mode III) as mentioned in Section III-C, and that with per-file randomness in top-1 similar file (Mode IV)) in both plaintext domain and ciphertext domain.

Fig. 6-(a) shows that UWare achieves the same deduplication ratios with the plaintext designs. That means that our security design does not affect the effectiveness of the deduplication methods. The reason is that the duplicate checking step in UWare is based on the equality testing of underlying blocks/files, and the used encryption method preserves the equality information. Besides that, the results confirm that the block-level mode can eliminate much more redundancy ($\approx$ 40%) than the file-level mode ($\approx$ 5%). In addition, our per-file randomness design incurs a little but acceptable loss of the deduplication ratio, i.e., 6% compared with the per-block randomness design, and 10% compared with the block-level mode. But it still achieves a much higher deduplication ratio ($\approx$ 30%) than the file-level mode ($\approx$ 5%).

2) Index Space Overhead: We now report the system performance in terms of required index space in the four modes. Compared with the plaintext designs, our secure deduplication designs lead to the index expansion, up to 27%, because of the randomness r, as shown in Fig. 6-(b). In particular, we have the following findings: 1) The block-level mode, which achieves the best deduplication performance, requires much larger memory space (near 1,000 MB for our dataset) compared with the file-level mode (about 61 MB). 2) The secure similarity-based dual-level deduplication mode with per-block randomness design requires even (a little) more memory space than the block-level mode due to the extra RPTIndex and more randomnesses. 3) Our per-file randomness design (about 140 MB) just requires a little more memory space than the file-level mode, which is much less than the block-level mode (i.e., over 80% in-memory index space saving).

3) Service Overhead: The service overhead mainly comes from the tag generation and data encryption at client side, and the secure deduplication procedure at UWare. Fig. 7-(a) shows the average time cost at client side for secure deduplication in four modes, and Fig. 7-(b) illustrates the detailed time consumption for three major steps, i.e., tag generation, file/block key generation, and data encryption. We see that the overall time cost increases linearly to the file size, where the tag generation and the file/blocks encryption dominate the cost. Note that Fig. 7-(b) does not show the time cost of the RP tags sampling step in our similarity-based design, because it is negligible to the above cost, e.g., it takes about 0.06 ms for a 64 MB file.

To better evaluate the time cost at UWare, we record the processing time for the small files (smaller than 64 MB) and the large files (from 64 MB to 2 GB), respectively. Table II reports the average time cost of the four modes, which includes the two major steps, i.e., duplicate checking and verification. We can see that our per-file randomness design by using the top-1 similar file is orders of magnitude faster than the per-block randomness design. The reason is that the number of randomnesses is the same as the number of unique files, which is relatively small and can be stored in memory (i.e., FTIndex) for accelerating the process of initial checking.

In addition, Fig. 8 illustrates the bandwidth consumptions for secure deduplication with UWare. Specially, our per-file randomness design (i.e., mode IV) is more efficient than the per-block randomness design (i.e., mode III), because only the RP tags are required for initial checking and the returned challenge message only contains one pair (r, r^*).

V. RELATED WORK

• **Deduplication in plaintext domain.** The problem of data deduplication has been extensively studied in the plaintext domain. In brief, it is achieved by storing and uploading a single copy of identical files/blocks, where the duplicate

checking procedure mostly relies on some designed indexes resided in main memory [1]. To improve system efficiency, there are a line of studies that focus on similarity-based near-exact deduplication, e.g., [19]. Particularly, Fu et al. [18] presented a general-purpose framework to evaluate various deduplication solutions. Their experimental results indicated that no single solution can perform the best in all metrics, but near-exact deduplication can achieve lower memory cost at a cost of decreasing deduplication ratio.

• **Deduplication over encrypted data.** To enable cross-user data deduplication over encrypted data, Douceur et al. [16] first proposed convergent encryption (CE), which encrypts a file by using its hash value as the key. Thus, users who own the same file can produce the same key. After that, Bellare et al. [6] formalized CE to the cryptographic primitive MLE. It is known that MLE and its variants are vulnerable to offline brute-force attacks when facing predictable data [6]. Accordingly, follow-up studies either rely on an additional independent server [9], [13], [22] or a number of online users [15] to defend against this threat, which is also compatible with UWare.

Moreover, secure deduplication has been studied in various settings. For example, Stanek et al. [17] proposed a two-layer scheme that only deduplicates popular files. Zhou et al. [10] combined cross-user file-level and single-user block-level deduplication together to balance the security and performance. Chen et al. [7] proposed BL-MLE, encapsulating the block keys into the block tags to reduce the metadata size but requiring linearly scanning all blocks with bilinear map operations. Recently, Li et al. [20] adopted MinHash encryption to encrypt each group of consecutive blocks with a key derived from a sampled block, which can resist frequency analysis. Different from these studies, we focus on designing a bandwidth-efficient middleware for secure deduplication services and patching the side-channel leakage in the deployment of a PoW protocol.

VI. Conclusion

In this paper, we presented UWare, a bandwidth-efficient middleware that enables cross-user deduplication over encrypted cloud storage. It aims to bring deduplication benefits of storage and bandwidth savings to users. It patches a practically feasible side-channel leakage of the existence of a target file when integrating the PoW protocol. It also leverages the similarity characteristics to balance the deduplication effectiveness and system efficiency. Evaluation results show that UWare can achieve comparable deduplication performance, efficient memory cost, and modest service overhead.

Acknowledgment

This work was supported in part by the Research Grants Council of Hong Kong under Grant CityU 11276816, Grant CityU 11212717, and Grant CityU C1008-16G, in part by the Innovation and Technology Commission of Hong Kong under ITF Project ITS/168/17, in part by the National Natural Science Foundation of China under Grant 61572412 and 61772212, and in part by an AWS Education Research Grant.

References

[1] W. Xia, H. Jiang, D. Feng, F. Douglis, P. Shilane, Y. Hua, M. Fu, Y. Zhang, and Y. Zhou, "A Comprehensive Study of the Past, Present, and Future of Data Deduplication," *Proceedings of the IEEE*, vol. 104, no. 9, pp. 1681–1710, 2016.

[2] "IDC: Public Cloud IaaS Revenues to Triple, by 2020," https://solutionsreview.com/cloud-platforms/idc-public-iaas-revenues, 2016.

[3] F. Armknecht, J.-M. Bohli, G. O. Karame, and F. Youssef, "Transparent Data Deduplication in the Cloud," in *Proc. of ACM CCS*, 2015.

[4] Y. Shin, D. Koo, and J. Hur, "A Survey of Secure Data Deduplication Schemes for Cloud Storage Systems," *ACM Computing Surveys (CSUR)*, vol. 49, no. 4, p. 74, 2017.

[5] V. Rabotka and M. Mannan, "An Evaluation of Recent Secure Deduplication Proposals," *JISA*, vol. 27, pp. 3–18, 2016.

[6] M. Bellare, S. Keelveedhi, and T. Ristenpart, "Message-Locked Encryption and Secure Deduplication," in *Proc. of EUROCRYPT*, 2013.

[7] R. Chen, Y. Mu, G. Yang, and F. Guo, "BL-MLE: Block-Level Message-Locked Encryption for Secure Large File Deduplication," *IEEE TIFS*, vol. 10, no. 12, pp. 2643–2652, 2015.

[8] Y. Zhao and S. S. Chow, "Updatable Block-Level Message-Locked Encryption," in *Proc. of ACM ASIACCS*, 2017.

[9] M. Bellare, S. Keelveedhi, and T. Ristenpart, "DupLESS: Server-Aided Encryption for Deduplicated Storage," in *Proc. of USENIX Security*, 2013.

[10] Y. Zhou, D. Feng, W. Xia, M. Fu, F. Huang, Y. Zhang, and C. Li, "SecDep: A User-Aware Efficient Fine-Grained Secure Deduplication Scheme with Multi-Level Key Management," in *Proc. of IEEE MSST*, 2015.

[11] S. Halevi, D. Harnik, B. Pinkas, and A. Shulman-Peleg, "Proofs of Ownership in Remote Storage Systems," in *Proc. of ACM CCS*, 2011.

[12] J. Xu, E.-C. Chang, and J. Zhou, "Weak Leakage-Resilient Client-Side Deduplication of Encrypted Data in Cloud Storage," in *Proc. of ACM ASIACCS*, 2013.

[13] Y. Zheng, X. Yuan, X. Wang, J. Jiang, C. Wang, and X. Gui, "Toward Encrypted Cloud Media Center With Secure Deduplication," *IEEE Trans. on Multimedia*, vol. 19, no. 2, pp. 251–265, 2017.

[14] D. Harnik, B. Pinkas, and A. Shulman-Peleg, "Side Channels in Cloud Services: Deduplication in Cloud Storage," *IEEE Security&Privacy*, vol. 8, no. 6, pp. 40–47, 2010.

[15] J. Liu, N. Asokan, and B. Pinkas, "Secure Deduplication of Encrypted Data without Additional Independent Servers," in *Proc. of ACM CCS*, 2015.

[16] J. R. Douceur, A. Adya, W. J. Bolosky, D. Simon, and M. Theimer, "Reclaiming Space from Duplicate Files in a Serverless Distributed File System," in *Proc. of IEEE ICDCS*, 2002.

[17] J. Stanek, A. Sorniotti, E. Androulaki, and L. Kencl, "A Secure Data Deduplication Scheme for Cloud Storage," in *Proc. of FC*, 2014.

[18] M. Fu, D. Feng, Y. Hua, X. He, Z. Chen, W. Xia, Y. Zhang, and Y. Tan, "Design Tradeoffs for Data Deduplication Performance in Backup Workloads," in *Proc. of USENIX FAST*, 2015.

[19] W. Xia, H. Jiang, D. Feng, and Y. Hua, "Similarity and Locality based Indexing for High Performance Data Deduplication," *IEEE Trans. on Computers*, vol. 64, no. 4, pp. 1162–1176, 2015.

[20] J. Li, C. Qin, P. P. Lee, and X. Zhang, "Information Leakage in Encrypted Deduplication via Frequency Analysis," in *Proc. of IEEE/IFIP DSN*, 2017.

[21] F. Armknecht, C. Boyd, G. T. Davies, K. Gjøsteen, and M. Toorani, "Side Channels in Deduplication: Trade-Offs Between Leakage and Efficiency," in *Proc. of ACM ASIACCS*, 2017.

[22] P. Puzio, R. Molva, M. Önen, and S. Loureiro, "ClouDedup: Secure Deduplication with Encrypted Data for Cloud Storage," in *Proc. of IEEE CloudCom*, 2013.

[23] P. Zuo, Y. Hua, C. Wang, W. Xia, S. Cao, Y. Zhou, and Y. Sun, "Mitigating Traffic-Based Side Channel Attacks in Bandwidth-Efficient Cloud Storage," in *Proc. of IEEE IPDPS*, 2018.

[24] C.-M. Yu, S. P. Gochhayat, M. Conti, and C.-S. Lu, "Privacy Aware Data Deduplication for Side Channel in Cloud Storage," *IEEE Trans. on Cloud Computing*, pp. 1–1, 2018.

[25] I. Brown, *Research Handbook on Governance of the Internet*. Edward Elgar Publishing, 2013.

[26] Z. Yan, W. Ding, X. Yu, H. Zhu, and R. H. Deng, "Deduplication on Encrypted Big Data in Cloud," *IEEE Trans. on Big Data*, vol. 2, no. 2, pp. 138–150, 2016.