Privacy Aware Data Deduplication for Side Channel in Cloud Storage

Chia-Mu Yu, Sarada Prasad Gochhayat, Mauro Conti, and Chun-Shien Lu

Abstract—Cloud storage services enable individuals and organizations to outsource data storage to remote servers. Cloud storage providers generally adopt data deduplication, a technique for eliminating redundant data by keeping only a single copy of a file, thus saving a considerable amount of storage and bandwidth. However, an attacker can abuse deduplication protocols to steal information. For example, an attacker can perform the duplicate check to verify whether a file (e.g., a pay slip, with a specific name and salary amount) is already stored (by someone else), hence breaching the user privacy.
In this paper, we propose ZEUS *(zero-knowledge deduplication response)* framework. We develop ZEUS and ZEUS$^+$, two privacy-aware deduplication protocols: ZEUS provides weaker privacy guarantees while being more efficient in the communication cost, while ZEUS$^+$ guarantees stronger privacy properties, at an increased communication cost. To the best of our knowledge, ZEUS is the first solution which addresses *two-side privacy* by neither using any extra hardware nor depending on heuristically chosen parameters used by the existing solutions, thus reducing both cost and complexity of the cloud storage. In summary, through the evaluation on real datasets and comparison to existing solutions, our proposed framework demonstrates its capability of eliminating data deduplication-based side channel and at the same time keeping the deduplication benefits.

Index Terms—Cloud Storage, Side Channel, Data Deduplication, Privacy.

1 Introduction

In recent years, the amount of data stored at the cloud storage (e.g., Dropbox [6]) is increasing rapidly due to the prevalence of data outsourcing. In order to be cost-effective and to reduce the bandwidth consumption, cloud storages use *cross-user client-side data deduplication* [25], [31] which eliminates the need to store redundant copies by keeping only a single copy of the data at the cloud storage (see Section 2.1). More specifically, when a user wants to upload a file, (s)he sends a *duplicate check request* (dc request) to the cloud storage. Upon receiving the request, the cloud storage determines whether it has a copy of the requested file in its storage. If a copy is found, it sends a particular *duplicate check response* (dc response) that indicates the existence of the file, and adds a reference to the existing file, hence the explicit transmission of file from the user to the cloud storage is no longer needed; otherwise, the user uploads entire file to the cloud storage.

Despite the benefits of storage and bandwidth savings, the above signaling behavior, where the cloud sends a dc response indicating the file existence status to the user before the explicit file uploading, creates a side channel for privacy leakage. In particular, an attacker can identify the presence of a specific file by partly following the uploading procedures and checking whether the deduplication occurs. For example, an attacker can upload several versions of a pay slip of a particular organization, with a specific name and different salary amounts to check which version of the pay slip gets deduplicated. Such a limited privacy exposes from snooping the file existence status actually leads to various security and privacy threats, such as confirmation-of-a-file [15], learn-the-remaining [15], related-files attack [26], and covert channel [15] (see Section 2.2).

The root cause of the deduplication-based side channel can be attributed to the deterministic relation between the dc request and dc response. More specifically, the cloud deterministically replies a positive dc response to deactivate the explicit file uploading upon finding the dc requested file in its storage. Based on the above observation, a straightforward strategy for the side channel defense is to randomize the duplicate check procedures. Unfortunately, only very few countermeasures [14], [15], [20], [26], [30] have been deployed in the cloud storage system or been proposed in the literature.

Contribution. We propose *zero-knowledge deduplication response* (ZEUS) as a side channel defense based on the framework of *zero-knowledge response* for cross-user client-side deduplication which achieves the two-side privacy with limited extra communications based on a weak assumption on user behavior. Moreover, we also propose the advanced countermeasure, ZEUS$^+$, by the combined use of ZEUS and the random threshold solution [15] to achieve a stronger privacy guarantee with slightly increased communications. In general, in contrast to the prior methods, ZEUS and ZEUS$^+$ possess the following advantages.

- **(A1) Parameterless Configuration.** Existing solutions [15], [26] usually involve parameters to be heuristically chosen. The most prominent feature of ZEUS is that it does not have any parameter to be manually selected, thus avoiding the difficulty in having a

- *Chia-Mu Yu is with Department of Computer Science and Engineering, National Chung Hsing University, Taiwan, and Taiwan Information Security Center (TWISC@NCHU), Taiwan. chiamuyu@nchu.edu.tw*
- *Sarada Prasad Gochhayat and Mauro Conti are with Department of Mathematics, University of Padua, Italy. sarada1987@gmail.com and conti@math.unipd.it*
- *Chun-Shien Lu is with Institute of Information Science, Academia Sinica, Taiwan. lcs@iis.sinica.edu.tw*

TABLE 1: Comparisons Between Different Side Channel Defenses (✓: has this property, ×: does not have this property)

	No Parameter	No Indep. Server	Two-Side Privacy
Mozy [22]	×	✓	×
Harnik et al. [15]	×	✓	×
Lee and Choi [20]	×	✓	×
Heen et al. [14]	✓	×	✓
Wang et al. [30]	×	✓	×
Shin and Kim [26]	✓	×	✓
Armknecht et al. [1]	×	✓	×
ZEUS (this paper)	✓	✓	△
ZEUS$^+$ (this paper)	×	✓	✓

proper security-performance trade-off in real world implementation.

- **(A2) No Independent Server.** ZEUS and ZEUS$^+$ do not assume the use of extra hardware while existing solutions require the independent gateway/server [18]. In essence, the effectiveness of ZEUS and ZEUS$^+$ relies solely on how the cloud react to the dc request.
- **(A3) Stronger Privacy.** Compared to existing solutions that can only have *inexistence privacy*, ZEUS and ZEUS$^+$ achieve a much stronger privacy notion, *two-side privacy* (see Section 3.3).

A comparison table about privacy and assumptions made by different side channel defenses is shown in Table 1. More discussions on the comparison can be found in Section 6.

Organization. The rest of the paper is organized as follows. We first review the related work in Section 2. The system model is introduced in Section 3. We present our proposed ZEUS and ZEUS$^+$ solutions in Section 4, followed by the real dataset evaluation in Section 5. Comparison and Discussion are presented in Section 6. Afterwards, we conclude our research in Section 7.

2 Background and Related Works

Here, we first describe how cross-user client-side data deduplication works (Section 2.1), and the corresponding privacy threats (Section 2.2). Then, we present an overview of the state-of-the-art solutions and their weaknesses (Section 2.3).

2.1 Data Deduplication

Data deduplication is a very popular technique adopted by the cloud storage to eliminate the need to store redundant data by creating logical pointer to the single instance of data piece, whenever the cloud storage receives identical files. The implementation of data deduplication can have different options, depending on where the deduplication occurs, the scope of the data that deduplication applies, and the deduplication granularity. More specifically, the cloud storage in *server-side deduplication* determines the need of an additional copy only after receiving the entire file, whereas the user in *client-side deduplication* pro-actively performs the duplicate check via the interaction with the cloud storage. Client-side data deduplication is featured by the use of *dc request* and *dc response*, and is illustrated in Fig. 1, where the dc request (e.g., "Is file f in cloud?") and dc response (e.g., "Yes/No") are used to check whether the user needs to upload the entire data. Note that the *duplicate check* refers to the procedures of exchanging dc request and dc response. In practice, the dc request is usually implemented by the cryptographic hash (e.g., SHA-256) of the data. Due to the collision avoidance of the cryptographic hash function, the user may detect the existence status of f by only examining the existence status of the hash in the cloud.

On the other hand, in *single-user (or intra-user) deduplication*, the deduplication takes place only among the data uploaded by the same user, while in *cross-user (or inter-user) deduplication*, only a single copy of the data will be stored, irrespective of the ownership of data. In other words, virtually all of the users in cross-user deduplication share a single disk in the cloud storage.

In addition, the deduplication can apply to either files or chunks, depending on the deduplication granularity. For example, Dropbox [6] performs the chunk-level deduplication with 4MB chunk size; i.e., each file is partitioned to chunks of fixed-size and the deduplication works over chunks. Moreover, the chunk size can also be varied [32]; the use of rolling hash (e.g., Rabin fingerprint [23]) proves to be useful in identifying the common parts of two similar contents.

As the above three notions of deduplication are orthogonal to each other, throughout this paper, unless stated otherwise, the cloud performs *cross-user client-side fixed-size chunk-level data deduplication*, seeking the greatest opportunity to deduplicate the data and therefore reaching the highest potential of storage and bandwidth savings.

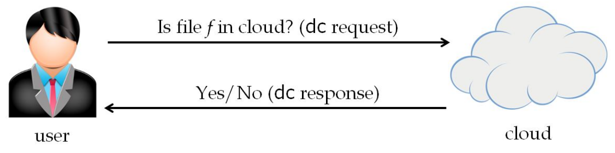


Fig. 1: Client-side data deduplication.

2.2 Side Channel in Deduplicated Cloud Storage

The deduplication signal (i.e., dc response), by which the user detects whether a chunk is already in the cloud, creates a side channel. Such a side channel is first formally presented in [15] and may lead to the following privacy leakage and abuses.

- **Confirmation-of-a-File.** The confirmation-of-a-file [11] is originally presented in the context of convergent encryption [8] but can be naturally applied to our context. More precisely, with the goal of detecting the existence status of a specific chunk, an attacker verifies his/her suspect by performing the duplicate check to see whether the deduplication occurs. The confirmation-of-a-file can be seen as the most straightforward privacy leakage due to the side channel.
- **Learn-the-Remaining-Information.** Learn-the-remaining-information [15] is a brute force-like strategy, by which the attacker exhaustively generates all possible unknown pieces and do the duplicate check. The positive (negative) dc response

(i.e., the occurrence of deduplication) indicates the existence (inexistence) of the corresponding chunk. Here, due to the low min-entropy nature (i.e., high predictability) [17] of the user content, learn-the-remaining-information strategy can be thought of as a repeated invocation of confirmation-of-a-file to learn the sensitive information of victim.

- **Related-chunks Attack.** Related-chunks attack [26] can also be seen as an enhanced version of the confirmation-of-a-file. More precisely, with the fact that all of the individual chunks from a particular file are actually dependent on each other, the existence of a given proportion of chunks of the file serve as an evidence of file existence. In this way, related-chunks can claim the file existence in a more efficient and more effective manner.
- **Covert Channel.** Covert channel is a steganographic channel, by which the information is exchanged by two parties that are not allowed to communicate. By using the side channel here, attackers can create a covert channel to bypass the censorship and communicate with each other [13], [15]. For example, one party selectively uploads or deletes a predefined chunk c. Another performs the duplicate check on c and checks the dc response to detect the absence or presence of c, which are decoded as bit 0 or 1, respectively.

We particularly note that the side channel here is powerful but would be easy to be exploited; any attacker with the privileges of only a normal cloud storage user can abuse the dc response as a side channel.

2.3 State-of-the-Art Solutions and Their Weaknesses

Harnik et al. [15] first identified the risk of side channel in deduplicated cloud storage and proposed to randomize the *deduplication threshold* for each chunk. The deduplication threshold is defined as the number of copies needed for triggering the deduplication; a popular choice is 1 for all chunks, which means that the subsequent uploadings of the same chunk will be deduplicated if a copy can be found. Harnik et al. made an observation that the deduplication-based side channel is due to the publicly known and fixed deduplication threshold. More precisely, when seeing the positive (negative) dc response[1], the attacker knows that deduplication is (not) triggered and chunk is (not) in the storage. To obfuscate the dc response, Harnik et al. [15] proposed to use random threshold (RT) on a per-chunk basis. In RT, the per-chunk deduplication thresholds are kept secret. As a consequence, even when seeing the negative dc response, it might be the case where a couple of copies have been in the storage but the number of copies still does not reach the threshold. In practice, a deduplication threshold t_i for the chunk c_i is selected uniformly at random from $[1, B]$, where B is a predefined upper bound of deduplication threshold. Despite the simplicity of RT, it has many disadvantages. For example, the choice of B remains unclear, despite an obvious trade-off between the overhead savings and privacy (e.g., larger B implies better privacy). Most importantly, in RT, when the attacker sees a positive dc response, (s)he can be fully confident of chunk existence, hence breaching the *existence privacy* (see Section 3.3).

1. Positive (negative) dc response means a copy is (not) found.

Different implementations of RT actually have the subtle performance and privacy differences. The subtlety mainly lies in how a deduplication threshold t_i for chunk c_i is determined. For example, Harnik et al. [15] proposed to select t_i for chunk c_i uniformly at random *beforehand*. Lee and Choi [20] determined t_i by making a random choice *at each upload* and claimed to have better privacy than Harnik et al.'s method. Nevertheless, Armknecht et al. [1] stated that the methods in [15] and in [20] are equivalent in terms of the privacy guarantee. Instead of uniform sampling over $[1, B]$, Wang et al. [30] determined the deduplication thresholds based on a game-theoretic approach. In particular, Wang et al. modeled the deduplication as a dynamic non-cooperative game between the attacker and cloud. The attacker is aimed to learn the cloud's payoff through repeated game iterations. Wang et al. claimed to have more efficiency but remain the same privacy level if game-theoretic deduplication thresholds are used. Nevertheless, the payoff matrix in [30] is fixed and cannot be adapted with the attacker's alternative strategies. In fact, Armknecht et al. [1] recently proved that deduplication thresholds uniformly sampled from $[1, B]$ achieve the optimal defense for the natural privacy measure. Unfortunately, all of the proposals here fall in the category of RT and therefore share the same weaknesses.

Another approach for the side channel defense is to use an extra hardware to obfuscate the network traffic. The rationale is that if a proxy, sitting between the user and cloud, able to cache dc requests, it can obfuscate the network traffic by manipulating the order of transmitting data. For example, Heen et al. [14] assumed that each user will be equipped with a gateway from the cloud storage provider. In this sense, the gateway can break the deterministic relation between dc requests and dc responses by the late forwarding policy. On the other hand, Shin and Kim [26] assumed an independent trusted server that can perform the similar task and achieve the differentially private duplicate check. The downside of the solutions in this category is the use of extra hardware. Though Heen et al. claimed the practicality by showing the real applications such as NeufGiga and BT Digital Vault, their gateway setting is still not a popular implementation choice, restricting their applications. The use of the method in [26] also completely sacrifices the bandwidth saving at the user side.

Mozy [22] conducted an alternative approach; its belief is that only small-size files contain sensitive information and their existence status matters. In this sense, given a threshold for the file size for the deduplication, the deduplication functions normally if the size of incoming file is larger than the threshold and the deduplication is deactivated otherwise. Nonetheless, the biggest challenge in this method is also the choice of the threshold for the file size.

3 System Model

In this section, we discuss the Network Model and the Threat Model that we will consider in this paper.

3.1 Network Model

We consider a cloud storage, which employs cross-user client-side fixed-sized chunk-level data deduplication. As mentioned in Section 2.1, client-side deduplication is featured by the check-first-data-later framework. In particular, the file to be uploaded is partitioned to chunks c of bit length ϕ. The user performs the *duplicate check* on c, consisting of an exchange of dc request and dc response. More precisely, the dc request $h(c)$, where $h(\cdot)$ is a cryptographic hash function (e.g., SHA-256), is used to query the existence status of c. The user uploads the chunks only when receiving negative dc responses (i.e., a deduplication signal indicating the chunk inexistence).

Formally, we define the *one-chunk, one-round interaction* duplicate check protocol as follows.

Definition 1. A one-chunk, one-round-interaction duplicate check protocol is defined as $f(c, aux)$, where c is the chunk that the uploading user has interest to know the existence status and aux denotes the auxiliary information required for duplicate check. In particular, the arguments c and aux of $f(\cdot)$ can be seen as materials uploaded by the user while $f(c, aux)$ denotes the materials responded by the cloud.

In general, the duplicate check protocol can be designed such that the user performs multi-round interactions with the cloud to determine whether the existence status of multiple chunks. However, to ease the notational complexity, we only present the definition of one-chunk, one-round-interaction duplicate check protocol. The above definition can be easily extended to the case of multi-chunk, multi-round-interaction duplicate check protocol. One can also see that, in essence, dc request consists of both c and aux, and dc response refers to $f(c, aux)$, from Definition 1 point of view.

3.2 Threat Model

For an arbitrarily chosen chunk c, the user (including attacker) does not know its existence status, except that the user uploads c previously. More specifically, if p denotes the probability that an arbitrary chunk is in the cloud, then p is assumed to be very small[2]. By using side channel attack, the attacker aims to learn the existence status of a given chunk c. In other words, the objective of the attacker is to know whether there is already a copy of c in the cloud storage. Under the conventional deduplication framework, to determine whether c is stored in a cloud server, the attacker with the knowledge of c also performs duplicate check. If the corresponding dc response is positive (negative), then the attacker would conclude the (non)existence of chunk c in the storage system.

The attacker does not have any obligation to complete the uploading and thus can abruptly abort the uploading at any time instant. Furthermore, we consider a *Sybil attacker*. In particular, due to the easy-to-register nature of the current commercial cloud storage[3], the Sybil attacker is able to create a number of legitimate accounts (termed as *Sybil accounts*) of the cloud storage, and to repeatedly perform independent deduplication checks on chunks. The Sybil accounts also can freely choose to obey or disobey the file uploading procedure at any time instant. We do not assume the ratio of Sybil accounts that can be created by the attacker. In other words, the privacy of our proposed solutions (see Section 4.2 and Section 4.3) is independent of the ratio of Sybil accounts. Thus, even in the extreme case that all the accounts controlled by the attacker are Sybil accounts (i.e., the ratio of Sybil accounts is 100%), our proposed solutions can still achieve their claimed privacy.

The attacker also seeks the assistance by using extra hardware and software. For example, the attacker may have a network sniffer sitting between the host and cloud to check the packet content. Moreover, the attacker is allowed to check whether the chunk c is accessed. In essence, if a chunk is not accessed or not transmitted after the duplicate check, this implies the existence of the chunk.

3.3 Privacy Notion

Here, two privacy notions, *existence privacy* and *inexistence privacy*, are defined. The existence privacy refers to the case where the attacker cannot confirm the chunk existence except for the chunks uploaded by himself/herself. More formally, we have the following definition of existence privacy.

Definition 2. Suppose that c is the chunk of attacker's interest; i.e., the attacker without prior knowledge of c is aimed to know the existence status of c. Let C be the event that c is in the cloud, $f(\cdot)$ the duplicate check protocol, and aux the auxiliary data. The duplicate check protocol $f(\cdot)$ achieves existence privacy if $P[C|f(c, aux)] = P[C]$, except that $f(c, aux)$ has clearly indicated the existence of c.

Basically, when a duplicate check protocol satisfies existence privacy, its dc response leaks no information on the existence status of c. On the other hand, the inexistence privacy refers to an opposite case, where the attacker cannot confirm the chunk inexistence. Similarly, we have the following definition of inexistence privacy.

Definition 3. Suppose that c is the chunk of attacker's interest; i.e., the attacker without prior knowledge of c is aimed to know the existence status of c. Let $\bar{C}$ be the event that c is not in the cloud, $f(\cdot)$ the duplicate check protocol, and aux the auxiliary data. The duplicate check protocol $f(\cdot)$ achieves inexistence privacy if $P[\bar{C}|f(c, aux)] = P[\bar{C}]$, except that $f(c, aux)$ has clearly indicated the inexistence of c.

2. This assumption can be justified as follows. Consider the case where the chunk size is 4MB (also the chunk size currently used by Dropbox). For an attacker without the prior knowledge of a chunk c, the probability p that an arbitrary chunk c is in the cloud could be approximately calculated as $\frac{(500\times 2^{50})/2^{2+20+3}}{2^{2^{2+20+3}}}$, where the denominator denotes the total number of combinations for chunks of 4MB size and the numerator denotes the estimated number of distinct chunks stored in Dropbox. Note that it is reported by Dropbox Tech Blog (https://blogs.dropbox.com/tech/2016/03/magic-pocket-infrastructure/) that Dropbox hosts approximately 500 petabytes of user data. We can see from the above calculation that, if no prior knowledge on c is available, the probability that an arbitrary chunk is in the cloud is negligible.

3. One can register a new account and have GBs of free space from cloud storage by simply presenting email address.

As mentioned in Section 2.3, RT [15] does achieve the inexistence privacy, but does not achieve the existence privacy; once receiving the positive dc response, the attacker immediately knows that the chunk already has a copy in the cloud. The above claim can be formally stated as $P[\bar{C}|\mathsf{RT}(c) = \text{“}-\text{''}] = P[\bar{C}]$ and $P[C|\mathsf{RT}(c) = \text{“}+\text{''}] = 1 \neq P[\bar{C}]$, where $\mathsf{RT}(c) = \text{“}-\text{''}$ ($\mathsf{RT}(c) = \text{“}+\text{''}$) means a negative (positive) dc response returned by the cloud running RT.

Here, we argue that the existence privacy is more important than the inexistence privacy because the former actually leaks more information. Consider the threats due to the deduplication-based side channel in Section 2.2. Most of the threats rely on the fact that a specific chunk is in the cloud; such a fact gives the attacker information about the content in the specific chunk, which could be a severe privacy leakage. An obvious example is learn-the-remaining-information; once confirming the chunk existence, the attacker learns the sensitive content. The existence privacy does not completely eliminate the possibility of four threats in Section 2.2, but significantly mitigates these threats.

A deduplicate check protocol with side channel defense achieves two-side privacy if both existence privacy and inexistence privacy are fulfilled. According to Definition 2 and Definition 3, a two-side private deduplicate check protocol means that the dc response does not include any information about the existence status of a specific chunk. We conjecture that such a deduplicate check protocol cannot have any deduplication gain either. We, instead, have a weaker version of existence privacy defined as follows.

Definition 4. Suppose that c is the chunk of attacker's interest; i.e., the attacker without prior knowledge of c is aimed to know the existence status of c. Let C be the event that c is in the cloud, $f(\cdot)$ the duplicate check protocol, and aux the auxiliary data. The duplicate check protocol $f(\cdot)$ achieves weaker existence privacy if $P[C|f(c, aux)] = 1/2$, except that $f(c, aux)$ has clearly indicated the existence of c.

The difference between Definition 2 and Definition 4 is that, while $P[C|f(c, aux)] = P[C] = p$ in the former, $P[C|f(c, aux)] = 1/2$ in the latter. The above definition of weak existence privacy states that, even after seeing the dc response, the attacker can only still make a random guess on the existence status of c. We particularly note that, though numerically $P[C|f(c, aux)]$ is increased from p in Definition 2 to $1/2$ in Definition 4, this makes nearly no impact on the probability that the attacker confirms the existence status of a *single* chunk c, in the absence of correlation among multiple chunks. In the following, a two-side private deduplicate check protocol means the fulfillment of both weak existence privacy and inexistence privacy, unless stated otherwise.

As mentioned in Section 2.3, RT [12] and its variants [1], [20], [30] achieve only one-side privacy. The heuristic in [14] assumes a trusted gateway but does not have formal privacy guarantee. The solution in [26] achieves differential privacy, which is a privacy notion similar to two-side privacy. Mozy [22] eliminates the side channel, only for small-size files.

There are the other security and privacy issues around the design of cloud storage[4], such as the reconciliation between the encryption and deduplication [18], proof of ownership (POW) [12], key management [19], and poison attack [3]. These issues are orthogonal to the side channel. Throughout this paper, we focus only on the side channel.

4 Proposed Solutions

In this section, we first present the strawman countermeasure and their security flaws in Section 4.1. After that, we propose the first solution ZEUS with the minimal overhead in Section 4.2. The second solution ZEUS$^+$ via the integration of ZEUS and the existing RT solution having two-side privacy guarantee is presented in Section 4.3.

4.1 Strawman Solutions

In addition to the deduplication threshold randomization [15], the most straightforward idea to uncorrelate the dc request and dc response is to randomize the dc response. A naïve implementation of such a *random response* approach goes as follows. Let chunk existences 0 and 1 represent the absence and presence of the chunks in the cloud, respectively. The dc response $-$ ($+$) indicates that the user has to (does not need to) upload the chunk. Then, the naïve random response strategy can be illustrated in Table 2. Note that we refer *dc table* to as the table describing the dc requests and dc responses (e.g., Table 2) thereafter. We can observe that when the chunk is absent, the cloud has no choice but returns a negative dc response, instructing the user to upload the chunk. On the other hand, when the chunk is present, the server has the flexibility for the dc response.

Though the design objective of random response is to reveal nothing about the chunk existence status, unfortunately, the positive response in Table 2 invariably indicates the chunk existence, thus violating existence privacy. Even worse, the Sybil attacker can also conclude the inexistence of chunk c by using Sybil accounts to perform independent duplicate checks on c. More specifically, each Sybil account uploads $h(c)$, gains the dc response, but aborts the communication with the cloud right before c is uploaded. By doing so, the attacker does not change the existence status of c. When all Sybil accounts gain negative dc responses, the attacker is highly confident that c is not in cloud.

TABLE 2: dc Table for Strawman zero-knowledge response Approach

chunk c existence	dc response
0	$-$
1	$+/-$

4. Currently, to the best of our knowledge, Mozy is the only public cloud storage provider that attempts to eliminate the privacy leakage from the side channel in client-side deduplicated cloud storage systems. However, there are other techniques that address the data privacy, rather than privacy issue of side channel. For example, convergent encryption is used by the cloud for encrypted data deduplication. Convergent encryption has been adopted by public cloud storage providers including Mega [21] and Bitcasa [2] and by a free/open decentralized cloud storage system, Tahoe-LAFS [27].

One might consider that the use of time limit between subsequent duplicate checks could be useful in reducing the ability of independent duplicate checks. Nevertheless, the attacker can easily circumvent the countermeasure by using different Sybil accounts. Another plausible approach is that, with the observation that the independent duplicate checks require the incomplete uploading, the cloud may ask users to complete the uploading; otherwise, the account will be blocked. Here, the *complete uploading* means that once receiving negative dc response, the user explicitly uploads the chunk. However, because Sybil accounts can be created with limited cost or even for free, a Sybil attacker can still bypass this approach by performing the duplicate check, disconnecting to the cloud right before chunk uploading, and leaving some Sybil accounts blocked.

4.2 ZEUS

Here, we present the design principle behind ZEUS, followed by the formal description of ZEUS.

4.2.1 Design Principle

Basically, the failure of the naïve random response in counteracting side channel can be attributed to the following two reasons.

- **(R1)** A distinguishable response exists. Here, the *distinguishable response* is defined as the dc response appearing only once in dc table. For example, $+$ in Table 2 is a distinguishable response. The attacker seeing $+$ immediately knows the chunk existence, breaching the existence privacy.
- **(R2)** An incomplete uploading can always be exploited by the attacker. The incomplete uploading enables the attacker to repeatedly run duplicate checks. The potentially different or the same dc responses may leak information on the chunk existence status.

Now, we propose the following three techniques, **(T1)**$\sim$**(T3)**, to circumvent the above difficulties and to develop our side channel defense, ZEUS. Note that **(T1)** and **(T2)** are developed to counteract **(R1)** while **(T3)** nullifies the effect of **(R2)**.

TABLE 3: dc table for naive implementation of double chunk uploading

c_1 existence	c_2 existence	dc response
0	0	$-,-$
0	1	$-,+$
1	0	$+,-$
1	1	$+,+$

TABLE 4: dc table for the implementation of XOR obfuscation

c_1 existence	c_2 existence	dc response
0	0	2
0	1	1
1	0	1
1	1	0

- **(T1) Double Chunk Uploading.** The first technique is, instead of uploading a single chunk, to upload two chunks at once. Table 3 shows the corresponding dc table. Unfortunately, the naïve implementation of double chunk uploading, as shown in Table 3, where the user sends two individual dc requests and receives two individual dc responses, is not helpful in preventing the privacy leakage, because this can only be regarded as doing the ordinary duplicate check twice. However, combined with *XOR obfuscation* described below, the randomness in the dc response makes the attacker much more difficult to distinguish between chunk existence and inexistence.
- **(T2) XOR Obfuscation.** The direct use of **(T1)** does not prevent the privacy leakage; however, if we perform encodings on both dc responses and the uploaded chunks, then the chunk existence status is hidden behind the deduplication result. More specifically, for the dc request $\langle h(c_1), h(c_2) \rangle$ in double chunk uploading, the dc response consists of a *single number* that indicates the number of chunks needed to be uploaded, instead of a pair of ordinary dc responses, as shown in Table 4. Now, there would only be three cases for the possible dc response here, which are 0, 1 and 2. If the dc response is 2 (0), two (no) chunks need to be uploaded; otherwise, the exclusive-or (XOR) of the two chunks, $c_1 \oplus c_2$, is uploaded. In this way, the user always uploads $c_1 \oplus c_2$ and cannot distinguish between the case where c_1 is but c_2 is not in the cloud and the opposite case, when the cloud has owned a copy of one of them.
- **(T3) Dirty Chunk List.** If the complete uploading can always be ensured, the privacy leakage can be significantly mitigated. Nonetheless, as mentioned in Section 4.1, the Sybil attacker may still perform independent duplicate checks by leveraging the Sybil accounts. To counteract such an abuse of free cloud accounts, we mark the chunk that has been requested but eventually is not uploaded as a *dirty* chunk. Afterwards, dc requests containing dirty chunks will invariably receive the dc response 2. The rationale behind this design is that dirty chunks could potentially be exploited by the attacker and therefore all the subsequent duplicate checks relevant to dirty chunks will always not trigger the deduplication. The above policy of using dirty chunks can be implemented by keeping a list (called *dirty chunk list*, $\mathcal{L}$) containing all of the hashes of dirty chunks. When receiving the dc request, the cloud first checks whether the hashes appear in $\mathcal{L}$. If so, the cloud returns 2; otherwise, return the value according to a publicly known dc table in ZEUS (described below).

TABLE 5: Design space of dc table for ZEUS

	c_1 existence	c_2 existence	dc response
z_1	0	0	x_1
z_2	0	1	x_2
z_3	1	0	x_3
z_4	1	1	x_4

TABLE 6: dc table for ZEUS

c_1 existence	c_2 existence	dc response
0	0	2
0	1	1
1	0	1
1	1	1

The Design of dc table in ZEUS. Despite the above three techniques, we face a challenge of how to assign proper values to x_i's in Table 5 such that both privacy and deduplication benefits are kept. In fact, the design space is rather limited, though the number of obvious options is $3^4 = 81$, because each x_i has three choices 0, 1, and 2. Here, for notation simplicity, we abuse the vector representation to represent different dc table realizations. For example, the vector $[2\ 1\ 1\ 0]^T$ represents the table 4. In fact, we can have the following four observations, which can be used to significantly shrink the design space. Also for the representation simplicity, the notation $x_i \in \{0, 1, 2\}$ means the dc response in different cases. The notation z_1 represents the case, where both chunks c_1 and c_2 are not in the cloud, the notation z_2 represents the case where c_1 is not but c_2 is in the cloud, and so on.

- **(O1)** When c_1 and c_2 in the dc request $\langle h(c_1), h(c_2)\rangle$ are all not in the cloud, the cloud is forced to return 2; otherwise, at least one chunk will be missing. Thus, x_1 has to be 2.
- **(O2)** Similarly, except for x_4, all of x_1, x_2, and x_3 cannot be 0; otherwise, at least one chunk will be missing. Thus, we derive the constraints of $x_1 \neq 0$, $x_2 \neq 0$, and $x_3 \neq 0$ in the design.
- **(O3)** Though being unable to know the existence status of a given chunk *before* processing the chunk (as mentioned in Section 3.2), the attacker can confirm the existence of a chunk c_1 if the attacker himself/herself uploads c_1 previously. We particularly note that this does not violate the privacy because the user who uploaded chunks has no doubt to be confident that his/her chunk is in the cloud. However, with the knowledge that a given chunk c_1 exists in the cloud, the attacker can infer the existence status of c_2, if the table is ill-designed. More specifically, if $x_3 \neq x_4$, the attacker can distinguish between the cases z_3 and z_4 by observing the dc response corresponding to the duplicate check on $\langle h(c_1), h(c_2)\rangle$. In this sense, the constraint of $x_3 = x_4$ needs to be satisfied.
- **(O4)** The observation **(O3)** can also apply to the symmetric case where the existence of c_2 has been known by the attacker interested in the existence status of c_1. Thus, the constraint of $x_2 = x_4$ also needs to be satisfied.

Based on **(O1)**∼**(O4)**, only two configurations, $[2\ 2\ 2\ 2]^T$ and $[2\ 1\ 1\ 1]^T$, are eligible as a candidate of dc table design. Nonetheless, though the $[2\ 2\ 2\ 2]^T$ design offers the strongest privacy guarantee, it does not have any storage and bandwidth savings. Hence, the only feasible solution for the dc table is $[2\ 1\ 1\ 1]^T$, as shown in Table 6.

The use of $\mathcal{L}$ implies that the attacker no longer has the capability of performing independent duplicate checks. More specifically, the attacker has two options, uploading the chunk and aborting the connection, after the duplicate check. The former results in the chunk existence, and therefore the subsequent duplicate checks involving the uploaded chunks provide no additional information to the user because of the same dc response in z_2, z_3, and z_4 (see Table 6). The latter results in the nullification of the deduplication on dirty chunks, and therefore the subsequent duplicate checks always return 2 to the user and also provide no additional information to the user.

4.2.2 Algorithmic Procedures

The formal description of ZEUS is shown in Fig. 2, where the user attempts to upload a file f to the cloud. The file f is first partitioned to chunks (step 1). Due to the double chunk uploading in ZEUS, the user checks whether the number of chunks is even and whether the size of the last chunk is equal to the predefined chunk size. If not, we generate bit sequence of appropriate length and concatenate it to the file f (steps 3∼6). After that, the user performs duplicate check on $\langle h(c_i), h(c_{i+1})\rangle$, $i \in [1, 3, \ldots, \hat{n} - 1]$, on pairs of chunks (step 8). The cloud, after receiving $\langle h(c_i), h(c_{i+1})\rangle$, checks whether the involved chunks are dirty (step 9). The cloud always returns 2 to the user if so, and returns either 1 or 2 according to the dc table shown in Table 6 (step 10). Depending on the received dc response, the user either uploads $c_1 \oplus c_2$ or uploads c_1 and c_2 explicitly to the cloud (steps 13∼20).

```
Algorithm: ZEUS
Input: file $f$ with chunk size $\phi$, and dirty chunk list $\mathcal{L}$
01 user partitions $f$ into chunks $c_1, \ldots, c_n$
02 user sets $\hat{n} = n$
03 if bit length $|c_n| \neq \phi$
04    user performs padding to $c_n$
05 if $n$ is odd
06    user picks random chunk $c_{n+1}$ and $\hat{n} = n + 1$
07 for $i \in \{1, 3, \ldots, \hat{n} - 1\}$
08    user performs duplicate check on $\langle h(c_i), h(c_{i+1})\rangle$
09    if $h(c_i) \notin \mathcal{L}$ and $h(c_{i+1}) \notin \mathcal{L}$
10       cloud replies 1 or 2 according to Table 6
11    else
12       cloud replies dc response 2
13    if user receives dc response 1
14       user uploads $c_i \oplus c_{i+1}$ to the cloud
15       if cloud does not receive $c_i \oplus c_{i+1}$
16          $\mathcal{L} = \mathcal{L} \cup \{c_i, c_{i+1}\}$
17    else
18       user uploads $c_i$ and $c_{i+1}$ to the cloud
19       if cloud does not receive $c_i$ and $c_{i+1}$
20          $\mathcal{L} = \mathcal{L} \cup \{c_i, c_{i+1}\}$
```

Fig. 2: ZEUS.

The design of the dc table in ZEUS has a distinguishable response **(R1)** for the case of z_1. Thus, at the first sight, one may consider that ZEUS violates the inexistence privacy, because the attacker receiving the response 2 can always confirm the chunk inexistence. The details of the argument that ZEUS achieves the two-side privacy (i.e., both existence privacy and inexistence privacy) will be described in Section 4.2.4.

4.2.3 Performance Evaluation of ZEUS

One can easily see that ZEUS does not sacrifice the storage saving because ZEUS only affects how the cloud reacts to the dc request. Here, we evaluate the performance mainly in terms of the extra communication cost incurred by the use of ZEUS. One can see from Table 4 and Table 6 that the difference between the original data deduplication and ZEUS lies in the case of z_4. More precisely, in such a case, the user in ZEUS needs to upload $c_1 \oplus c_2$ of length ϕ whereas the user in the original deduplication does not.

Let p be the probability that an arbitrary chunk is in the cloud. In the original deduplication, the expected communication cost of uploading two chunks can be calculated as

$$2(1-p)^2\phi + (1-p)p\phi + p(1-p)\phi. \tag{1}$$

On the other hand, the expected communication cost of uploading two chunks in ZEUS can be calculated as

$$2(1-p)^2\phi + (1-p)p\phi + p(1-p)\phi + p^2\phi. \tag{2}$$

The difference in the communication cost between Eq. (1) and Eq. (2) is $p^2\phi$, which also shows the price to pay for the privacy.

4.2.4 Privacy of ZEUS

To learn the existence status of a specific chunk, the only option left to the attacker is to perform duplicate check. In the following, the privacy is evaluated based on what the attacker learns from the duplicate checks.

Single Duplicate Check. From Table 6, at this moment, we know that ZEUS violates the inexistence privacy, because the dc response 2 invariably indicates the chunk inexistence of queried chunks. Now, we have the following privacy result of ZEUS.

Theorem 1. ZEUS achieves weak existence privacy *under the condition of single duplicate check.*

Proof: If both c_1 and c_2 in the dc request $\langle h(c_1), h(c_2)\rangle$ are not controlled by the attacker, where c_2 is the chunk of attacker's interest, the attacker can only confirm the chunk inexistence when receiving the dc response 2 by chance.

Moreover, as mentioned in Section 3.2, we assume that the attacker cannot know the existence status of c_1 if the user does not upload it. Thus, though the duplicate check $\langle h(c_1), h(c_2)\rangle$ with the very likely inexistence of c_1 can be used to identify the existence status of c_2 (see Table 6), the attacker is unable to have such kind of duplicate check. In fact, one might consider that an arbitrarily chosen chunk c_1 is very unlikely to be in the cloud. In this way, the attacker can claim to detect the existence status of c_2 by differentiating z_1 and z_2. However, given the dc response 1, the probability of c_2 in the cloud can be formulated as

$$\begin{aligned}
&P[C_2|R_1] && (3)\\
=&\frac{P[R_1|C_2]P[C_2]}{P[R_1|C_2]P[C_2] + P[R_1|\bar{C_2}]P[\bar{C_2}]} && (4)\\
=&\frac{1 \times P[C_2]}{1 \times P[C_2] + P[R_1|\bar{C_2}](1 - P[C_2])} && (5)\\
=&\frac{P[C_2]}{P[C_2] + (1 - P[C_2])(\varpi_1 + \varpi_2)}, && (6)\\
&\text{where } \varpi_1 = P[R_1|\bar{C_2}, \bar{C_1}]P[\bar{C_1}] \\
&\text{and } \varpi_2 = P[R_1|\bar{C_2}, C_1]P[C_1] \\
=&\frac{p}{p + (1-p)(0 \times p + 1 \times p)} = \frac{p}{2p - p^2}, && (7)
\end{aligned}$$

where R_1 denotes the event that the cloud response is 1, C_2 denotes the event that chunk 2 is in cloud, $\bar{C_2}$ denotes the event that chunk 2 is not in cloud, p denotes the probability that an arbitrary chunk is in cloud, the first equality comes from $P[A|B] = \frac{P[B|A]P[A]}{P[B|A]P[A]+P[B|\bar{A}]P[\bar{A}]}$, and the second equality comes from $P[R_1|C_1] = 1$. From the above, we see that $P[C_2|R_1] \approx \frac{1}{2}$ and therefore $P[\bar{C_2}|R_1] \approx \frac{1}{2}$, if $P[C_2] = p$, p is rather small (as mentioned in Section 3.2). Hence, the attacker still cannot make sure c_2 is in the cloud even when seeing dc response 1.

On the other hand, the attacker can upload c_1 explicitly to ensure the existence of c_1 before performing duplicate check. However, the duplicate check $\langle h(c_1), h(c_2)\rangle$ with the guaranteed existence of c_1 does not help the attacker gain extra existence information of c_2 (see Table 6). ■

Multiple Duplicate Checks. Consider the case where the attacker attempts to learn the existence status of c_2 by invoking duplicate checks more than once. Here, the multiple duplicate checks can be categorized as three types based on the relation among the chunks in the consecutive dc requests.

- **(M1)** The chunks in dc requests are all different.
- **(M2)** The chunks in two dc requests have a single chunk overlapping.
- **(M3)** The chunks in all the dc requests are the same.

In **(M1)**, the attacker can only confirm the chunk inexistence when receiving the dc response 2 by chance. In **(M2)**, without loss of generality, we assume that two dc requests are $\langle h(c_1), h(c_2)\rangle$ and $\langle h(c_1), h(c_3)\rangle$, respectively. After performing duplicate check $\langle h(c_1), h(c_2)\rangle$, the attacker has the option of whether he/she uploads the chunk(s), irrespective of dc response. If the attacker chooses to abort the communication, this results in the case where the next duplicate check $\langle h(c_1), h(c_3)\rangle$ invariably returns dc response 2 because of dirty chunk c_1. If the attacker chooses to upload the chunk(s), c_1 will be in the cloud after the duplicate check $\langle h(c_1), h(c_2)\rangle$. As a consequence, the next duplicate check $\langle h(c_1), h(c_3)\rangle$ with the guaranteed existence of c_1 gives no additional information about the existence status of c_2.

In **(M3)**, the attacker also faces the same difficulty; if the attacker chooses to upload the chunk(s), c_1 will be in the cloud after the duplicate check $\langle h(c_1), h(c_2)\rangle$. As a consequence, the next duplicate check $\langle h(c_1), h(c_3)\rangle$ with the guaranteed existence of c_1 gives no additional information about the existence status of c_2.

Hence, because of the above arguments, due to our design of ZEUS from the perspective of attacker's gain, the multiple duplicate checks collapse to single duplicate check.

Inexistence Privacy of ZEUS. Here, we state a special case where ZEUS in fact also (very occasionally) achieves the inexistence privacy by considering the user behaviors and network conditions. In particular, in the real world, due to the unpredictable user behaviors (e.g., shut down the PC/laptop even when the task is unfinished), the instability of Internet connection (e.g., unstable 3G/4G communication during the high-speed moving), and the software/hardware error (e.g., the crash of OS), it may be normal even for users to have short period of disconnection, naturally resulting in the incomplete uploading. Recall that when seeing the indistinguishable response 2, the attacker originally can invariably claim the inexistence of both queried chunks. Nevertheless, with the above consideration, the cloud can also claim that dc response 2 for $\langle h(c_1), h(c_2)\rangle$ results from the previous incomplete uploading of either c_1 or c_2. In this case, c_1 and c_2 might already have a copy in the cloud. The cloud has possibility that it has c_1 (or c_2) but is still forced to return 2, gaining the inexistence privacy.

We particularly note that the inexistence privacy from the above consideration is just an accidental case; this argument is too strong and impractical, which cannot cover most of the other typical cases with reliable data uploading. Therefore, despite the above argument, we still consider that ZEUS cannot achieve inexistence privacy.

4.2.5 Implementation Details

The dirty chunk list $\mathcal{L}$ can be implemented by a Bloom filter[5] (termed as *dirty Bloom filter, DBF*) for efficient space utilization and queries. The use of DBF in ZEUS incurs the additional processing overhead. Note that DBF needs to be maintained in the memory for avoiding slow disk I/O operations. Both processing and memory overhead from the DBF are minor. Since DBF has to be kept in the memory and the query for a Bloom filter is only constant time, the check of dirty chunks merely imposes minor computation burden.

Two options are available for implementing DBF. First, the cloud allocates a fixed-size memory space for the conventional Bloom filter as DBF. Let p_a be the probability that a chunk will be marked as dirty because of incomplete uploading and N be the expected number of distinct chunks processed by the cloud every year. Then, the memory space of size $-p_a\theta N\ln(p_{fp})/(\ln 2)^2$ bits for DBF suffices to provide the membership query of DBF with the desired false positive probability p_{fp} for θ years [4]. In the case of $p_{fp} = 10^{-6}$, $N = 10^{18}$, $\theta = 1$, and $p_a = 10^{-12}$, Bloom filter occupies approximately 3MB. The second method is to implement dynamic Bloom filter [10] as DBF. Dynamic Bloom filter gradually grows the size when more and more elements are inserted. Compared to the first implementation choice of DBF, dynamic Bloom filter offers adaptive memory usage, at the cost of slightly increased programming effort.

The step 4 of Fig. 2 is used for the data padding when the size of the last chunk is not ϕ. The padding is not necessary in the original client-side deduplication; however, due to the use of XOR obfuscation, the padding is needed in our setting. Thus, in the cloud, each chunk is associated with a *real chunk size*. Most of the chunks will be associated with ϕ as real chunk size. However, a few of the chunks in the cloud with be with a number smaller than ϕ. The real chunk size gives the user ability to remove the padding 0's in the last chunk after the user downloads the file in the future.

The step 6 of Fig. 2 is used if the number of chunks is odd in total. In this case, the user generates and uploads one extra random chunk. This extra chunk can also be seen as the additional bandwidth overhead incurred by ZEUS. However, the extra bandwidth consumption due to this extra chunk is negligible with the consideration of a large number of files are uploaded. Hence, we do not consider the bandwidth consumption in Eq. (2).

5. Bloom filter [4] is a space-efficient probabilistic data structure that supports the membership query with false positives

4.3 ZEUS$^+$

The inexistence privacy of ZEUS is in fact achieved by assuming that even benign users may also abort the uploading due to the unreliable connection or unpredictable user behavior. In practice, the occurrence of such an undesirable situations for benign users can be substantially mitigated by developing a more reliable client-side software. Moreover, compared to the number of real inexistent chunks, the number of chunks that are marked as dirty due to the abnormal communication abortion is rather limited. Thus, one may claim that the inexistence privacy of ZEUS relies on a strong assumption, making it impractical in reality.

Here, the idea of eliminating the strong assumption while keeping the two-side privacy in ZEUS is to combine the use of ZEUS and RT. With the observation that ZEUS and RT offer the existence privacy and inexistence privacy, respectively, the advantage of such a hybrid use (termed as ZEUS$^+$) is obvious; the two-side privacy of ZEUS$^+$ is based on the protocol design, instead of the exterior assumption on the user behavior and network conditions. The formal description of ZEUS$^+$ is shown in Fig. 3. In essence, ZEUS$^+$ can be thought of as a two-stage obfuscation; the first stage randomizes the dc response through the RT technique and the second stage further obfuscates the output of the first stage through the dc table.

One can see from Fig. 3 that the only difference between ZEUS and ZEUS$^+$ is the use of RT (steps 10$\sim$17). In essence, ZEUS$^+$ works like ZEUS, except that the chunk existences (i.e., c_1 existence and c_2 existence) in Table 6 are determined base on the RT principle. Recall that RT principle (see Section 2.3) is that each chunk c_i is associated with a deduplication threshold t_i known only to the cloud and the deduplication will not be triggered if the number of copies of c_i does not exceed t_i. Note that "# c_i" denotes the number of times that the cloud receives c_i. We also note that in ZEUS$^+$ the cloud does not need to keep t_i copies in the storage; an additional counter for keeping track of # c_i suffices to fulfill RT principle without sacrificing disk utilization.

4.3.1 Performance and Privacy Evaluation of ZEUS$^+$

As the communication cost due to the use of RT is dependent on t_i's and the chunk distribution, no closed-form formulation like Eq. (2) can be obtained. As a result, as a

```
Algorithm: ZEUS$^+$
Input: file $f$ with chunk size $\phi$, and dirty chunk list $\mathcal{L}$
01 user partitions $f$ into chunks $c_1, \ldots, c_n$
02 user sets $\hat{n} = n$
03 if bit length $|c_n| \neq \phi$
04    user performs padding to $c_n$
05 if $n$ is odd
06    user picks random chunk $c_{n+1}$ and $\hat{n} = n + 1$
07 for $i \in \{1, 3, \ldots, \hat{n} - 1\}$
08    user performs duplicate check on $\langle h(c_i), h(c_{i+1}) \rangle$
09    if $h(c_i) \notin \mathcal{L}$ and $h(c_{i+1}) \notin \mathcal{L}$
10       if # $c_i$ in the cloud $< t_i$
11          $c_i$ existence is set as 0
12       else
13          $c_i$ existence is set as 1
14       if # $c_{i+1}$ in the cloud $< t_{i+1}$
15          $c_{i+1}$ existence is set as 0
16       else
17          $c_{i+1}$ existence is set as 1
18       cloud replies 1 or 2 according to Table 6
19    else
20       cloud replies dc response 2
21    if user receives dc response 1
22       user uploads $c_i \oplus c_{i+1}$ to the cloud
23       if cloud does not receive $c_i$ and $c_{i+1}$
24          $\mathcal{L} = \mathcal{L} \cup \{c_i, c_{i+1}\}$
25    else
26       user uploads $c_i$ and $c_{i+1}$ to the cloud
27       if cloud does not receive $c_i$ and $c_{i+1}$
28          $\mathcal{L} = \mathcal{L} \cup \{c_i, c_{i+1}\}$
```

Fig. 3: ZEUS$^+$.

combined use of ZEUS and RT, the bandwidth consumption of ZEUS$^+$ will be empirically evaluated based on the real dataset in Section 5

On the other hand, we have the privacy result of ZEUS$^+$ as follows.

***Theorem* 2.** ZEUS$^+$ achieves two-side privacy *under the condition of single duplicate check.*

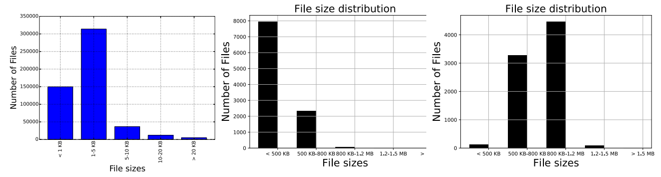
Proof: Due to the similarity between ZEUS and ZEUS$^+$, we can easily know that ZEUS$^+$ achieves weak existence privacy, based on Theorem 1.

Simply speaking, ZEUS$^+$ can be seen as the cloud running RT first to determine positive/negative dc response and then running ZEUS to obfuscate the dc response outputted by RT. The breach of inexistence privacy of ZEUS is due to the distinguishable dc response 2. However, with the use of RT, the dc response 2 for the dc request $\langle h(c_1), h(c_2) \rangle$ can also possibly be attributed to the unsaturated t_1 and t_2. Consider an extreme case that the attacker has perfect confidence that the chunk c_1 is not in the cloud and aims to detect the existence status of the chunk c_2. In this way, ZEUS$^+$ will be degenerated to RT, which has been known to achieve inexistence privacy, as shown in Section 3.3. Thus, we know that ZEUS$^+$ also achieves inexistence privacy. In fact, the lack of inexistence privacy in ZEUS is now compensated by the use of RT, resulting in the two-side privacy in ZEUS$^+$. ■

5 Real Dataset Evaluation

The storage saving will not be affected by the side channel defense. In particular, the schemes belonging to random threshold category [1], [15], [20], [30] gain the privacy by sacrificing the bandwidth saving and the schemes belonging to extra hardware category [14], [26] gains the privacy mainly by forwarding the request packets with certain delays. ZEUS and ZEUS$^+$ also do not sacrifice the storage saving; instead, certain communications that are not needed will be required in ZEUS and ZEUS$^+$. Thus, in our evaluation, we only focus on the communication cost. Note that the communication cost used in our measurements is defined as the number of bits required during the entire chunk uploading process, including the duplicate check (i.e., dc request and dc response) and explicit chunk uploading (i.e., the chunk c, if necessary).

The datasets we used in our evaluation are Enron Email Dataset [9], The Oxford Buildings Dataset [29], and traffic-signs-dataset [28]. We chose these datasets because we believe that ordinary users have the demand to backup the email and multimedia content to the cloud storage. We carried out the evaluations on Fedora 12 Linux operating systems of kernel 2.6.35.9 SMP on Intel Core 2 Duo 3GHz. The evaluation program was written in Python 2.7.6 . We implemented the hash function SHA-256 from OpenSSL library. Fig. 4 reports the statistics of the above three datasets. In the setting of our evaluation, we picked 1000 files uniformly at random and uploaded them to the cloud. Afterwards, we chose 200 files uniformly at random to perform duplicate checks and explicit chunk uploading if necessary.

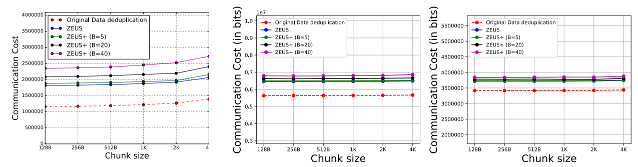


(a) Enron. (b) Oxford. (c) Traffic.

Fig. 4: The statistics of datasets used in our experiment.

Since the schemes belonging to random threshold category [1], [15], [20], [30] only has inexistence privacy guarantee while the schemes belonging to extra hardware category [14], [26] assume the assistance of extra hardware, the comparison is made among the original data deduplication (no privacy is considered, maximum deduplication opportunity), ZEUS, ZEUS$^+$ ($B = 5$), ZEUS$^+$ ($B = 20$), and ZEUS$^+$ ($B = 40$), where B is a deduplication threshold specified in RT (see also Section 2.3). Such a comparison shows addition communication burden incurred by the use of ZEUS and ZEUS$^+$ with different parameter settings. Moreover, both ZEUS and ZEUS$^+$ have an inherent dirty chunk list **(T3)** to prevent the attacker from gaining existence status information by iteratively performing dc requests on the same chunks. Dirty chunks refer to the chunks on which no deduplication is triggered, and furthermore all of the dc requests relevant to dirty chunks will not trigger deduplication. Thus, the use of dirty chunks actually compromise the deduplication benefit. Thus, the same set of evaluations applies with different ratios of dirty chunks (0%, 10%, and 25%)[6]. The evaluation results show the impact of the ratio of dirty chunks on the communication cost.

Fig. 5 shows the communication costs with varying chunk sizes without dirty chunks. This reflects the case of the real communication cost in the sense that the attacker may be unwilling to create the deduplication-based side channel given the use of ZEUS and ZEUS$^+$ and benign users infrequently have the abnormal disconnection. Thus, there would be very few percentages of dirty chunks in the cloud.

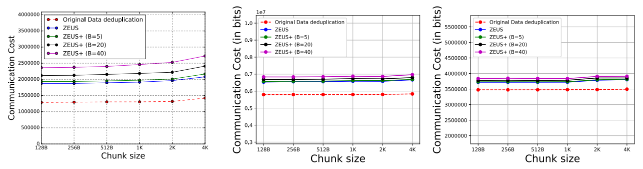


(a) Enron. (b) Oxford. (c) Traffic.

Fig. 5: Communication cost for different chunk sizes (no dirty chunk).

Obviously, since the original data deduplication finds the maximum opportunity for the data deduplication, it has the lowest communication cost. ZEUS has the second lowest communication cost because, compared to ZEUS$^+$, ZEUS does not have the communication cost incurred by RT. Moreover, a comparison among ZEUS$^+$ with different B's clearly shows the larger B implies more communication cost. The gap between the communication costs of original data deduplication and ZEUS actually is dependent on the dataset characteristic. As shown in Eqs. (1) and (2), ZEUS incurs only $p^2\phi$ extra communications, compared to the original data deduplication. Thus, in an extreme case where no duplicate chunk can be found, ZEUS and the original deduplication have the same communication cost, because the duplicate check will not fall in the case of z_4 in Table 5. The above argument also shows that the gap will be varied on different datasets.

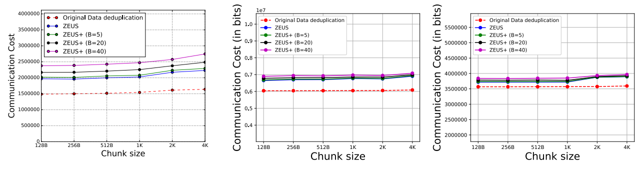
Figs. 5, 6, and 7 show the comparison of different side channel defenses with varying ratios of dirty chunks. Here, for simplicity, though the number of dirty chunks would grow with the increasing number of dc requests, we choose to randomly select a fixed percentage of chunks as dirty chunks, to see the impact of the number of dirty chunks on the communication cost. One can see from Figs. 5, 6, and 7 that, given the deduplication with the side channel defense, more dirty chunks imply more communication cost required. The reason is obvious; if the cloud finds either chunk in the dc request dirty, it cancels the deduplication functionality for the dc request. Thus, more dirty chunks in the cloud imply more communication cost required.



(a) Enron. (b) Oxford. (c) Traffic.

Fig. 6: Communication cost for different chunk sizes (10% dirty chunks).

One can also see that the ratio of dirty chunks has more impact on ZEUS, in contrast to ZEUS$^+$, in terms of communication cost. In fact, one can see that ZEUS$^+$ with $B = 40$ in Figs. 5, 6, and 7 have similar communication cost. This can be attributed to the fact that the chunks already have a very low probability to be deduplicated, even if no dirty is in the cloud. Thus, the consideration of the increased number of dirty chunks makes only negligible impact on the communication cost. On the other hand, the chunks in the original deduplication and ZEUS can find a better deduplication opportunity in the cloud without dirty chunks. However, this deduplication benefit will be nullified to some extent in the cloud with dirty chunks. Hence, we can see a clear increase of communication cost when comparing Figs. 5, 6, and 7.



(a) Enron. (b) Oxford. (c) Traffic.

Fig. 7: Communication cost for different chunk sizes (25% dirty chunks).

6 Comparison and Discussions

The schemes belonging to random threshold category [1], [15], [20], [30] only has inexistence privacy guarantee because the positive dc response always corresponds to the chunk existence, irrespective of the setting of B. In addition, the choice of B determines the trade-off between the deduplication benefits and privacy, but unfortunately remains unclear. Thus, only the requirement of no independent server is satisfied in Table 1. On the other hand, the schemes belonging to extra hardware category [14], [26] assume the assistance of extra hardware, making the requirement of no independent server unsatisfied.

One can see from Section 4.2 that ZEUS serves as the side channel defense without the additional server and manual

6. By its definition, dirty chunks cannot be deduplicated for privacy concern, sacrificing the corresponding deduplication benefit. Thus, "no dirty chunk" is used to represent the basis (or say, ideal case) of deduplication benefit for ZEUS and ZEUS$^+$. On the other hand, as mentioned in Section 4.2.4, dirty chunks could be due to either the attacker who issues the dc request but does not upload the chunk or the benign user who suffers from, for example, network connection problem. We believe that, in reality, the attacker may cause only a few dirty chunks, and there will not be too many network connection problems. Note that obviously 10% and 25% are over-estimations of the ratio of dirty chunks. For example, if the cloud handle 1EB (10^{18} bytes) data, this means that at least 10^{17} bytes are dirty, which is impossible in commercial clouds. We chose 10% and 25% as the maximum ratios of dirty chunks to see their impact on the deduplication benefit.

parameter. ZEUS can also achieve the two-side privacy, however, with the assumption 4.2.4 that not only the attacker but also benign users also create dirty chunks. Recall that due to the unstable connection to the cloud, benign users may indeed create dirty chunks. Nonetheless, the dc response 2 due to such dirty chunks is very unlikely to occur, compared to the dc response 2 due to two inexistent chunks. As a result, the attacker receiving the dc response 2 may still have strong confidence that the queried chunks are not in the cloud. Thus, we only have a triangle sign for two-side privacy in Table 1.

ZEUS$^+$ is the hybrid solution; it inherits the problem of choosing a proper B from RT and therefore has a cross sign in Table 1. However, because RT and ZEUS protect the inexistence privacy and existence privacy, respectively, ZEUS$^+$ works as the side channel defense with two-side privacy.

7 Conclusion

Although client-side data deduplication has been widely adopted by cloud storage services to eliminate redundant data and communications, it leaks the privacy of the chunk existence status, resulting in more sophisticated threats. In this paper, we develop two solutions, ZEUS and ZEUS$^+$, based on the framework of zero-knowledge deduplication response, preventing the attacker from gaining the existence status information from duplicate checks. While ZEUS and ZEUS$^+$ is able to offer a stronger privacy notion, two-side privacy, our real dataset evaluations also confirm that ZEUS and ZEUS$^+$ incur slightly increased communications.

Acknowledgment

This research was partly funded by MOST 105-2923-E-001-002-MY2 (MOST-CNR joint project), MOST 104-2628-E-155-001-MY2, MOST 105-2923-E-002-014-MY3, MOST 105-2218-E-155-010. Mauro Conti is supported by a Marie Curie Fellowship funded by the European Commission (agreement PCIG11-GA-2012-321980). This work is also partially supported by the EU TagItSmart! Project (agreement H2020-ICT30-2015-688061), the EU-India REACH Project (agreement ICI+/2014/342-896), and by the projects "Physical-Layer Security for Wireless Communication", and "Content Centric Networking: Security and Privacy Issues" funded by the University of Padua. This work is partially supported by the grant n. 2017-166478 (3696) from Cisco University Research Program Fund and Silicon Valley Community Foundation.

References

[1] F. Armknecht, C. Boyd, G. T. Davies, and Gjøsteen. Side channels in deduplication: trade-offs between leakage and efficiency. *ACM Conference on Computer and Communications Security (ASIACCS)*, 2017.
[2] Bitcasa. `http://www.bitcasa.com`
[3] 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.
[4] A. Broder and M. Mitzenmacher. Network applications of bloom filters: a survey. *Internet Mathematics*, vol. 1, no. 4, pp. 485 - 509, 2004.
[5] R. Chen, Y. Mu, G. Yang, and F. Guo. BL-MLE: block-level message-locked encryption for secure large file deduplication. *IEEE Transactions on Information Forensics and Security*, vol. 10, no. 12, pp. 2643 - 2652, Dec 2015.
[6] Dropbox. `https://www.dropbox.com`
[7] M. Dutch. Understanding data deduplication ratios. *SNIA Data Management Forum*, 2008.
[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)*, 2001.
[9] Enron Email Dataset. `https://www.cs.cmu.edu/~./enron/`
[10] D. Guo, J. Wu, H. Chen, Y. Yuan, and X. Luo. The dynamic bloom filters. *IEEE Transactions on Knowledge and Data Engineering*, vol. 22, no. 1, pp. 120 - 133, 2010.
[11] Hack Tahoe-LAFS! `https://tahoe-lafs.org/hacktahoelafs/drew_perttula.html`
[12] 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.
[13] H. Hovhannisyan, K. Lu, R. Yang, W. Qi, J. Wang, and M. Wen. A novel deduplication-based covert channel in cloud storage service. *IEEE Global Communications Conference (GLOBECOM)*, 2015.
[14] O. Heen, C. Neumann, L. Montalvo, and S. Defranc. Improving the resistance to side-channel attacks on cloud storage services. *International Conference on New Technologies, Mobility and Security (NTMS)*, 2012.
[15] 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.
[16] T. Jiang, X. Chen, Q. Wu, J. Ma, W. Susilo, and W. Lou. Secure and efficient cloud data deduplication with randomized tag. *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 3, pp. 532 - 543, Mar 2017.
[17] S. Keelveedhi, M. Bellare, and T. Ristenpart. Dupless: Server-aided encryption for deduplicated storage. *USENIX Security Symposium*, 2013.
[18] 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.
[19] 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 Systems*, vol. 25, no. 6, pp. 1615-1625, 2014.
[20] S. Lee and D. Choi. Privacy-preserving cross-user source-based data deduplication in cloud storage. *International Conference on ICT Convergence (ICTC)*. 2012.
[21] Mega. `https://mega.nz`
[22] Mozy. `https://mozy.com/`
[23] M. O. Rabin. Fingerprinting by random polynomials. Center for Research in Computing Technology, Harvard University. Tech Report TR-CSE-03-01. Retrieved 2007-03-22.
[24] H. Ritzdorf, G. O. Karame, C. Soriente, and S. Čapkun. On information leakage in deduplicated storage systems. *ACM Cloud Computing Security Workshop (CCSW)*, 2016.
[25] V. Rabotka and M. Mannan. An evaluation of recent secure deduplication proposals. *Journal of Information Security and Applications*, vol. 27, pp. 3–18, Apr. 2016.
[26] Y. Shin and K. Kim. Differentially private client-side data deduplication protocol for cloud storage services. *Security and Communication Networks*, vol. 8, pp. 2114 - 2123, 2015.
[27] Tahoe-LAFS. `https://tahoelafs.org`
[28] Traffic signs dataset. `http://www.cvl.isy.liu.se/en/research/datasets/traffic-signs-dataset/download/`
[29] The Oxford Buildings Dataset. `http://www.robots.ox.ac.uk/\%7Evgg/data/oxbuildings/`
[30] B. Wang, W. Lou, and Y. T. Hou. Modeling the side-channel attacks in data deduplication with game theory. *IEEE Conference on Communications and Network Security (CNS)*, 2015.
[31] 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, pp. 1681–1710, Sept 2016.
[32] W. Xia, Y. Zhou, H. Jiang, D. Feng, Y. Hua, Y. Hu, Y. Zhang, and Q. Liu. FastCDC: a fast and efficient content-defined chunking ap-

proach for data deduplication. *USENIX Annual Technical Conference*, 2016.