Secure Storage with Replication and Transparent Deduplication

Iraklis Leontiadis, Reza Curtmola

New Jersey Institute of Technology, USA
{leontiad,crix}@njit.edu

Abstract. We seek to answer the following question: *To what extent can we deduplicate replicated storage ?* To answer this question, we design ReDup, a secure storage system that provides users with strong integrity, reliability, and transparency guarantees about data that is outsourced at cloud storage providers. Users store multiple replicas of their data at different storage servers, and the data at each storage server is deduplicated across users. At the same time, users rely on remote data integrity mechanisms to check the integrity of their data. In this setting, it is adequate to consider a stronger and more realistic adversarial model, one in which we also allow collusions between storage servers and users of the system. As such, a cloud storage provider (CSP) could store less replicas than agreed upon by contract, unbeknownst to the users. ReDup defends against such adversaries by making replica generation to be time consuming so that a dishonest CSP cannot generate replicas on the fly when challenged by the users.
In addition, ReDup employs transparent deduplication, which means that users get a proof attesting the deduplication level used for their files at each replica server, and thus are able to benefit from the storage savings provided by deduplication. The proof is obtained by aggregating individual proofs from replica servers, and has a constant size regardless of the number of replica servers. Our solution scales better than state of the art and is provably secure under standard assumptions.

1 Introduction

Outsourcing storage to cloud storage providers (CSPs) has become a popular and convenient practice. Despite its cost-saving benefits, cloud storage remains rife with security issues [12]. There are reported incidents of lost data or service unavailability due to power outages [11], hardware failure, software bugs [10], external or internal attacks, negligence, or administrator error. Moreover, cloud infrastructures lack transparency and data owners have to fully trust the CSPs. All these factors limit the suitability of cloud platforms for applications that require long-term data integrity and reliability. Of particular concern to data owners is that although storage can be outsourced, the liability in case data is lost, damaged, or stolen cannot be outsourced.

Several approaches can be used to ease these concerns. First, to improve *reliability*, data can be stored redundantly by replicating it across geographically dispersed cloud storage servers. Whenever data is damaged at one replica server (RS), data can be retrieved from healthy replication servers in order to repair the damaged data and restore the desired level of redundancy. Second, the transparency of cloud infrastructures can be improved by using an auditing mechanism such as *remote data integrity checking* (RDIC) [4, 7, 19], which allows data owners to efficiently check the integrity of data stored at untrusted CSPs.

At the same time, a popular trend is that of *data deduplication*, which allows CSPs to reduce their storage costs by exploiting common properties of files stored by different users. When different users upload the same file at a CSP, deduplication ensures that only one copy is stored. Recent studies show that cross-user data deduplication can lead to significant savings in storage costs, ranging from 50%[17] to 95%[17, 18].

Although deduplication across multiple users' files is economically beneficial for CSPs, the individual users whose files get deduplicated do not benefit from these savings. Typically, each user gets charged an amount that is proportional with the amount of data stored and any savings due to deduplication with other users' data are not passed to the end user. Recently, Armknecht et al. [3] introduced *transparent deduplication*, which gives users full transparency on the storage savings achieved through deduplication. This enables a new pricing model which takes into account the level of deduplication of the data: The more users store the same piece of data, the lower each individual user gets charged for storing that piece of data.

We wish to design a system that provides both integrity and reliability (via RDIC and replication) as well as cost-efficient storage via transparent deduplication, when faced with an economically motivated adversary that controls some or all of the storage servers. Adversarial servers will try to "cut corners" and gain an economic advantage as long as it remains undetected. This can be achieved either by using less storage than required to fulfill their contractual obligations for replication, or by charging users according to a deduplication level that is

lower than the real one. To achieve this goal, we are faced with two main challenges that were not addressed by previous work:

Challenge 1: *Overcoming the ROTF attack*. Previous work has established that the storage servers should be required to store different and incompressible replicas [8, 9]. Otherwise, if all replicas are identical, an economically motivated set of colluding servers may try to save storage by simply storing only one replica and redirecting all data owner's RDIC challenges to the one server storing the replica. One approach to generate different replicas is by encrypting the original file with different keys. This mitigates the "redirection" attack described earlier: A storage system cannot successfully pass RDIC challenges for the t replicas without actually storing the t replicas.

However, in order to enable deduplication across users, the replicas generated by two users for the same file for the same storage server should be identical. For example, two users must generate identical replicas H_1 for storage server RS_1, identical replicas H_2 for storage server RS_2, etc. To achieve this, users should use the same keys to generate replicas for the same storage server. This introduces the *replicate on the fly (ROTF)* attack, a novel attack unique to this setting: if at least one user shares with the CSP the keys used to generate replicas, then the CSP can recover and store only the original file instead of storing the t replicas. The CSP can then generate on the fly a particular replica to pass an RDIC challenge for that replica. This will hurt the reliability of the storage system, because the CSP does not store t replicas, unbeknownst to the client.

Challenge 2: *Efficient transparent deduplication for multiple replicas*. Transparent deduplication has been investigated only when the data is stored at a single cloud server [3]. When data is replicated at multiple storage servers, the previous solution does not scale well and transparent deduplication becomes more challenging to achieve securely and efficiently.

Contributions: In this work, we propose ReDup, a secure storage solution with Replication and transparent deDuplication. ReDup provides users with strong integrity, reliability, and transparency guarantees about data that is outsourced at cloud storage providers. To the best of our knowledge, ours is the first proposal to provide all these guarantees at the same time. Specifically, ReDup offers:

- *Integrity*: ReDup employs a remote data integrity checking (RDIC) mechanism to allow users to check the integrity of their outsourced data. Each user runs periodically a RDIC protocol to check the health of her data at each replica server. Whenever data damage is detected at a replica, data from healthy replica servers can be used to restore the desired replication level. Such a RDIC mechanism allows users to assess the health of their data by periodically verifying the integrity and replication level of their data.
- *Reliability*: ReDup provides data reliability by replicating a user's data at multiple storage servers that are geographically dispersed. Since different users may have different reliability needs, ReDup offers multiple replication levels and allows users to choose a replication level suitable for their needs. We consider a more realistic adversarial model which includes not only collusions between storage servers, but also between storage servers and users of the system. This introduces a novel attack, the *replicate on the fly (ROTF)* attack, which allows the CSP to store only one copy of the data and generate replicas on the fly to respond to RDIC challenges. To defend against the ROTF attack, we make the replica generation be time consuming and we enhance the standard RDIC challenge-response model to include an additional check regarding the time needed to generate the RDIC proof. In this way, dishonest CSPs that try to generate replicas on the fly will not be able to pass the RDIC challenges. In ReDup, replicas are generated from the original file by applying a novel *shortcut-free time consuming function (SFTCF)*, which we define formally and then instantiate with a butterfly construction.
- *Efficient and transparent deduplication for multiple replicas*: When a user's data is replicated at multiple servers, ReDup provides a proof to the user attesting the deduplication level that occurs at each replica server. The proof is obtained by aggregating individual deduplication level proofs from replica servers, and has a constant size regardless of the number of replica servers. Users are charged inversely proportional to the deduplication level of each of their replicas. ReDup reconciles the seemingly contradictory notions of replication and deduplication: The data of each user is replicated at multiple servers to increase reliability, whereas deduplication is applied independently at each replication server across different users' data to reduce storage costs.
- *Collusion resistance*: These guarantees hold even in the presence of collusion between replica servers or between replica servers and users.

The remainder of the paper is organized as follows: In Section 2 we present background information and related work. We describe the system and adversarial model in Section 3. In Section 4 we provide some preliminaries for

our basic building blocks. A solution overview of the protocol is depicted in Section 5 and its full description is described in Section 6. Finally, Section 7 analyses the security of ReDup.

2 Background and Related Work

Remote data integrity checking for multiple replicas. Remote data integrity checking (RDIC) [4, 13, 19] is a mechanism that allows to check the integrity of data stored at an untrusted cloud storage provider (CSP). A data owner uploads at the CSP their data together with metadata consisting of a set of verification tags, and then periodically challenges the CSP to provide a proof about the health of the data. The CSP is able to create such a proof based on the data and the metadata initially uploaded by the owner.

To ensure data reliability over time, the data owner creates multiple replicas of the data and stores them at multiple storage servers. The data owner then uses RDIC to periodically check the health of each replica, and if a replica is found corrupt, data from the other healthy servers is used to restore the desired redundancy level in the system [8, 9]. Previous work has established that the storage servers should be required to store different and incompressible replicas [8, 9].

Transparent Deduplication. Armknecht *et al.* [3] introduced the notion of *transparency* for deduplicated storage: The cloud provides to users proofs that attest the level of deduplication across users employed by the cloud over their files. This enables a new pricing model which takes into account the level of deduplication of the data, allowing end users to get the benefits of deduplication. Users are protected against a cloud provider that uses a certain deduplication level, but charges users based on a lower level.

The solution lies in a Merkle tree tailored for this application, which allows an honest user to verify a) how many users have also uploaded the same file and b) that information about the user's file has been correctly incorporated in the bill issued by the cloud. Although this solution is efficient when files are stored at a single storage server, when translated to a multiple replica scenario it becomes inefficient as it would require multiple instances of the Merkle tree, one per each replica.

2.1 Other Related Work

Current literature in remote data integrity checking protocols either does not address deduplication in a multiple replica scenario, or does not consider the challenging multi-user scenario with collusions between users and economically-motivated replica servers.

Multi-User with Tags Deduplication. Vasilopoulos *et al.* [21] proposed a combination of existing deduplication schemes with proofs of retrievability to further reduce the storage cost of tags for identical blocks. In their model, there is a single replica storage policy and users do not collude with the cloud provider. Armknecht *et al.* [2] considered the same model, whereby a single replica server stores only once tags coming from different users for the same data block. The solution lies on shared aggregated tags based on BLS signatures [6] incorporating the secret keys of all users and can tolerate collusions between users and a malicious cloud storage provider: Deleting a deduplicated block tag and obtaining the secret key from a malicious cannot help the cloud to reconstruct the tag without the participation of all the other users. Their model, however, does not consider providing both multiple replica storage and deduplication.

Replicated Storage. Curtmola *et al.* [9] considered a model in which a single user stores replicas of a file at multiple storage servers to tolerate faults. The user uses an RDIC protocol to verify faithful storage at each replica server. However, this scenario does not consider multiple users or the deduplication functionality. Armknecht *et al.* [1] considered a multiple replica storage scenario enhanced with proofs of correct replication by the user.

3 System and Adversarial Model

3.1 System Model

A set of users, $\mathcal{U} = U_1, U_2, U_3, \ldots, U_m$, store their files at a cloud storage provider (CSP). To ensure data reliability and protect against data damage, the CSP exposes an interface that allows users to store multiple replicas of their files at different replication servers. Each user uses remote data integrity checking (RDIC) to check the integrity of their replicas stored at each replica server; in case data damage is detected at a replica server, the user leverages replicas from other healthy replica servers to restore the desired level of redundancy.

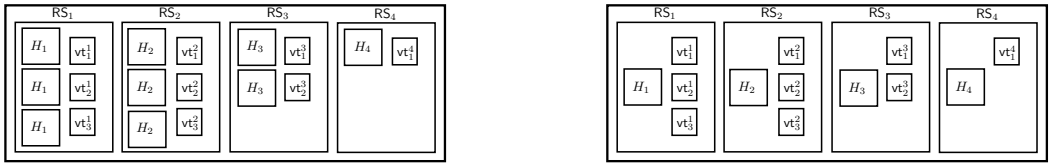
Replication level. As users have different budgets and needs, the CSP allows users to choose the desired *replication level* (rl) for their files. Without loss of generality, we assume the CSP offers a fixed number of replication levels (*e.g.*, in practice it may offer three levels, corresponding to high, medium, and low reliability). Fig. 1(a) shows an example with three users choosing different replication levels, $\mathsf{rl}_1 = 4, \mathsf{rl}_2 = 3, \mathsf{rl}_3 = 3$. User U_1, who chose $\mathsf{rl}_1 = 4$, will generate four replicas H_1, H_2, H_3, H_4 and the corresponding RDIC verification tags $\mathsf{vt}_1^1, \mathsf{vt}_1^2, \mathsf{vt}_1^3, \mathsf{vt}_1^4$, and will store them at replication servers $\mathsf{RS}_1, \mathsf{RS}_2, \mathsf{RS}_3$ and RS_4. Whereas user U_3, who chose $\mathsf{rl}_3 = 2$, will generate two replicas H_1, H_2 and RDIC verification tags $\mathsf{vt}_3^1, \mathsf{vt}_3^2$, and store them at servers $\mathsf{RS}_1, \mathsf{RS}_2$.

We assume users agree on a common key fk that will be used to generate the replicas. This ensures that if two users want to store the same file, the replicas generated for the file will be identical, thus allowing deduplication to be applied at each replica server. The mechanism used to agree on this common key is outside the scope of this paper; for example, users can rely on variants of convergent encryption to derive this key securely, either with the aid of a trusted server [14] or with a multiparty computation protocol between users [15].

Deduplication level. Whenever possible, the CSP employs deduplication across different users' files at each replication server: If multiple users store identical files, the CSP keeps only one copy. In the example of Fig. 1(b), servers $\mathsf{RS}_1, \mathsf{RS}_2, \mathsf{RS}_3$ perform deduplication for the files H_1, H_2, H_3, and the *deduplication level* (dl) is $\mathsf{dl}_1 = 3, \mathsf{dl}_2 = 3, \mathsf{dl}_3 = 2$, respectively. Server RS_4 does not perform deduplication, as it already stores only one copy of file H_4. Notice that deduplication occurs at each replication server independently, meaning that different copies of the same file will be dispersed along replica servers to ensure reliability, but at each replica server deduplication is applied and only one copy of multiple identical files is stored.

Pricing model. The system divides time into epochs (*e.g.*, one epoch is one day) and users get charged at the end of each epoch. A user's bill for each epoch is directly proportional to the chosen replication level and inversely proportional to the deduplication level that occurs at each replica server. This means that if a user is uploading a file at a replica server and that file is already stored by r other users, then each of the $r + 1$ users that store the file will get charged an amount that is $r + 1$ smaller compared to the case when no deduplication occurs.

To prevent a dishonest CSP from charging users more by claiming a lower deduplication level, the system employs transparent deduplication: the CSP provides to each user at the end of each epoch a proof that attests to the deduplication level that occured at each replication server.



(a) Before Deduplication **(b)** After Deduplication

Fig. 1: Multi-replica Deduplication

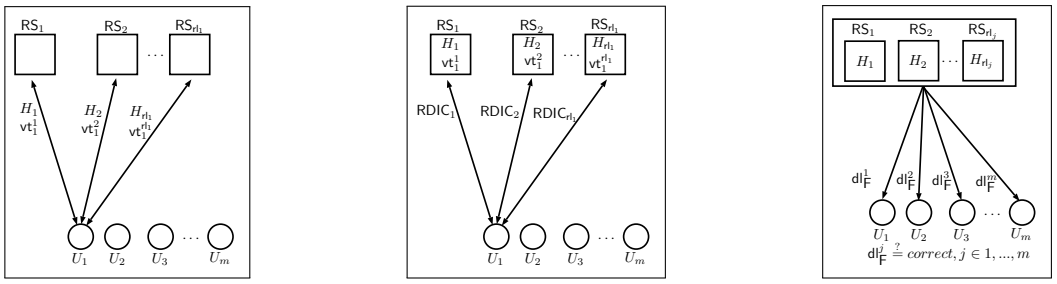
System overview. As depicted in Fig. 2, the system consists of four protocols: $\mathsf{Setup}, \mathsf{Replicate}, \mathsf{RDIC}$, and $\mathsf{AttestDedup}$. Each user U_j, with $1 \leq j \leq n$, runs these protocols. We give an overview of these protocols next:

$\mathsf{Setup}(1^\lambda, n, \mathsf{rl}_j)$: During Setup, each user U_j chooses rl_j, the replication level for her files. Users also generate the secret keys that will be used during the other protocols of the system.

$\mathsf{Replicate}(\mathsf{F}, \mathsf{fk}, \mathsf{k}_j, \mathsf{rl}_j)$: Each user U_j runs the $\mathsf{Replicate}$ protocol to generate replicas $H_1, H_2, \cdots, H_{\mathsf{rl}_j}$ for file F, using the key fk. Identical files by different users are stored only once at each replica server, but are stored multiple times according to the replication level choice rl_j to ensure reliability. User U_j also computes the set of RDIC verification tags vt_j^i on top of each replica H_i, with $1 \leq i \leq \mathsf{rl}_j$. Finally, user U_j uploads replica H_i and verification tags vt_j^i at server RS_i, with $1 \leq i \leq \mathsf{rl}_j$.

$\mathsf{RDIC}(\mathsf{F}, < U_j : Q >, < \mathsf{RS}_i : \sigma_i >)$: Each user U_j engages in a remote data integrity checking protocol (RDIC) with replica server RS_i to check faithful storage of the replica file H_i, for $1 \leq i \leq \mathsf{rl}_j$. In the RDIC protocol, the user issues a challenge Q to a replica server, and the server responds with a proof σ_i that attests

the integrity of the replica stored at that server (this proof is constructed using the challenged replica file and its corresponding verification tags). The user verifies the correctness of the proof received from the server. Unlike in a standard RDIC protocol, the user performs an additional check in order to prevent the ROTF attack: whether the server's response time is below a threshold T.



(a) Replicate **(b)** RDIC **(c)** AttestDedup

Fig. 2: Replicate, RDIC, AttestDedup.

AttestDedup(ep, U_j, F): Each user U_j runs the AttestDedup protocol during each epoch ep to verify the CSP's claim about the deduplication level employed for the user's replica files during that epoch. During each time epoch, the CSP issues a bill to each user based on the replication level chosen by the user for her files, and on whether the user's replica files benefited from deduplication. The bill includes a proof that allows the client to verify the deduplication level of its replicas. The AttestDedup protocol is needed to prevent dishonest CSPs from claiming a lower deduplication level that the one deployed at its servers in order to charge users a higher bill.

3.2 Adversarial Model

In ReDup we assume a *rational* adversary who does not deviate from the protocol execution, unless economic incentives force the cloud service provider to act maliciously. Herewith, we refer to the rational cloud as malicious cloud whenever it does not follow the protocol rules in order to increase its revenues or decrease its costs violating the contractual promises it offers to a signer. Recall the goal of a user U_j is to get sound guarantees that the cloud stored rl_j replicas of a file F. The second requirement of the user is to get assurances that the cloud correctly computed the deduplication level of a file and did not cheat by overcharging users. In contrast with existing single replica scenario, a multi-replica service storage service scenario poses challenging attacks:

1. **Replicate on the fly** (ROTF): A client has agreed with the cloud for rl_j replica servers to ensure reliability in case of a catastrophic damage of a server. The client creates rl_j copies of the same file and uploads them to the CSP. However a malicious cloud in order to reduce its storage costs, stores the file in less than rl_j replica servers and whenever the client challenges the CSP with a remote data checking protocol to assure faithful file replica storage, it generates replica copies on the fly and computes correct proofs from the existing replicas, thus passing the remote data checking protocol undetectably.
   To remedy the aforementioned the client creates rl_j different replica copies for the same file. However a colluding user can share its secret key information with a colluding replica server thus enabling the latter to reduce its storage cost and cheat under the radar of an honest user whenever the latter challenges the replica server to prove file possession.
2. **Incorrect Deduplication level**: In order to increase its revenues a cloud service provider may advertise appealing charging costs for deduplicated files: Users are charged at each period inversely proportional to the number of times a files has been stored in a replica server. A malicious cloud though can always claim unique file storage on similar files, thus increasing its revenues.

3.3 Security Requirements

Inspired by the aforementioned adversarial model we define the security requirements of our system. The security guarantees protect an honest user from a coalition of malicious replica servers and users who will try to not follow the contractual agreement with respect to **1)** faithful storage of a file at rl_j replica servers and **2)** the correct deduplication level on the bill.

3.3.1 Collusion Resistant Replicas Integrity We do not consider any confidentiality guarantees or privacy attacks due on deduplication. In contrast with previous multi-user Cloud storage services, whereby the client can collude with a single replica server in order to convince an honest user for the faithful storage of a file, in our model multiple replica servers RS can collude either in between them, or they can maliciously collaborate with a malicious user. We build a stronger and more realistic adversarial model in terms collusions. ReDup exhibits similar security requirements with multiple replica remote data checking guarantees, whereby a malicious cloud proves to a user that it faithfully stores rl_j copies of a file at their entire form. In contrast with single user multiple replica remote data checking protocols [1, 9] ReDup aims to assure data integrity of each replica file under ROTF attacks between malicious users and replica servers.

More specifically we say that ReDup assures:

- *SG1: Replica integrity*, if each replica server RS_i can convince a user U_j with high probability that the replica H_i remains intact in its entirety, for $1 \leq i \leq \mathsf{rl}_j$.
- *SG2: Storage Allocation*, if the amount of data stored by a CSP for a file F of size $|\mathsf{F}|$ on a replication level rl is at least $\mathsf{rl}|\mathsf{F}|$.

SG1 protects the users from a CSP that does not store replica files in their entirety. SG2 protects users from a CSP that does not respect its contractual obligations of storing rl_j replicas and tries to reduce its costs by storing less replicas. Together, SG1 and SG2 imply that the CSP faithfully stores all rl_j replica copies of a file F. We capture these two guarantees under the *Collusion Resistant Replicas Integrity* ($\mathsf{CR^2P}$) property, formulated with a standard security game between the adversary and the challenger:

Following previously formats for game based definitions [4, 7, 19] in RDIC protocols we extend them in order to follow our stronger adversarial model whereby a replica server can collude either with another replica server or with malicious users. During the game an adversary $\mathcal{A}$ creates the *environment*, consisting of users, replica servers, files, replicas, authentication tags and keys. As in our adversarial model we assume $\mathcal{A}$ can collude with another user U or another replica server RS we allow $\mathcal{A}$ during the game to have full control on them by having access to the oracles, which provide all the secret transcripts. We denote by $\mathcal{O}^{\mathsf{abc}}(k, l, m; x, y, z)$ the abc oracle which takes as inputs the parameters k, l, m and executes its code with local variables x, y, z, which are unknown to the caller–the adversary $\mathcal{A}$. We denote with capital lowercase $\mathbf{a}_i$ a list which can take up to i elements. With $\mathcal{U}'$ we refer to the set of corrupted users and with $\mathcal{U} - \mathcal{U}'$ to the uncorrupted users. Subsequently with $\mathcal{RS}'$ we refer to the corrupted replica servers and with $\mathcal{RS} - \mathcal{RS}'$ to the faithful servers. The notation $U_i \rightarrow \mathcal{U}$ states the insertion of element U_i to the set $\mathcal{U}$. For simplicity of presentation we do not assume two different adversaries $\mathcal{A}$ which are stateless but we make the implicit assumption derived from the soundness proofs of zero-knowledge protocols that the extractor does not participate in the real execution of the protocol. $\mathcal{A}$ has access to the following oracles:

- $\mathcal{U}', \mathcal{RS}' \leftarrow \mathcal{O}^{\mathsf{Setup}}(\mathbf{uid}_{m-1}, \mathbf{sid}_t; m, \mathsf{rl}_{\mathsf{uid}})$: Whenever invoked with parameters a list of users ids $\mathbf{uid}_{m-1}$ and replica servers ids $\mathbf{sid}_t$, the $\mathcal{O}^{\mathsf{Setup}}$ oracle stores the ids to the appropriate sets $\mathcal{U}', \mathcal{RS}'$, denoting the list of corrupted users and replica servers, respectively.
- $H_i \leftarrow \mathcal{O}^{\mathsf{GenReplica}}(\mathsf{F}, \mathsf{uid}; \mathsf{fk}, i)$: The $\mathcal{O}^{\mathsf{GenReplica}}$ oracle takes as input a file F and a user id uid. It first checks if $\mathsf{uid} \in \mathcal{U}'$. If that user is corrupted then it outputs the replica copy H_i for the replica server RS_i for that user on file F using the key fk. The oracle keeps track of the uploaded files and for similar files it uses the same key in order to simulate the deduplication process. Finally $\mathcal{O}^{\mathsf{GenReplica}}$ also stores $H_i \rightarrow H$ in the list H and sends H_i to $\mathcal{A}$.
- $\mathsf{vt}^i_{\mathsf{uid}} \leftarrow \mathcal{O}^{\mathsf{TagFile}}(H_i, \mathsf{uid}; \mathsf{k}_{\mathsf{uid}})$: The $\mathcal{O}^{\mathsf{TagFile}}$ oracle on input H_i and uid first checks if $\mathsf{uid} \in \mathcal{U}'$ and $H_i \in H$. If both hold, then computes the tags $\mathsf{vt}^i_{\mathsf{uid}}$ using $\mathsf{k}_{\mathsf{uid}}$ and forwards them to $\mathcal{A}$.
- $c^{\mathsf{uid}}_{\mathsf{F}} \leftarrow \mathcal{O}^{\mathsf{Challenge}}(\mathsf{F}, \mathsf{uid}; \mathsf{k}_{\mathsf{uid}})$: The $\mathcal{O}^{\mathsf{Challenge}}$ oracle outputs a challenge for file F for the user $U_{\mathsf{uid}} \in \mathcal{U} - \mathcal{U}'$.
- $\beta \leftarrow \mathcal{O}^{\mathsf{Verify}}(\mathsf{proof}^{\mathsf{uid},i}_{\mathsf{F}}, \tau_i; T)$: The $\mathcal{O}^{\mathsf{Verify}}$ oracle takes as input a proof $\mathsf{proof}^{\mathsf{uid},i}_{\mathsf{F}}$ and a response time τ_i. It outputs $\beta = 0$ if either the proof is not valid or $\tau_i > T$, otherwise it sets $\beta = 1$.

During the $\mathbf{Game}^{\mathsf{CR^2P}}_{\mathcal{A}}$ game the adversary communicates with the oracles in order to create the environment to be challenged upon as follows:

$\mathbf{Game}_{\mathcal{A}}^{\mathsf{CR^2P}}$

1 : $\mathcal{U}', \mathcal{RS}' \leftarrow_{\$} \mathcal{A}^{\mathcal{O}^{\mathsf{Setup}}}$ //$\mathcal{A}$ compromises users and servers
2 : **for** $i = 1...\mathsf{rl}_{\mathsf{uid}}$ **do**
3 : $H_i \leftarrow_{\$} \mathcal{A}^{\mathcal{O}^{\mathsf{GenReplica}}(\mathsf{F},\mathsf{uid};\mathsf{fk},i)}$ //$\mathcal{A}$ learns replica copies
4 : $\mathsf{vt}_{\mathsf{uid}}^{i} \leftarrow_{\$} \mathcal{A}^{\mathcal{O}^{\mathsf{TagFile}}(H_i,\mathsf{uid};\mathsf{k}_{\mathsf{uid}})}$ //$\mathcal{A}$ asks for verifications tags
5 : $c_{\mathsf{F}}^{\mathsf{uid}} \leftarrow_{\$} \mathcal{A}^{\mathcal{O}^{\mathsf{Challenge}}(\mathsf{F},\mathsf{uid};\mathsf{k}_{\mathsf{uid}})}$ //$\mathcal{A}$ is challenged
6 : $\mathsf{proof}_{\mathsf{F}}^{\mathsf{uid},i}, \tau_i \leftarrow \mathcal{A}(\mathcal{U}', \mathcal{RS}', \mathsf{F}, H_i, \mathsf{vt}_{\mathsf{uid}}^{i}, c_{\mathsf{F}}^{\mathsf{uid}})$ //$\mathcal{A}$ issues a proof $\mathsf{proof}_{\mathsf{F}}^{\mathsf{uid},i}$
7 : $\beta_i \leftarrow \mathcal{O}^{\mathsf{Verify}}(\mathsf{proof}_{\mathsf{F}}^{\mathsf{uid},i}, \tau_i; T)$
8 : **return** $\beta = \bigwedge \beta_i$ //Experiment is successful if $\beta \stackrel{?}{=} 1$

Finally the game outputs a value $\beta \in \{0, 1\}$. We define the success probabilities of an adversary $\mathcal{A}$ playing the $\mathbf{Game}_{\mathcal{A}}^{\mathsf{CR^2P}}$ game as: $\mathbf{Succ}_{\mathcal{A}}^{\mathsf{CR^2P}} = \Pr[\mathbf{Game}_{\mathcal{A}}^{\mathsf{CR^2P}} = 1]$. The heuristic is that if the output of the experiment equals 1 then $\mathcal{A}$ should posses all replica copies $H_1 \dots H_{\mathsf{rl}_j}$. In order to formulate that heuristic we employ the notion of the extractor $\mathcal{E}$, which can communicate with the adversary and rewind her at different steps in order to extract a file F from all replica copies $H_1 \dots H_{\mathsf{rl}_j}$. We define the success probability of the extractor $\mathcal{E}$ as follows: $\mathbf{Succ}_{\mathcal{A}}^{\mathsf{Extract}} = \Pr[\mathsf{F} = \mathsf{F}_{\mathsf{fh}} | \mathsf{F}_{\mathsf{fh}} \leftarrow \mathcal{E}^{\mathcal{A}}]$.

Definition 1. *($\mathsf{CR^2P}$: Collusion Resistant Replica Possession)* ReDup *system guarantees Collusion Resistant Replica Possession if under any collusions for a set users $|U|$ who have stored the file F in rl_j replica servers $\mathsf{RS}_1, \mathsf{RS}_2, \mathsf{RS}_3, \cdots, \mathsf{RS}_{\mathsf{rl}_j}$ and for any PPT adversary $\mathcal{A}$, for any security parameter λ and a negligible quantity $\mathsf{negl}(\lambda)$, it holds that:*

$$\Pr[\mathbf{Succ}_{\mathcal{A}}^{\mathsf{Extract}} \leq \mathsf{negl}(\lambda) \wedge \mathbf{Succ}_{\mathcal{A}}^{\mathsf{CR^2P}} > \mathsf{negl}(\lambda) : \mathcal{E} \overset{\mathcal{U}', \mathcal{RS}', \mathsf{F}, H_i, \mathsf{vt}_{\mathsf{uid}}^{i}, c_{\mathsf{F}}^{\mathsf{uid}}}{\longleftrightarrow} \mathcal{A} \leftrightarrow \mathbf{Game}_{\mathcal{A}}^{\mathsf{CR^2P}}] \leq \gamma$$

Intuitively, the $\mathsf{CR^2P}$ definition establishes an upper bound γ on the event that an adversary $\mathcal{A}$ wins the $\mathbf{Game}_{\mathcal{A}}^{\mathsf{CR^2P}}$ game with non-negligible probability and that an extractor $\mathcal{E}$ is not able to extract the file after interacting with $\mathcal{A}$.

3.3.2 Deduplication Correctness A rational CSP aims at increasing its revenues. Following a greedy strategy it can deny deduplication on identical files on its storage log and charge users as owning a file that is not identical with any other. Claiming a smaller deduplication level on the storage log included at each bill, allows the CSP to enjoy higher charges. As such, the cloud ought to prove to a user U_j that is legitimately charged: All users, who have deduplicated the file, have been accumulated in order to enjoy reduced costs. The cloud can accumulate the deduplication level with respect to a file in an authenticated data structure. For each file it keeps track of user ids as input to the data structure for a file and whenever a user wants to verify deduplication correctness for a file, then it shows the user the size of the structure. However, acting maliciously it can delete elements undetectably. Therefor for deduplication correctness and to detect the malicious behavior of the CSP apart from the size of the data structure users ask for membership proofs of their id associated to the file they have stored. In that way when the CSP claims a wrong, smaller deduplication level by deleting user ids, it can be detected by the membership proofs verification.

Thus to assure deduplication correctness it suffices to attest:

- **dc1**) membership of each user who has stored identical files on the authenticated data structure.
- **dc2**) a correct size of the authenticated data structure.

Finally grouping together **dc1** and **dc2** we define (informally) deduplication correctness as:

SR3: Deduplication Correctness assures that a malicious CSP cannot claim a deduplication level $\mathsf{dl}_{\mathsf{F}}^{i'}$ for a file F at replica server RS_i different than the real one $\mathsf{dl}_{\mathsf{F}}^{i}$, in an undetectable way.

Definition 2. *(Deduplication Correctness) During an epoch ep, each user U_j stores file replicas at replica servers $\mathsf{RS}_1, \mathsf{RS}_2, \mathsf{RS}_3, \cdots, \mathsf{RS}_{\mathsf{rl}_j}$ and the deduplication level for a file F at each replica server RS_i is $\mathsf{dl}_{\mathsf{F}}^{i}$. The system guarantees Deduplication Correctness if, for any epoch ep, an honest user U who runs the $\mathsf{AttestDedup}(ep, U, \mathsf{F})$ algorithm will detect if a dishonest CSP claims a deduplication level $\mathsf{dl}_{\mathsf{F}}^{i'} \neq \mathsf{dl}_{\mathsf{F}}^{i}$ for file F at replica server RS_i.*

4 Preliminaries

In this section we present the preliminaries building blocks of our design.

4.1 Shortcut Free - Time Consuming Function (SFTCF)

We put forward the definition of a Shortcut Free and Time Consuming Function S. S is a symmetric trapdoor function which takes input I with v blocks and outputs H with v blocks. Moreover S should adhere to the shortcut free property which states that the holder of any output H' with $v' < v$ blocks will not help her to recover the remaining $v - v'$ output blocks in time less than a threshold T, even when it knows the trapdoor of S. Finally the running time of S should be considerably greater than the running time of a well known functionality G. For instance G in our case is the time to fetch from the hard disk v blocks and send them to a destination. The properties of a SFTCF are:

1. **Shortcut Freeness**: Storing any intermediate state st, which is smaller than the original size v of the input, does not result in evaluation time smaller than the running time of S on the original input of size v: S cannot be decomposed in $S_1, S_2, \ldots, S_v$, such that $S(v) = S_1() \circ S_2() \circ, \ldots, \circ S_v()$
2. **G-Detectable Time Consumption**: Evaluation of the function requires computational resources, which results in a considerable detectable time for its evaluation. That is, for another function G whose complexity is $\Omega_G(v)$ we say that S guarantees G-Detectable Time Consumption if $\Omega_S(v) \gg \Omega_G(v)$.

Security. An SFTCF is correct if it allows the recovery of the original input I from the output H. Evaluating S and S^{-1} cannot be done without having the secret key.

Definition 3. *An* SFTCF *S is secure if it assures shortcut freeness and is G-Detectable Time Consuming for any G with $\Omega_S(v) \gg \Omega_G(v)$.*

4.1.1 Instantiation We present our SFTCF S instantiated with the butterfly function from [20]. Let I and H be the input and output domain consisting of v block files. The butterfly construction is split in d rounds. For each round an atomic operation w takes as input pairs of blocks and outputs another pair, acting as input for the next same round. A PRP such as AES can be used for w. More formally $S : I^v \rightarrow H^v$. The input blocks at level 1 are denoted as $I_1[u], u \in [1, \ldots, v]$ and the final output blocks as $H_d[u], u \in [1, \ldots, v]$. Any intermediate level blocks $H_j[u], j \in [1 \ldots d]$ are computed from the output of the previous level blocks H_{j-i}. Thus, w is invoked $O(v/2 \log_2 v)$ times. An example is shown in figure 3. $v = 8 = 2^3$ and $d = 4 = \log_2 8$. Intuitively each round mixes each block with another one. At the final round the result of each block $H_4[u]$ is the result of mixing uth block with all the other $u - 1$ blocks. The final block $H_4[3]$ from round 4 it is a mix of block 3 and block 7 with w. Blocks $3, 7$ are the mix of blocks $1, 3$ and $5, 7$ accordingly from round 3. Blocks $1, 3, 5, 7$ are the output blocks from the input blocks $1, 2, 3, 4, 6, 7, 8$. Deleting only the block $H_4[3]$ will not aid the malicious cloud to evaluate S any time faster than $O(v)$ since $H_4[3]$ as we show depends on all blocks and S needs to be invoked from level 1. In the aforementioned example we assume the number of input blocks to be a power of 2. Otherwise more rounds than $\log_2 v$ are needed to present a complete mix of all blocks, or a bigger branching factor of w (e.g: mix& slice technique from [5]).

An alternative approach would be to apply a symmetric encryption algorithm multiple times on all the blocks of the input file F. However this would have a negative effect at the user side concerning the decoding time of each block. Namely, to achieve the same time delay as with the butterfly construction and assuming the running time $|w|$ of w equals the computation time of a single block encryption with the multiple encryption approach a user has to apply the inverse of encryption more times than with the butterfly approach. E.g: The time threshold is set to $6|w|$. User encrypts a file composed of 8 blocks 7 times, resulting in 56 encryptions of a single block. In contrast with the butterfly construction user needs to invoke w 12 times to guarantee the same threshold.

4.2 Merkle Trees

The use of Merkle Trees to authenticate streams of data has already been proposed by Merkle [16]. A binary tree whereby the leaf nodes correspond to data and intermediate nodes keep digest thereof reduce the authentication procedure to logarithmic costs on the height of the tree and the size of data subsequently. Let W be a collision resistant hash function: $\mathtt{W}\{0,1\}^* \rightarrow \{0,1\}^\lambda$, mapping strings of arbitrary length to λ-bit strings. Assume a vector

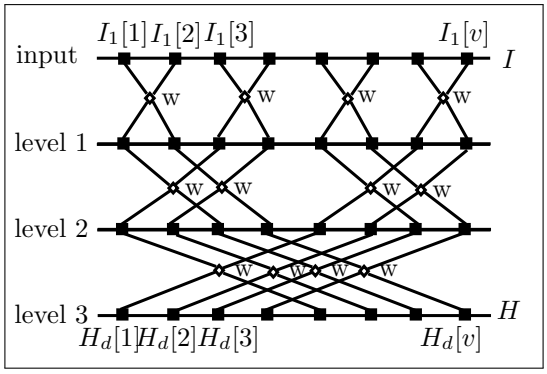


Fig. 3: A butterly based SFTCF.

of data elements $\mathbf{l} = \{l_1, l_2, l_3, \ldots, l_n\}$. The $\mathtt{MT}(\mathbf{l})$ algorithm computes (cf. figure 5) the Merkle tree for the data vector $\mathbf{l}$ and outputs its $\mathsf{root}_\mathbf{l}$ (cf. figure 4-left). All leaf nodes correspond to the hash of each element $\mathtt{W}(l)$ and parent nodes are computed as the hash of the concatenated children hashes. A prover who claims membership of data element l_x runs the $\mathtt{ProveMT}(x, \mathbf{l})$ algorithm and sends the authentication path $\mathbf{ap}_{l_x}$ to the verifier. A verifier can check the correctness of the authentication path with respect to the membership of the element l_x in $\mathbf{l}$ by recomputing the Merkle tree based on the authentication path $\mathbf{ap}_{l_x}$ running the `CheckPath` algorithm. Finally it checks if the computed root $\mathtt{W}' \stackrel{?}{=} \mathsf{root}_l$. For example in figure 4-right $\mathbf{ap}_{l_3} = \{h_4, h_9, h_{14}\}$.

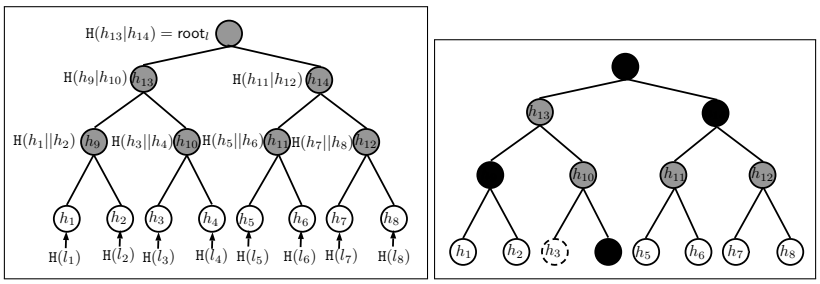


Fig. 4: Merkle Tree construction for the dataset $\mathbf{l} = l_1, l_2, l_3, l_4, l_5, l_6, l_7, l_8$. White nodes are leaf nodes. Grey nodes are intermediate-parent nodes.Black nodes consist the authentication path $\mathbf{ap}_{l_4} = \{h_4, h_9, h_{14}\}$ of the dashed line leaf node l_3. Note that there is no requirement the number of elements to be hashed to be a power of two. In that case the tree is not balanced.

4.2.1 Security The security guarantee for a Merkle tree hash algorithm demonstrates that it is impossible for an adversary to forge a Merkle Tree Hash $\mathsf{root}_l \leftarrow \mathtt{MT}(\mathbf{l})$ claiming valid membership proof for elements which do not belong in $\mathbf{l}$.

Definition 4. *If* $\mathtt{W}$ *is a collision resistant hash function then the* $\mathtt{MT}(\mathbf{l})$ *hash algorithm outputs a merkle hash tree for the dataset* $\mathbf{l}$ *which is also collision resistant, i.e: an adversary* $\mathcal{A}$ *can claim to authenticate an element* $x \ni \mathbf{l}$ *by generating the same* $\mathsf{root}_l \leftarrow \mathtt{MT}(\mathbf{l})$ *algorithm with negligible probability.*

5 ReDup Overview

We present an overview of ReDup. Recall the two challenges ReDup addresses: **CH1a)** resiliency against collusions between servers, **CH1b)** resiliency against collusions between a user and replica servers and **CH2)** scalable transparent deduplication for multiple replica servers.

CH1a: To avoid collusions between servers, we rely on encrypting the replicas with different keys. As such, when RS'_i does not store its replica copy H_i and acquires it from another server RS_i on the fly to correctly reply

$\mathsf{root}_l \leftarrow \mathtt{MT}(\mathbf{l})$

1 : $n = |\mathbf{l}|$
2 : **if** $n == 1$
3 : $\quad \mathsf{root}_l = \mathtt{W}(l_1)$
4 : $\quad$ **return** root_l
5 : **else**
6 : $\quad q = \left\lfloor \frac{n}{2} \right\rfloor$
7 : $\quad \mathtt{MT}(\mathbf{l}_0 \dots \mathbf{l}_q) || \mathtt{MT}(\mathbf{l}_{q+1} \dots \mathbf{l}_n)$

$\mathsf{apm}_x \leftarrow \mathtt{ProveMT}(x, \mathbf{l})$

1 : $n = |\mathbf{l}|$
2 : **if** $n == 1$
3 : $\quad$ **return** $\perp$
4 : **else**
5 : $\quad q = \left\lfloor \frac{n}{2} \right\rfloor$
6 : $\quad$ **if** $x < q$
7 : $\quad\quad$ **return** $\mathtt{ProveMT}(x, \mathbf{l}_0 \dots \mathbf{l}_q) || \mathtt{MT}(\mathbf{l}_{q+1} \dots \mathbf{l}_n)$
8 : $\quad$ **else**
9 : $\quad\quad$ **return** $\mathtt{ProveMT}(x - q, \mathbf{l}_q \dots \mathbf{l}_n) || \mathtt{MT}(\mathbf{l}_1 \dots \mathbf{l}_q)$

$\mathsf{root} \leftarrow \mathtt{ComputePath}(\mathbf{ap}, x, n, l_x)$

1 : / l_x:=the data element,x:=its ranking in the merkle tree
2 : / n:=the total number of leaves in the merkle tree
3 : **if** $n == 1$
4 : $\quad$ **return** $\mathtt{W}(l_x)$
5 : **else**
6 : $\quad q = \left\lfloor \frac{n}{2} \right\rfloor$
7 : $\quad$ **if** $x < q$
8 : $\quad\quad$ **return** $\mathtt{ComputePath}(\mathbf{ap}_1 \dots \mathbf{ap}_{|\mathbf{ap}|-1}, x, q, l_x)$
9 : $\quad$ **else**
10 : $\quad\quad$ **return** $\mathtt{ComputePath}(\mathbf{ap}_1 \dots \mathbf{ap}_{|\mathbf{ap}|-1}, x - q, n - q, l_x)$

$0, 1 \leftarrow \mathtt{CheckPath}(\mathbf{ap}, x, n, l, \mathsf{root})$

1 : / l_x:=the data element,x:=its ranking in the merkle tree
2 : / n:=the total number of leaves in the merkle tree
3 : $\mathsf{root}' = \mathtt{ComputePath}(\mathbf{ap}, x, n)$
4 : **return** $(\mathsf{root}' == \mathsf{root})$

Fig. 5: Merkle tree algorithms.

to the challenges of the RDIC protocol, it fails since each user U_j differentiates replica copies H_i with different keys.

CH1b: To mitigate collusions between a user and a replica server, U_j encrypts each replica with a SFTCF function S during the Replicate phase. U_j evaluates S function on input the file blocks $f_1, \dots, f_v$, keyed by a different key for each replica. Authentication tags are computed based on the output of S by each user. Finally, U_j uploads tags and replica copies to each replica server and whenever it wants to get assurances for the faithful storage of each replica file at each replica server RS_i, it runs a remote data integrity checking protocol RDIC with each RS_i. In contrast with previous RDIC protocols, the verification procedure during RDIC at ReDup, succeeds if and only if the time to verify the integrity of each replica copy is below a threshold value T, which is greater the time to evaluate $S(F)$. As such U_j can detect malicious behavior of a replica server RS_i who colludes with a user to answer the challenges of RDIC without storing the replica file.

CH2: To assure correctness of the deduplication level, the CSP during the AttestDedup phase creates a digest per file, per time epoch ep over the authenticated data structure of each file, which accumulates all users who have stored that file. For the authenticated data structure creation we employ a Merkle tree construction (cf Fig. 6). User ids are the leaves of the tree coupled with information about the identifier of the accumulated file and the epoch, which are accumulated to create the public root of the Merkle tree $h_0^{\mathsf{id_F}}$, acting as the digest. For membership queries (**dc1** cf. subsection 3.3.2) it suffices for U_j to obtain the sibling path, which corresponds to its leaf and to check the consistency of the Merkle tree: check equality of the constructed root with the one published by the CSP.

To attest the correct size of the deduplication level of a file (**dc2** cf. subsection 3.3.2), we follow the accumulation Merkle tree construction in [3, Section 3.2.1], dubbed as CARDIAC, which enables the CSP to prove an upper bound on the number of leaves of the Merkle tree. In CARDIAC there is an upper bound of users X, which are accumulated in the same Merkle tree $h_0^{\mathsf{id_F}}$ as for **dc1**, corresponding to users who stored file F. The tree is augmented with $2^{fh} - |X|$ zero leaves, which are leaves with the hash of value of 0, for a fixed height fh of the Merkle tree. A rational CSP shows less elements in X in order to increase its revenues. During the deduplication

level proof the CSP sends the right-most non-zero leaf and its sibling path. At the verification phase, the client checks the sibling path of this last node and also builds up the tree starting from the zero-leaves. If a server is cheating by excluding leaves of users, it should add more zero nodes, otherwise the client would detect it, when it builds up the tree from the zero leaves. However, adding more zero-leaves will destroy the structure of the tree and the verification during **dc1** will fail. We show first a naive adaptation of CARDIAC to our multi-replica model and then we present an improved solution.

Naive solution. A superficial adaptation of CARDIAC to a multi-replica scenario does not scale well with different replication level policies per user. Imagine 10 replica servers, one file and three users (e.g Fig. 6) with three different replication levels: $\mathsf{rl}_1 = 3, \mathsf{rl}_2 = 5, \mathsf{rl}_3 = 10$ for users U_1, U_2, and U_3 accordingly. Then the CSP following the naive CARDIAC approach has to maintain 10 different trees, one per each replica server RS_i. Whenever U_3 wants to check for **dc1** has to obtain Merkle sibling paths for 10 different trees, increasing the proof computation and verification time and the communication bandwidth. Alike with **dc1**, U_3 must reconstruct the 10 trees from the zero leaves for **dc2** and check membership in 10 Merkle trees for all the rightmost non-zero leafs. E.g in Figure 6 $fh = 2$ and U_2 to attest a correct deduplication level in the tree corresponding to the $5th$ replica gets as part of the proof for **dc1** the sibling path of node h_2 which is h_1, h_6. For **dc2** the CSP returns the claimed correct deduplication level number $\mathsf{dl}_{\mathsf{F}}^5 = 3$, the rightmost non-zero leaf = h_3 and its sibling path that allows U_2 to build up that tree starting from the remaining zero leafs = h_5. The client for **dc1** checks the correctness of the Merkle tree with the sibling path. For **dc2** U_2 verifies **1**) whether $\mathsf{dl}_{\mathsf{F}}^5 + |\text{zero leafs}| \stackrel{?}{=} 2^{fh}$ and **2**) whether the rightmost non-zero leaf h_3 with its sibling path h_5 correctly build the Merkle tree for RS_5.

Optimized solution. Recall that the CSP offers a finite number of different replication policies, which in a real world example could be $3, 5, 10$ ($\mathsf{RP} = 3, \mathsf{rp}_1 = 3, \mathsf{rp}_2 = 5, \mathsf{rp}_3 = 10$) different replica servers and users choose their replication level rl_j accordingly. Our solution reduces the number of Merkle trees per file from rl_j to a constant number 3, which is the number of different replication policies offered by the CSP. Our observation is that different replica server Merkle trees which are symmetric: the same number of users are accumulated in different trees, can be further aggregated. To accomplish the aggregation we annotate the leaves of the tree with the id of all symmetric RS_i (cf. Fig. 7). It is not difficult to see that all the different sets of symmetric trees equal the different replication level RP offered by the CSP. For example, for replication policies $\mathsf{rp}_1 = 3, \mathsf{rp}_2 = 5, \mathsf{rp}_3 = 10$ and three users with replication levels $\mathsf{rl}_1 = 3$, $\mathsf{rl}_2 = 5$ and $\mathsf{rl}_3 = 10$, then the three symmetric replica server groups are $\mathsf{agg}_{\mathsf{rp1}} = \mathsf{RS}_1, \mathsf{RS}_2, \mathsf{RS}_3$ for U_1, U_2 and U_3, $\mathsf{agg}_{\mathsf{rp2}} = \mathsf{RS}_4, \mathsf{RS}_5$, for U_2 and U_3 and $\mathsf{agg}_{\mathsf{rp3}} = \mathsf{RS}_6, \mathsf{RS}_7, \mathsf{RS}_8, \mathsf{RS}_9, \mathsf{RS}_{10}$ for U_3. Thus, following the example in Figure 6 now user U_3 has to perform 3 verifications for **dc1** and **dc2** for one file F stored in 10 replicas instead of 10 verifications. Consequently the CSP aggregates proofs for 10 Merkle trees in 3 trees as shown in Figure 7.

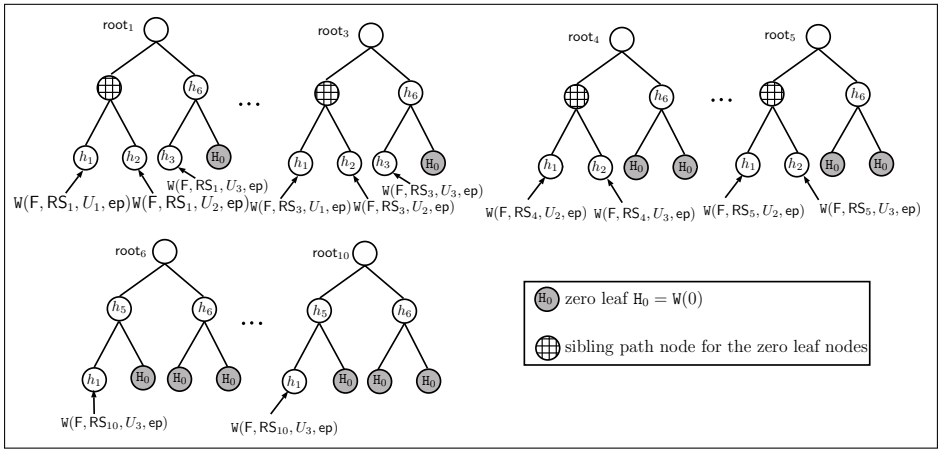


Fig. 6: Merkle Trees for Deduplication Level Proofs.

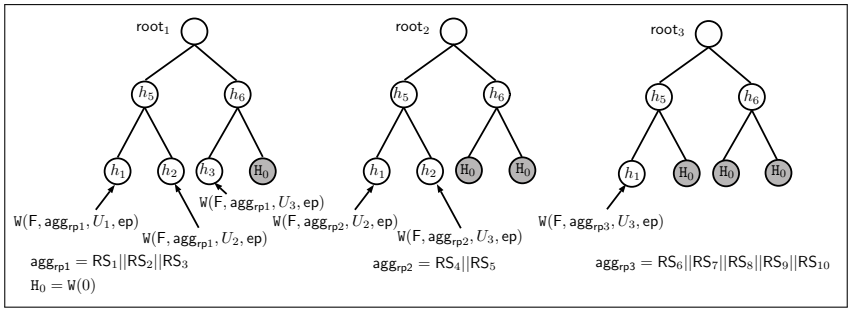


Fig. 7: Optimized Merkle Trees for Deduplication Level Proofs. The grey node is the zero node for that tree: its value is the hash digest of 0. During the AttestDedup.V phase the user to verify the correctness of the deduplication level proof checks the sibling path of the right-most non-zero node h_3 and then recomputes the tree from the remaining zero nodes and their sibling path: the crossed filled h_5 node.

6 Protocol

In this section we present our protocol in full details in figures 8, 9 and 10. We start with some common notation (cf. table 1).

RS_i	Replica server i, $1 \leq i \leq n$
U_j	User j, $1 \leq j \leq m$
f_u	File block u, $1 \leq u \leq v$
S	Shortcut Free and Time Consuming Function (SFTCF)
fk	Secret key for S
$\mathtt{W}$	A hash function $\mathtt{W} : \{0,1\}^* \rightarrow \{0,1\}^\lambda$
vt_j^i	Verification tags created by user U_j for her replica stored at RS_i
dl_F^i	Deduplication level of file F at RS_i
rl_j	Replication level for user U_j
fh	Fixed height of the Merkle tree
nz	Index of the rightmost non-zero leaf of a Merkle tree
$Z^{\mathsf{id_F}}$	A list of users owning file with id $\mathsf{id_F}$
$l_\mathcal{U}^{\mathsf{id_F}}$	A list of hashes of each element of the list $Z^{\mathsf{id_F}}$
$\mathsf{root}^{\mathsf{id_F}}$	The root of the Merkle tree for file $\mathsf{id_F}$
$h_0^{\mathsf{id_F}}$	The signed root of the Merkle tree for file $\mathsf{id_F}$
$\mathsf{apm}_j^{\mathsf{id_F}}$	The authentication path for node h_j of the Merkle tree with root $h_0^{\mathsf{id_F}}$
$h_{nz}^{\mathsf{id_F}}$	The rightmost non-zero leaf of the Merkle tree with root $h_0^{\mathsf{id_F}}$
$\mathsf{apc}_j^{\mathsf{id_F}}$	The authentication path of the rightmost non-zero leaf of the Merkle tree with root $h_0^{\mathsf{id_F}}$
$\pi_j^{\mathsf{id_F},ep}$	The proof of deduplication correctness for epoch ep of user U_j and file with id $\mathsf{id_F}$

Table 1: Notations

A file F consists of file blocks $f_1, f_2, ..., f_v$. Moreover, each replica server RS_i maintains of a log file $\mathtt{FL}$ to keep track which users have stored a specific file. $\mathtt{FL}$ is abstracted as a dictionary keyed by the id of a file $\mathsf{id_F}$. $\mathtt{FL}[\mathsf{id_F}].\mathsf{append}(U_j)$ denotes the insertion of user U_j under the key $\mathsf{id_F}$ and $\mathtt{FL}[\mathsf{id_F}]$ returns a tuple set with the id of all users who have stored file F with id $\mathsf{id_F}$. In turn, the CSP stores in a log dictionary $\mathtt{RL}$ the replication level choice of a user and the files each user has stored: $\mathtt{RL}[U_j] = (\mathsf{rl} : \mathsf{rl}_j, f : ())$. We write $\mathtt{RL}[U_j]$ to fetch the tuple $(\mathsf{rl} : \mathsf{rl}_j, f : ())$ and $\mathtt{RL}[U_j].\mathsf{rl}$ to retrieve the replication level of user U_j. $\mathtt{RL}[U_j].f$ denotes the identifiers of the files U_j has stored. The protocol overall can be described in four phases: Setup, Replicate, RDIC, AttestDedup. In the Setup phase users agree on replica generation keys and tag keys.

Setup: Each user U_j runs the Setup algorithm, which outputs tag keys k_j (line 1, Setup algorithm, fig. 8). Users agree on a key fk to compute the replica copies, different per replica, but identical inside a replica server in case of identical files. The derivation of fk is a black box in our protocol. Furthermore, each user chooses its replication level rl_j and forwards it to the CSP (line 2), which in turn stores it at the replication level dictionary $\mathtt{RL}[U_j].\mathsf{rl} = \mathsf{rl}_j$ (line 3).

Replicate: Each user calls this algorithm, which outputs the replica copies to be stored at each replica server RS_i and the verification tags thereof. More specifically, U_j for each replica server $\mathsf{RS}_i, 1 \leq i \leq \mathsf{rl}_j$ generates the replica copy H_i (Replicate, fig 8) with a Shortcut-free time consuming function as the butterfly construction

from the hourglass scheme [20]. A key fk_i is derived for each replica server RS_i with the use of a PRF, which is keyed by fk and takes as input the identifier of the replica server i. Finally the SFTCF S is applied to all blocks $f_1, f_2, \ldots, f_v$ and the replica copy H_i is obtained. For the generation of the verification tag files vt_j^i for each replica RS_i and user U_j we base our solution to CPOR for remote data integrity checking such as CPOR. In contrast with previous RDIC protocols though the TagFile does not take the file F but the output H_i of the S time consuming function.

TagFile algorithm first splits the replica copy H_i to v blocks $H_i^1, H_i^2, \ldots, H_i^v$. For each block H_i a random element $a \xleftarrow{\$} \mathbb{Z}_p$ is chosen uniformly at random. The final tag is coupled with an unpredictable function $PRF(i|u)$, encompassing information about the current block u and the replica server RS_i. Finally, the verification tag for user U_j for replica server RS_i is $\mathsf{vt}_j^i = \{a_j^u H_i^u + PRF_{\mathsf{k}_j}(i|u)\}_{u=1}^v$ (lines 2-4, TagFile algorithm). U_j runs the TagFile for all the replica servers of its choice rl_j.

RDIC: For the remote data checking protocol the users and the CSP invoke the Challenge, Prove and Verify algorithms (cf. 9) as in typical remote data integrity checking protocols. For our protocol instantiation we employ the CPOR technique. User issues challenges for each replica server RS_i and the latter computes a proof for each challenge. Finally the user verifies the correctness of the proof. The verification of RDIC protocol, in contrast with previous remote data integrity checking protocols, succeeds if and only if the time to verify the integrity of each replica copy is below a threshold value $T > \mathtt{Time}(S(F))$.

AttestDedup: The CSP computes the Merkle trees per epoch with the AttestDedup.P algorithm (lines 1-14, alg. AttestDedup.P, fig. 10). Each tree associates the users storing a file F at the leaves of the tree along with all the symmetric replica servers of each replication policy inside each leaf. More specifically for each different replication policy rp offered by the CSP (line 2, alg. AttestDedup.P) the CSP fetches the files that have been stored in the system (line 3) and retrieves the id of the users who have stored these files at any replica server (line 4). Each user will be a different leaf of the tree (line 5). In lines 6-9 the correct symmetric replica servers for rp are accumulated in the leaf of each user and finally the leaf $v_{\mathcal{U}}^{\mathsf{id_F}}$ is being hashed with a collision resistant hash function $\mathtt{W}$ to output the digest $l_{\mathcal{U}}^{\mathsf{id_F}}$. For the rest $2^{fh} - |l_{\mathcal{U}}^{\mathsf{id_F}}|$ nodes, zero leaves are computed as $\mathtt{W}(0)$ to fill in the tree of height fh. Once all the leaves of the tree have been computed, the CSP calls the $\mathtt{MT}$ algorithm, which computes the root of the Merkle tree $\mathsf{root}^{\mathsf{id_F}}$ (line 12) and finally signs it (line 13) with an unforgeable signature Sig to compute $\mathsf{Sig}(\mathsf{root}^{\mathsf{id_F}}) = h_0^{\mathsf{id_F}}$.

To compute the proof for a user U_j the CSP computes the sibling paths $\mathsf{apm}_j^{\mathsf{id_F}}$ for all the trees the user is included (line 4), using the common Merkle tree membership proof ($\mathtt{ProveMT}$ algorithm). That assures that the user has been inserted in the Merkle tree for that file F. To establish a correct deduplication level of the tree (total non-zero leaves), the CSP fetches the right-most non zero leaf node $h_{nz}^{\mathsf{id_F}}$ of each tree whereby U_j has been inserted, with the $\mathtt{FetchR}$ algorithm and computes its sibling path $\mathsf{apc}_j^{\mathsf{id_F}}$ as well (lines 5-6). Finally it forwards the proof $\pi_j^{\mathsf{id_F},ep} = (\mathsf{apm}_j^{\mathsf{id_F}}, h_{nz}^{\mathsf{id_F}}, \mathsf{apc}_j^{\mathsf{id_F}}, fh, |l_{\mathcal{U}}^{\mathsf{id_F}}|)$ to U_j.

U_j upon receipt of the proof $\pi_j^{\mathsf{id_F},ep}$ invokes AttestDedup.V to verify the proof. It first checks whether the claimed deduplication level is consistent with the zero leafs for a tree of height fh (line 4, alg AttestDedup.V). For the **dc1** part, which assures that its id is included in the tree(s) of the corresponding files, U_j calls the $\mathtt{CheckPath}$ (which also verifies the signed root of the Merkle tree) to verify the consistency of the returned sibling path $\mathsf{apm}_j^{\mathsf{id_F}}$ (line 5, alg AttestDedup.V). To verify **dc2**: the correctness of the deduplication level $|l_{\mathcal{U}}^{\mathsf{id_F}}|$, U_j first verifies the paths of all $h_{nz}^{\mathsf{id_F}}$ with the $\mathtt{CheckPath}$ Merkle tree algorithm (line 6, alg. AttestDedup.V). Afterwards the user checks if $(2^{fh} - 1) - |l_{\mathcal{U}}^{\mathsf{id_F}}|$ nodes computed as zero leaf nodes, along with their sibling nodes, compute correctly the Merkle tree with the $\mathtt{CheckZeroNodes}$ algorithm (line 6, alg AttestDedup.V). Finally if all the checks succeed (lines 4-6 alg. AttestDedup.V), U_j outputs 1 as a successful deduplication level verification, otherwise it outputs 0 (line 7, alg. AttestDedup.V).

7 Analysis

7.1 Security

ReDup scheme is secure as long as it guarantees *Collusion Resistant Replicas Integrity* and *Deduplication Correctness*. We split the *Collusion Resistant Replicas Integrity* proof in two parts. The first part focuses on the correctness property of the proof returned by each RS_i during the RDIC subprotocol. Namely, we prove that a rational adversary $\mathcal{A}$ cannot forge an incorrect proof to be accepted by an honest user U_j. At the second part we

– $\underline{\mathsf{Setup}(1^\lambda, n, \mathsf{rl}_j)}$: // Run by U_j.

1 : $(\mathsf{k}_j, \mathsf{fk}) \leftarrow \mathsf{KeyGen}(1^\lambda, n)$

2 : $U_j \xrightarrow{\mathsf{rl}_j} \mathsf{CSP}$

3 : $\mathsf{CSP} : \mathtt{RL}[U_j].\mathsf{rl} = (\mathsf{rl}_j)//\mathsf{CSP}$ stores the replication level in the log file rl.

– $\underline{\mathsf{Replicate}(\mathsf{F}, \mathsf{fk}, \mathsf{k}_j, \mathsf{rl}_j)}$: // Run by U_j.

1 : **for** $(i = 1, i \leq \mathsf{rl}_j, i++)$ **do**

2 : $H_i \leftarrow \mathsf{GenReplica}(i, \mathsf{F}, \mathsf{fk})$:

3 : $\mathsf{vt}_j^i \leftarrow \mathsf{TagFile}(H_i, \mathsf{k}_j)$

4 : $U_j \xrightarrow{H_i, \mathsf{vt}_j^i} \mathsf{RS}_i$

5 : $\mathsf{CSP} : \mathtt{FL}_i[\mathsf{id}_\mathsf{F}].\mathsf{append}(U_j)$

– $\underline{\mathsf{RDIC}(\mathsf{F}, < U_j : Q >, < \mathsf{RS}_i : \sigma_i >)}$: // Run by U_j and CSP.

1 : **for** $(i = 1, i \leq \mathsf{rl}_j, i++)$ **do**

2 : $Q \leftarrow \mathsf{Challenge}(l, n)$

3 : $\sigma_i \leftarrow \mathsf{Prove}(Q, H_i, \mathsf{vt}_i)$

4 : $\tau_i \leftarrow (\mathtt{Time}(\mathsf{Verify}(\sigma_i) \leq T)) : 1?0$

5 : **if** $\bigwedge \tau_i \stackrel{?}{=} 1$ **return** 1 **else return** 0

– $\underline{\mathsf{AttestDedup}(ep, U_j, \mathsf{F})}$:// Run by U_j and CSP.

1 : **for** $(ep \in \mathcal{T} \wedge \mathsf{F} \in \mathtt{FL} \wedge U_j \in \mathcal{U})$ **do**

2 : $\pi_j^{\mathsf{id}_\mathsf{F}, ep} \leftarrow \mathsf{AttestDedup.P}(ep, U_j, \mathsf{F})$

3 : $0, 1 \leftarrow \mathsf{AttestDedup.V}(\pi_j^{\mathsf{id}_\mathsf{F}, ep})$

Fig. 8: The ReDup system.

build an extractor, who communicates with a prover and is able to extract the file from the transcripts of interaction with $\mathcal{A}$. That assures that as long as the each replica server RS_i answers correctly the challenges during the challenge phase of the RDIC then it faithfully stores the replica copy.

7.1.1 Collusion Resistant Replicas Integrity For the first part we show that the only way the Verify algorithm accepts the σ_i is the CSP to compute the correct response values $\{\mu_{d_c}\}_{c=1}^{l}$. Namely the proof relies on the unforgeability of the tags and the semantic security of S: an adversary $\mathcal{A}$ impersonating the cloud service provider CSP cannot make a honest U_j during the Verify procedure to accept with the submission of wrong responses.

Theorem 1. *Let PRF be a pseudorandom function, $\mathcal{A}$ a probabilistic polynomial time adversary and S a* SFTCF *with atomic building block w which is semantically secure. Then, there is negligible probability for the* Verify *algorithm to accept the $\{\mu'_{d_c}\}_{c=1}^{l} \neq \{\mu_{d_c}\}_{c=1}^{l}$, during its interaction through a challenger $\mathcal{C}$ with $\mathcal{A}$, during the* $\mathsf{CR}^2\mathsf{P}$ *game.*

For the proof of the theorem we employ the game hopping technique. Starting from the original game we show 1) how subsequent games are constructed and 2) the distinguishing probability of an adversary $\mathcal{A}$ to detect the changes in the game.

Proof. We show the sequence of games that will demonstrate the proof of Theorem 1.

– Game 0 is the original $\mathsf{CR}^2\mathsf{P}$ game. The adversary $\mathcal{A}$ communicates with the oracles: $\mathcal{O}^{\mathsf{Setup}}, \mathcal{O}^{\mathsf{GenReplica}}, \mathcal{O}^{\mathsf{TagFile}}$ to create its environment. Based on that environment the challenger issues a challenge and forwards it to $\mathcal{A}$ with the $\mathcal{O}^{\mathsf{Challenge}}$ oracle.

– $(a_j, \mathsf{k}_j, \mathsf{fk}) \leftarrow \mathsf{KeyGen}(1^\lambda, n)$:

– $H_i \leftarrow \mathsf{GenReplica}(i, \mathsf{E}, \mathsf{fk})$:

1 : **parse** F *as* $f_1, f_2, \ldots, f_v$
2 : $\mathsf{fk}_i = \mathsf{PRF}_{\mathsf{fk}}(i)$ //Derive the key for replica RS_i.
3 : $H_i = S_{\mathsf{fk}_i}(f_1, f_2, \ldots, f_v)$ //Run the SFTCF S on E.
4 : **return** H_i

– $\mathsf{vt}_j^i \leftarrow \mathsf{TagFile}(H_i, a_j, \mathsf{k}_j)$:

1 : **parse** H_i *as* $H_i^1, H_i^2, \ldots, H_i^v$
2 : **for** $(u = 1, u \leq v, u++)$ **do**
3 : $\mathsf{vt}_j^{u,i} = a_j H_i^v + \mathsf{PRF}_{\mathsf{k}_j}(i|u)$ //Compute the verification tag
4 : **return** $\mathsf{vt}_j^i = \{\mathsf{vt}_j^{u,i}\}_{u=1}^v$

– $Q = \{d_c, b_c\}_{c=1}^l \leftarrow \mathsf{Challenge}(l, n)$:

1 : **for** $(c = 1, c \leq l, c++)$ **do**
2 : $d_c \leftarrow [1...v]$
3 : $b_c \xleftarrow{\$} \mathbb{Z}_p$
4 : **return** $Q = \{d_c, b_c\}_{c=1}^l$

– $(\{\mu_{d_c}\}_{c=1}^l, \sigma_i) \leftarrow \mathsf{Prove}(Q, H_i, \mathsf{vt}_i)$:

1 : **parse** H_i *as* $H_i^1, H_i^2, H_i^3, \ldots, H_i^v$
2 : **parse** Q *as* $\{d_c, b_c\}_{c=1}^l$
3 : **for** $(d_c \in Q)$ **do**
4 : $\mu_{d_c} = \Sigma b_{d_c} \cdot H_i^c$
5 : $\sigma_i = \Sigma b_{d_c} \cdot \mathsf{vt}_i^c$
6 : **return** $(\{\mu_{d_c}\}_{c=1}^l, \sigma_i)$

– $\mathsf{Verify}(\{a_{d_c}\}_{c=1}^l, \{b_{d_c}\}_{c=1}^l, \{\mu_{d_c}\}_{c=1}^l, \sigma_i)$:

1 : **if** $\sigma_i \stackrel{?}{=} \Sigma_{c=1}^l b_{d_c} \cdot \mathsf{PRF}_{\mathsf{k}_j}(i|c) + \Sigma_{c=1}^l \mu_{d_c} \cdot a_j$
2 : **return** 1 //
3 : **else**
4 : **return** $\perp$

Fig. 9: Algoriths for Replicate and RDIC . Run by U: Client , CSP:Cloud Storage Provider ,

– In Game 1 the challenger, whenever A queries the $\mathcal{O}^{\mathsf{GenReplica}}$ oracle with input $\mathsf{F}, \mathsf{uid}, i$ then $\mathcal{C}$ instead of using the PRF it randomly chooses a random elements $\{r_v\}_{v=1}^u \xleftarrow{\$} \mathbb{Z}_p$ and stores in a lookup table $T[\mathsf{F}, \mathsf{uid}, i] = \{r_i^v\}_{v=1}^u$. If $\mathcal{A}$ is able to distinguish between Game 0 and Game 1 then it halts the game. The probability to halt the game is treated as a bad event and it is the probability of distinguishing pseudorandom output of a PRF from a real random element. Assuming PRF is pseudorandom then the distinguishing probability is negligible.
– In Game 2 $\mathcal{C}$ does not compute the correct replica H_i from the $\mathcal{O}^{\mathsf{GenReplica}}$ oracle. Instead it substitues the output of the SFTCF S with random outputs of size v $R_i = R_1 \ldots R_v$. and forwards to $\mathcal{A}$ R_i. Assuming a semantically secure S with indistinguishable encryptions $\mathcal{A}$ cannot distinguish Game 2 from Game 1.

$\pi_j^{ep} \leftarrow \mathsf{AttestDedup.P}(ep, U_j, fh)$: // Run by the CSP

1 : $pp = 1$
2 : **foreach** rp **do** // For replication levels 3,5,10, at every loop $\mathsf{rp} = 3, 5, 10$
3 : **for** $(\mathsf{id_F} \in \mathtt{RL}[U_j].f)$ **do** // For every file fetch the id thereof from RL
4 : **for** $\mathcal{U} \in \mathtt{FL}[\mathsf{id_F}]$ **do** // Retrieve the set of users who stored the file
5 : $Z^{\mathsf{id_F}} = \mathcal{U}||ep$
6 : **for** $(j = pp; j \leq \mathsf{rp})$ **do**
// Aggregate in one tree all the RS of that replication level group
7 : **if** $(\mathtt{RL}[\mathcal{U}].\mathsf{rl} > j)$ **continue**
8 : $Z^{\mathsf{id_F}} += ||\mathsf{RS}_j$ // Aggregate all the replica servers.
9 : $l_{\mathcal{U}}^{\mathsf{id_F}} += \mathtt{W}(Z^{\mathsf{id_F}})$ // Using a CRHF W hash the leaf value.
10 : **for** $(z = 1; z \leq (2^{fh} - 1) - |l_{\mathcal{U}}^{\mathsf{id_F}}|; z++)$
11 : $l_{\mathcal{U}}^{\mathsf{id_F}} += \mathtt{W}(0)$ // Pad with 0 leaf nodes
12 : $\mathsf{root}^{\mathsf{id_F}} \leftarrow \mathtt{MT}(l_{\mathcal{U}}^{\mathsf{id_F}})$ // Build the merkle tree for $l_{\mathcal{U}}^{\mathsf{id_F},\mathsf{rl}}$
13 : $h_0^{\mathsf{id_F}} = \mathsf{Sig}(\mathsf{root}^{\mathsf{id_F}})$ // Sign the root
14 : $pp = \mathsf{rp}$

1 : **foreach** rp **do** // For replication levels 3,5,10, at every loop rp=3,5,10
2 : **if** $(\mathtt{RL}[U_j].\mathsf{rl} < \mathsf{rp})$ **continue**
3 : **for** $(\mathsf{id_F} \in \mathtt{RL}[U_j].f)$ **do** // For every file fetch the id thereof from RL
4 : $\mathsf{apm}_j^{\mathsf{id_F}} \leftarrow \mathtt{ProveMT}(h_j, l_{\mathcal{U}}^{\mathsf{id_F}})$ // Compute the sibling path for U_j's leaf
5 : $h_{nz}^{\mathsf{id_F}} \leftarrow \mathtt{FetchR}(h_0^{\mathsf{id_F}})$ // Fetch the rightmost non-zero leaf
6 : $\mathsf{apc}_j^{\mathsf{id_F}} \leftarrow \mathtt{ProveMT}(h_{nz}^{\mathsf{id_F}}, l_{\mathcal{U}}^{\mathsf{id_F}})$ // Compute its sibling path
7 : $\pi_j^{\mathsf{id_F},ep} = (\mathsf{apm}_j^{\mathsf{id_F}}, h_{nz}^{\mathsf{id_F}}, \mathsf{apc}_j^{\mathsf{id_F}}, fh, \mathsf{dl}_{\mathsf{F}}^{\mathsf{rp}})$
8 : **return** $\pi_j^{\mathsf{id_F},ep}, \forall \mathsf{id_F} \in \mathtt{RL}[U_j].f$

$0, 1 \leftarrow \mathsf{AttestDedup.V}(\pi_j^{\mathsf{id_F},ep})$: // Run by U_j

1 : **foreach** rp **do** // For replication levels 3,5,10, at every loop $\mathsf{rp} = 3, 5, 10$
2 : **if** $(\mathtt{RL}[U_j].\mathsf{rl} < \mathsf{rp})$ **continue**
3 : **for** $(\mathsf{id_F} \in \mathtt{RL}[U_j].f)$ **do** // For every file fetch the id thereof from RL
4 : $\mathsf{dl}_{\mathsf{F}}^{\mathsf{rp}} + |\text{zero leafs}| \stackrel{?}{=} 2^{fh}$
5 : $1, 0 \leftarrow \mathtt{CheckPath}(h_j, \mathsf{apm}_j^{\mathsf{id_F}}, h_0^{\mathsf{id_F}})$ //Check if U_j's file F was included
6 : $1, 0 \leftarrow \mathtt{CheckPath}(h_{nz}^{\mathsf{id_F}}, \mathsf{apc}_j^{\mathsf{id_F}}, h_0^{\mathsf{id_F}})$ //Test inclusion of rightmost non-zero leaf
7 : $1, 0 \leftarrow \mathtt{CheckZeroNodes}(h_0^{\mathsf{id_F}})$ // Build up the tree starting from the zero nodes
8 : **return** $(all\ checks == 1)?1 : 0$

Fig. 10: Algorithms for AttestDedup: The CSP builds Merkle Trees and computes the deduplication correctness proof . The Client verifies the proof.

– In Game 3 the challenger does some bookkeeping. Whenever $\mathcal{A}$ queries the $\mathcal{O}^{\mathsf{GenReplica}}$ oracle keeps track of the result replica copies $H_1, \ldots, H_{\mathsf{rl}_j}$. If $\mathcal{A}$ does not query the $\mathcal{O}^{\mathsf{TagFile}}$ for replica copies $H_1, \ldots, H_{\mathsf{rl}_j}$, then $\mathcal{C}$ calls it and stores the verification tags vt_j^i for replica server RS_i. Then when the challenger $\mathcal{C}$ challenges $\mathcal{A}$, $\mathcal{C}$ gets the response and aborts if it is not the supposed one: $\{\mu_{d_c}\}_{c=1}^{l}, \sigma_i$. Let the expected responses be $\{\mu_{d_c}\}_{c=1}^{l}, \sigma_i$. Recall that $\mu_{d_c} = \Sigma b_{d_c} \cdot \mathsf{vt}_i^c$ and $\sigma_i = \Sigma b_{d_c} \cdot H_i^c$. A cheating $\mathcal{A}$ who succeeds without sending the expected $\{\mu_{d_c}\}_{c=1}^{l}, \sigma_i$ sends to the challenger: ${\sigma_i}' = \Sigma_{c=1}^{l} b_{d_c} \cdot r_i^v + \Sigma_{c=1}^{l} \mu'_{d_c} \cdot a_{d_c}$, where $\mu'_{d_c} \neq \mu_{d_c}$ for at least one μ_{d_c}. We define $\Delta_\sigma = \sigma_i - {\sigma_i}'$ and $\Delta_\mu = \mu - \mu'$. Therefore $\Delta_\sigma = \Sigma_{c=1}^{l} \Delta_{\mu_{d_c}} \cdot a_j$. The bad event occurs when $\mathcal{A}$ passes forged responses, yet $\mathcal{O}^{\mathsf{Verify}}$ verifies correctly and $\Delta_\sigma \neq 0$. For fixed Δ_σ and Δ_μ $\mathcal{A}$ has to guess the correct values of a_j for a suer U_j. The only view $\mathcal{A}$ has for a_j is from the $\mathcal{O}^{\mathsf{TagFile}}$ oracle which responds with $\mathsf{vt}_j^{u,i} = a_j H_i^u + r_i^u$, where $\{r_u\}_{u=1}^{v}$ are uniformly at random elements in $\mathbb{Z}_p$.

Thus the correct probability of guessing a_j is $1/p$ and after q queries is q/p. Thus with probability q/p $\mathcal{A}$ can compute convincing responses for a replica copy H_i corresponding to replica copy H_i.

For the second part of the proof we demonstrate the feasibility of building an extractor $\mathcal{E}$ which extracts the file F from its interaction with each replica server RS_i. In contrast with previous definitions of soundness in proofs of retrievability [7, 19] we enhance our model with a time parameter in which the colluding $\mathcal{A}$ with a user U'_j has to issue the proof for the faithful possession of replica copy.

Theorem 2. ReDup *system assures Collusion Resistant Replicas Integrity assuming a Shortcut Free and G-Detectable Time Consuming* SFTCF *S.*

Proof. (Sketch) A file F consists of v blocks: $f_1, f_2, f_3, \ldots, f_v$ and is uploaded in t replica servers. $\mathcal{A}_i$ corrupts RS_i and tries to cheat. $\mathcal{A}$ the CSP controls all $\mathcal{A}_i$. Let δ_i^u the probability $\mathcal{A}_i$ corrupts block f_u at replica server RS_i. We assume for the sake of lucidness that all δ_i^u are equal.[1] Let δ_i^u follow a Bernoulli distribution with success probability δ_i^u denoting the probability $\mathcal{A}$ corrupts block f_u at replica server RS_i and failure probability $1 - \delta_i^u$. Let U_{uid} be the user who challenges the CSP. Then all the $v\mathsf{rl}_{\mathsf{uid}}$ blocks have corruption probability δ_i^u for $u \in [1 \ldots v\mathsf{rl}_{\mathsf{uid}}]$. The success probability $\mathbf{Succ}_{\mathcal{A}}^{\mathsf{CR^2P}}$ for $\mathcal{A}$ to pass a challenge of size l depends on the failure probability $1 - \delta_i^u$ and the success probability δ_i^u to corrupt f_u by outputting the correct challenge on time less than T. We assume S is a secure SFTCF. The probability to correctly guess the challenged blocks equals the probability to randomly guess the output of S for each block of the challenge of size l, and is equal to $\mathbf{Succ}_{\mathcal{A}}^{\mathsf{CR^2P}} = \prod_{u=1}^{l} (1 - \delta_i^u + \frac{\delta_i^u \epsilon}{2^v})$, where ϵ is a negligible probability that corresponds to the event of evaluating S in time less than T. From that we conclude that $\mathbf{Succ}_{\mathcal{A}}^{\mathsf{CR^2P}} \leq \mathsf{negl}(\lambda)$. Thus if the corruption probability δ_i^u for a block f_u is ≈ 1 then the first summand annihilates and the second summand $\frac{\delta_i^u \epsilon}{2^v}$ is negligible small. When $\delta_i^u \approx 0$ then $\mathcal{A}$ does not follow an adversarial behavior and faithfully stores all blocks.

We analyze the success probabilities of the extractor $\mathcal{E}$ who communicates with the adversary $\mathcal{A}$ and aims to extract the file F from the transcripts of interaction with $\mathcal{A}$. $\mathcal{E}$ has access to the challenges given to $\mathcal{A}$ and tries to reconstruct the original file F. The extractor $\mathcal{E}$ simulates the $\mathcal{O}^{\mathsf{TagFile}}$ oracle. When $\mathcal{A}$ queries the $\mathcal{O}^{\mathsf{TagFile}}$ oracle with input (H_i, uid), $\mathcal{E}$ first checks if $\mathsf{uid} \in \mathcal{U}'$ and $H_i \in H$. If both hold then it computes the tags $\mathsf{vt}_{\mathsf{uid}}^i$ and forwards them to $\mathcal{A}$. We assume $\mathcal{A}$ stores only $s < v\mathsf{rl}_{\mathsf{uid}}$ blocks. By storing we mean both the blocks and the verifications tags. Thus, during the challenge, $\mathcal{A}$ has to correctly guess the blocks and tags of the challenge. Let some $s' < l$, s blocks of the total l-block challenge be stored by $\mathcal{A}$. We denote by E_1 the event $\mathcal{A}$ correctly guesses the remaining $l - s'$ challenged blocks (which are not stored), E_2 the event $\mathcal{A}$ computes the responses for that challenge correctly and E_3 the event the $\mathcal{O}^{\mathsf{TagFile}}$ oracle outputs a special malicious output h^*, from which $\mathcal{A}$ can compute the remaining $l - s'$ blocks and the responses on the fly. Accordingly, the probabilities for E_1, E_2, E_3 are p_1, p_2, p_3, respectively.

Clearly, $p_1 = p_2 = \frac{2^{l-s'}}{2^v}$, $p_3 = \frac{1}{2^q}$, where q is the digest size of the $\mathcal{O}^{\mathsf{TagFile}}$ response. As such, $\mathbf{Succ}_{\mathcal{A}}^{\mathsf{Extract}} = 1 - (p_1 p_2 + (1 - p_1 p_2) p_3) = 1 - p_3 + p_1 p_2 (p_3 - 1) = 1 - \frac{1}{2^q} + \frac{2^{2(l-s')}}{2^{2v}} (\frac{1}{2^q} - 1)$, meaning that $\mathbf{Succ}_{\mathcal{A}}^{\mathsf{Extract}} > \mathsf{negl}(\lambda)$. As such, $\Pr[\mathbf{Succ}_{\mathcal{A}}^{\mathsf{Extract}} \leq \mathsf{negl}(\lambda) \wedge \mathbf{Succ}_{\mathcal{A}}^{\mathsf{CR^2P}} > \mathsf{negl}(\lambda)] \leq \mathsf{negl}(\lambda)$.

[1] The CSP may vary that probability according to frequency of challenges per RS_i–i.e: RS_a is challenged more often than RS_b, thus $\delta_a \leq \delta_b$.

7.2 Deduplication Correctness

Theorem 3. ReDup *system guarantees Deduplication Correctness against a rational adversary $\mathcal{A}$ who controls all the replica servers RS_i assuming 1) a collision resistant hash function* W *during the construction of the Merkle trees at the* AttestDedup *phase and 2) honest users.*

Proof. (Sketch) A rational adversary $\mathcal{A}$ will only act maliciously when it has economic incentives from this behavior. Thus, a detectable malicious behavior, which in turn degrades its reputation is not part of its strategy. During the AttestDedup interaction with each user, the CSP provides proofs of memberships and proofs for the correct deduplication $\pi_j^{\mathsf{id_F},ep}$ of a file with identifier $\mathsf{id_F}$ for a time epoch ep, through Merkle hash trees. An incorrect $\mathsf{dl}_{\mathsf{F}}^{i'}$ can be either greater or smaller the correct one $\mathsf{dl}_{\mathsf{F}}^{i}$.

- **Case 1**: $\mathsf{dl}_{\mathsf{F}}^{i'} < \mathsf{dl}_{\mathsf{F}}^{i}$. As users act honestly they will verify the membership test of their leaf node at each symmetric Merkle tree at each time epoch ep. Assuming a rationale $\mathcal{A}$, and a collision resistant hash function W, used for the construction of all the symmetric trees, then an honest user U_j during the verification of the correctness of $\mathsf{dl}_{\mathsf{F}}^{i'} < \mathsf{dl}_{\mathsf{F}}^{i}$ will accept the proof $\hat{\pi_j}^{\mathsf{id_F},ep}$ with negligible probability $\mathsf{neg}(\lambda) \leq 2^{-\lambda}$, where 2λ is the image length of the collision resistant hash function W. The collision resistance property of W prevents $\mathcal{A}$ of computing a set of leaves $l_{\mathcal{U}'}^{\mathsf{id_F},\mathsf{rl}}$ different than the correct set of leaves $l_{\mathcal{U}}^{\mathsf{id_F},\mathsf{rl}}$ with the same root digest $h_0^{\mathsf{id_F},\mathsf{rl}}$. If that is doable then we show a reduction, which breaks the collision resistance property of W. The reduction is omitted in the full version.
- **Case 2**: $\mathsf{dl}_{\mathsf{F}}^{i'} > \mathsf{dl}_{\mathsf{F}}^{i}$. This event does not occur, as it will further reduce the storage allocation costs per user since by adding more leaves to a tree and reducing the number of the 0 leaf nodes to demonstrate that $\mathsf{dl}_{\mathsf{F}}^{i'} > \mathsf{dl}_{\mathsf{F}}^{i}$, the cost of a deduplicated file will be shared across more users than actually store the file.

This concludes the proof and ReDup system guarantees Deduplication Correctness with non negligible probability $1 - 2^{-\lambda}$.

7.3 Efficiency

We perform a theoretical evaluation of our design. More specifically we analyze the cost at user side for replica copies and challenges generation, the verification of the deduplication cardinality proof and the response verification overhead from the challenge during the RDIC subprotocol. Accordingly we analyze the costs at the CSP side for the responses of the challenge per replica server and the deduplication cardinality proof of correctness. All algebraic operations take place in $\mathbb{Z}_p$.

User overhead. U_j at the Replicate phase invokes rl_j times a PRF function to derive a secret key for each replica RS_i and then calls the SFTCF function S, rl_j times to generate the replica copies $H_i, i \in [1, \ldots, \mathsf{rl}_j]$ within the GenReplica algorithm. To tag a file F at each replica server $\mathsf{RS}_i, i \in [1, \ldots, \mathsf{rl}_j]$ user U_J computes n multiplications, n additions and n PRF evaluations. To compute a challenge U_j randomly choses l indexes $\{d_c\}_{c=1}^{l}$ and l random elements $\{b_c\}_{c=1}^{l} \xleftarrow{\$} \mathbb{Z}_p$. To verify the response $(\{\mu_{d_c}\}_{c=1}^{l}, \sigma_i)$ from a replica server, U_j validate the equation $\sigma_i \stackrel{?}{=} \Sigma_{c=1}^{l} b_{d_c} \cdot \mathsf{PRF}_{\mathsf{k}_j}(i|c) + \Sigma_{c=1}^{l} \mu_{d_c} \cdot a_j$, which results in $2l$ multiplications and $2l+1$ additions. For the verification of the deduplication cardinality proof $\pi_j^{\mathsf{id_F},ep}$ per epoch, U_j first checks membership of its node to each aggregated Merkle tree with the CheckPath algorithm, membership of the last non-zero node and re-evaluates each aggregated Merkle tree according to the starting position of the first zero leaf. Thus the total cost per epoch per aggregated tree is $3 \log zl + (zl - \mathsf{dl}_{\mathsf{F}}^{\mathsf{rp}}) \cdot \mathtt{W} = O(\log zl + zl)$, where W is the cost for the hash function evaluation.

Cloud overhead. For the computation of the response during the RDIC subprotocol each RS_i commits to $2l$ multiplication and $2l$ additions to compute $\mu_{d_c} = \Sigma b_{d_c} \cdot H_i^c$ and $\sigma_i = \Sigma b_{d_c} \cdot \mathsf{vt}_i^c$ at the Prove algorithm. For the AttestDedup subprotocol the CSP computes rp symmetric Merkle trees per file per time epoch ep. Thus the computational cost for a single file F per time epoch is $\mathsf{rp} \cdot \log zl$ for Merkle trees with zl leaves. To compute the deduplication cardinality proof $\pi_j^{\mathsf{id_F},ep}$ per epoch for a user U_j with replication level rl_j assuming $\mathsf{rl}_j = \max(\mathsf{rp} \in \mathsf{RP})$, then CSP computes the sibling path of two elements: the user node at the tree with root $h_0^{\mathsf{id_F},\mathsf{rp}}$ and the sibling path for the rightmost non-zero leaf $h_{nz}^{\mathsf{id_F}}$. It also computes the missing parent nodes for the re-evaluation of the tree from the zero nodes. It is highly likely that these nodes will overlap with the previous sibling paths. Thus the overall size of the proof computed by the server per epoch for a file F is $3 \cdot \log zl = O(\log zl)$.

Bibliography

[1] F. Armknecht, L. Barman, J. Bohli, and G. O. Karame. Mirror: Enabling proofs of data replication and retrievability in the cloud. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, pages 1051–1068, 2016.

[2] F. Armknecht, J.-M. Bohli, D. Froelicher, and G. O. Karame. Sport: Sharing proofs of retrievability across tenants. Cryptology ePrint Archive, Report 2016/724, 2016. http://eprint.iacr.org/2016/724.

[3] F. Armknecht, J.-M. Bohli, G. O. Karame, and F. Youssef. Transparent data deduplication in the cloud. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 886–900, New York, NY, USA, 2015. ACM.

[4] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song. Provable data possession at untrusted stores. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 598–609. ACM, 2007.

[5] E. Bacis, S. D. C. di Vimercati, S. Foresti, S. Paraboschi, M. Rosa, and P. Samarati. Mix&slice: Efficient access revocation in the cloud. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 217–228, 2016.

[6] D. Boneh, B. Lynn, and H. Shacham. *Short Signatures from the Weil Pairing*, pages 514–532. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[7] K. D. Bowers, A. Juels, and A. Oprea. Proofs of retrievability: Theory and implementation. In *Proceedings of the 2009 ACM Workshop on Cloud Computing Security*, CCSW '09, pages 43–54, New York, NY, USA, 2009. ACM.

[8] B. Chen and R. Curtmola. Towards self-repairing replication-based storage systems using untrusted clouds. In *Proceedings of the Third ACM Conference on Data and Application Security and Privacy*, CODASPY '13, pages 377–388, New York, NY, USA, 2013. ACM.

[9] R. Curtmola, O. Khan, R. Burns, and G. Ateniese. Mr-pdp: Multiple-replica provable data possession. In *Proceedings of the 2008 The 28th International Conference on Distributed Computing Systems*, ICDCS '08, pages 411–420, Washington, DC, USA, 2008. IEEE Computer Society.

[10] http://www.computerworld.com/. Oops: Google "loses" your cloud data, 2015. https://goo.gl/zXRAdR.

[11] http://www.datacenterknowledge.com/. Amazon data center loses power during storm, 2012. https://goo.gl/anNoI.

[12] http://www.infoworld.com/. The dirty dozen: 12 cloud security threats, 2016. ttps://goo.gl/i6tAsF.

[13] A. Juels and B. S. Kaliski. PORs: Proofs of retrievability for large files. In *Proc. of ACM Conference on Computer and Communications Security (CCS '07)*, 2007.

[14] S. Keelveedhi, M. Bellare, and T. Ristenpart. Dupless: server-aided encryption for deduplicated storage. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 179–194, 2013.

[15] J. Liu, N. Asokan, and B. Pinkas. Secure deduplication of encrypted data without additional independent servers. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 874–885. ACM, 2015.

[16] R. C. Merkle. A certified digital signature. In *Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings*, pages 218–238, 1989.

[17] D. T. Meyer and W. J. Bolosky. A study of practical deduplication. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, FAST'11, pages 1–1, Berkeley, CA, USA, 2011. USENIX Association.

[18] D. T. Meyer and W. J. Bolosky. A study of practical deduplication. *Trans. Storage*, 7(4):14:1–14:20, Feb. 2012.

[19] H. Shacham and B. Waters. *Compact Proofs of Retrievability*, pages 90–107. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[20] M. van Dijk, A. Juels, A. Oprea, R. L. Rivest, E. Stefanov, and N. Triandopoulos. Hourglass schemes: How to prove that cloud files are encrypted. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 265–280, New York, NY, USA, 2012. ACM.

[21] D. Vasilopoulos, M. Önen, K. Elkhiyaoui, and R. Molva. Message-locked proofs of retrievability with secure deduplication. In *Proceedings of the 2016 ACM on Cloud Computing Security Workshop*, CCSW '16, pages 73–83, New York, NY, USA, 2016. ACM.