Transparent Data Deduplication in the Cloud

Frederik Armknecht
University of Mannheim
68131 Mannheim, Germany
armknecht@uni-mannheim.de

Jens-Matthias Bohli
NEC Laboratories Europe
69115 Heidelberg, Germany
Jens-Matthias.Bohli@neclab.eu

Ghassan O. Karame
NEC Laboratories Europe
69115 Heidelberg, Germany
ghassan.karame@neclab.eu

Franck Youssef
NEC Laboratories Europe
69115 Heidelberg, Germany
franck.youssef@neclab.eu

ABSTRACT

Cloud storage providers such as Dropbox and Google drive heavily rely on data deduplication to save storage costs by only storing one copy of each uploaded file. Although recent studies report that whole file deduplication can achieve up to 50% storage reduction, users do not directly benefit from these savings—as there is no transparent relation between effective storage costs and the prices offered to the users.

In this paper, we propose a novel storage solution, ClearBox, which allows a storage service provider to transparently attest to its customers the deduplication patterns of the (encrypted) data that it is storing. By doing so, ClearBox enables cloud users to verify the effective storage space that their data is occupying in the cloud, and consequently to check whether they qualify for benefits such as price reductions, etc. ClearBox is secure against malicious users and a rational storage provider, and ensures that files can only be accessed by their legitimate owners. We evaluate a prototype implementation of ClearBox using both Amazon S3 and Dropbox as back-end cloud storage. Our findings show that our solution works with the APIs provided by existing service providers without any modifications and achieves comparable performance to existing solutions.

1. INTRODUCTION

Cloud storage services have become an integral part of our daily lives. With more and more people operating multiple devices, cloud storage promises a convenient means for users to store, access, and seamlessly synchronize their data from multiple devices.

The increasing adoption of the cloud is also fueled by the multitude of competing cloud storage services which offer relatively cheap services. For example, Dropbox offers its customers free 2 GB accounts, Google drive offers 100 GB for only 1.99 USD, while Box.com offers its business clients unlimited storage for only 12 EUR per month. These competitive offers are mainly due to the sharp plummet exhibited by modern hard drives, going from 20 USD per GB to just about few cents per GB in 2014 [9].

CCS'15, October 12–16, 2015, Denver, Colorado, USA.

DOI: http://dx.doi.org/10.1145/XXX.XXXXXXX.

To further increase their profits,[1] existing cloud storage providers adopt aggressive storage efficiency solutions when storing their clients' data. Namely, existing clouds store duplicate data (either at the block level or the file level) uploaded by different users only once—thus tremendously saving storage costs. Recent studies show that cross-user data deduplication can save storage costs by more than 50% in standard file systems [35, 36], and by up to 90-95% for back-up applications [35].

The literature features a large number of proposals for securing data deduplication (e.g., [14, 25, 42]) in the cloud. All these proposals share the goal of enabling cloud providers to deduplicate *encrypted* data stored by their users. Such solutions allow the cloud provider to reduce its total storage, while ensuring the confidentiality of stored data.

By doing so, existing solutions increase the profitability of the cloud, but do not allow users to directly benefit from the savings of deduplication over their data. Notice that cloud service providers charge their customers based on the amount of data that they store—irrespective of the level of data deduplication exhibited by their data. However, a user who is using the cloud as back-up storage should benefit—and rightly so—from reductions (by up to 90%), when compared to a client who is storing personal files in the cloud which are less likely to be deduplicated. In Figure 1, we estimate the cost reductions per user due to data deduplication in comparison to the price per user in existing commodity cloud providers such as Dropbox, Google drive, and Microsoft Onedrive. Our estimates clearly suggest that there is considerable room for price reductions for those users whose data undergoes considerable deduplication.

In this paper, we address this problem and we propose a novel secure storage solution, dubbed ClearBox, which enables a cloud provider to *transparently* and verifiably attest the deduplication patterns of every file stored at the cloud. Our solution relies on gateway to orchestrate cross-user file-based deduplication prior to storing files on (public) cloud servers. ClearBox ensures that files can only be accessed by legitimate owners, resists against a curious cloud provider, and enables cloud users to verify the effective storage space occupied by their encrypted files in the cloud (after deduplication). By doing so, ClearBox provides its users with full transparency on the storage savings exhibited by their data; this allows users to assess whether they are acquiring appropriate service and price reductions for their money—in spite of a *rational* gateway that aims at maximizing its profit.

ClearBox can be integrated with existing cloud storage providers such as Amazon S3 and Dropbox without any modifications, and motivates a new cloud pricing model which takes into account the

[1] Cloud services are contributing to a 150 billion USD market [6].

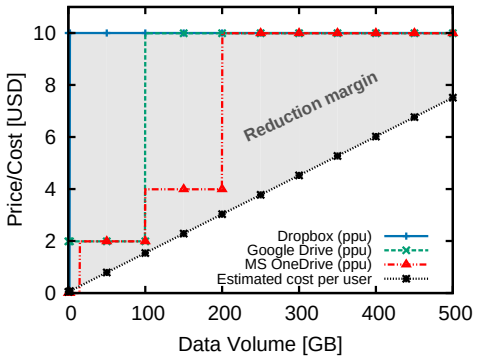


Figure 1: Cost reductions due to data deduplication vs. prices of commodity storage providers. The blue, green, and red curves show the price currently charged by Dropbox, Google Drive, and MS OneDrive, respectively. The dotted black line depicts the estimated cost of storage per user in Amazon S3 [1] after data undergoes deduplication. We assume that 50% of the data stored by clients is deduplicated [36] with the data pertaining to 2 other cloud users and that clients download 0.96% of the data stored in their accounts per day [34]. The "reduction margin" refers to the difference between the price borne by the users and the effective cost of users' storage after deduplication. "ppu" refers to the price per user.

level of deduplication undergone by data. We believe that such a model does not threaten the profitability of the cloud business and—on the contrary—gives considerable incentives for users to store large and popular data such as music and video files in the cloud (since the storage costs of popular data might be cheaper). In summary, we make the following contributions in this work:

Concrete Instantiation: We describe a cloud storage scheme, dubbed ClearBox, which employs a novel cryptographic tree-based accumulator, CARDIAC, to attest in logarithmic time (with respect to the number of clients that uploaded the file) the deduplication patterns of every file stored in the cloud. ClearBox additionally leverages Proofs of Ownership [23, 27], and self-expiring URL commands in order to effectively manage access control on the files stored in the cloud.

Security Analysis: We provide a model for ClearBox and analyze the security of our proposal according to our model. Namely, we show that ClearBox enables users to verify the deduplication undergone by their files in spite of a rational provider that aims at maximizing its profit in the system. In addition, we show that ClearBox resists against malicious clients and a curious storage provider.

Prototype Implementation: We implement and evaluate a prototype based on ClearBox using both Amazon S3 and Dropbox as back-end cloud storage, and we show that our proposal does not impair the experience witnessed by cloud users and incurs tolerable overhead on the gateway when orchestrating file operations amongst its clients.

The remainder of this paper is organized as follows. In Section 2, we introduce our model and goals. In Section 3, we present ClearBox and analyze its security. In Section 4, we evaluate a prototype implementation based on the integration of ClearBox with Amazon S3 and Dropbox. In Section 5, we review related work in the area, and we conclude the paper in Section 6. In Appendices A and B, we provide additional insights with respect to our scheme.

2. MODEL

Before we give a full description of ClearBox in Section 3, we present in this section our system and security models.

2.1 System Model

ClearBox comprises a number of clients $C_1, C_2, \ldots$ that are interested in storing their files at a storage provider S such that each client learns the level of deduplication undergone by his files, that is the number of clients that outsourced the same file. Since existing storage providers do not report the deduplication patterns of the stored data, a straightforward approach would be to rely on a decentralized scheme whereby users coordinate their file uploads prior to storing their data onto the cloud; such a decentralized scheme, however, requires interaction among users, and is unlikely to scale as the number of users storing the same data increases [41]. This is exactly why ClearBox makes use of an additional gateway G, which interfaces between the users and the cloud, and orchestrates data deduplication prior to storing the data on the cloud. Note that G is a logically centralized entity, and can be easily instantiated using distributed servers. We argue that our model is generic and captures a number of practical deployment scenarios. For instance, G could be an independent service which offers cheaper cloud storage, by performing data deduplication over existing public clouds; alternatively, G could be a service offered by S itself in order to offer differentiation to existing cloud storage services, etc.

In our scheme, we assume that G owns an account hosted by S, and orchestrates cross-user file-based deduplication[2]. By doing so, G can provide its users with full transparency on the storage savings due to deduplication exhibited by their data. For instance, G can offer price reductions for customers whose data is highly deduplicated (e.g., 50% discount if the file is deduplicated among at least n entities). Alternatively, G could fairly distribute storage costs amongst users who are storing deduplicated files; for example, assume that there are n clients all storing the same file f, then each client can effectively be charged a fraction of $\frac{1}{n}$ of the cost of storing f. Notice that these reductions do not threaten the profitability of G (nor S) which can still comfortably profit from the various offered services (e.g., resilience to failures, mobile access, data synch across devices). On the contrary, we argue that offering the option of sharing storage costs with other clients storing the same files may provide a clear differentiator compared to other competitors. Nevertheless, providers are clearly not keen on sharing parts of their profits with the users and may not correctly report the cost reductions due to data deduplication. The challenge here therefore lies in transparently and efficiently attesting data storage costs (i.e., including deduplication savings) across users in the presence of a storage provider which might not correctly report the deduplication (and access) patterns of the data that it is storing.

Conforming with the operation of existing storage providers, we assume that time is divided into epochs E_i of equal length, e.g., 12 hours [1]. Clients receive from G a list of their files and *deduplication patterns* at the end of every epoch. The deduplication pattern of a given file refers to the number of users that store the same deduplicated file in the cloud.

Similar to existing storage providers, ClearBox supports the following operations: Put, Get, Delete. In addition, ClearBox supports two additional protocols, Attest and Verify, which are used to prove/verify the deduplication patterns undergone by files. We

[2]Although block-based deduplication can lower storage consumption to as little as 32% of its original requirements, we point out that nearly three quarters of the improvement observed could be captured through whole-file deduplication [36].

start by describing these protocols. Here, the expression

$$\Pi : [P_1 : in_1;\ \ldots,\ P_n : in_n] \rightarrow [P_1 : out_1;\ \ldots,\ P_n : out_n]$$

denotes the event that a set of parties $P_1, \ldots, P_n$ run an interactive protocol Π where in_i and out_i denote the input and output of P_i, respectively.

The Put *Protocol.*

The Put protocol is executed between the gateway and a client C who aims to upload a file f. Initially, the client derives a key k_{FID} from the file which he uses to encrypt f to f^* which is eventually uploaded to S via G. Next, both parties derive a file ID FID that will serve as a practically unique handle to f. The gateway G maintains internally a set $\mathcal{F}$ which contains pairs $(FID, \mathcal{C}_{FID})$ where $\mathcal{C}_{FID}$ is the set of clients that are registered to the file referenced by FID. If no client is registered to a file with file ID FID, it holds that $\mathcal{F}$ does not contain an element of the form $(FID, *)$. Given this, the gateway checks if any users are registered to this file already, i.e., if $(FID, \mathcal{C}_{FID}) \in \mathcal{F}$. If so, it inserts C into $\mathcal{C}_{FID}$. Otherwise, a new set $\mathcal{C}_{FID} = \{C\}$ is created and $(FID, \mathcal{C}_{FID})$ inserted into $\mathcal{F}$. Moreover, the client gets a verification tag τ which later allows to validate the proofs generated by G.

$$\begin{aligned}\mathsf{Put} \quad : \quad & [C : f;\ G : \mathcal{F};\ S : \perp] \longrightarrow \\ & [C : FID, k_{FID}, \tau;\ G : FID, \mathcal{F};\ S : f^*]\end{aligned}$$

When we need to specify the verification tag of a specific client C, we write τ_C.

The Get *Protocol.*

When a client C wants to download a file f, it initiates this protocol with G and sends the corresponding file ID FID. The gateway first checks if $\mathcal{F}$ contains an entry $(FID, \mathcal{C}_{FID})$. In the positive case, it further verifies if $C \in \mathcal{C}_{FID}$. If this verification passes, actions are taken such that eventually the client can download the encrypted file f^* from S and decrypt it to f with k_{FID}. Observe that we do not specify additional tokens to authenticate the client as we assume the existence of authenticated channels between the clients and G.

$$\begin{aligned}\mathsf{Get} \quad : \quad & [C : FID, k_{FID};\ G : \mathcal{F};\ S : f^*] \longrightarrow \\ & [C : f;\ G : \perp;\ S : \perp]\end{aligned}$$

The Delete *Protocol.*

This protocol allows a client to delete a file or, more precisely, to cancel his registration. To this end, the client sends the file ID FID to the gateway who checks if $(FID, \mathcal{C}_{FID}) \in \mathcal{F}$ for some set $\mathcal{C}_{FID}$ and if $C \in \mathcal{C}_{FID}$. If this is not the case, the request is simply ignored and C receives $f = \perp$. Otherwise $\mathcal{C}_{FID}$ is updated to $\mathcal{C}_{FID} \setminus \{C\}$ at the beginning of the next epoch. If $\mathcal{C}_{FID}$ becomes empty by this action, this means that no user is any longer registered to this file. Hence, G can request to S to delete the file.

$$\begin{aligned}\mathsf{Delete} \quad : \quad & [C : FID;\ G : \mathcal{F};\ S : f^*] \longrightarrow \\ & [C : \perp;\ G : \mathcal{F};\ S : \perp]\end{aligned}$$

The Attest *Protocol.*

The purpose of the Attest procedure, which is executed by the gateway only, is twofold. On the one hand, it generates a proof of cardinality for a given file ID FID and an epoch E that attests an upper bound for $|\mathcal{C}_{FID}|$—the number of clients registered to this file within this epoch. On the other hand, it also includes a proof of membership for a given client C with respect to $\mathcal{C}_{FID}$. Formally, we have:

$$\mathcal{A} \leftarrow \mathsf{Attest}(FID, E, \mathcal{C}_{FID}, C).$$

The proof $\mathcal{A}$, i.e., the output of Attest, contains a claim on an upper bound of $|\mathcal{C}_{FID}|$, a compact digest for $\mathcal{C}_{FID}$, and possibly additional information, so that a client $C \in \mathcal{C}_{FID}$ can use the subsequent Verify to get convinced of the upper bound of the set $\mathcal{C}_{FID}$ and its own membership. As described in Section 2.2, an upper bound for $|\mathcal{C}_{FID}|$ is sufficient and necessary in our model. Namely, G does not have incentives to report a larger value of $|\mathcal{C}_{FID}|$ since this results in G under-charging clients and hence a reduction of the profit. Therefore, to increase its profits, G's interest is to claim the smallest possible upper bound at the end of each epoch.

The Verify *Protocol.*

In ClearBox, customers can verify (using the Verify protocol) the proof generated by the Attest protocol to confirm that they are part of the set of file users, and to verify the upper bound on the total number of file users. The Verify algorithm is executed by a client, and uses the verification tag which has been generated during the Put procedure.

It outputs either accept or reject to indicate whether the proof is accepted or not.

$$\mathrm{accept}|\mathrm{reject} \leftarrow \mathsf{Verify}(FID, E, \mathcal{A}, \tau).$$

2.2 Security Model

In the sequel, we assume that the communication between a client and the gateway is authenticated to provide non-repudiation and, in the case of need, encrypted. As already stated, we assume that time is divided into a sequence of epochs $E_1, E_2, \ldots$, where $E \leq E'$ means that E either took place before epoch E' or that both refer to the same epoch. If not mentioned otherwise, we will always refer to epochs that happened in the past. Moreover, we assume that all parties are synchronized. That is, at each point in time, all parties share the same view on the current epoch (e.g., its index number if epochs are represented by an increasing counter).

In each epoch, several protocol runs may be executed between the gateway G and a client. A protocol run is represented by a quadruple $p_i = (C, \mathrm{prot}, (in), (out))$ where C is a client, prot denotes one of the protocols Put, Get, Delete, in denotes the inputs of C, and out its outputs. For any past epoch E, we denote by $\mathbb{P}_E$ all protocol runs that occurred within this epoch. Here, we restrict ourselves to *full* protocol runs, i.e., which have not been aborted. We assume that these are uniquely ordered within the epoch. That is for $\mathbb{P}_E = \{p_1, \ldots, p_\ell\}$ where ℓ denotes the total number of protocol runs within this epoch, it holds for any $p, p' \in \mathbb{P}_E$ with $p \neq p'$ that either $p < p'$ (when p took place before p') or $p' < p$ (when p' happened first). We extend this order to the set of all protocol runs in a straightforward way. More precisely, for any two epochs $E \neq E'$ with $E < E'$ it holds that $p < p'$ for all $p \in \mathbb{P}_E$ and all $p' \in \mathbb{P}_{E'}$. Finally, we denote by $\mathbb{P}_{\leq E} = \bigcup_{E' \leq E} \mathbb{P}_{E'}$ all protocol runs that took place up to epoch E inclusive and define $\mathbb{P}_{<E}$ analogously.

DEFINITION 1 (FILE REGISTRATION).
We say that a client C is registered to a file *ID FID at some epoch E if there exists a $p = (C, \mathsf{Put}, (f), (FID, k_{FID}, \tau)) \in \mathbb{P}_{\leq E}$ for which one of the following two conditions hold:*

1. *$p \in \mathbb{P}_E$*
2. *$p \in \mathbb{P}_{<E}$ and for all $p' \in \mathbb{P}_{<E}$ with $p < p'$ it holds that $p' \neq (C, \mathsf{Delete}, (FID), (\perp))$.*

We denote by $\mathcal{C}_{FID}(E)$ the set of all clients that are registered to FID at epoch E. If the considered epoch is clear from the context, we simply write $\mathcal{C}_{FID}$. Finally, we denote by $\mathbb{E}_{\mathrm{reg}}(E, C, f, FID, \tau)$ the event that C is registered to FID within epoch E where f is the file specified in the protocol run p mentioned above

(that is the file used in the last upload). Obviously, it holds that $C \in \mathcal{C}_{FID}(E)$ *if and only if* $\mathbb{E}_{\text{reg}}(E, C, f, FID, \tau)$ *holds for some file* f.

The first condition in Def.1 means that if C uploaded the file within epoch E, he is registered to the file, no matter if he requests to delete it later in the same epoch. The second condition means that if the upload took place in a previous epoch, C must not have subsequently asked to delete it in an older epoch. Observe that also here, if C asks to delete it within epoch E, he is still registered to FID within this epoch. Likewise, we allow the client to download the file any time within an epoch if he is registered to the file within this epoch.

We are now ready to formally define correctness and soundness.

Correctness

For correctness, two requirements arise. First, a user who uploaded a file with Put, must be able to obtain it with Get for those epochs during which he is registered to this file. As files are identified by their file ID, we address the file ID generation process first. Namely, we say that the file ID generation process creates *ε-unique* file IDs if it holds for any two protocol runs $(C, \mathsf{Put}, (f), (FID, k_{FID}))$ and $(C', \mathsf{Put}, (f'), (FID', k'_{FID}))$ that

$$Pr\left[FID = FID' | f = f'\right] = 1, \quad (1)$$

$$Pr\left[FID = FID' | f \neq f'\right] \leq \varepsilon. \quad (2)$$

The first condition means that the same file leads always to the same file ID while the probability that different files result into the same file ID is at most ε. This allows us to define correct access for our scheme:

DEFINITION 2 (CORRECT ACCESS). *Assume that the file ID generation process is ε-unique. We say that the scheme provides* correct access *if it holds for any client C, for any epoch E, and any file f that if the event $\mathbb{E}_{\text{reg}}(E, C, f, FID)$ holds and if there exists a protocol run $(C, \mathsf{Get}, (FID, k_{FID}), (f')) \in \mathbb{P}_E$, then:*

$$Pr\left[f' = f\right] \geq 1 - \varepsilon. \quad (3)$$

The reason that one cannot require the probability to be equal to 1 is that with some probability ε, two different files may get the same file ID. In these cases, correctness cannot be guaranteed anymore.

DEFINITION 3 (CORRECT ATTESTATION). *Let FID be an arbitrary file ID and E be an arbitrary epoch. Moreover, let $\mathcal{A}$ be a proof generated by the gateway for a file FID and epoch E. That is, $\mathcal{A} \leftarrow \mathsf{Attest}(FID, E, \mathcal{C}_{FID})$. Moreover, let bd denote the upper bound for $|\mathcal{C}_{FID}(E)|$ claimed in $\mathcal{A}$ and let C be an arbitrary client.*

We say that the scheme provides correct attestation *if under the conditions of $\mathbb{E}_{\text{reg}}(E, C, f, FID, \tau)$ and $|\mathcal{C}_{FID}(E)| \leq bd$ it holds that:*

$$Pr\left[\text{accept} \leftarrow \mathsf{Verify}(FID, E, \mathcal{A}, \tau)|\right] = 1. \quad (4)$$

Soundness

We assume that each party (client, gateway, service provider) can potentially misbehave but aim for different attack goals. Consequently, soundness requires that security is achieved against all these attackers. In what follows, we motivate each attacker type and define the corresponding security goals.

Malicious Clients: In principle, we assume that a client may be arbitrarily malicious with various aims: disrupting the service, repudiate actions, and gain illegitimate access to files. The first two can be thwarted with standard mechanisms like authenticated channels. Thus, we focus on the third only: a user must only be able to access a file f via Get if he had uploaded it before to G and if he is still registered to it.

DEFINITION 4 (SECURE FILE ACCESS). *We say that the scheme provides ε-*secure file access *if it holds for any client C, for any epoch E, and any file f that for any $(C, \mathsf{Get}, (FID, k_{FID}), (f')) \in \mathbb{P}_E$ while $\mathbb{E}_{\text{reg}}(E, C, f, FID, \tau)$ does not hold, then*

$$Pr\left[f' = \perp\right] \geq 1 - \varepsilon. \quad (5)$$

Rational Gateway: We assume that both the gateway and the service provider are *rational* entities, e.g., see [43] for a similar assumption. By rational, we mean that the gateway and the storage provider will only deviate from the protocol if such a strategy increases their profit in the system. Notice that gateway has access to all the information witnessed by the service provider (e.g., access to the stored files). This implies that any attack by a rational service provider may equally be executed by the gateway. Likewise, a rational gateway could not mount stronger attacks by colluding with the service provider. Hence, we restrict our analysis in the sequel to the security of our scheme given a rational gateway.

Recall that the gateway performs two types of interactions: attesting the number of registered clients and handling clients' files. Consequently, we see two different types of strategies that may allow the gateway to increase its advantage in the system: *(i)* overcharging clients and *(ii)* illegitimately extracting file content.

With respect to the first strategy, if a rational gateway manages to convince clients of a smaller level of deduplication, the gateway can charge higher prices. On the other hand, with respect to the second strategy, the gateway may try to derive information about the contents of the uploaded files[3]; notice that acquiring information about users' data can be economically beneficial for the gateway since the stored data can be misused, e.g. sold. Consequently, two security goals need to be ensured: secure attestation and data confidentiality, that we formalize next.

DEFINITION 5 (SECURE ATTESTATION). *Let FID be an arbitrary file ID and E be an arbitrary epoch. Moreover, let $\mathcal{A}$ be a proof generated by the gateway for a file FID, an epoch E,and a client C, that is $\mathcal{A} \leftarrow \mathsf{Attest}(FID, E, \mathcal{C}_{FID}, C)$. Moreover, let bd denote the upper bound for $|\mathcal{C}_{FID}(E)|$ claimed in $\mathcal{A}$. We say that the scheme provides ε-*secure attestation *if $\neg\mathbb{E}_{\text{reg}}(E,$ $C, f, FID, \tau)$ or $|\mathcal{C}_{FID}(E)| > bd$ implies that*

$$Pr\left[\text{accept} \leftarrow \mathsf{Verify}(FID, E, \mathcal{A}, \tau)|\right] \leq \varepsilon. \quad (6)$$

Observe that it is not in the interest of a rational gateway in our model to report a larger value for $|\mathcal{C}_{FID}(E)|$ as this would allow the clients to demand further cost reductions.

With respect to the confidentiality of the uploaded data, notice that standard semantically secure encryption schemes cannot be used in our context since they effectively prevent the deduplication of data [14, 25, 42]. Schemes that are suitable for deduplication produce the same ciphertext for multiple encryptions of the same message and are referred to as message-locked encryption (MLE) [13, 15]. The achievable security in our case is therefore that of MLE schemes.

To describe the security of MLE schemes, we adapt in what follows the security notion introduced in [15] which guarantees privacy under chosen distribution attacks.

[3]We assume the gateway to be curious in this respect.

An MLE scheme consists of a tuple of algorithms ($setup$, $keygen$, enc,dec, tag), where $setup$ generates (public) parameters P, the key generation $keygen$ generates a key k given P and a message f; finally, enc and dec are the algorithms to encrypt and decrypt f with key k. The tag generation algorithm tag creates a tag for a ciphertext. This is covered in our model by the unique file identifier FID and does not play a role in the privacy notion.

A message space is given by an algorithm $\mathcal{M}$ which samples messages $M \in \{0,1\}^*$ according to a distribution and may provide additional context information. The message sampling must be unpredictable, i.e., the probability to predict the output of $\mathcal{M}$ is negligible given context information. The security of an MLE scheme is defined by the following experiment PRV$-CDA$^A_{MLE,\mathcal{M}}$, an unpredictable sampling algorithm $\mathcal{M}$ and an adversary A [15]:

1. The environment randomly generates a parameter set P and a bit $b \leftarrow \{0,1\}$.
2. The environment samples a set of messages M and context information Z: $(M, Z) \leftarrow \mathcal{M}$. As the same message would result in the same encryption, the restriction here is that all messages are distinct.
3. For all messages $M[i]$ in M, the environment encrypts message $M[i]$ with the MLE scheme:
$$C_0[i] \leftarrow enc_{keygen(M[i])}(M[i])$$
and generates a random string of the same length:
$$C_1[i] \leftarrow \{0,1\}^{|C_0[i]|}.$$
4. The adversary obtains the set C_b and outputs a guess for b: $\widetilde{b} \leftarrow A(P, C_b, Z)$
5. The adversary wins if b equals $\widetilde{b}$ in this case the experiment returns 1, otherwise 0.

DEFINITION 6 (DATA CONFIDENTIALITY). *We say that the MLE scheme for message sampling algorithm $\mathcal{M}$ is secure, if the adversary's advantage in winning the PRV$-CDA$^A_{MLE,\mathcal{M}}$ experiment is negligible, i.e.*

$$2 \cdot Pr\left[1 \leftarrow \textit{PRV\$-CDA}^A_{MLE,\mathcal{M}}\right] - 1 \leq \text{negl}(\kappa),$$

for a security parameter κ.

Notice that it is integral for both G and S to know the file sizes and the download patterns of files in order to perform correct accounting[4]. Therefore, hiding this information cannot be part of our goals.

2.3 Design Goals

In addition to the security goals stated above, our proposed solution should satisfy the following functional requirements. ClearBox should work within the APIs provided by current service providers, without deteriorating the performance witnessed by users when compared to standard solutions where users directly interface with S. Similar to existing cloud storage providers such as Amazon S3 and Google Cloud Storage, we assume that S exposes to its clients a standard interface consisting of a handful of basic operations, such as storing a file, retrieving a file, deleting a file, generating a signed URL for sending HTTP commands for storage/retrieval, etc. (cf. Table 1). Where appropriate, we also discuss the case where S is a commodity cloud service provider and exposes a simpler interface, e.g., that does not allow storing a file using a URL.

[4]Current cloud services maintain a detailed log containing all the activities per account.

Command	Description
createBucket(B)	Creates a bucket B
PUT(B, FID)	Upload a file FID to B
GET(B, FID)	Download a file FID from B
DELETE(B, FID)	Delete a file FID from B
generateURL(COMMAND, t)	Generate a URL expiring at time t supporting PUT, GET, DELETE.

Table 1: Sample API exposed by Amazon S3 and Google Cloud Storage. COMMAND refers to an HTTP command such as PUT(B, FID).

Moreover, our solution should scale with the number of users, the file size, and the number of uploaded files, and should incur tolerable overhead on the users when verifying the deduplication patterns of their files at the end of every epoch.

3. ClearBox

In this section, we present our solution, and analyze its security according to the model outlined in Section 2.

3.1 Overview

ClearBox ensures a transparent attestation of the storage consumption of users whose data is effectively being deduplicated—without compromising the confidentiality of the stored data.

To attest the deduplication patterns to its customers, one naive solution would be for the gateway to publish the list of all clients associated to each deduplicated file (e.g., on a public bulletin board) such that each client could first check if *(i)* he is a member of the list and *(ii)* if the size of the list corresponds to the price reduction offered by the gateway for storing this file. Besides the fact that this solution does not scale, it is likewise within the interest of G not to publish the entire list of its clients and their files; for example, competitors could otherwise learn information on the service offered by G (e.g., total turnover).

To remedy this, ClearBox employs a novel Merkle-tree based cryptographic accumulator which is maintained by the gateway to efficiently accumulate the IDs of the users registered to the same file within the same time epoch. Our construct ensures that each user can check that his ID is correctly accumulated at the end of every epoch. Additionally, our accumulator encodes an upper bound on the *number* of accumulated values, thus enabling any legitimate client associated to the accumulator to verify (in logarithmic time with respect to the number of clients that uploaded the same file) this bound.

Clearly, a solution that requires the gateway to publish details about all the accumulators for all the stored files does not scale with the number of files stored in the cloud. This is why ClearBox relies on a probabilistic algorithm which selectively reveals details about a number of file accumulators in each epoch. However, if the gateway could select which file accumulators to publish, then G could easily cheat by only creating correct accumulators for the selected files, while misreporting the deduplication patterns of the remaining files. In ClearBox, the choice of which accumulators are published in each epoch is seeded by an external source of randomness which cannot be predicted but can be verified by any entity. We show how such a secure source of randomness can be efficiently instantiated using Bitcoin. This enables any client to validate that the sampling has been done correctly, and as such that he is acquiring the promised price reductions—without giving any advantage for G to misbehave.

ClearBox enforces fine-grained access control on shared files by leveraging self-expiring URLs when accessing content. Namely, whenever a user wishes to access a given resource, the gateway

generates a URL for that resource on the fly, which expires after a short period of time. As shown in Table 1, existing cloud APIs support the dynamic generation of expiring URLs. By doing so, ClearBox does not only ensure that G can restrict access to the data stored on the cloud, but also enables G to keep track of the access patterns of its users (e.g., to be used in billing). ClearBox also relies on an oblivious server-aided key generation protocol to ensure that the stored files are encrypted with keys that are dependent on both the hash of the file and the gateway's secret. This protects against brute force search attacks when the message content is predictable, but also ensures that a curious gateway/storage provider which does not know the file hash cannot acquire the necessary keys to decrypt the file (since the key generation protocol is oblivious). To protect against malicious users who otherwise have obtained the file hash (e.g., by theft/malware) but do not possess the full file, ClearBox employs proofs of ownership over the *encrypted* file to verify that a given user is indeed in possession of the full file.

In the following subsections, we go into greater detail on the various parts of ClearBox, starting with the building blocks that we will use in our solution, then moving on to the protocol specification, and finally to its security analysis.

3.2 Building Blocks

Before describing ClearBox in detail, we start by outlining the building blocks that will be used in ClearBox.

3.2.1 CARDIAC

Cryptographic accumulators (e.g. [12, 20, 21, 22, 30, 33, 39]) basically constitute one-way membership functions; these functions can be used to answer a query whether a given candidate belongs to a set.

In what follows, we show how to construct a cardinality-proving accumulator (CARDIAC) which leverages Merkle trees in order to efficiently provide proofs of membership *and* (non-public) proofs of maximum set cardinality. As we show in Section 3.3, proofs of cardinality are needed to attest the file deduplication patterns to users.

A Merkle tree is a binary tree, in which the data is stored in the leaves. Let $a_{i,j}$ denote a node in the tree located at the i-th level and j-th position. Here, the level refers to the distance (in hops) to the leaf nodes; clearly, leaf nodes are located at distance 0. On the other hand, the position within a level is computed incrementally from left to right starting from position 0; for example, the leftmost node of level 1 is denoted by $a_{1,0}$. In a Merkle tree, the intermediate nodes are computed as the hash of their respective child nodes; namely $a_{i+1,j} = \mathtt{H}(a_{i,2j}, a_{i,2j+1})$.

Given a tree of height ℓ, CARDIAC accumulates elements of a set X by assigning these to the leaf nodes (starting from position 0) while the remaining leaf nodes $a_{0,|X|}, \ldots, a_{0,2^\ell-1}$ are filled with a distinct symbol $\mathbf{0}$. We call these the *zero leaves*. Nodes that can be computed from the zero leaves play a special role. We denote these as *open* nodes in the sense that their values are openly known. More formally, the zero leaves $a_{0,|X|}, \ldots, a_{0,2^\ell}$ are open. Moreover, if $a_{i,2j}$ and $a_{i,2j+1}$ are both open, so is $a_{i+1,j} = \mathtt{H}(i+1, a_{i,2j}, a_{i,2j+1})$.

We now outline the main algorithms $(\mathtt{Acc}, \mathtt{Prove}_M, \mathtt{Verify}_\mathtt{M}, \mathtt{Prove}_C, \mathtt{Verify}_\mathtt{C})$ provided by CARDIAC. Observe that $\mathtt{Prove}_M$ and $\mathtt{Prove}_C$ will be used to implement Attest and likewise $\mathtt{Verify}_\mathtt{M}$ and $\mathtt{Verify}_\mathtt{M}$ to instantiate Verify.

$\delta \leftarrow \mathtt{Acc}(X)$**.** This algorithm accumulates the elements of a set X into a digest δ. In CARDIAC, δ corresponds to the hash of the root node of the modified Merkle tree, $a_{\ell,0}$, and the height ℓ of the tree, i.e., $\delta = \mathtt{H}(a_{\ell,0}, \ell)$.

$\pi_M \leftarrow \mathtt{Prove}_M(X, x)$**.** Given a set X and element $x \in X$, this algorithm outputs a proof of membership π_M asserting that $x \in X$. π_M consists of the sibling path of x in the modified Merkle tree and the root $a_{\ell,0}$.

$\mathtt{Verify}_\mathtt{M}(\delta, x, \pi_M)$**.** Given δ, an element x, its sibling path and the root $a_{\ell,0}$, this algorithm outputs `true` if and only if $\delta = \mathtt{H}(a_{\ell,0}, \ell)$ where ℓ is the length of the sibling path and the sibling path of x matches the root $a_{\ell,0}$.

$\pi_C \leftarrow \mathtt{Prove}_C(X)$**.** Given a set X, this algorithm outputs a proof π_C attesting an upper bound on the size $|X|$ of the set. Here, π_C consists of the size of X, the right-most non-zero element $a_{0,|X|-1}$, its sibling path, and the root of the tree $a_{\ell,0}$.

$\mathtt{Verify}_\mathtt{C}(\delta, c, \pi_C)$**.** Given the digest δ, the cardinality $|X|$ of the set X, and a proof of cardinality consisting of $a_{0,|X|-1}$, its sibling path, and the root of the tree $a_{\ell,0}$, this algorithm outputs `true` if the following conditions are met:

- it holds that $\delta = \mathtt{H}(a_{\ell,0}, \ell)$ where ℓ is the length ℓ of the sibling path,
- it holds that $2^{\ell-1} < |X| \leq 2^\ell$,[5]
- the sibling path of $a_{0,|X|-1}$ matches the root $a_{\ell,0}$, and
- all nodes on the sibling path that are open do contain the right value.

Here, $\mathtt{Verify}_\mathtt{C}$ assumes that all the leafs with position larger than $|X| - 1$ in level 0 are filled with the $\mathbf{0}$ element which is not contained in X.

In Appendix A, we show that, in addition to proofs of membership, our CARDIAC instantiation provides a proof that the number of non-zero leaves in the Merkle tree with root $a_{\ell,0}$ is at most $|X|$.

3.2.2 Time-Dependent Randomness

ClearBox leverages a time-dependent randomness generator $\mathsf{GetRandomness} : T \to \{0,1\}^{\ell_{\text{seed}}}$ where T denotes a set of discrete points in time. In a nutshell, GetRandomness produces values that are unpredictable but publicly reconstructible.

More formally, let cur denote the current time. On input $t \in T$, GetRandomness outputs a uniformly random string in $\{0,1\}^{\ell_{\text{seed}}}$ if $t \leq \mathsf{cur}$, otherwise GetRandomness outputs $\perp$. We say that GetRandomness is secure if the output of $\mathsf{GetRandomness}(t)$ cannot be predicted with probability significantly better than $2^{-\ell_{\text{seed}}}$ as long as $t < \mathsf{cur}$.

Similar to [10], we instantiate GetRandomness by leveraging functionality from Bitcoin, since the latter offers a convenient means (e.g., by means of API) to acquire time-dependent randomness. Namely, Bitcoin relies on blocks, a hash-based Proof of Work concept, to ensure the security of transactions. In particular, Bitcoin peers need to find a nonce, which when hashed with the last block hash and the root of the Merkle tree accumulating recent transactions, the overall hash is smaller than a 256-bit threshold.

The difficulty of block generation in Bitcoin is adjusted so that blocks are generated once every 10 minutes on average; it was shown in [29] that the block generation in Bitcoin follows a shifted geometric distribution with parameter $p = 0.19$. Recent studies show that a public randomness beacon—outputting 64 bits of min-entropy every 10 minutes—can be built atop Bitcoin [8].

Given this, GetRandomness then unfolds as follows. On input time t, GetRandomness outputs the hash of the latest block that has appeared since time t in the Bitcoin block chain. Clearly, if $t > \mathsf{cur}$ corresponds to a time in the future, then GetRandomness

[5]This condition minimizes the number of open nodes in the tree, and hence the effort in proving/verifying membership and cardinality.

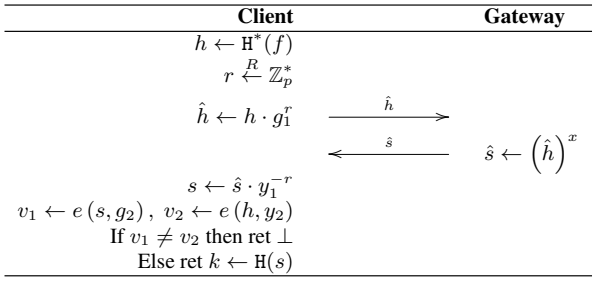


Figure 2: Server-aided key generation module based on blind BLS signatures. Here, Γ_1, Γ_2 are two groups with order p, g_1, g_2 generators of Γ_1 and Γ_2 respectively, the pairing function $e : \Gamma_1 \times \Gamma_2 \rightarrow \Gamma_T$, the hash function $\mathtt{H}^* : \{0,1\}^* \rightarrow \Gamma_1$, the secret key $x \in \mathbb{Z}_p^*$, with corresponding public keys $y_1 = g_1^x, y_2 = g_2^x$.

will output $\perp$, since the hash of a Bitcoin block that would appear in the future cannot be predicted. On the other hand, it is straightforward to compute GetRandomness(t), for a value $t \leq$ cur (i.e., t is in the past) by fetching the hash of previous Bitcoin blocks. In this way, GetRandomness enables an untrusted party to sample randomness—without being able to predict the outcome ahead of time.

The purpose of GetRandomness is to ensure that the selection of files to which the gateway attests the deduplication pattern is randomly chosen and not precomputed.

While using a time-dependent source of randomness is a viable option, other alternatives may be imaginable. For example, one may couple the selection of files with Fiat-Shamir heuristics [26] to ensure that the selection is randomly chosen from the point of view of the gateway and couple it with proofs of work. If the time effort to generate a valid proof of work is about the duration of an epoch, a malicious gateway may not be able to precompute the selections. We leave the question of investigating viable alternatives to external sources of randomness as an interesting direction for future research.

3.2.3 Server-Aided Key Generation

ClearBox employs an oblivious protocol adapted from [14] which is executed between clients and the gateway to generate the keys required to encrypt the stored files. Unlike [14], our protocol does not rely on RSA, and is based on blind BLS signatures [17, 18]. Although the verification of BLS signatures is more expensive than its RSA counterpart, BLS signatures are considerably shorter than RSA signatures, and are faster to compute by the gateway.

As shown in Figure 2, we assume that at setup, the gateway chooses two groups Γ_1 and Γ_2 with order p, and a computable bilinear map $e : \Gamma_1 \times \Gamma_2 \rightarrow \Gamma_T$. Additionally, the gateway chooses a private key $x \in \mathbb{Z}_p$, and the corresponding public keys $y_1 = g_1^x \in \Gamma_1$ and $y_2 = g_2^x \in \Gamma_2$. Let $\mathtt{H}^* : \{0,1\}^* \rightarrow \Gamma_1$ be a cryptographic hash function which maps bitstrings of arbitrary length to group elements in Γ_1. Prior to storing a file f, the client computes $h \leftarrow \mathtt{H}^*(f)$, blinds it by multiplying it with g_1^r, given a randomly chosen $r \in \mathbb{Z}_p$, and sends the blinded hash $\hat{h}$ to the gateway. The latter derives the signature on the received message and sends the result back to the client, who computes the unblinded signature s and verifies that: $e(s, g_2) = e\left(h^x g_1^{rx} g_1^{-rx}, g_2\right) = e(h, y_2)$.

The encryption key is then computed as the hash of the unblinded signature: $k \leftarrow \mathtt{H}(s)$. The benefits of such a key generation module are twofold:

- Since the protocol is oblivious, it ensures that the gateway does not learn any information about the files (e.g., about the file hash) during the process. On the other hand, this protocol enables the client to check the correctness of the computation performed by the gateway (i.e., verify the gateway's signature). As we show later, this verification is needed to prevent a rational G from registering users of the same file to different file versions with reduced level of deduplication.
- By involving the gateway in the key generation module, brute-force attacks on predictable messages can be slowed down by rate-limiting key-generation requests to G. Notice that, similar to [14], this scheme does not prevent a curious G from performing brute-force searches on predictable messages, acquiring the hash, and the corresponding key k. In this sense, the security offered by our scheme reduces to that of existing MLE schemes (cf. Section 3.4).

3.2.4 Proof of Ownership

The aforementioned server-aided key generation ensures that an adversary which is not equipped with the correct file hash cannot acquire the file encryption key k. However, a user who has wrongfully obtained the file hash would be able to claim file ownership. In this case, Proofs of Ownership (PoW) can be used by the gateway to ensure that the client is in possession of the file in its entirety (and not only of its hash) [16, 23, 27].

In this paper, we rely on the PoW due to Halevi *et al.* [27] which, as far as we are aware, results in the lowest computational and storage overhead on the gateway [16]. This PoW computes a Merkle tree over the file f, such that the root of the Merkle tree constitutes a commitment to the file. In the sequel, we denote by $MT_{\mathtt{Buf}(f)}$ the root of the Merkle tree as output by the PoW of [27] given an input $\mathtt{Buf}(f)$, being an encoding of a file f. The verifier can challenge the prover for any block of the file, and the prover is able to prove knowledge of the challenged block by submitting the authentication path of this block. In order to reduce the number of challenges for verifying the PoW, the file is encoded into $\mathtt{Buf}(f)$ before computing the Merkle tree. An additional trade-off can be applied by limiting $\mathtt{Buf}(f)$ to a maximum size of 64 MB in the case of larger files.

The security of this scheme is based on the minimum distance of random linear codes. We refer the readers to Appendix B for more details on the PoW of [27].

3.3 ClearBox: Protocol Specification

We now detail the specifications for the procedures of ClearBox. As mentioned earlier, we assume in the sequel that G owns an account hosted by S, and that the communication between C, S, and G occurs over authenticated and encrypted channels.

Specification of the Put *Procedure.*

In ClearBox, when a client C wishes to upload a new file f onto S, C issues an upload request to G. Subsequently, C and G start executing the server-aided key generation protocol described in Section 3.2.3. The outcome of that protocol is the key $k \leftarrow \mathtt{H}(s)$, where $s \leftarrow \mathtt{H}^*(f)^x$ given a cryptographic hash function $\mathtt{H}^*$.

C then encrypts f using encryption algorithm enc under key k, computes and sends to G the root of the Merkle tree output by the PoW of [27], that is $FID \leftarrow MT_{\mathtt{Buf}(enc(k,f))}$ where Buf is an encoding function (cf. Section 3.2.4).

Subsequently, G checks if any other client has previously stored a file indexed by FID. Here, two cases emerge:

f has not been stored before: In this case, G issues a timed generateURL command allowing the client to upload the data onto G's account within a time interval. Recall that a timed

generateURL command results in a URL which expires after the specified period of time[6]. After the upload of the encrypted file $f^* \leftarrow enc(k, f)$ terminates, G accesses S, computes $MT_{\texttt{Buf}(f^*)}$ using the PoW of [27], and verifies that it matches FID. If the verification matches, G stores the metadata associated with f^* in a newly generated structure indexed by FID (such as the client ID C and the size of f^*, see Section 4 for more details). Otherwise, if $MT_{\texttt{Buf}(f^*)}$ does not match FID, G deletes the file and appends C to a blacklist.

f has been stored before: In this case, G requests that C proves that it owns the file f. For that purpose, G and C execute the PoW protocol of [27] (we refer the reader to Appendix B for more details). In essence, G chooses a random number u of leaf indexes of the Merkle tree computed over $\texttt{Buf}(f^*)$, and asks C for for the sibling-paths of all the u leaves. In response, C returns the sibling paths corresponding to the chosen u leafs associated with the Merkle tree of $\texttt{Buf}(f^*)$. G accepts if all the sibling paths are valid with respect to the stored FID. If this verification passes, G appends C to the file structure FID, and sends an ACK to C. In turn, C deletes the local copy of the file, and only needs to store FID and the key k.

Specification of the Get *Procedure.*

To download a file with index FID, C submits FID to G; the latter checks that C is a member of the user list added to the metadata structure of FID. If so, G generates a timed URL allowing C to download the requested file from S.[7]

Notice that if C did not locally cache the decryption key associated with FID, then C can leverage its knowledge of $\texttt{H}^*(f)$ in order to acquire the corresponding key by executing the server-aided generation protocol with G.

Specification of the Delete *Procedure.*

When C wants to delete file FID, it informs G. G marks C for removal from the metadata structure associated with FID in the subsequent epoch (see Section 3.3 for more details). If no further clients are registered for this file, G sends a request to S to delete it.

Specification of the Attest *Procedure.*

At the end of each epoch, G attests the deduplication patterns of its clients' files, e.g., in their bills. Notice that if a client requested the deletion of a file f during the epoch, G only removes the marked clients from the clients list subscribed to FID after the end of the epoch.

At the end of epoch E_j, the bill of every client C of G includes for each stored file f the number of accesses by C and the cardinality of $\mathcal{C}_{FID}$, which denotes the set of clients registered to f. Here, we assume a static setting where clients are charged for files that they are storing within each epoch; this conforms with the functionality of existing providers, such as Amazon S3, which rely on fixed epochs for measuring storage consumption (e.g., the epoch interval is 12 hours in Amazon S3). Since our Attest procedure is efficient (cf. Section 4), our solution can be made practically dynamic by relying on extremely small epochs—in which case the bill can accumulate the fine-grained storage consumption over a number of epochs.

[6]In our implementation, we set the expiry timeout of the URL to 30 seconds; this means that clients have 30 seconds to *start* the upload process.

[7]Here, G additionally notes the number of download requests performed by C for file FID.

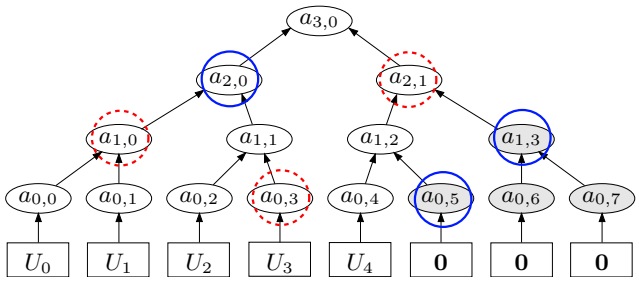


Figure 3: Sketch of a proof of set membership and set cardinality given a set of 5 elements in CARDIAC.

The bill issued by the provider acts as a binding commitment by G to the deduplication (and access) patterns of its clients' files. After the bills are issued, G needs to convince his clients that they are correctly formed. Notice that G keeps a record of the authenticated download requests by clients, which can be used to prove the number of their accesses per file. Moreover, G needs to prove to all clients of a file f that:

- Each of these clients is included in the set of clients $\mathcal{C}_{FID}$ storing FID.
- The size of $\mathcal{C}_{FID}$ is as claimed in the bill.
- Clients storing f are referenced to the *same* set $\mathcal{C}_{FID}$.

As described in Section 3.4, this last condition prevents G from including users of same file into different accumulators, resulting in the under-reporting of the file deduplication patterns.

The first two proofs can be realized by G using CARDIAC. More specifically, G accumulates the IDs of the clients storing FID; here, for each user $C_i \in \mathcal{C}_{FID}$, a leaf node with value $U_i \leftarrow \texttt{H}(FID||C_i||E_j||\text{seed}_{i,j})$ is created, where $\text{seed}_{i,j}$ denotes a nonce which is sampled for this client and this epoch and which is communicated within the bill. As we show in Section 3.4, this protects user privacy and ensures that the ID of any user is not given in the clear to other users when G issues proofs of size and/or membership. The accumulation of all $U_i \in \mathcal{C}_{FID}$ results into a digest denoted by δ. As described earlier, CARDIAC enables G to prove to each client in the set that *(i)* the client is part of the accumulated set $\mathcal{C}_{FID}$ and *(ii)* an upper bound of the size $|\mathcal{C}_{FID}|$ of the accumulated set. Observe that an upper bound on the cardinality is sufficient to protect against a rational gateway. This is the case since the gateway does not benefit from reporting a larger $|\mathcal{C}_{FID}|$, since this might entail price reductions to the clients storing f.

Figure 3 shows an example for a set comprising 5 clients $U_0, \ldots, U_4$. The elements circled in dotted red depict the membership proof for U_2, while the elements circled in solid blue depict the proof of set cardinality which consists of the sibling path of the last non-zero element. The grey elements consist of the open nodes, i.e., the zero leaves which are not associated to any client and hence filled with symbol 0, and all nodes that can be derived from these. Note that the proofs of membership and set cardinality can be further compressed in case the respective sibling paths contain the same elements.

G still needs to show that all the clients storing FID are accumulated in the same accumulator. This can be achieved by publishing the association between FID, and its corresponding accumulator digest on a public bulletin board (e.g., on the public website of G). However, this approach does not scale with the number of files processed by G. For that reason, we elect to only publish those associations for a randomly chosen subset of all files. Here, G first

obtains a random seed by invoking $\mathsf{GetRandomness}(t_j)$, where t_j denotes a point in time after the issuance of the bills for epoch E_j which is determined by a deterministic and publicly known process. This seed is used to sample the files whose accumulators need to be published during this epoch; for instance, G can extract ν-bits of the seed and use them as a mask to decide which file accumulators to publish. The probability that any file FID is selected at the end of a given epoch is subsequently given by $2^{-\nu}$.

Notice that, in this case, G needs to compute proofs of membership and set cardinality only for the selected files; this information is sent to each client storing the selected files. The pairs (FID, δ) for the selected files are subsequently published by G (e.g., on its own website).

Specification of the Verify *Procedure.*

The Verify procedure is only conducted by clients storing files for which the corresponding pairs (FID, δ) have been published by G at the end of the epoch. Notice that clients can easily check which FID are sampled by invoking $\mathsf{GetRandomness}(t_j)$.

Given the proofs of membership and set cardinality issued by G for their files, clients invoke the $\mathtt{Verify_C}$ and $\mathtt{Verify_M}$ algorithms of CARDIAC in order to verify that their ID is included in the set of clients $\mathcal{C}_{FID}$ storing FID, and that $|\mathcal{C}_{FID}|$ is as claimed in the bill. Moreover, clients check the pairs (FID, δ) published by G to verify that there is only one set $\mathcal{C}_{FID}$ which they are referenced to.

In this case, the proof of membership consists of the sibling-paths of leaves of the tree of height $\lceil \log_2(|\mathcal{C}_{FID}|) \rceil$. Hence, the size of the membership proof is at most $2 \cdot \lceil \log_2(|\mathcal{C}_{FID}|) \rceil$ hash values and verification takes at most $2 \cdot \lceil \log_2(|\mathcal{C}_{FID}|) \rceil$ executions of the hash function. On the other hand, the proof of cardinality requires at most $2^{\ell-1}$ hash operations to check the open leaves (since we assume that at least half of the leaves are occupied by the clients); that is, the size of the proof of cardinality is at most $2 \cdot \lceil \log_2(|\mathcal{C}_{FID}|) \rceil$ hash values, but verification requires $O(|\mathcal{C}_{FID}|)$ hash operations. Notice that this is at most half the effort required to construct δ in CARDIAC. As shown in Section 4, this process is extremely efficient; notice that the verification of $\mathtt{Verify_C}$ and $\mathtt{Verify_M}$ can be made even more efficient if the client pre-computes and stores the open (zero) leaves in the tree. As the values of these nodes are independent of the file ID and the IDs of the clients, this precomputation could be performed once for a selection of trees of different heights.

Additional Operations.

Directories and other functionality: ClearBox hides the clients' directory structures from G by working on a single directory structure hosted within S's account on the cloud. This has the benefit of reducing the overhead borne by G (i.e., no path related overhead) and minimizes information leakage towards G.

Directory operations such as directory creation, directory renaming, etc. are locally handled by the software client of the users. Here, local directories contain pointers to the files stored therein and outsourced to the cloud; this enables the local client to perform operations such as directory listing and file renaming without the need to contact G—thereby minimizing the overhead incurred on G. Only operations that affect the client files stored on the cloud (e.g., file deletion/creation) are transmitted to G.

Other APIs: Recall that ClearBox leverages expiring URL-based PUT commands (exposed by Amazon S3 and Google Cloud Storage [3]) to enable clients to upload new objects directly to S; expiring URLs are also important in ClearBox to revoke data access to clients.

A number of commodity cloud service providers such as Dropbox, and Google drive, however, do not support URL commands for file creation, and only provide (non-expiring) URL-based file download. To integrate ClearBox with such commodity storage providers, we note the following differences to the protocol specification of ClearBox (cf. Section 3.3):

- At file upload time, the URL-based PUT is replaced by the clients uploading the file to G, which in turn uploads the file to S. Recall that G has to compute the Merkle tree over the uploaded file; this can be done asynchronously before G uploads the file to S—therefore reducing the performance penalty incurred on G.
- Files are stored under random identifiers, and can be accessed by means of permanent URLs which map to the file ID. When the user requests to delete a file, G renames the file to a new randomly selected ID. Other legitimate clients who require access to the file have to contact G who informs them of the new URL corresponding to the renamed file object.

In Section 4, we present a prototype implementation of ClearBox which interfaces with Dropbox, and we use it to evaluate the performance of this alternative technique.

Rate-Limiting: Similar to [14], we rate-limit requests to G to prevent possible brute-force search attacks on predictable file contents (in the assisted key generation phase), and to prevent resource exhaustion attacks by malicious clients. For this purpose, we limit the number of requests R_i that a client can perform in a given time frame T_i to a threshold θ_{max}.

3.4 Security Analysis

In this section, we address the security of ClearBox with respect to the model outlined in Section 2.

Secure Access: We start by showing that ClearBox ensures that a client who uploaded a file with Put will be able to recover the file with Get as long as he does not delete it. Since we assume that G and S will not tamper with the storage protocol, the threat to the soundness argument can only originate from other malicious clients. Moreover, since the procedures Get and Verify do not modify the stored data, we will focus the analysis on the operations Delete and Put.

Clearly, the Delete procedure only removes the access of the user requesting the deletion. Impersonation is not possible here since we assume a proper identity management by G. Namely, the gateway will only delete a file with ID FID if $|\mathcal{C}_{FID}| = 0$.

On the other hand, the Put procedure can only incur in the modification of data when a file f is uploaded for the first time. A subsequent upload of f is deduplicated and therefore does not entail any data modification at S. Notice that during initial upload, a malicious client can try to defect from the protocol, and construct an FID that does not fit to f, upload another file, or encrypt the file using the wrong key, etc. Recall that G verifies FID by downloading the uploaded file f^* to check whether $FID \stackrel{?}{=} MT_{\mathtt{Buf}(f^*)}$. Notice that if a malicious user creates a malformed f^*, that is $f^* \neq enc(\mathtt{H}(\mathtt{H}^*(f)^x), f)$ for any file f, then this will result into a random FID value which, with overwhelming probability, will not collide with any other file ID. That is, f^* would be stored by S without being deduplicated—which incurs storage costs on the malicious client (as he is fully charged for storing f^*) without affecting the remaining honest clients.

With respect to illegitimate client access, we distinguish between *(i)* the case that a client obtains ownership of a file when uploading the file for the first time with Put and *(ii)* the case where the client accesses the file with Get at some later point in time without having

the ownership in the file. For case *(i)*, uploading of a file that is not yet stored will not help the client. Namely, the client cannot pretend to upload a file with a different FID as G will check the integrity of the file. When a user requests to upload a file which has been stored already, the adopted proof of ownership scheme (PoW) (cf. Section 3.2.4 and Appendix B) ensures that the client must know the entire file, or the corresponding encoded data buffer (in the case of larger files) which is larger than an assumed leakage threshold. More details of the security of the PoW scheme can be found in Appendix B. In case *(ii)*, when a client requests access to f, G first checks if this client is still subscribed to f. Observe that this verification is handled internally by the gateway—without giving clients the ability to interfere with this process. If access is granted, the client gets a temporary URL which expires after a short time. Thus, this information cannot be re-used for further accesses; in particular, clients that are no longer subscribed to this file are not able to access the file any further.

Notice that also external adversaries cannot acquire access to f due to reliance on authenticated channels. Namely, we assume that all exchanged messages are appropriated signed, thus no repudiation of requests will be possible.

Secure Attestation: Let $\mathcal{C}_{FID}$ denote the set of clients that are storing file f.

Recall that f is referenced with $FID \leftarrow MT_{\texttt{Buf}(enc(k,f))}$ where $k \leftarrow \texttt{H}(\texttt{H}^*(f)^x)$ and `Buf` denotes the encoding used in PoW. The key k is deterministically computed from the hash $\texttt{H}^*(f)$ of the file according to our server-assisted encryption scheme. Due to the collision resistant hash function used throughout the process, FID is uniquely bound to f. Therefore, all clients that are registered to f are likewise expecting the same ID FID.

According to the work flow of ClearBox, each client $C_i \in \mathcal{C}_{FID}$ is informed about the size of $\mathcal{C}_{FID}$ referencing to the according file ID FID. Therefore, G first commits to the pair $FID, |\mathcal{C}_{FID}|$. Subsequently, G samples a subset of files for which it proves the correctness of the deduplication patterns using the accumulator. For these files the gateway G publishes the association between FID and the digest δ of the corresponding accumulator. This sampling is enforced by the GetRandomness function which acts as an unpredictably time-dependent source of randomness. This gives negligible advantage for G in learning the files that will be sampled when committing to $|\mathcal{C}_{FID}|$ at the end of the epoch. As the output of GetRandomness can be verified by the clients, each client $C_i \in \mathcal{C}_{FID}$ can check if G has reported δ for the corresponding FID.

Following from the security of CARDIAC (cf. Appendix A), G can only prove set membership and cardinality to its clients, if the underlying set and δ were computed correctly. Recall that δ constitutes a commitment to the set $\mathcal{C}_{FID}$. By publishing a list of associations (FID, δ) for the sampled file IDs, clients can ensure that G did not split $\mathcal{C}_{FID}$ into separate subgroups. We refer the readers to Appendix A for a security treatment of CARDIAC.

Data Confidentiality: As explained in Section 2, the knowledge of files size and user access patterns is essential for G and S to operate their business. Hence, hiding this information cannot be achieved in our solution. In the sequel, we analyze the confidentiality of the stored files in the presence of curious G and S.

Observe that both, the gateway G and the service provider S, only see the encrypted files, i.e., $f^* \leftarrow enc(k, f)$. Therefore, if the underlying encryption scheme is secure, the contents of the files are protected against any eavesdropper unless the key k is leaked. Our server-assisted encryption scheme is an instance of a message-locked encryption scheme [14]. As the key generation is oblivious, G does not have any additional advantage when compared to a standard message-locked encryption adversary. As such, similar to MLE schemes, ClearBox achieves indistinguishability for unpredictable files.

Notice that a curious gateway G may guess $\texttt{H}^*(f_i)$ for popular (predictable) content f_i and compute the corresponding key (as he knows the secret x) in order to identify if this f_i is stored by some clients. A client who stores a low-entropy confidential file, can protect against this attack, by appending a high-entropy string so that the file cannot be guessed anymore. However, the deduplication of the file is no longer possible. ClearBox offers a stronger protection towards any other entity who does not know the secret value x, e.g., the service provider S. Recall that G rate-limits client requests for encryption keys, to slow down brute-force search attacks on predictable file contents (via the interface of gateway).

4. DEPLOYMENT IN AMAZON S3 AND DROPBOX

In what follows, we evaluate a prototype implementation of ClearBox using Amazon S3 and Dropbox as a back-end storage.

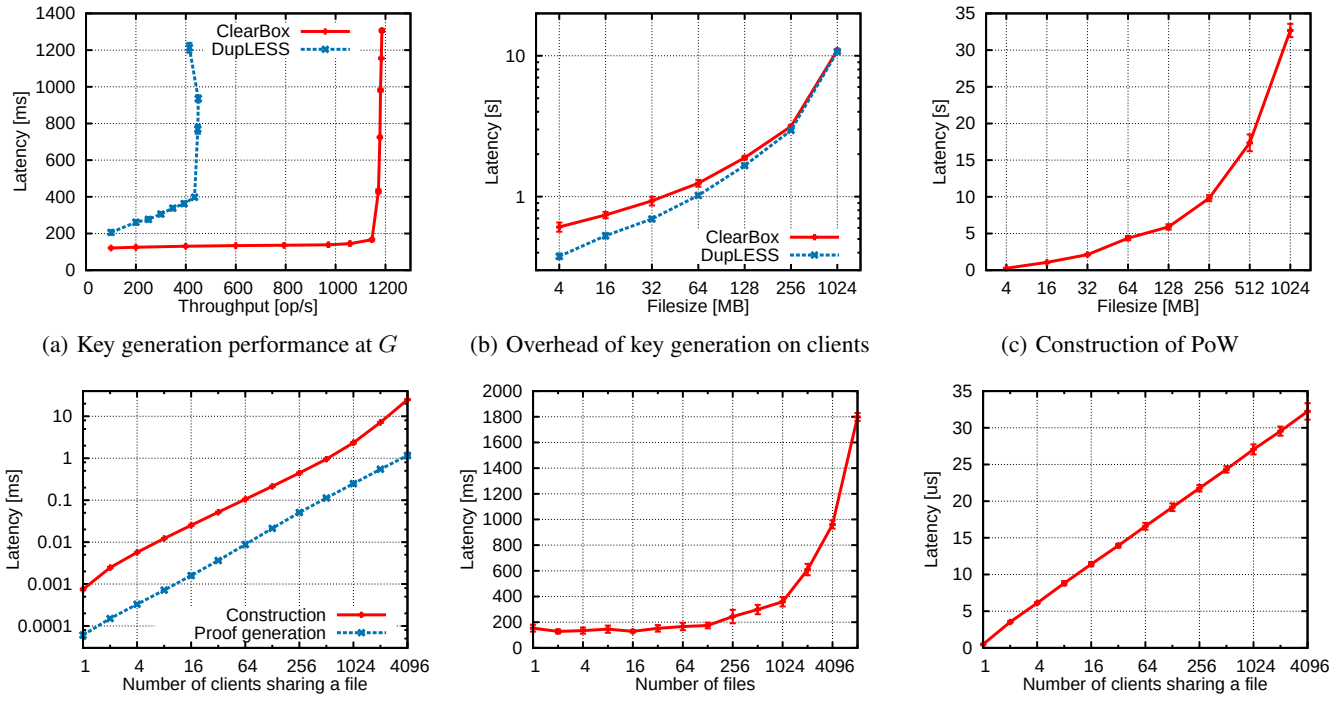
4.1 Implementation Setup

We implemented a prototype of ClearBox in Java. In our implementation, we relied on SHA-256, the Java built-in random number generator, and the JPBC library [7] (based on the PBC cryptographic library [5]) to implement BLS signatures. For a baseline comparison, we also implemented the server-based deduplication DupLESS of Bellare *et al.* [14] and integrated it with Amazon S3. Recall that, in DupLESS, clients directly interact with the cloud providers when storing/fetching their files. To generate keys, DupLESS uses a server-assisted oblivious protocol based on RSA blind signatures [14]. Notice that we did not implement a plain (unencrypted) cloud storage system since the performance of plain storage can be directly interpolated from the line rate (i.e., network capacity); where appropriate, we will discuss the performance of ClearBox relative to a plain cloud storage system.

We deployed our implementations on two dual 6-core Intel Xeon E5-2640 clocked at 2.50GHz with 32GB of RAM, and 100Mbps network interface card. The ClearBox gateway, and the assisting server of DupLESS were running on one dual 6-core Xeon E5-2640 machine, whereas the clients were co-located on the second dual 6-core Xeon E5-2640 machine; this ensures a fair comparison between ClearBox and DupLESS. To emulate a realistic Wide Area Network (WAN), we relied on NetEm [38] to shape all traffic exchanged on the networking interfaces following a Pareto distribution with a mean of 20 ms and a variance of 4 ms (which emulates the packet delay variance of WANs [24]).

Our implementation interfaces with both Amazon S3 and Dropbox (respectively), which are used to store the user files. To acquire Bitcoin block hashes, our clients invoke an HTTP request to a getblockhash tool offered by the Bitcoin block explorer[8] [2]. In our setup, each client invokes an operation in a closed loop, i.e., a client may have at most one pending operation. We spawned multiple threads on G's machine—each thread corresponding to a unique worker handling requests/bills of a given client. We bounded the maximum number of threads that can be spawned in parallel to 100. In our implementation, the gateway and clients store the metadata information associated to each file in a local MySQL database.

[8]For example, the hash of Bitcoin block 'X' can be acquired by invoking `https://blockexplorer.com/q/getblockhash/X`.



(a) Key generation performance at G

(b) Overhead of key generation on clients

(c) Construction of PoW

(d) CARDIAC construction and proof generation

(e) Latency in CARDIAC w.r.t. number of published files.

(f) CARDIAC verification by clients

Figure 4: Performance evaluation of the building blocks used in ClearBox with respect to a number of parameters.

The gateway leverages the caching feature of MySQL in order to reduce I/O costs. Recall that the Query Caching engine of MySQL [4] maps the text of the SELECT queries to their results. For each file, the gateway stores FID, the size of each file, and the IDs of the clients sharing the file.

Each data point in our plots is averaged over 10 independent measurements; where appropriate, we include the corresponding 95% confidence intervals. To accurately measure (micro-)latencies, we made use of the benchmarking tool due to Boyer [19].

4.2 Performance Evaluation

Before evaluating the overall performance of ClearBox, we start by analyzing the performance of the building blocks with respect to a number of parameters. Unless otherwise specified, we rely in our evaluation on the default parameters listed in Table 2.

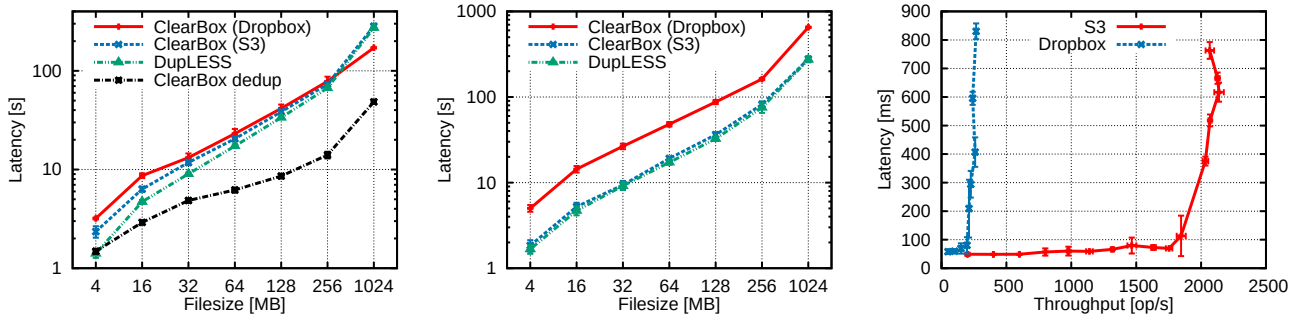
Gateway-assisted key generation: In Figure 4(a), we evaluate the overhead incurred by the oblivious key generation module (cf. Section 3.2.3) on the gateway. Here, we require that the gateway handles key generation requests back to back; we then gradually increase the number of requests in the system (until the throughput is saturated) and measure the associated latency. Our results show that our scheme incurs almost 125 ms latency per client key generation request on the gateway and attains a maximum throughput of 1185 operations per second; this considerably improves the maximum throughput of key generation of DupLESS (449 operations per second). This is mainly due to the fact that BLS signatures are considerably faster to compute by the gateway when compared to RSA signatures. In contrast, as shown in Figure 4(b), BLS signatures are more expensive to verify by the clients than the RSA-variant employed in DupLESS. However, we argue that the overhead introduced by our scheme compared to DupLESS can be easily tolerated by clients, as the client effort is dominated by hashing the file; for instance, for 16 MB files, our proposal only incurs an additional latency overhead of 213 ms on the clients when compared to DupLESS.

Parameter	Default Value
Default file size	16 MB
RSA modulus size	2048 bit
$\lvert\mathcal{C}_{FID}\rvert$	100
Num. of PoW challenges	50
Elliptic Curve (BLS)	PBC Library Curve F

Table 2: Default parameters used in evaluation.

Proofs of ownership: Figure 4(c) depicts the overhead incurred by FID required to instantiate the PoW scheme of [27] with respect to the file size. As explained in Appendix B, the PoW scheme of [27] reduces the costs of verifying PoW by encoding the original file in a bounded size buffer (64 MB in our case). Our results show that the computation of FID results in a tolerable overhead on both the gateway and the clients. For example, the computation of FID requires 1.07 seconds for a file with size 16 MB.

CARDIAC: In Figure 4(d), we evaluate the overhead incurred by the construction of the accumulators and the proof generation in CARDIAC with respect to $|\mathcal{C}_{FID}|$, i.e., the number of clients subscribed to the same file f. Our results show that the latency incurred by the construction of CARDIAC can be easily tolerated by G; for instance, the construction of an accumulator for 1000 users requires 2.34 ms. Clearly, this overhead increases as the number of users sharing the same file increases. Notice that once the entire Merkle tree is constructed, proof generation can be performed considerably faster than the construction of the accumulator. In this case, G only needs to traverse the tree, and record the sibling paths for all members of the accumulator.



(a) PUT Performance (b) GET Performance (c) Latency vs. throughput

Figure 5: Performance evaluation of ClearBox using Amazon S3 and Dropbox as back-end cloud storage.

In Figure 4(e), we evaluate the overhead incurred on G by the proof generation in CARDIAC with respect to the files selected for attestation at the end of each epoch. Our results show that this latency is around 150 ms when less than 100 files are selected, since the accumulators of these files can be handled in parallel by separate threads in our pool. When the number of selected files increases beyond our pool threshold (i.e., 100), the latency of proof generation increases e.g., to reach almost 1 second for 4000 files.

In Figure 4(f), we evaluate the overhead of the proof verification by the clients. Given that the verification only requires $\lceil \log |\mathcal{C}_{FID}| \rceil$ hashes, this operation only incurs marginal overhead on the clients. For example, the verification of membership and cardinality in CARDIAC when $|\mathcal{C}_{FID}| = 1000$ only requires 27 μs.

ClearBox: In Figure 5(a), we evaluate the latency witnessed by users in the PUT operation with respect to the file size. Our results show that ClearBox achieves comparable performance than DupLESS over S3. For example, when uploading 16 MB files on Amazon S3, DupLESS incurs a latency of 4.71 seconds, while ClearBox requires 6.33 seconds. The additional overhead in ClearBox mainly originates from the computation of FID by the users (cf. Figure 4(c)); the latency is mainly dominated by the upload of the file to Amazon.

In contrast, when users want to upload a file which is already stored on the cloud, ClearBox results in faster upload performance than DupLESS, since users are no longer required to upload the file, but have to execute the PoW protocol with G—which incurs negligible latency when compared to the upload of modest-sized files. Recall that this comes at the expense of additional load on G; in contrast, the server in DupLESS bears no load when users upload/download files.

When using Dropbox, recall that clients upload the file directly to G, which in turn uploads it to its Dropbox account (since Dropbox does not provide URL commands for file creation). Although this process requires less communication rounds between the clients and G (to acquire the signed URL), our results show that the latency incurred when uploading un-deduplicated files in ClearBox using Dropbox is marginally larger than its Amazon S3 counterpart; we believe that this discrepancy is due to the lower upload bandwidth between our clients and G when compared to Amazon S3. Notice that the performance of uploading deduplicated files in ClearBox does not depend on the back-end cloud provider and is therefore identical for both our Amazon S3 and Dropbox implementations.

In Figure 5(b), we evaluate the latency witnessed by users in the GET operation in ClearBox. Our results show that GET operation incurs comparable latencies in both ClearBox and DupLESS. Recall that in ClearBox, clients have to first contact G and acquire the timed GET URL to download resources. Given that the latency of the GET operation is dominated by the download speed, the overhead incurred by this additional communication round is marginal. Notice that the latency exhibited by ClearBox users is tolerable compared to that witnessed in a plain cloud storage system. For instance, assuming a 100 Mbps line rate, the download of a 32MB file in plain clouds requests almost 3 seconds; on the other hand, downloading this file in ClearBox incurs a latency of 10 seconds.

In Figure 5(c), we evaluate the latency incurred on the gateway in ClearBox with respect to the achieved throughput. Here, we assume that 50% of the requests handled by G correspond to PUT requests, while the remaining 50% are GET requests. We further assume that 50% of the upload requests correspond to files which are already stored at S. When uploading (un-deduplicated) files to S, we do not measure the overhead incurred on G when verifying FID since this verification is asynchronous and can be done by G at a later point in time. We further simulate a large download bandwidth at G by emulating client requests from a local socket. Our findings show that ClearBox achieves a maximum throughput of approximately 2138 operations per second when integrated with Amazon S3. This shows that our solution scales to a large number of users in the system. When interfacing with Dropbox, the performance of ClearBox deteriorates since the load on G considerably increases in this case;[9] the maximum throughput exhibited by our scheme decreases to almost 289 operations per second.

5. RELATED WORK

Data deduplication in cloud storage systems has acquired considerable attention in the literature.

In [28], Harnik *et al.* describe a number of threats posed by client-side data deduplication, in which an adversary can learn if a file is already stored in a particular cloud by guessing the hashes of predictable messages. This leakage can be countered using Proofs of Ownership schemes (PoW) [23, 27], which enable a client to prove it possesses the file in its entirety. PoW are inspired by Proofs of Retrievability and Data Possession (POR/PDP) schemes [11, 40], with the difference that PoW do not have a pre-processing step at setup time. Halevi *et al.* [27] propose a PoW construct based on Merkle trees which incurs low overhead on the server in constructing and verifying PoW. Xu *et al.* [44] build upon the PoW of [27] to construct a PoW scheme that supports client-side deduplication in a bounded leakage setting. Di Pietro and Sorniotti [23] propose a

[9] For comparison purposes, we did not include in our measurements the overhead introduced by the uploading of files by G onto S; this process can be executed at a later point in time by G.

PoW scheme which reduces the communication complexity of [27] at the expense of additional server computational overhead. Blasco *et al.* [16] propose a PoW based on Bloom filters which further reduces the server-side overhead of [23].

Douceur *et al.* [25] introduced the notion of convergent encryption, a type of deterministic encryption in which a message is encrypted using a key derived from the plaintext itself. Convergent encryption is not semantically secure [15] and only offers confidentiality for messages whose content is unpredictable. Bellare *et al.* [31] proposed DupLESS, a server-aided encryption to perform data deduplication scheme; here, the encryption key is obliviously computed based on the hash of the file and the private key of the assisting server. In [42], Stanek *et al.* propose an encryption scheme which guarantees semantic security for unpopular data and weaker security (using convergent encryption) for popular files. In [41], Soriente *et al.* proposed a solution which distributively enforces shared ownership in agnostic clouds. This solution can be used in conjunction with ClearBox to enable users to distributively manage the access control of their deduplicated files.

Several proposals for accumulating hidden sets exist such as the original RSA-based construction [12], which has been extended to dynamic accumulators [21], and non-membership proofs [32]. There exist as well constructions based on bilinear groups [22, 39] and on hash-functions [20, 33]. A related concept are zero-knowledge sets [30, 37]. These structures provide proofs of set membership. Notice, however, that cryptographic accumulators do not typically provide information about the accumulated set, such as content or cardinality.

Our attacker model shares similarities with the one considered in [43], where the cloud provider is assumed to be economically rational. This scheme relies on an hourglass function—to verify that the cloud provider stores the files in encrypted form—which imposes significant constraints on a rational cloud provider that tries to apply the hourglass functions on demand.

6. CONCLUSION

In this paper, we proposed ClearBox, which enables a cloud provider to transparently attest to its clients the deduplication patterns of their stored data. ClearBox additionally enforces fine-grained access control over deduplicated files, supports data confidentiality, and resists against malicious users. Evaluation results derived from a prototype implementation of ClearBox show that our proposal scales well with the number of users and files in the system.

As far as we are aware, ClearBox is the first complete system which enables users to verify the storage savings exhibited by their data. We argue that ClearBox motivates a novel cloud pricing model which promises a fairer allocation of storage costs amongst users—without compromising data confidentiality nor system performance. We believe that such a model provides strong incentives for users to store popular data in the cloud (since popular data will be cheaper to store) and discourages the upload of personal and unique content. As a by-product, the popularity of files additionally gives an indication to cloud users on their level of privacy in the cloud; for example, the user can verify that his private files are not deduplicated—and thus have not been leaked.

Acknowledgements

The authors would like to thank Nikos Triandopoulos and the anonymous reviewers for their valuable feedback and comments. This work was partly supported by the EU H2020 TREDISEC project, funded by the European Commission under grant agreement no. 644412.

References

[1] Amazon S3 Pricing. `http://aws.amazon.com/s3/pricing/`.

[2] Bitcoin real-time stats and tools. `http://blockexplorer.com/q`.

[3] Google Cloud Storage. `https://cloud.google.com/storage/`.

[4] The MySQL Query Cache. `http://dev.mysql.com/doc/refman/5.1/en/query-cache.html`.

[5] PBC Library. `http://crypto.stanford.edu/pbc/`, 2007.

[6] Cloud Market Will More Than Triple by 2014, Reaching $150 Billion. `http://www.msptoday.com/topics/msp-today/articles/364312-cloud-market-will-more-than-triple-2014-reaching.htm`, 2013.

[7] JPBC:Java Pairing-Based Cryptography Library. `http://gas.dia.unisa.it/projects/jpbc/#.U3HBFfna5cY`, 2013.

[8] Bitcoin as a public source of randomness. `https://docs.google.com/presentation/d/1VWHm4Moza2znhXSOJ8FacfNK2B_vxnfbdZgC5EpeXFE/view?pli=1#slide=id.g3934beb89_034`, 2014.

[9] These are the cheapest cloud storage providers right now. `http://qz.com/256824/these-are-the-cheapest-cloud-storage-providers-right-now/`, 2014.

[10] F. Armknecht, J. Bohli, G. O. Karame, Z. Liu, and C. A. Reuter. Outsourced proofs of retrievability. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 831–843, 2014.

[11] G. Ateniese, R. C. Burns, R. Curtmola, J. Herring, L. Kissner, Z. N. J. Peterson, and D. X. Song. Provable data possession at untrusted stores. In *ACM Conference on Computer and Communications Security*, pages 598–609, 2007.

[12] N. Baric and B. Pfitzmann. Collision-free accumulators and fail-stop signature schemes without trees. In W. Fumy, editor, *EUROCRYPT*, volume 1233 of *Lecture Notes in Computer Science*, pages 480–494. Springer, 1997.

[13] M. Bellare and S. Keelveedhi. Interactive message-locked encryption and secure deduplication. In J. Katz, editor, *Public-Key Cryptography - PKC 2015 - 18th IACR International Conference on Practice and Theory in Public-Key Cryptography, Gaithersburg, MD, USA, March 30 - April 1, 2015, Proceedings*, volume 9020 of *Lecture Notes in Computer Science*, pages 516–538. Springer, 2015.

[14] M. Bellare, S. Keelveedhi, and T. Ristenpart. DupLESS: Server-aided encryption for deduplicated storage. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 179–194, Berkeley, CA, USA, 2013. USENIX Association.

[15] M. Bellare, S. Keelveedhi, and T. Ristenpart. Message-locked encryption and secure deduplication. In T. Johansson and P. Q. Nguyen, editors, *Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings*, volume 7881 of *Lecture Notes in Computer Science*, pages 296–312. Springer, 2013.

[16] J. Blasco, R. Di Pietro, A. Orfila, and A. Sorniotti. A tunable proof of ownership scheme for deduplication using bloom filters. In *Communications and Network Security (CNS), 2014 IEEE Conference on*, pages 481–489, Oct 2014.

[17] A. Boldyreva. Efficient threshold signature, multisignature and blind signature schemes based on the gap-diffie-hellman-group signature scheme. 2002.

[18] D. Boneh, B. Lynn, and H. Shacham. Short signatures from the weil pairing. *J. Cryptology*, 17(4):297–319, 2004.

[19] Brent Boyer. Robust Java benchmarking. `http://www.ibm.com/developerworks/library/j-benchmark2/j\-benchmark2\-pdf.pdf`.

[20] A. Buldas, P. Laud, and H. Lipmaa. Eliminating counterevidence with applications to accountable certificate management. *Journal of Computer Security*, 10(3):273–296, 2002.

[21] J. Camenisch and A. Lysyanskaya. Dynamic accumulators and application to efficient revocation of anonymous credentials. In *Advances in Cryptology - CRYPTO 2002*, pages 61–76. Springer, 2002.

[22] I. Damgård and N. Triandopoulos. Supporting non-membership proofs with bilinear-map accumulators. *IACR Cryptology ePrint Archive*, 2008:538, 2008.

[23] R. Di Pietro and A. Sorniotti. Boosting efficiency and security in proof of ownership for deduplication. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '12, pages 81–82, New York, NY, USA, 2012. ACM.

[24] D. Dobre, G. Karame, W. Li, M. Majuntke, N. Suri, and M. Vukolić. Powerstore: Proofs of writing for efficient and robust storage. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 285–298, New York, NY, USA, 2013. ACM.

[25] J. R. Douceur, A. Adya, W. J. Bolosky, D. Simon, and M. Theimer. Reclaiming space from duplicate files in a serverless distributed file system. In *ICDCS*, pages 617–624, 2002.

[26] A. Fiat and A. Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Proceedings on Advances in cryptology—CRYPTO '86*, pages 186–194, London, UK, UK, 1987. Springer-Verlag.

[27] S. Halevi, D. Harnik, B. Pinkas, and A. Shulman-Peleg. Proofs of ownership in remote storage systems. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 491–500, New York, NY, USA, 2011. ACM.

[28] D. Harnik, B. Pinkas, and A. Shulman-Peleg. Side channels in cloud services: Deduplication in cloud storage. *IEEE Security & Privacy*, 8(6):40–47, 2010.

[29] G. O. Karame, E. Androulaki, and S. Capkun. Double-spending fast payments in bitcoin. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, pages 906–917, New York, NY, USA, 2012. ACM.

[30] A. Kate, G. M. Zaverucha, and I. Goldberg. Constant-size commitments to polynomials and their applications. In *Advances in Cryptology-ASIACRYPT 2010*, pages 177–194. Springer, 2010.

[31] 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, Washington, D.C., 2013. USENIX.

[32] J. Li, N. Li, and R. Xue. Universal accumulators with efficient nonmembership proofs. In *Applied Cryptography and Network Security, 5th International Conference, ACNS 2007, Zhuhai, China, June 5-8, 2007, Proceedings*, pages 253–269, 2007.

[33] H. Lipmaa. Secure accumulators from euclidean rings without trusted setup. In *Applied Cryptography and Network Security - 10th International Conference, ACNS 2012, Singapore, June 26-29, 2012. Proceedings*, pages 224–240, 2012.

[34] S. Liu, X. Huang, H. Fu, and G. Yang. Understanding data characteristics and access patterns in a cloud storage system. In *13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, CCGrid 2013, Delft, Netherlands, May 13-16, 2013*, pages 327–334, 2013.

[35] 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.

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

[37] S. Micali, M. Rabin, and J. Kilian. Zero-knowledge sets. In *Foundations of Computer Science, 2003. Proceedings. 44th Annual IEEE Symposium on*, pages 80–91. IEEE, 2003.

[38] NetEm. NetEm, the Linux Foundation. Website, 2009. Available online at `http://www.linuxfoundation.org/collaborate/workgroups/networking/netem`.

[39] L. Nguyen. Accumulators from bilinear pairings and applications. In *Topics in Cryptology - CT-RSA 2005, The Cryptographers' Track at the RSA Conference 2005, San Francisco, CA, USA, February 14-18, 2005, Proceedings*, pages 275–292, 2005.

[40] H. Shacham and B. Waters. Compact Proofs of Retrievability. In *ASIACRYPT*, pages 90–107, 2008.

[41] C. Soriente, G. O. Karame, H. Ritzdorf, S. Marinovic, and S. Capkun. Commune: Shared ownership in an agnostic cloud. In *Proceedings of the 20th ACM Symposium on Access Control Models and Technologies, Vienna, Austria, June 1-3, 2015*, pages 39–50, 2015.

[42] J. Stanek, A. Sorniotti, E. Androulaki, and L. Kencl. A secure data deduplication scheme for cloud storage. In *Financial Cryptography and Data Security - 18th International Conference, FC 2014, Christ Church, Barbados, March 3-7, 2014, Revised Selected Papers*, pages 99–118, 2014.

[43] 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.

[44] J. Xu, E.-C. Chang, and J. Zhou. Weak leakage-resilient client-side deduplication of encrypted data in cloud storage. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ASIA CCS '13, pages 195–206, New York, NY, USA, 2013. ACM.

APPENDIX

A. SECURITY ANALYSIS OF CARDIAC

In what follows, we analyze the security of CARDIAC. Let a set X be given and the accumulated digest for the set is $\delta \leftarrow \mathtt{Acc}(X)$. Recall that $\delta = \mathtt{H}(a_{\ell,0}, \ell)$ (with $\mathtt{H}$ being a cryptographically secure hash function) acts as a commitment to the root $a_{\ell,0}$ and the height ℓ of a Merkle tree. For the purpose of our analysis, we distinguish between two security goals: *(i)* proving membership using $\mathtt{Prove}_M$ and $\mathtt{Verify}_\mathtt{M}$ and *(ii)* proving an upper bound on $|X|$ using $\mathtt{Prove}_C$ and $\mathtt{Verify}_\mathtt{C}$. As the security of Merkle trees is well understood with respect to the first goal [20], we focus in the sequel on analyzing the second goal.

Since ℓ is verified (indirectly) using the length of the sibling path and assuming that the hash function is secure, CARDIAC ensures that the Merkle tree can encode at most 2^ℓ elements. $\mathtt{Prove}_C$ outputs the sibling path of the last set element $a_{0,|X|-1}$. The verification algorithm $\mathtt{Verify}_\mathtt{C}$ requires to check the values of all open nodes in the sibling path. Recall that the open nodes are those nodes that depend only on the zero leaves and hence represent publicly known values. In fact, this verification step constitutes membership proofs for every single zero leaf $a_{0,j}$ for $j = |X|, \ldots, 2^\ell - 1$. In turn, this ensures that at least $2^\ell - |X|$ leaves of the tree are zero.

A direct consequence is that at most $|X|$ leaves can be non-zero, giving an upper bound on X. Observe that the proof assumes that further leaves located on the left of the last non-zero element are not set to zero. This assumption can be safely made since G has no incentives to place more zero-leaves (since this means that less users are registered than claimed and would pay too little to G).

B. POW BASED ON [24]

In Figure 6, we detail the Proof of Ownership (PoW) due to Halevi *et al.* [27]. The protocol is divided into three phases: in the first phase, the file f is reduced into a buffer with maximum 64 MB in size. Then, the contents of the buffer are pseudo-randomly mixed and finally, the Merkle tree of the buffer is computed. The first two phases can be seen as applying a linear code to the file, and

Input: A file f of length M bit, broken into $m = \lceil \frac{M}{512} \rceil$ blocks of length 512 bit.

Initialize positions:

- Compute bit-length $\ell = \min\{20, \lceil \log_2 m \rceil\}$, being close to M, but at most 64 MB, i.e. 2^{20} blocks with 512 bits.
- Initialize the buffer `Buf` with 2^ℓ blocks of 512 bits.
- Initialize a table ptr of m rows and 4 columns of ℓ bits which holds pointers into the buffer `Buf`. A temporary value IV will be initialized with SHA-256' IV.

Reduction phase:
For each $i \in [m]$:

1. Update $IV := \text{SHA-256}(IV; File[i])$, where $File[i]$ denotes the i-th block of the file.
   Initialize $ptr[i] = \text{trunc}_{4\ell}(IV)$ with IV truncated to 4ℓ bits..
2. For $j = 0:3$, do
3. $\quad Block =$ Cyclic Shift of $File[i]$ by $j * 128$ bits
4. $\quad$ XOR $Block$ into location $ptr[i][j]$ in `Buf`

Mixing phase:
Repeat 5 times:

1. For each block $i \in [\ell]$, For $j = 0:3$, do
2. $\quad Block =$ Cyclic Shift of `Buf`$[i]$ by $j * 128$ bits
3. $\quad$ If $i \neq ptr[i \mod 2^\ell][j]$ then XOR $Block$ into location $ptr[i \mod 2^\ell][j]$ in `Buf`

Tree construction: The final buffer content is denoted by `Buf`(f). Construct a Merkle tree using the blocks of `Buf`(f) as the leaves.

Figure 6: Detailed PoW Protocol of [27].

outputting a buffer `Buf`(f). Subsequently, to verify a PoW, the verifier asks for the sibling paths of a random number of leaves from `Buf`(f) and checks that the authentication path matches the root of the Merkle tree.

The buffer is an encoding of the file with a random linear code. The code is generated using SHA-256 to generate an array of pointer ptr. Due to the collision resistance of the hash function, any modification of the file will therefore lead to a different code. Under the assumption that the code has a minimum distance of $L/3$, where L is the number of blocks in the buffer, any two valid buffers will be different in at least $L/3$ positions. Thus, the prover will only succeed with a probability of $(2/3)^t$ in answering t challenges without knowing the correct buffer. For example, when $t = 20$ challenges are made, the probability of success is at most 2^{-12}. Note that without knowing the file in its entirety, two different random codes are derived in the first two steps due to the collision-resistance of the deployed hash function. In particular, it is highly unlikely that any block in the buffer can be predicted. We refer the readers to [27] for more details on the security of this construct.