Auditable Compressed Storage

Iraklis Leontiadis[1] and Reza Curtmola[2]

[1] Inpher, USA and Switzerland (iraklis@inpher.io)
[2] New Jersey Institute of Technology, Newark, NJ, USA (reza.curtmola@njit.edu)

Abstract. Outsourcing data to the cloud for personal use is becoming an everyday trend rather than an extreme scenario. The frequent outsourcing of data increases the possible attack window because users do not fully control their personal files. Typically, once there are established secure channels between two endpoints, communication is considered secure. However, in the cloud model the receiver–the cloud–cannot be fully trusted, either because it has been under adversarial control, or because it acts maliciously to increase its revenue by deleting infrequent accessed file blocks. One approach used by current literature to address the aforementioned security concerns is via Remote Data Integrity Checking (RDIC) protocols, whereby a data owner can challenge an untrusted cloud service provider (CSP) to prove faithful storage of its data.
Current RDIC protocols assume that the original data format remains unchanged. However, users may wish to compress their data in order to enjoy less charges. In that case, current RDIC protocols become impractical because, each time compression happens on a file, the user has to run a new RDIC protocol. In this work we initiate the study for *Auditable Compressed Storage* (ACS). After defining the new model we instantiate two protocols for different widely used compression techniques: run length encoding and Huffman encoding. In contrast with conventional RDIC, our protocols allow a user to delegate the compression to the cloud in a provably secure way: The client can verify correctness of compression without having to download the entire uncompressed file and check it against the compressed one.

1 Introduction

The proliferation of information available to individuals, companies and institutions in conjunction with the adoption of the cloud as the *de facto* outsourcing service, drives the delegation of data storage to third party cloud services. As the cloud may misbehave by not storing data at their entire form or tampering with it, new mechanisms for remote data integrity checking (RDIC) are vital. It is almost a decade since the first protocols for RDIC paved the way for secure outsourced storage: *Provable Data Possession* (PDP) (2; 1) and *Proofs of Retrievability* (POR) (9; 11).

Despite the remarkable scientific literature impact of remote data integrity checking protocols (Google Scholar reports $\approx 6,300$ citations for PDP (2; 1) and POR (9; 11) at the time of this writing), there is a common restrictive setting

under which current RDIC protocols operate: A cloud service provider (CSP) can only provide proofs of data possession for the original data file format that was initially uploaded. This model can abstract the procedure of outsourcing storage of archival data in general. However, specific use case scenarios require the transformation of raw data to a different format at the cloud, e.g: *compressing* the original data. As the current RDIC protocols do not allow for such versatility, neither the CSP can take advantage of current *compression* techniques to reduce its storage space, nor the user can demand compression of the original data without skyrocketing egress costs: The need to download the uncompressed file, compress it and upload both the new compressed file and the verification metadata for the RDIC protocol increases the communication cost and the charges subsequently.

Motivating Scenario: We consider an online storage service equipped with backup functionality. At first, the user uploads data with the corresponding verification metadata. The user engages with the CSP in a challenge-response protocol part of the RDIC scheme, in order to attest intact storage of its data. Periodically, at the end of fixed time slots, the CSP compresses the original files into backup files. The purpose is to allow the user retrieve old versions of its data and provide also reliability in case of a catastrophic attack of the current file. The user needs to attest integrity of the compressed file as well, but this is now impossible based on the current verification metadata which is computed over the original uncompressed data.

In this paper, we seek to design and analyze protocols for *Accountable Compressed Storage* (ACS). Such protocols will allow an honest user who uploads its data to an economically motivated CSP—with adversarial behavior—to attest faithful and intact storage of the compressed version of the data. The CSP is motivated to proceed with a wrong compression in order to maximize its profit. For example, the CSP does not compress the data optimally in order to charge for more storage; or, it stores an incorrect (*i.e.*, smaller) compressed version of the data, in order to save on storage. One way to overcome this adversarial behavior is to transfer the task of compression to the user, who then engages in a new RDIC protocol with the CSP based on the compressed data. However, this results in increased communication costs and, ultimately, higher charges for the user: The user has to download the uncompressed file and upload the compressed version thereof. Thus, to accomplish our goal, we need to address two challenges:

Challenge 1: *Reduce communication overhead during compression*: As the original uncompressed data is deleted locally at the user side and rests (ostensibly) at the cloud side, whenever data needs to be compressed, the user can retrieve this data, verify its integrity, compress the data and compute the new verification metadata, and upload the compressed data with the corresponding metadata to the CSP. This solution incurs high communication costs, as the user has to download the entire uncompressed data.

Challenge 2: *Enable tag versatility according to compression without sacrificing security.* To avoid the increased communication costs, a solution could be to delegate the compression to the cloud service provider. However, as the CSP

holds only verification metadata for the uncompressed data, the user would not be able to verify the integrity of the compressed data. The CSP could forward the compressed data to the user, who can then compute the new verification metadata, but the user does not hold the original data to check the correctness of compression. Retrieving the original uncompressed data to verify compression correctness would also incur increased communication costs.

For the aforementioned reasons, delegating the compression of the data and the computation of the new verification metadata to the CSP without violating the security goals, renders the design of an *Accountable Compressed Storage* protocol challenging.

Contributions. In this work, we make the following contributions. We first introduce the framework of *Auditable Compressed Storage* (ACS). ACS allows the user to delegate the compression of presumably stored data to an untrusted CSP. The CSP , however, may not store the correctly compressed data in order to maximize its profits. In contrast with previous work on RDIC protocols, ACS expands the model with **1)** an extra challenge-response protocol whereby the CSP proves to the user not only that it faithfully stores the compressed version of the original data, but also that the compression is correct, and **2)** a tag transformation procedure, which allows the untrusted CSP to compute the new verification metadata for the compressed data without holding secret key information. Second, we design two protocols for *Auditable Compressed Storage*: ACS-RLE for run-length-encoding compression and ACS-HUFF for Huffman encoding. We analyze their security in a provable way following the novel security model for ACS schemes and we make a thorough efficiency analysis thereof.

2 Background

2.1 Compression Algorithms

Compression reduces the original size of a transmitted file in order to save storage when saving it and also reduces time and costs when transmitting it. Below we elaborate on two popular compression techniques: Run Length Encoding and Dictionary-based Encoding with Huffman Prefix trees.

2.1.1 Run Length Encoding (RLE)

RLE compresses a stream of data F composed of symbols b from some alphabet $b \in \mathcal{S}$ with a compact representation thereof. The compact representation consists of tuples of the form $(b : \mathsf{frq})$: where frq is the frequency of each symbol b in file F. RLE exploits the redundancy that occurs over the symbols of the stream in order to compactly encode the stream in a format that can decompress in a lossless manner the compressed stream. Obviously the higher the redundancy of the stream the higher the compression ratio. Redundancy is considered as a consecutive subset of equal symbols inside the original stream F. As an example, $\mathsf{F} =$ "$AAAAAAGGGGTTTTTTTCCCCCCEF$" then $\mathsf{F}' = (A:6, G:4, T:7, C:5, E:1, F:1)$.

2.1.2 Dictionary-based Encoding (DBE)

With DBE, variable length frequent patterns of symbols are replaced with shorter codewords c from a code alphabet $c \in \Phi$. The mapping is implemented using a dictionary $D := (b, \mathsf{frq}, c)$ where c corresponds to a binary codeword for symbol $b \in S$ and frq is the frequency of b in F. Hereafter, we write D_i to refer to the i^{th} row and $D_i[b], D_i[\mathsf{frq}], D_i[c]$ to refer to the i^{th} symbol, frequency or code, correspondingly. With $D_i[b]_k$, in case of a block with multiple symbols we refer to the k symbol of block b in row i. For the implementation of the dictionary the invariant is: the more frequent symbols or patterns are assigned to shorter codes. Huffman tree prefix encoding avoids the ambiguity of codewords which share common prefixes for different symbols by building a prefix binary tree. Leaves of the tree correspond to the symbols/patterns, left path nodes are assigned the 0 value and right side nodes the 1 value. To ensure the invariant of DBE, less frequent items are placed at the lowest level of the tree. The order of elements in a level does not matter for the correctness of the prefix tree encoding. Huffman (8) proposed a bottom-up approach to recursively build the tree. The algorithm starts by picking the two less frequent symbols b_1, b_2. Assign them to left-0 and right-1 leaf and put them under the parent of the meta symbol $b_1||b_2$. Remove from the original data stream b_1, b_2, add $b_1||b_2$ and recompute the frequencies. The process is repeated until $|\mathsf{F}| = 2$, where the algorithm returns the left and the right subtree of the root node. The codeword of each symbol is the path to its leaf. The DBE outputs the dictionary D and the encoded stream F' according to D. For the stream $\mathsf{F} =$ "$AAAAAAGGGGTTTTTTTCCCCCCEF$" the Huffman Prefix Tree T is shown in Figure 3. The encoding dictionary D is shown in Table 4 and the compressed stream (in binary) is $\mathsf{F}' = 10101010101001101101101111111111111111000000000001000101$. The size of the compressed stream is $6 \cdot 2 + 4 \cdot 3 + 2 \cdot 7 + 2 \cdot 5 + 2 \cdot 4 = 56$ bits, while the uncompressed stream has size $24 \cdot 8 = 192$ bits, assuming symbols are 8 bits long. Thus compression ratio equals $3.42 : 1$.

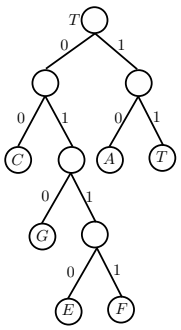


Fig. 1: Huffman Encoding Prefix Tree.

Symbol	Frequency	Codeword
T	7	11
A	6	10
C	5	00
G	4	011
E	1	0100
F	1	0101

Table 1: Dictionary Huffman Encoding

2.2 Related Work

2.2.1 Remote Data Integrity Remote Data Integrity Checking (RDIC) (2; 9; 11), allows data owners to efficiently audit the integrity of their outsourced data stored at an untrusted CSP. During a Setup phase, the user preprocesses its file F and computes verification metadata σ using some secret key information. User $\mathcal{U}$ then uploads F and σ to the CSP and deletes them from her local storage. $\mathcal{U}$ keeps in storage a small, constant, amount of information related to her secret key. At a later point in time, the data owner challenges the CSP to prove possession of a randomly chosen subset of file blocks. The CSP can only compute a valid proof as long as it stores the challenged file blocks with the corresponding verification metadata. Finally, the owner checks the correctness of the proof and is convinced about the faithful storage of the file F by the CSP when the proof is successfully verified.

2.2.2 Dynamic Remote Data Integrity RDIC was initially designed to handle static datasets. To handle dynamic data that changes over time (*i.e.*, data blocks are altered, new blocks are inserted and existing blocks are deleted), dynamic RDIC protocols have been proposed.

Dynamic PDP. Ateniese *et al.* (3) introduced the notion of scalable provable data possession. Their solution allows deletions, changes and appends of new data blocks but not insertions. Erway *et al.* (5) addressed the dynamicity of existing PDP schemes with an authenticated skip list, which keeps track of ranking information. The proposed solution enables insertions of blocks as well without the need to recompute the tags of the existing blocks. Wang *et al.* (14) gave a solution for dynamic PDP based on Merkle trees. Other solutions with extra properties (replication and transparency(7), variable block size (6), multiple access to shared data (15)) have been presented in the literature.

Dynamic POR. Stefanov *et al.* (13) proposed *Iris*, a PoR scheme for authentic outsourced storage, which supports dynamic datasets. The client has to locally store erasure-code data to allow edits on the uploaded file. Shi *et al.* (12) used a combination of erasure encoded log buffers with authenticated data structures to achieve dynamicity of data for a proof of retrievability protocol. In contrast with previous solutions Cast *et al.* (4) employed an oblivious RAM scheme to support dynamic data and Etemad *et al.* (10) presented generic constructions for DPoR schemes.

In contrast with all dynamic RDIC protocols, whereby some blocks of data are inserted, deleted, or changed, our *auditable compressed storage* framework requires a different approach. The compressed file is treated as a completely new file consisting of different blocks. Thus, employing a dynamic RDIC scheme is not suitable, because the blocks of the compressed file are different compared to those in the original file.

3 Model and Security Guarantees

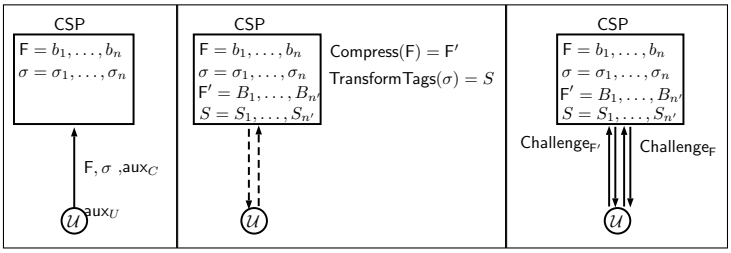
In this section, we describe the model for an *Auditable Compressed Storage* (ACS) system. First, we set up the environmental setting with the functional requirements of an ACS. Next, we analyze the adversarial model and we present the sufficient security guarantees for an ACS scheme.

3.1 System Model

A user $\mathcal{U}$ uploads a file F to an untrusted cloud service provider CSP. The file F consists of n blocks. Each file block consists of 8-bit ω symbols, and is thus $w = 8 \cdot \omega$ bits long. As the CSP may act maliciously, in order to guarantee storage integrity for the file F, $\mathcal{U}$ uploads verification metadata for each block, computed with a secret key. This metadata allows $\mathcal{U}$ to get guarantees about faithful storage by running an RDIC protocol such as PDP (2), POR (9; 11).

At a later point in time, the CSP compresses F into F' using a compression algorithm ($\mathsf{Comp} = \mathsf{Encode}, \mathsf{Decode}$). $\mathcal{U}$ needs to check the integrity of the compressed file F' and to get assurances that **1)** $\mathsf{Decode}(\mathsf{Encode}(\mathsf{F}')) = \mathsf{F}$, where F' is the correct compression encoding of the original uploaded file F, and **2)** the compressed file F' is faithfully stored. ACS introduces a Compress phase whereby the CSP compresses F to F' and transforms the tags of the uncompressed data blocks to correspond to the compressed blocks. $\mathcal{U}$ is also exposed to the traditional RDIC algorithms: $\mathsf{TagFile}, \mathsf{Challenge}_{\mathsf{F}}, \mathsf{Prove}, \mathsf{Verify}$, whereby $\mathcal{U}$ computes authentication tags on top of each file block $b_1, \ldots, b_n$ and later on during a challenge response protocol, $\mathcal{U}$ challenges the CSP on a random subset of blocks. The CSP as long as it faithfully stored the blocks of F proves to $\mathcal{U}$ that it stores F and $\mathcal{U}$ verifies the proof. In contrast with a traditional RDIC protocol, in ACS during the $\mathsf{Challenge}_{\mathsf{F}'}$ phase the CSP demonstrates that **1)** F' is the correct compression encoding version of F and **2)** it faithfully stores F' in its entire form. We can describe an ACS system in 4 phases (also illustrated in Figure 5):

- Setup: The system parameters params are initialized and user $\mathcal{U}$ chooses secret key information sk. $\mathcal{U}$ according to params splits the file in n blocks: $\mathsf{F} = b_1, b_2, b_3, \ldots, b_n$ and computes auxiliary information aux related to the compression of the file. Some of aux may be stored at the user (aux_U), whereas some of it may be sent to the CSP (aux_C). For each block b_i, it calls TagFile, which computes an authentication tag σ_i. Finally, $\mathcal{U}$ uploads F, $\{\sigma_i\}_{i=1}^{n}$, and aux_C to the CSP.
- Compress: The CSP compresses F to F' using the Encode algorithm. Note that F' has n' blocks, where n' may normally be different than n. The CSP then transforms the tags $\sigma_1, \ldots, \sigma_n$ of the uncompressed file blocks into the ones for the compressed version of the file, $S_1, \ldots, S_{n'}$.
- $\mathsf{Challenge}_{\mathsf{F}'}$: The user engages in a challenge-response protocol to verify that F' is a correct compression of F and to attest the faithful storage of F' at the CSP.
- $\mathsf{Challenge}_{\mathsf{F}}$: The user engages in a challenge-response protocol to attest the faithful storage of F at the CSP.



(a) Setup **(b)** Compress **(c)** $\mathsf{Challenge}_{\mathsf{F}'}$

Fig. 2: ACS model. Interrupted lines during the Compress phase denote that the user may not need to interact with the CSP.

3.2 Adversarial Model

We assume an adversary $\mathcal{A}$ controlling the CSP to be a rational player in the protocol. That is, it will deviate from the protocol as long as it has some economic incentive to proceed in a malicious behavior. An ACS adversary $\mathcal{A}$ can misbehave as follows:

AM1 (Incorrect compression): The CSP may claim storage of a file F'', which does not correspond to the compressed version F' of F. The CSP acting rationally has the economical motivation to compute and store F'' such that the size of of F'' is either smaller (to save on storage costs) or larger (to charge the user for more storage) than the size of F'. A rational CSP can potentially misbehave in the following two ways when compressing or transforming the tags: **AT1)** reduce the frequency of a block (cf. Figure 6), or **AT2)** increase the frequency of a block (cf. Figure 7).

AM2 (Compression Integrity): As in traditional RDIC protocols, the CSP may discard rarely accessed blocks of the compressed file F' or it may try to hide data loss incidents to maintain its reputation.

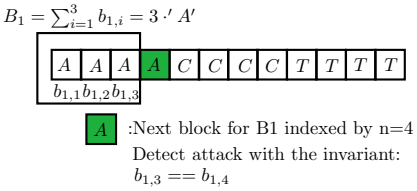


Fig. 3: Remove block attack.

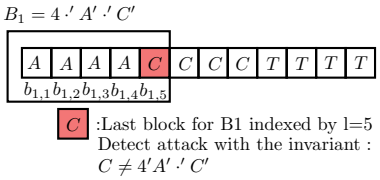


Fig. 4: Add block attack.

3.3 Security Guarantees

We seek to design an ACS system with the following security guarantees:

SG1 (Proof of correct compression): Through an interactive proof the user can check the correctness of the compression F'.

SG2 (Remote compressed data integrity): The user can detect with high probability if the CSP stores a large fraction of blocks of the compressed file F'.

As ACS extends the current RDIC protocols with the compression functionality, we extend the standard RDIC security game in order to capture the proof of correct compression property. We model the security of an ACS scheme through the $\mathbf{G}_{\mathcal{A}}^{\mathsf{ACS}}$ game (cf. Figure 8), in which a challenger $\mathcal{C}$ interacts with the adversary $\mathcal{A}$ with the following interface:

- Setup: $\mathcal{C}$ first runs the $\mathcal{O}^{\mathsf{Setup}}$ which forwards the public parameters to $\mathcal{A}$.
- TagFile: $\mathcal{A}$ asks for tags of its choice based on file blocks and gets the correct authentication tags from the $\mathcal{O}^{\mathsf{TagFile}}$ oracle. It gets the tags $\sigma_1, \ldots, \sigma_n$ for file blocks $b_1, \ldots, b_n$. Finally, it stores $b_1, \ldots, b_n$ and tags $\sigma_1, \ldots, \sigma_n$.
- Compress: After $\mathcal{A}$ having asked for the tags of all file blocks, $\mathcal{C}$ asks $\mathcal{A}$ to compress the file F. $\mathcal{C}$ also transforms the old tags to new ones $S_1 \ldots S_{n'}$ corresponding to the compressed file. That is an interactive process as in the real protocol the CSP interacts with $\mathcal{U}$ to transform the tags in case of the dictionary based encoding, while in the run-length-encoding protocol there is no interaction. At the end of the interaction $\mathcal{C}$ outputs a bit $b = 0, 1$. It sets the bit to 0 if $\mathcal{A}$ did not compute the correct compression metadata and 1 otherwise.
- $\mathsf{Challenge}_{\mathsf{F}'}$: $\mathcal{C}$ generates a challenge Q' for the compressed file F' and asks $\mathcal{A}$ to provide a proof for the possession of blocks in the challenge.
- $\mathsf{Challenge}_{\mathsf{F}}$: $\mathcal{C}$ generates a challenge Q for the uncompressed file F and asks $\mathcal{A}$ to provide a proof for the possession of blocks in the challenge.

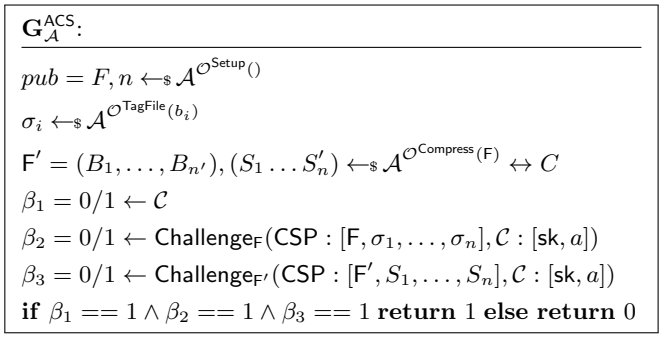


Fig. 5: $\mathbf{G}_{\mathcal{A}}^{\mathsf{ACS}}$ game

We say that $\mathcal{A}$ wins the $\mathbf{G}_{\mathcal{A}}^{\mathsf{ACS}}$ game if $\mathbf{G}_{\mathcal{A}}^{\mathsf{ACS}}$ outputs 1: $\mathbf{G}_{\mathcal{A}}^{\mathsf{ACS}} \Rightarrow 1$. The advantage of $\mathcal{A}$ in winning the game is: $\mathbf{Adv}_{\mathsf{ACS}}(\mathcal{A}) = \Pr[\mathbf{G}_{\mathcal{A}}^{\mathsf{ACS}} \Rightarrow 1]$

Definition 1. *A scheme* $S = (\mathsf{ACS.Setup}, \mathsf{ACS.Compress}, \mathsf{ACS.Challenge}_{\mathsf{F}'}, \mathsf{ACS.Challenge}_{\mathsf{F}})$ *securely instantiates* ACS *if whenever* $\mathcal{A}$ *wins the* $\mathbf{G}_{\mathcal{A}}^{\mathsf{ACS}}$ *game with high probability, then an extractor* $\mathcal{E}$ *can interact with* $\mathcal{A}$ *during the* $\mathsf{Challenge}_{\mathsf{F}}$ *and* $\mathsf{Challenge}_{\mathsf{F}'}$ *algorithms to construct the uncompressed and compressed file respectively.*

Intuitively our security definition demonstrates the concept of knowledge extractor as it has already been presented in the RDIC literature (2; 9; 11), whereby

whenever $\mathcal{A}$ succeeds in the game, then $\mathcal{E}$ can extract the files F, F' or putting another way: $\mathcal{A}$ succeeds in the game $\mathbf{G}_{\mathcal{A}}^{\mathsf{ACS}}$ whenever it successfully stores F, compresses it and stores the compressed version F'. The difference with the previous games for remote data integrity protocols is the proof for possession of both the uncompressed and compressed files F, F' (check bits β_2 and β_3 in the $\mathbf{G}_{\mathcal{A}}^{\mathsf{ACS}}$ game) and also the proof of correct compression, which is embed in the game through the interaction with the challenger $\mathcal{C}$ (check bit β_1), which represents a user $\mathcal{U}$ in the system.

4 Auditable Compressed Storage

In this section, we present our two core contributions. A protocol for auditable compressed storage for run-length encoding dubbed ACS-RLE, and second a protocol for auditable compressed storage tailored for dictionary-based Huffman encoding: ACS-HUFF. Before delving into the details of the the two protocols, we highlight the challenges and sketch a first attempt with its shortcomings.

Strawman solution 1. During the Compress phase, the user downloads the uncompressed file, compresses it, computes the authentication tags on the compressed files and finally uploads the tags and the compressed data blocks to the CSP. Despite its simplicity, this approach is fraught with increased costs. The user is burdened with increased communication costs as it has to download the uncompressed file and then upload the compressed one with the new tags. Moreover, the compression of the file and the computation of the tags on the compressed blocks is performed entirely by the user, thus amplifying its computation costs.

Strawman solution 2. The user may try to reduce its computation costs by delegating to the CSP the compression of the file. The CSP then sends the compressed file to the user and the user computes the new tags. However, this approach raises two concerns:

1. The user still has to download compressed file data blocks, but now the total communication overhead will be less than downloading the entire uncompressed file.
2. Most importantly the user has no mean of verifying the correctness of compression as it has deleted the uncompressed file. To circumvent that, the user has to store some metadata in order to check correctness of compression e.g: sign or tag with an authentication mechanism the original file and keep the signatures/tags to verify it when it uncompresses the compressed file. However, that approach incurs extra computation and storage costs.

$(a, \mathsf{sk}, b_1, \ldots, b_n, \sigma_1, \ldots, \sigma_n, \mathsf{aux}) \leftarrow \mathsf{RLE.Setup}(1^\lambda)$:

1 : User runs $\mathsf{sk} \leftarrow \mathsf{KeyGen}(1^\lambda)$ to generate uniformly random key sk.
2 : User runs $((\sigma_1, \ldots, \sigma_n), a) \leftarrow \mathsf{RLE.TagFile}(\mathsf{sk}, \mathsf{F})$
3 : User runs $\mathsf{RLE.Aux}$ to generate the aux auxiliary data.
4 : User uploads $b_1, \ldots, b_n$ and $\sigma_1, \ldots, \sigma_n$, to the CSP and stores $a, \mathsf{sk}, \mathsf{aux}$.
5 : **return** $(a, \mathsf{sk}, b_1, \ldots, b_n, \sigma_1, \ldots, \sigma_n, \mathsf{aux})$

$(\mathsf{F}', S_1, \ldots, S_{n'}) \leftarrow \mathsf{RLE.Compress}(\mathsf{F})$:

1 : CSP runs $\mathsf{F}' \leftarrow \mathsf{RLE}(\mathsf{F})$ to generate the compressed version
$\mathsf{F}' = B_1, \ldots, B_{n'}$ of F.
2 : CSP transforms $\sigma_1, \ldots, \sigma_n$ to $S_1, \ldots, S_{n'}$, for the compressed blocks
$B_1, \ldots, B_{n'}$ by running $S_1, \ldots, S_{n'} \leftarrow \mathsf{RLE.TransformTags}(B_1, \ldots, B_{n'}, \sigma_1, \ldots, \sigma_n)$.
3 : CSP stores $B_1, \ldots, B_{n'}, S_1, \ldots, S_{n'}$
4 : **return** $(\mathsf{F}', S_1, \ldots, S_{n'})$

$(1, 0) \leftarrow \mathsf{RLE.Challenge}_{\mathsf{F}'}(\mathsf{CSP} : [\mathsf{F}', S_1, \ldots, S_{n'}, \sigma_1, \ldots, \sigma_n], \mathcal{U} : [\mathsf{sk}, a])$:

1 : User generates the challenge $Q = (i, r_i)$ where $i \in [n']$ and $r_i \in \mathbb{Z}_p$
2 : CSP runs $\mathsf{RLE.Prove}(\mathsf{F}', S_1, \ldots, S_{n'}, \sigma_1, \ldots, \sigma_n, Q)$ to generate the
$\mathsf{proof} = (\{\mu_i\}_{i \in Q}, S, \mathsf{pcc})$.
3 : User runs $\mathsf{RLE.Verify}(\mathsf{proof}, \mathsf{sk}, Q, a)$ to verify correct compression
of the compressed blocks.
4 : **if** $\mathsf{RLE.Verify}(\mathsf{proof}, \mathsf{sk}, Q, a) == 1$ **return** 1 **else return** 0

$(1, 0) \leftarrow \mathsf{RLE.Challenge}_{\mathsf{F}}(\mathsf{CSP} : [\mathsf{F}, \sigma_1, \ldots, \sigma_n], \mathcal{U} : [\mathsf{sk}, a])$:

1 : User generates the challenge $Q = (i, r_i)$ where $i \in [n]$ and $r_i \in \mathbb{Z}_p$
2 : CSP runs $\mathsf{CPOR.Prove}(\mathsf{F}, \sigma_1, \ldots, \sigma_n, Q)$ to generate the
$\mathsf{proof} = (\{\mu_i\}_{i \in Q}, S, \mathsf{pcc})$
3 : User runs $\mathsf{CPOR.Verify}(\mathsf{proof}, \mathsf{sk}, Q, a)$ to verify correct data possession
of the compressed blocks
4 : **if** $\mathsf{CPOR.Verify}(\mathsf{proof}, \mathsf{sk}, Q, a) == 1$ **return** 1 **else return** 0

Fig. 6: ACS-RLE scheme.

4.1 ACS-RLE Scheme

We assume f is a secure pseudorandom function (PRF), indexed by a key $\mathsf{sk} \xleftarrow{\$} \mathcal{K}$. Let $f : \{0,1\}^* \times \mathcal{K} \rightarrow \mathbb{Z}_p$, where p is a prime. All operations are computed modulo p, unless otherwise noted. The ACS-RLE scheme is described in Figure

10, supported by algorithms in Figures 11 and 13. In the $\mathsf{RLE.Challenge_F}$ phase, we use the CPOR.Prove and CPOR.Verify algorithms from (11).

$\mathsf{sk} \leftarrow \mathsf{KeyGen}(1^\lambda)$:

1 : $\mathsf{sk} \xleftarrow{\$} \{0,1\}^\lambda$

2 : **return** sk

$\mathsf{FRQ} \leftarrow \mathsf{RLE.Aux}(b_1, \ldots, b_n)$:

1 : Compute the frequencies $\mathsf{FRQ} = \{e_i - s_i + 1 = \mathsf{frq}_i\}$ of consecutive blocks

2 : **return** $MI_i = (s_i, e_i), 1 \leq i \leq n' = \sum \mathsf{frq}_i$

$(\{\sigma_i\}_{i=1}^n, a) \leftarrow \mathsf{RLE.TagFile}(\mathsf{sk}, \mathsf{F})$:

1 : Initialize ω, n, w.

2 : User selects uniformly at random $a \in \mathbb{Z}_p$.

3 : User splits F in n blocks $b_1, \ldots b_n$ of size $w = 8 \cdot \omega$ bits each.

4 : **for** $(i = 1, i \leq n, i++)$ **do**

5 : $\sigma_i = f_{\mathsf{sk}}(i) + \sum_{k=1}^{\omega} a b_{ik}$ // Compute the verification tag

6 : **return** $\{\sigma_i\}_{i=1}^n, a$

$(S_1, \ldots, S_{n'}) \leftarrow \mathsf{RLE.TransformTags}(B_1, \ldots, B_{n'}, \sigma_1, \ldots, \sigma_n)$:

Compute the new tags $S_1, \ldots, S_{n'}$ by aggregating identical σ_i.

1 : **for** $(i = 1, i \leq n', i++)$ **do**

2 : **parse** $B_i = \mathsf{frq}_i \cdot' b'_i$

3 : $S_i = \sum_{j=i}^{\mathsf{frq}_i + i} \sigma_j$

4 : **return** $(S_1, \ldots, S_{n'})$

Fig. 7: ACS-RLE algorithms.

$(\{\mu_i\}_{i\in Q}, S, \mathsf{pcc}) \leftarrow \mathsf{RLE.Prove}(\mathsf{F}', S_1, \ldots, S_{n'}, \sigma_1, \ldots, \sigma_n, B_i, \ldots, B'_n, Q)$:

1 : **foreach** $(i, r_i) \in Q$ **do** $\mu_i = \sum_{j=1}^{\omega} r_i B_{ij}$

2 : $S = \sum_{(i,r_i)\in Q} r_i S_i$

Add the checkpoints for each $(i, r_i) \in Q$:

3 : **foreach** $i \in Q$ **do**

map i to i', i' is the starting position of the compressed block in the file

4 : $i' \in [1 \ldots n] \leftarrow \mathsf{FindRange}(i \in [1 \ldots n'])$

// E.g: F=′rrrrree′, then $6 \leftarrow \mathsf{FindRange}(2)$

5 : $\mathsf{pcc} += \sigma_{i'+\mathsf{frq}_i-1}, \sigma_{i'+\mathsf{frq}_i}, i', \mathsf{frq}_i$

6 : **return** $\mathsf{proof} = (\{\mu_i\}_{i\in Q}, S, \mathsf{pcc})$

$(0, 1) \leftarrow \mathsf{RLE.Verify}(\mathsf{proof}, \mathsf{sk}, Q, a)$:

1 : **parse** pcc as $(\sigma_l, \sigma_n, i', \mathsf{frq}_i), l = i' + \mathsf{frq}_i - 1, w = i' + \mathsf{frq}_i \forall i \in Q$

Check whether the CSP accumulated the correct range for each symbol:

2 : **foreach** $(i, r_i) \in Q$ **do**

Check whether the CSP reduced the frequency:

3 : $s = (\sigma_w - f_{\mathsf{sk}}(w))/\sum_{j=1}^{\omega} a$ // Extract the block from the uncompressed block tag

4 : $s' = \mu_{i+1}/\mathsf{frq}_i r_i \sum_{j=1}^{\omega} a$ // Extract the block from the compressed block tag

5 : **if** $s == s'$ **then return** 0

Check whether the CSP increased the frequency:

6 : $s = (\sigma_l - f_{\mathsf{sk}}(l))/\sum_{j=1}^{\omega} a$ // Extract the block from the uncompressed block tag

7 : $s' = \mu_i/\mathsf{frq}_i r_i \sum_{j=1}^{\omega} a$ // Extract the block from the compressed block tag

8 : **if** $s! = s'$ **then return** 0

Check correctness of mapping ranges to indexes using the MI dictionary

9 : **foreach** $i \in Q$ **if** $MI_i \neq (i', \mathsf{frq}_i - i = 1)$**return** 0

Check correctness of data possession proof on the compressed blocks:

10 : **if** $S \neq \sum_{i,r_i\in Q} (r_i f_{\mathsf{sk}}(i) + \sum_{j=1}^{\omega} a\mu_i)$ **then return** 0

11 : **return** 1

Fig. 8: ACS-RLE algorithms for the challenge protocol of the compressed blocks.

Intuitively, we mitigate **AT1)** by requiring the CSP to provide extra information about the next uncompressed block after the currently challenged one during the challenge response phase of $\mathsf{RLE.Challenge}_{\mathsf{F}'}$. Namely, the user $\mathcal{U}$ can extract from the tag S'_i, which encodes identical blocks, the encoded block and check whether it equals the extracted block from the next indexed uncompressed tag (lines $3-5$, algorithm RLE.Verify, Figure 13). If the check is correct, $\mathcal{U}$ infers that the CSP conducted an **AT1** type of attack. For instance, in Figure 6, a malicious CSP during RLE.Compress(F), computed a wrong block $B_1 = \sum_{i=1}^{3} b_{1,i} = 3 \cdot A$. It transforms the tags that correspond to $b_{1,1}, b_{1,2}, b_{1,3}$ into a singleton tag $S = \sigma_1 + \sigma_2 + \sigma_3$, excluding the block $b_{1,4}$, which should have been included in the transformation. However, the user during the challenge response protocol can detect the malicious behavior of the CSP by demanding the tag σ_4 and checking whether the encoded symbol 'A' equals the symbol of the tag returned by the CSP for the compressed block B_1. Notice that the tags σ_i for the uncompressed blocks are unforgeable and incorporate the block index. To mitigate **AT2)**, a similar detection mechanism is adopted: By obtaining next block tag, $\mathcal{U}$ can check whether the last symbol of the block does not match with the block of the challenged block (lines $6-8$, algorithm RLE.Verify, Figure 13). In the case of a mismatch, $\mathcal{U}$ detects a malicious behavior by the CSP. Those two invariant checks are incorporated in the $\mathsf{RLE.Challenge}_{\mathsf{F}'}$ phase which follows a challenge-response approach of probabilistically checking intact storage of the compressed file F'.

4.2 ACS-HUFF Scheme

Our ACS-HUFF scheme is based on Huffman encoding to create a dictionary, which assigns shorter codes to blocks. As the mapping translates symbols-blocks of the uncompressed file to the new ones with different codes, the aggregate technique of uncompressed tags to compute the new tags will be inconsistent with the new codes of the blocks-symbols. A naive approach would be the user to compute the new tags on the compressed blocks based on the compressed file and the dictionary the cloud provides to her. However, that increases the communication costs of the protocol overall. In our approach, the user stores a small state during the Setup phase, which allows her to validate the correctness of the Huffman dictionary. The user then computes some auxiliary information for each new tag which is used by the CSP to convert the tags of uncompressed file to the compressed ones.

The details of the ACS-HUFF scheme are given in Figures 14 and 15. We use the same conventions with the ACS-RLE scheme regarding the PRF f and the encoding of each block to a number in $\mathbb{Z}_p$. All operations are performed modulo p. For the $\mathsf{DBE.Challenge}_{\mathsf{F}}$ and $\mathsf{DBE.Challenge}_{\mathsf{F}'}$ phases, we use the CPOR.Prove and CPOR.Verify algorithms from (11).

$(a, \mathsf{sk}, b_1, \ldots, b_n, \sigma_1, \ldots, \sigma_n, \mathsf{aux}) \leftarrow \mathsf{DBE.Setup}(1^\lambda)$:

1 : User runs $\mathsf{sk} \leftarrow \mathsf{KeyGen}(1^\lambda)$ to generate uniformly random key sk.
2 : User runs $((\sigma_1, \ldots, \sigma_n), a) \leftarrow \mathsf{DBE.TagFile}(\mathsf{sk}, \mathsf{F})$
3 : User runs $\mathsf{DBE.Aux}$ to generate the aux auxiliary data
4 : User uploads data $b_1, \ldots, b_n$ and authentication tags $\sigma_1, \ldots, \sigma_n,$ to the CSP and stores $a, \mathsf{sk}, \mathsf{aux}$.
5 : **return** $(a, \mathsf{sk}, b_1, \ldots, b_n, \sigma_1, \ldots, \sigma_n, \mathsf{aux})$

$(\mathsf{F}', S_1, \ldots, S_n) \leftarrow \mathsf{DBE.Compress}(\mathsf{F})$:

1 : CSP runs $D \leftarrow \mathsf{HuffmanTree}(\mathsf{F})$ and sends D to $\mathcal{U}$
2 : $\mathcal{U}$ verifies if D is correct by calling $\mathsf{Check}(\mathsf{aux}, D)$
3 : **if** $\mathsf{Check}(\mathsf{aux}, D) == 0$ **then return** ;
4 : **else** $\mathcal{U}$ runs $T \leftarrow \mathsf{DBE.Metatag}(D)$
5 : CSP runs $\mathsf{F}' \leftarrow \mathsf{DBE}(\mathsf{F}, D)$ to generate the compressed version $\mathsf{F}' = B_1, \ldots, B_n$
6 : CSP transforms $\sigma_1, \ldots, \sigma_n$ to $S_1, \ldots, S_n,$ for the compressed blocks $B_1, \ldots, B_n$ by running $S_1, \ldots, S_n \leftarrow \mathsf{TransformTags}(\sigma_1, \ldots, \sigma_n, T)$.
7 : CSP stores $B_1, \ldots, B_n, S_1, \ldots, S_n$

$(1, 0) \leftarrow \mathsf{DBE.Challenge}_{\mathsf{F}'}(\mathsf{CSP} : [\mathsf{F}', S_1, \ldots, S_{n'}], \mathcal{U} : [\mathsf{sk}, a])$:

1 : User generates the challenge $Q = (i, r_i)$ where $i \in [n], r_i \xleftarrow{\$} \mathbb{Z}_p, |Q| = m \leq n$.
2 : CSP runs $\mathsf{CPOR.Prove}(\mathsf{F}', S_1, \ldots, S_n, Q)$ to generate the $\mathsf{proof} = (\{\mu_i\}_{i \in Q}, S)$.
3 : User runs $\mathsf{CPOR.Verify}(\mathsf{proof}, \mathsf{sk}, Q, a)$ to verify correct compression

---

$(1, 0) \leftarrow \mathsf{DBE.Challenge}_{\mathsf{F}}(\mathsf{CSP} : [\mathsf{F}, \sigma_1, \ldots, \sigma_n], \mathcal{U} : [\mathsf{sk}, a])$:

1 : User generates the challenge $Q = (i, r_i)$ where $i \in [n], r_i \xleftarrow{\$} \mathbb{Z}_p, |Q| = m \leq n$.
2 : CSP runs $\mathsf{CPOR.Prove}(\mathsf{F}, \sigma_1, \ldots, \sigma_n, Q)$ to generate the $\mathsf{proof} = (\{\mu_i\}_{i \in Q}, S)$.
3 : User runs $\mathsf{CPOR.Verify}(\mathsf{proof}, \mathsf{sk}, Q, a)$ to verify data possession

Fig. 9: ACS-HUFF scheme.

4.3 Differences between ACS-RLE and ACS-HUFF

Notice that ACS-RLE and ACS-HUFF have a different algorithmic design, which nonetheless follows our unified model for ACS as presented in Section 3. Namely, due to the nature of the RLE compression technique and the transformation of the tags by the CSP, the Compress algorithm is non-interactive and does not demonstrate any proof of compression correctness. The proof of compression correctness is embedded in the $\mathsf{Challenge}_{\mathsf{F}'}$ phase, whereby $\mathcal{U}$ is convinced not

$\{\sigma_i\}_{i=1}^{n} \leftarrow \mathsf{DBE.TagFile}(\mathsf{sk}, \mathsf{F})$:

1 : Initialize ω, n, w.
2 : User selects uniformly at random $a = \{a_k\}, k \in [1 \dots \omega] \in \mathbb{Z}_p$.
3 : User splits F in n blocks $b_1, \dots b_n$ of size $w = 8 \cdot \omega$ bits each.
4 : **for** $(i = 1, i \leq n, i++)$ **do**
5 : $\sigma_i = f_{\mathsf{sk}}(i) + \sum_{k=1}^{\omega} a_k b_{ik}$ // Compute the verification tag
6 : **return** $\{\sigma_i\}_{i=1}^{n}, a$

$\mathsf{FRQ} \leftarrow \mathsf{DBE.Aux}(b_1, \dots, b_n)$:

1 : Compute the frequencies $\mathsf{FRQ} = \{\mathsf{frq}_i\} \forall s_i \in \mathsf{F}$
2 : **return** FRQ

$(1, 0) \leftarrow \mathsf{DBE.Check}(\mathsf{aux}, D)$:

1 : **parse** aux as $\mathsf{FRQ_i}$
2 : **for** $i = \{1, \dots, |D|\}$
3 : **if** $D_i[\mathsf{frq}] \stackrel{?}{=} \mathsf{FRQ_i}$

$(T) \leftarrow \mathsf{DBE.Metatag}(D)$:

1 : $\mathcal{U}$ selects uniformly at random $a' \in \mathbb{Z}_p$.
2 : $\mathcal{U}$ **for** $i = \{1, \dots, |D|\}$ computes $t_i = \sum_{k=1}^{\omega} a' D_i[b]_k - \sum_{k=1}^{\omega} a D_i[s]_k$
and sends $T = \{t_i\}_{i=1}^{|D|}$
3 : **return** T

$(S_1, \dots, S_n) \leftarrow \mathsf{DBE.TransformTags}(\sigma_1, \dots, \sigma_n, T)$:

1 : **parse** $T = \{t_i\}_{i=1}^{|D|}$
2 : **for** $(i = 1, i \leq n, i++)$ **do**
3 : CSP computes $S_i = \sigma_i + t_i$
4 : **return** $(S_1, \dots, S_{n'})$

Fig. 10: ACS-HUFF algorithms.

only about the faithful storage of F', but also about the valid compression of F'. This is possible because RLE encoding does not change the codes of the new compressed blocks. We changed the challenge-response protocol of CPOR in order to establish the following: correct proof of faithful storage of F' demonstrates correctness during compression. With the ACS-HUFF this is not doable and a

different treatment was needed. Basically, the ACS-HUFF.Compress is interactive and deviates from ACS-RLE.Compress because 1) $\mathcal{U}$ checks the correctness of intermediate compression information: the Huffman dictionary and 2) $\mathcal{U}$ provides the metatags to CSP in order the latter to compute the new tags on F'. Moreover since the proof of correct compression is established at this step, in the ACS-HUFF.Challenge$_{\mathsf{F}'}$ phase there is no need to demonstrate a proof of correct compression and a standard CPOR challenge-response protocol is called for the proof of intact storage of F'.

5 Analysis

In this section we analyze both ACS-RLE and ACS-HUFF protocols. We start with the security analysis before exploring the efficiency.

5.1 Security

Theorem 1. ACS-RLE *is a secure accountable compressed storage system according to Definition 1 against a PPT adversary $\mathcal{A}$ and a polynomial extractor $\mathcal{E}$, as long as f is a secure* PRF.

Proof. Throughout, n' is the number of blocks of the compressed file F', m is the size of the challenge Q, and s is the number of blocks $\mathcal{A}$ stores. $\mathsf{Adv}_f(\mathcal{B})$ is a negligible quantity on the success probability of probabilistic polynomial time adversary $\mathcal{B}$ against the security of the underlying PRFf: fixed on a key, the output of the PRF should be indistinguishable from a truly random function.

Learning phase Adversary $\mathcal{A}$ plays the $\mathbf{G}_{\mathcal{A}}^{\mathsf{ACS}}$ game with a challenger $\mathcal{C}$. $\mathcal{C}$ first calls the $\mathcal{O}^{\mathsf{Setup}}$ oracle which returns $\mathsf{pk} = n, \omega, w$ and sk. $\mathcal{C}$ forwards the public key information pk to $\mathcal{A}$. $\mathcal{A}$ chooses a file F and asks the tags σ_i for each block b_i from $\mathcal{C}$. $\mathcal{A}$ forwards tuples (i, b_i) to $\mathcal{C}$, who calls $\mathcal{O}^{\mathsf{TagFile}}(b_i)$ and forwards the tags to $\mathcal{A}$. There are no restrictions here at the side of $\mathcal{A}$. From those tags it can extract a and forge the tags. When $\mathcal{A}$ finishes with queries on tags, then it stores all tags σ_i and blocks b_i.

Compress phase $\mathcal{C}$ asks $\mathcal{A}$ to compress the file F. $\mathcal{A}$ runs $\mathsf{RLE.Compress}(\mathsf{F})$ to compress file F to F' and $S_1, \ldots, S_{n'} \leftarrow \mathsf{RLE.TransformTags}(B_1, \ldots, B_{n'}, \sigma_1, \ldots, \sigma_n)$ to transform tags $\sigma_1, \ldots, \sigma_n$ to $S_1, \ldots, S_{n'}$.

Challenge phase $\mathcal{C}$ creates a random challenge $Q = (i, r_i), i \in [1 \ldots n']$ and sends it to $\mathcal{A}$. $\mathcal{A}$ constructs the $\mathsf{proof} = (\{\mu_i\}_{i \in Q}, S, \mathsf{pcc})$ calling the $\mathsf{RLE.Prove}(\mathsf{F}', S_1, \ldots, S_{n'}, \sigma_1, \ldots, \sigma_n, Q)$ algorithm and returns it to $\mathcal{C}$.

We consider two events bad_1 and bad_2. $\mathsf{bad}_1 = 1$ happens whenever $\mathcal{A}$ convinces $\mathcal{C}$ during the challenge protocol without holding the required challenged blocks and $\mathsf{bad}_2 = 1$ when $\mathcal{A}$ does not transform correctly the blocks for the compressed blocks or does not store them at all. We categorize bad_2 event with two possible cases: bad_{21} considers the *remove block attack* event and bad_{21} the event when *add block attack* occurs. The success probability of $\mathcal{A}$ in winning the

$\mathbf{G}_{\mathcal{A}}^{\mathsf{ACS}}$ game without storing the requested blocks or compressing and transforming the tags without being detected by a benign verifier ($\mathcal{C}$, or user in the real protocol) is bounded by:

$$\Pr[\mathsf{bad}_1 = 1] + \Pr[\mathsf{bad}_{21} = 1] + \Pr[\mathsf{bad}_{22} = 1] \tag{1}$$

$$= 2 \cdot \sum_{i=0}^{m} \frac{i}{m} \cdot \mathsf{Adv}_f(\mathcal{B}) + 1 - \frac{n' - s}{m} \tag{2}$$

$$= 2 \cdot \frac{m(m-1)}{2m} \cdot \mathsf{Adv}_f(\mathcal{B}) + 1 - \frac{n' - s}{m} \tag{3}$$

$$= (m-1) \cdot \mathsf{Adv}_f(\mathcal{B}) + 1 - \frac{n' - s}{m} \tag{4}$$

Thus the success probability of an extractor $\mathcal{E}$ to extract the file equals:

$$\begin{aligned}
\Pr[\mathcal{E} \Rightarrow 1] &= 1 - (\Pr[\mathbf{G}_{\mathcal{A}}^{\mathsf{ACS}} \Rightarrow 1] \wedge (\Pr[\mathcal{A} \text{ not store } \mathsf{F}'] \\
&\quad \vee \Pr[\mathcal{A} \text{ not compress correctly}])) \\
&= 1 - (\Pr[\mathsf{bad}_1 = 1] + \Pr[\mathsf{bad}_{21} = 1] + \Pr[\mathsf{bad}_{22} = 1]) \\
&= 1 - (m-1) \cdot \mathsf{Adv}_f(\mathcal{B}) + 1 - \frac{n' - s}{m} \\
&\leq 1 - \mathsf{negl}(\lambda)
\end{aligned}$$

Theorem 2. ACS-HUFF *is a secure accountable compressed storage system according to Definition 1 against a PPT adversary $\mathcal{A}$ and a polynomial extractor $\mathcal{E}$, as long as f is a secure* PRF.

Proof. Similarly with the ACS-RLE proof, we analyze the success probabilities of an extractor to extract the correct file or providing proofs of compression.

$$\begin{aligned}
\Pr[\mathcal{E} \Rightarrow 1] &= 1 - (\Pr[\mathbf{G}_{\mathcal{A}}^{\mathsf{ACS}} \Rightarrow 1] \wedge (\Pr[\mathcal{A} \text{ not store } \mathsf{F}'] \vee \\
&\quad \Pr[\mathcal{A} \text{ not compress correctly}])) \\
&= 1 - (\Pr[\mathsf{bad}_1 = 1] + \Pr[\mathsf{bad}_2 = 1]) \\
&= 1 - (\sum_{i=0}^{m} \frac{i}{m} \cdot \mathsf{Adv}_f(\mathcal{B}) + 1 - \frac{n' - s}{m}) \\
&= 1 - (\frac{(m-1)}{2} \cdot \mathsf{Adv}_f(\mathcal{B}) + 1 - \frac{n' - s}{m}) \\
&\leq 1 - \mathsf{negl}(\lambda)
\end{aligned}$$

5.2 Performance

We perform a theoretical performance analysis of ACS-RLE and ACS-HUFF, two vanilla protocols V1 and V2 as baseline. V1 consists of a straightforward approach in which the user $\mathcal{U}$ whenever seeks to compress its already outsourced data file F, downloads the entire file F, compresses it to F′, computes the new tags $\{S\}_{i=1}^{n}$

Protocol	**PreComputation**	**Storage**	**Compress** $[\mathcal{U}\|\mathsf{CSP}\|C]$		
V1	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(n+n')$	0	$\mathcal{O}(n+2n')$
V2	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(n')$	$\mathcal{O}(n)$	$\mathcal{O}(2n')$
ACS-RLE	$\mathcal{O}(n)$	$\mathcal{O}(n')$	0	$\mathcal{O}(n+n')$	0
ACS-HUFF	$\mathcal{O}(n)$	$\mathcal{O}(\|D\|)$	$\mathcal{O}(\|D\|)$	$\mathcal{O}(n+\|D\|)$	$\mathcal{O}(\|D\|)$

Table 2: Comparison table. **PreComputation** depicts the computation time for the user $\mathcal{U}$ for the Setup phase. **Storage** is the space overhead for $\mathcal{U}$ after uploading file F and tags. **Compress** shows computation time for $\mathcal{U}$ and CSP, and the communication overhead (C) needed for the compression at the CSP.

Protocol	**Challenge**	**Prove**	**Verify**
V1	$\mathcal{O}(1)$	$\mathcal{O}(m)$	$\mathcal{O}(1)$
V2	$\mathcal{O}(1)$	$\mathcal{O}(m)$	$\mathcal{O}(1)$
ACS-RLE	$\mathcal{O}(1)$	$\mathcal{O}(m)$	$\mathcal{O}(m)$
ACS-HUFF	$\mathcal{O}(1)$	$\mathcal{O}(m)$	$\mathcal{O}(1)$

Table 3: Comparison table. **Challenge**, **Prove** and **Verify** present the computation overhead to compute the Challenge, the proof time of the CSP and the time $\mathcal{U}$ spends to verify the proof respectively. n is the number of blocks in the uncompressed file F, n' is the number of the compressed blocks, m is the size of the challenge and $|D|$ is the size of entries for the dictionary of Huffman encoding for ACS-HUFF scheme.

and finally uploads both F' and $\{S\}_{i=1}^{n}$ to the CSP. In V2, the user asks the CSP to compress the file F', downloads it to check correctness of compression, computes the new tags $\{S\}_{i=1}^{n}$ and uploads them to the CSP.

Tables **??** and **??** show the comparison. We take into account the following factors in order to compare the protocols: The **PreComputation** time for the user during the Setup phase. The **Storage** overhead after the Setup $\mathcal{U}$ needs to allocate. The computation time for the Compress algorithm for $\mathcal{U}$ and the CSP, denoted as **Compress** in the table. The communication overhead (**Compress**.C) that is required for compression. And finally, the running time overhead during the challenge response protocol $\mathsf{Challenge}_{\mathsf{F}'}$ for the computation of the **Challenge**, the **Prove** and the verification (**Verify**).

PreComputation is the same $\mathcal{O}(n)$ for all protocols as it consists of splitting the file F in n blocks and computing n tags with the same mechanism (CPOR tags). The **Storage** in both vanilla protocols is $\mathcal{O}(1)$. In ACS-RLE, the user storage cost is increased to $\mathcal{O}(n')$ due to the auxiliary table MI, which stores ranges of symbols. In ACS-HUFF, $\mathcal{U}$ has to store the frequencies of all symbols of the file F in order to check for the correctness of compression during the ACS-HUFF.Compress phase. This results in a storage complexity of $\mathcal{O}(|D|)$. Notice that $|D|$ is considerably smaller than n and depends only on the vocabulary of the file F and the block size w of each block. The latter is due to the heuristic that the bigger the w the less unique patterns, thus entries, in D.

The advantages of our two auditable compressed storage protocols ACS-RLE and ACS-HUFF are obvious during the Compress algorithm which is assessed in

the **Compress** column of **??**. As in both ACS-RLE and ACS-HUFF the compression is delegated to the CSP, $\mathcal{U}$'s computation time complexity is minimal: 0 for ACS-RLE and $\mathcal{O}(|D|)$ for ACS-HUFF. V1 requires $\mathcal{O}(n+n')$ computation time for $\mathcal{U}$ as the user compresses the file and computes the new tags. It also results in $\mathcal{O}(n+2n')$ total communication complexity. V2 splits the computation overhead for compression in $\mathcal{O}(n')$ for $\mathcal{U}$ and $\mathcal{O}(n)$ for the CSP. Recall that in V2 the user computes the new tags for the compressed file and the CSP compresses. In contrast with the vanilla protocols, ACS-RLE and ACS-RLE demand constant communication complexity for compression. ACS-RLE communication complexity is reduced to 0, whereas in ACS-HUFF the complexity is slightly increased to $\mathcal{O}(|D|)$ due to the extra check $\mathcal{U}$ has to perform to check the correctness of Huffman based compression. Finally, during the $\mathsf{Challenge}_{\mathsf{F}'}$ phase the verification time for the ACS-RLE protocol is increased from $\mathcal{O}(1)$ to $\mathcal{O}(m)$ due to the extra checks for compression correctness. Note that m refers to the size of the challenge Q on the compressed file F'. ACS-RLE verification time is the same as with the vanilla protocols, as it is a conventional CPOR verification (11). We let as future work the design of an auditable compress storage protocol which will enjoy the low Compress complexity user running time and communication overhead as with ACS-RLE, with the constant verification time of ACS-HUFF.

Another baseline method could use an existing RDIC scheme for the uncompressed file and one for the compressed one. The asymptotic performance of such a scheme would be optimal in terms of compression costs compared with all the aforementioned ones. However, there is a metric which is not included in that case. The Cloud has to store $2n$ tags instead of n tags, which asymptotically is equivalent with the existing baselines but for large files it might be a prohibitive solution for the CSP.

Notice also that for high entropy files both ACS-RLE and ACS-HUFF require the client to store up to $O(n)$ information. That however imposes that the compression ratio would be very small due to high entropy; so the client would not have any incentive a-priori to use any compression techniques.

6 Conclusion

We initiated the study of *Auditable Compressed Storage*, whereby the original data changes format during its lifetime, when outsourced to a cloud infrastructure. This setting poses new threats and challenges in secure outsourcing storage as the task of compressing the data is delegated to the cloud, who is in charge of transforming the authentication tags of the original data to tags corresponding to the compressed data. As the Cloud Service Provider is economically motivated to cheat by not applying a correct compression to the original data, we demonstrate how to extend existing Remote Data Integrity Check protocols to encompass a proof of correct compression in conjunction with proofs of faithful storage. We designed and analyzed two protocols: ACS-RLE for Run Length Encoding compression and ACS-HUFF for Huffman Dictionary-based encoding.

Our protocols fulfill the security guarantees for correct compression and faithful storage.

Acknowledgments. This research was supported by the US National Science Foundation under Grants No. CNS 1801430, CNS 1054754, and DGE 1565478.

Bibliography

[1] Ateniese, G., Burns, R., Curtmola, R., Herring, J., Khan, O., Kissner, L., Peterson, Z., Song, D.: Remote data checking using provable data possession. ACM Trans. Inf. Syst. Secur. **14**, 12:1–12:34 (June 2011)

[2] Ateniese, G., Burns, R., Curtmola, R., Herring, J., Kissner, L., Peterson, Z., Song, D.: Provable data possession at untrusted stores. In: Proc. of the 14th ACM CCS. pp. 598–609 (2007)

[3] Ateniese, G., Di Pietro, R., Mancini, L.V., Tsudik, G.: Scalable and efficient provable data possession. In: Proc. of the 4th SecureComm. pp. 9:1–9:10 (2008)

[4] Cash, D., Küpçü, A., Wichs, D.: Dynamic proofs of retrievability via oblivious ram. J. Cryptol. **30**(1), 22–57 (2017)

[5] Erway, C., Küpçü, A., Papamanthou, C., Tamassia, R.: Dynamic provable data possession. In: Proc. of the 16th ACM CCS. pp. 213–222 (2009)

[6] Esiner, E., Kachkeev, A., Braunfeld, S., Küpçü, A., Özkasap, O.: Flexdpdp: Flexlist-based optimized dynamic provable data possession. Trans. Storage **12**(4), 23:1–23:44 (Aug 2016)

[7] Etemad, M., Küpçü, A.: Transparent, distributed, and replicated dynamic provable data possession. In: Proc. of the 11th ACNS. pp. 1–18 (2013)

[8] Huffman, D.A.: A method for the construction of minimum-redundancy codes. Proc. of the Institute of Radio Engineers **40**(9), 1098–1101 (1952)

[9] Juels, A., Kaliski, Jr., B.S.: Pors: Proofs of retrievability for large files. In: Proc. of the 14th ACM CCS. pp. 584–597 (2007)

[10] Mohammad Etemad, M., Küpçü, A.: Generic efficient dynamic proofs of retrievability. In: Proc. of the ACM CCSW. pp. 85–96 (2016)

[11] Shacham, H., Waters, B.: Compact Proofs of Retrievability, pp. 90–107. Berlin, Heidelberg (2008)

[12] Shi, E., Stefanov, E., Papamanthou, C.: Practical dynamic proofs of retrievability. In: Proc. of the 2013 ACM CCS. pp. 325–336 (2013)

[13] Stefanov, E., van Dijk, M., Juels, A., Oprea, A.: Iris: A scalable cloud file system with efficient integrity checks. In: Proc. of the 28th ACSAC. pp. 229–238 (2012)

[14] Wang, Q., Wang, C., Li, J., Ren, K., Lou, W.: Enabling public verifiability and data dynamics for storage security in cloud computing. In: Proc. of the 14th ESORICS. pp. 355–370 (2009)

[15] Zhang, Y., Blanton, M.: Efficient dynamic provable possession of remote data via update trees. Trans. Storage **12**(2), 9:1–9:45 (Feb 2016)