A Framework for the Design of Secure and Efficient Proofs of Retrievability

Françoise Levy-dit-Vehel[1] and Maxime Roméas[2]

[1]LIX, ENSTA Paris, INRIA, Institut Polytechnique de Paris, 91120 Palaiseau, France
levy@ensta.fr
[2]LIX, École polytechnique, INRIA, Institut Polytechnique de Paris, 91120 Palaiseau, France
romeas@lix.polytechnique.fr

Abstract

Proofs of Retrievability (PoR) protocols ensure that a client can fully retrieve a large outsourced file from an untrusted server. Good PoRs should have low communication complexity, small storage overhead and clear security guarantees with tight security bounds. The focus of this work is to design good PoR schemes with simple security proofs. To this end, we use the Constructive Cryptography (CC) setting by Maurer [13]. We propose a framework for the design of secure and efficient PoR schemes based on Locally Correctable Codes (LCC). We give a first instantiation of our framework using the high rate lifted codes introduced by Guo *et al.* [5]. This yields an infinite family of good PoRs. We assert their security by solving a finite geometry problem, giving an explicit formula for the probability of an adversary to fool the client. Using the local correctability properties of Tanner codes, we get another instantiation of our framework and derive an analogous formula for the success probability of the audit.

1 Introduction

1.1 Context and state-of-the-art

With the continuous increase in data creation, individuals and business entities call upon remote storage providers to outsource their data. This new dependency raises some issues, as the storage provider can try to read and/or modify the client's data. Besides, when a client does not often access his data, the service provider can delete it to make room for another client's data. In this context, it appears important to deploy client side protections designed to bring security guarantees like confidentiality and integrity. In this work, we focus on the following problem : given a client who stored a file on a server and erased its local copy, how can he check if he is able to retrieve his file from the server in full ? Addressing this issue is the goal of a class of cryptographic protocols called Proofs of Retrievability (PoRs).

The first PoR scheme was proposed in 2007 by Juels and Kaliski [9] and was based on checking the integrity of some sentinel symbols secretly placed by the client before uploading its file. This scheme has low communication but its drawback is that it is bounded-use only, as the number of possible verifications depends on the number of sentinels. Shacham and Waters [17] proposed to correct this drawback by appending some authenticator symbols to the file. Verification consists in checking random linear combinations of file symbols and authenticators. Then comes a few PoR schemes based on codes. Bowers *et al.* [2] proposed a double-layer encoding with the use of an inner code to recover information symbols and an outer code to correct the remaining erasures. Dodis *et al.* [4] formalize the verification process as a request to a code which models the space of possible answers to a challenge. They use Reed-Solomon codes to design their PoR scheme. In

2013, Paterson *et al.* [16] laid the foundation for studying PoR schemes using a coding theoretic framework. Following these ideas, Lavauzelle and Levy-dit-Vehel [12] (2016) used the local structure of the lifted codes introduced by Guo *et al.* [5] to build a PoR scheme, that compares favourably to those presented above w.r.t. storage overhead.

Unfortunately, PoR schemes have a few issues. Indeed, their security definitions are often unclear, making it hard to understand what they really achieve. Moreover, when a client wants to retrieve his data, the security guarantees brought by the use of the PoR scheme only holds under the condition that both client and server unveil some private information (client's secret material and server's state). We give a detailed explanation of this in sec. 2.2. In 2018, Badertscher and Maurer [1] used the Constructive Cryptography (CC) framework introduced by Maurer [13] to propose a new PoR definition, that avoids the aforementioned flaws. They also designed a PoR scheme based on generic erasure codes. Generalizing [1] and [12], we introduce a framework for designing secure, composable and efficient PoR protocols based on locally correctable codes.

We formulate our framework using the terminology of the CC model. This approach allows us to design and study the security of PoR schemes in a modular fashion, that achieves stronger security and clearer security guarantees than previous schemes (whose security was based on so-called ϵ adversaries or related notions). Using another definitional model such as the Universal Composability one by Canetti [3] would probably give closely related results. We chose to use CC because it makes the resources available to the parties (namely, untrusted server storage, local memories, communication channels) explicit. It also makes the switching between computational, statistical and information-theoretic security notions easy. Finally, using CC, we give exact or tight security bounds for our schemes as opposed to the asymptotic bounds of other models.

In our new framework, we design secure and efficient PoR schemes from a family of codes : the locally correctable codes (LCCs) formally introduced by Katz and Trevisan [10] in 2000. Reed-Muller codes are well known to be locally correctable, but with poor rate as their length grows. The year 2011 has seen a breakthrough in the theory of codes with locality, with the construction by Kopparty *et al.* [11] of a class of high-rate LCCs - the multiplicity codes - generalizing the Reed-Muller class. Other high rates LCCs are notably the lifted codes introduced by Guo *et al.* [5], and the expander codes of Hemenway *et al.* [6]. The high rate of these codes permits to minimize the server storage overhead, making them best suited for the outsourced storage context. We give an instantiation of our framework using the lifted codes of Guo *et al.* . In a nutshell, we exploit the geometric properties of lifted codes and the CC security model for PoRs to give simple security proofs with tight bounds. This is a key difference between our approach and the one of Lavauzelle and Levy-dit-Vehel [12], which is also based on lifted codes and can in fact be seen as a different instantiation of our framework.
We also sketch another instantiation of our framework using the graph codes of Tanner [18].

1.2 Contributions

Given a locally correctable code, we propose a canvas for deriving a PoR scheme. We get efficiency by taking advantage of the local correctability of the code to design an audit with low communication complexity. Using the CC security model for PoRs of Badertscher and Maurer [1], we give clear and composable security guarantees for our PoR construction. We are also able to give tight security bounds derived from geometric/combinatorial proofs.

As in many protocols, the client first encodes its file and uploads it to the server. Retrieving the file, *i.e.* decoding, is done by iterating the local correction algorithm of the code [1]. With such a decoding process in mind, we identify the adversarial configurations of corruptions that would prevent the extraction of the file. This analysis of adversarial impact permits us to phrase the security of the audit - which heavily relies on the local correction step - as a problem about the structure of the code. For example, if the code uses geometric properties of $\mathbb{F}_q^m$, we reduce the security of the audit to a finite geometry problem. If the code is a graph code, we reduce the security to a graph theoretic problem.

[1] This is needed to ensure that extraction is indistinguishable from a sequence of audits. Indeed, if a malicious server could distinguish audits from extraction, it could adapt its behaviour accordingly.

Instantiating our framework with the lifted Reed-Solomon codes : we characterize all the configurations of corruptions that are impossible to correct using the local correctability of those codes. More precisely, we show that these configurations of corruptions correspond to sets of points verifying a geometric property inside a vector space over a finite field. Then, we show that these sets of points belong to a large number of affine lines. From this we derive an explicit formula for the probability of the adversary to fool the client. Wrapping this analysis in our PoR framework, we get :

Theorem (Lifted Reed-Solomon Codes PoR)**.** *Let $d, m \in \mathbb{N}$, $\mathbb{F}_q$ be a finite field. The protocol* $\mathsf{audit} := (\mathsf{init}_{\mathsf{audit}}, \mathsf{audit}_{\mathsf{lcc}}, \ldots, \mathsf{audit}_{\mathsf{lcc}})$ *(with k copies of* $\mathsf{audit}_{\mathsf{lcc}}$*) for the lifted Reed-Solomon code* $\mathrm{Lift}_m(\mathrm{RS}_q[q,d])$ *of dimension ℓ constructs the auditable and authentic SMR, say* $\mathbf{aSMR}^{k,\mathsf{audit}}_{\Sigma,\ell}$*, from* $\mathbf{aSMR}^{k}_{\Sigma,q^m}$*, with respect to the simulator* $\mathsf{sim}_{\mathsf{audit}}$ *and the pair (*honSrv*,* honSrv*). More precisely, for all distinguisher* $\mathbf{D}$ *making at most r audits, we have*

$$\Delta^{\mathbf{D}}(\mathsf{honSrv}^{\mathsf{S}}\mathsf{audit}_{\mathcal{P}}\,\mathbf{aSMR}^{k}_{\Sigma,q^m}, \mathsf{honSrv}^{\mathsf{S}}\,\mathbf{aSMR}^{k,\mathsf{audit}}_{\Sigma,\ell}) = 0$$

$$\textit{and}\quad \Delta^{\mathbf{D}}(\mathsf{audit}_{\mathcal{P}}\,\mathbf{aSMR}^{k}_{\Sigma,q^m}, \mathsf{sim}^{\mathsf{S}}_{\mathsf{audit}}\,\mathbf{aSMR}^{k,\mathsf{audit}}_{\Sigma,\ell}) \leq r \cdot \left(1 - \frac{1}{q^{m-1}}\right)\left(1 - \frac{d-1}{q-1}\right)^{q-d+1}$$

Thus, we get a family of PoR schemes with precise security guarantees. Efficiency of this construction is shown in figures 5 and 4 of sec. 6, where we also give a comparison between our parameters and those of the PoR scheme of Lavauzelle and Levy-dit-Vehel [12].

Instantiating our framework with Tanner codes: we proceed analogously as in the lifted Reed-Solomon codes case to first design a global decoder, and then to characterize the configuration of erased edges that are unrecoverable by the decoding algorithm. This way, we derive a bound on the failure probability of the audit, yielding the following :

Theorem (Tanner Codes PoR)**.** *Let $G = (V, E)$ be a q-regular graph with $|V| := n$ and let $\mathcal{C}_0 \subseteq \mathbb{F}^q$ be a linear code with minimal distance d and rate R. Let s be the minimal size, number of vertices, of a subgraph of G with minimal degree d. The protocol* $\mathsf{audit} := (\mathsf{init}_{\mathsf{audit}}, \mathsf{audit}_{\mathsf{graph}}, \ldots, \mathsf{audit}_{\mathsf{graph}})$ *(with k copies of* $\mathsf{audit}_{\mathsf{graph}}$*) for a Tanner code $\mathcal{C}(G, \mathcal{C}_0)$ of length $nq/2$ and rate at least $2R - 1$ constructs the auditable and authentic SMR, say* $\mathbf{aSMR}^{k,\mathsf{audit}}_{\mathbb{F},(2R-1)nq/2}$*, from* $\mathbf{aSMR}^{k}_{\mathbb{F},nq/2}$*, with respect to the simulator* $\mathsf{sim}_{\mathsf{audit}}$ *and the pair (*honSrv*,* honSrv*). More precisely, for all distinguisher* $\mathbf{D}$ *making at most r audits, we have*

$$\Delta^{\mathbf{D}}(\mathsf{honSrv}^{\mathsf{S}}\mathsf{audit}_{\mathcal{P}}\,\mathbf{aSMR}^{k}_{\mathbb{F},nq/2}, \mathsf{honSrv}^{\mathsf{S}}\,\mathbf{aSMR}^{k,\mathsf{audit}}_{\mathbb{F},(2R-1)nq/2}) = 0$$

$$\textit{and}\quad \Delta^{\mathbf{D}}(\mathsf{audit}_{\mathcal{P}}\,\mathbf{aSMR}^{k}_{\mathbb{F},nq/2}, \mathsf{sim}^{\mathsf{S}}_{\mathsf{audit}}\,\mathbf{aSMR}^{k,\mathsf{audit}}_{\mathbb{F},(2R-1)nq/2}) \leq r \cdot \left(1 - \frac{s}{n}\right)$$

In order to design our framework, we constructed a protocol to authenticate outsourced data (**aSMR**, for Authentic Server Memory Resource[2]) that is tailored for PoR purposes (see sec. 4). Our **aSMR** is different from the one of [1] in several aspects, notably, when dealing with encoded data, we halve the extra storage needed in comparison to [1]. Further details are to be found in sec. 4. This new construction might be useful in other code based cryptographic protocols for outsourced storage. It can also be used to improve the efficiency of the generic PoR of [1].

2 Background

2.1 The Constructive Cryptography model

The CC model, introduced by Maurer [14] in 2011, and augmented since then [13, 15, 8, 7] aims at asserting the real security of cryptographic primitives. To do so, it redefines them in terms of so-called *resources* and

[2]Following the terminology of [1].

converters. In this model, starting from a basic resource (e.g. communication channel, shared key, memory server...), a converter (a cryptographic protocol) aims at constructing an enhanced resource, *i.e.*, one with better security guarantees. The starting resource, lacking the desired security guarantees, is often called the *real* resource and the obtained one is often called the *ideal* resource, since it does not exist as is in the real world. An example of such an ideal resource is a confidential server, where the data stored by a client is readable by this client only. The only information that leaks to other parties is its length. This resource does not exist, but it can be emulated by an insecure server on which the client uses an encryption protocol where the encryption scheme is $\mathsf{IND}-\mathsf{CPA}$ secure. We say that this *construction* of the confidential server is secure if the real world - namely, the insecure server together with the protocol - is *just as good* as the ideal world - namely, the confidential server. This means that, whatever the adversary can do in the real world, it could as well do in the ideal world. We use the fact that the ideal world is by definition secure and contraposition to conclude. This construction notion is illustrated in fig. 1.

The CC model follows a top-down approach, allowing to get rid of useless hypotheses made in other models. A particularity of this model is its composability, in the sense that a protocol obtained by composition of a number of secure constructions is itself secure.

We give the required material to understand how we use CC in app. A. We follow the presentation of [7].

Server-Memory Resources We recall the constructions of [1] that we will use or improve in this work. The first resource is the basic server-memory resource (SMR) denoted by $\mathbf{SMR}^{k}_{\Sigma,n}$ where k is the number of clients, Σ the alphabet and n the number of data blocks. The resource allows cooperating clients to read and write data blocks that are encoded as elements of a finite alphabet Σ via their interfaces $\mathsf{C}_1,\ldots,\mathsf{C}_k$. The interface C_0 is the initialization interface used to set up the initial state of the resource. The server can be "honest but curious" by obtaining the entire history of accesses made by the clients (a log file) and reading their data at interface S_H. The server can also be intrusive and overwrite data using its interface S_I when the resource is set into a special write mode. This write mode can be toggled by the distinguisher at the world interface W. The specification of the resource $\mathbf{SMR}^{k}_{\Sigma,n}$ is given in fig. 6 in app. B.

We recall the figure 1 given in [1] to illustrate the CC construction notion on **SMR**s. The SMR security guarantees can be augmented to provide authenticity by using a suitable protocol in this construction notion. This new server-memory resource is called authentic SMR, denoted by $\mathbf{aSMR}^{k}_{\Sigma,n}$, and is constructed in [1]. In the **aSMR**, the behavior of the server at its interface S_I is weakened as the server cannot modify the content of data blocks but is limited to either delete or restore previously deleted data blocks at this interface. A deleted data block is indicated by the special symbol ϵ. In this work, we use a different **aSMR** specification that the one used in [1]. We modify the `restore` behavior to only restore data blocks that were deleted after the last client update of the database. We introduce a version number that tracks the number of said updates in the history of the **aSMR** and clients are now allowed to overwrite corrupted data blocks. These changes decrease the storage overhead along with the communication complexity of read operations while the communication complexity of write operations is increased in comparison to the specification of [1]. Our changes to the **aSMR** yield substantial improvements for the parameters of our code-based PoR schemes. Our take on the **aSMR** resource is described in fig. 2 and our changes are precised in sec. 4.

2.2 Proofs of Retrievability

Proofs of Retrievability (PoR) are cryptographic protocols whose goal is to guarantee that a file stored by a client on a server remains retrievable in full. PoRs thus involve two parties : a client who owns a file F and a server, here modeled as a SMR, on which F is stored. We recall the commonly used definitions for PoR security as presented in [12]. A PoR system is composed of three main procedures:

- *An initialization phase.* The client encodes his file F with an initialization function $\mathbf{Init}(F) = (\tilde{F}, \mathtt{data})$. He keeps `data` (*e.g.* keys, etc.) for himself, then he sends $\tilde{F}$ to the server and erases F.
- *A verification phase.* The client produces a challenge c with a randomized **Chall** function and sends it to the server. The latter creates a response $r = \mathbf{Resp}(\tilde{F}, c)$ and sends it back to the client. The

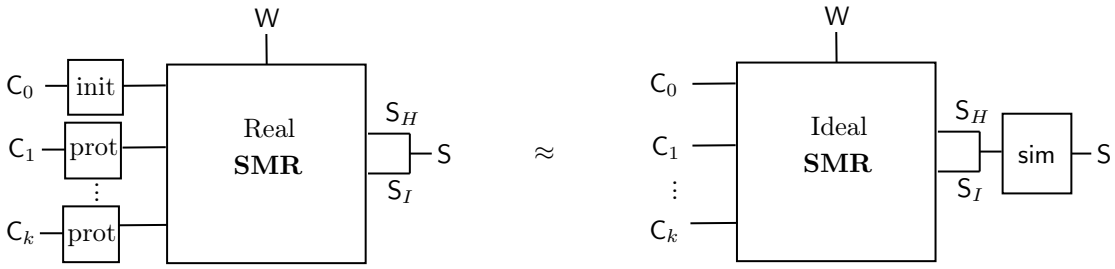

Figure 1: Illustration of the construction notion for SMRs. On the left, we have a real **SMR** with a protocol for each client. On the right, we have an ideal **SMR** with stronger security guarantees. The construction is secure if there exists a simulator that makes these two resources indistinguishable.

Resource aSMR$^k_{\Sigma,n}$

The **aSMR** definition is identical to **SMR** except for the influence of an adversary at interface S_I and the addition of a version number ctr.

Interface Interface $\mathsf{C}_t, t \in \{1, \ldots, k\}$

Input: $(\texttt{read}, i) \in [1, n]$
  if ACTIVE **and not** INTRUSION **then**
    HIST $\leftarrow$ HIST $||$ $(t, \texttt{R}, i)$
    return $\mathbb{M}[i]$

Input: $(\texttt{write}, i, x) \in [1, n] \times \Sigma$
  if ACTIVE **and not** INTRUSION **then**
    $ctr \leftarrow ctr + 1$
    HIST $\leftarrow$ HIST $||$ $(t, \texttt{W}, i, x, ctr)$
    $\mathbb{M}[i] \leftarrow x$

Interface Interface S_I

Input: $(\texttt{delete}, i) \in [1, n]$
  if INTRUSION **then**
    $\mathbb{M}[i] \leftarrow \epsilon$

Input: $(\texttt{restore}, i) \in [1, n]$
  if INTRUSION **then**
    if $\exists k, x, t :$ HIST$[k] = (t, \texttt{W}, i, x, ctr)$ **then**
      $\mathbb{M}[i] \leftarrow x$

Figure 2: Our new authentic SMR (only the differences with **SMR** are shown)

client checks if r is correct by running **Verif**(c, r), which also access `data`, and outputs `accept` if r is considered correct and `reject` otherwise.

- *An extraction phase.* If the client has been convinced by the verification phase, he can use his **Extract** algorithm to recover his whole file with high probability.

The security of PoR schemes is usually defined with ϵ-adversaries. In a PoR scheme, the client wants to use the **Verif** procedure to be sure that he will be able to retrieve his file in full by using the **Extract** procedure. The following definition models the fact that, if the server's answers to client's challenges make him look like he owns the file, then the client must be able to recover it entirely.

Definition 1 (ϵ-adversary)**.** Let $\mathcal{P}$ be a PoR system and X be the space of challenges generated by **Chall**. An ϵ-adversary $\mathcal{A}$ for $\mathcal{P}$ is an algorithm such that, for all files F,

$$\Pr_{x \in X}\left[\textbf{Verif}(x, \mathcal{A}(x)) = \texttt{false}\right] \leq \epsilon$$

The client models the server as an ϵ-adversary and uses his verification process to maintain an approximation of ϵ. Depending on this estimate, the client can decide whether his file is retrievable or not. The security of PoRs is thus usually measured as follows :

Definition 2 (PoR security)**.** Let $\epsilon, \rho \in [0,1]$. A PoR system is said to be (ϵ, ρ)-sound if, for all ϵ-adversaries $\mathcal{A}$ and for all files F, we have:

$$\Pr\left[\mathbf{Extract}^{\mathcal{A}} = F\right] \geq \rho$$

where the probability is taken over the internal randomness of $\mathbf{Extract}^{\mathcal{A}}$.

As pointed out by Badertscher and Maurer in [1], this model has a major drawback concerning client-side security guarantees. The most important thing for the client, the availability of his data, is conditioned to the execution of the **Extract** algorithm which needs to access the client's private data and the server's strategy (as indicated in def. 2). In practice, no server would reveal its entire state to a client. This problem is addressed in [1], where the authors used the CC framework to propose a definition of PoRs that fixes this drawback. In their work, they introduced an ideal abstraction of PoRs in the form of an ideal SMR that sees the clients' interfaces augmented with an `audit` mechanism. On an `audit` request, the resource checks whether the current memory content is indeed the newest version that the client wrote to the storage. If a single data block has changed, the ideal audit will detect this and output `reject` to the client. In case of a successful audit (returning `accept`), this guarantee holds until the server gains write-access to the storage, in which case a new audit has to reveal whether modifications have been made. We present the specification of the auditable and authentic SMR $\mathbf{aSMR}_{\Sigma,n}^{k,\mathsf{audit}}$ in fig. 7 of app. C. In addition to the advantages we discussed, we believe that this CC based security model is simpler and more intuitive than the one of ϵ-adversaries.

2.3 Locally Correctable Codes

In [1], Badertscher and Maurer give a protocol based on generic erasure codes to construct the auditable **aSMR**. Due to the use of classical codes, a client who wants to read a single data block needs to read the entire memory in order for him to run the decoding algorithm of the code to recover (or not) the data block. In this work, we show how one can use LCCs, so that one has to read only a small number of memory positions to recover one data block, while keeping the auditable property of the constructed resource. We now briefly present LCCs, which were formally introduced by Katz and Trevisan [10] in 2000.

Definition 3 (Locally correctable code)**.** Let $r \in \mathbb{N}, \delta \in [0,1]$ and $\epsilon : [0,1] \mapsto [0,1]$. A code $\mathcal{C} \subseteq \mathbb{F}_q^n$ is said to be (r, δ, ϵ)-locally correctable if there exists a probabilistic decoding algorithm $\mathcal{A}$ such that,

1. For all $\mathbf{c} \in \mathcal{C}$, for all $i \in [\![1, n]\!]$ and for all vectors $\mathbf{y} \in \mathbb{F}_q^n$ with relative Hamming distance $\Delta(\mathbf{c}, \mathbf{y}) \leq \delta$, we have $\Pr[\mathcal{A}^{\mathbf{y}}(i) = \mathbf{c}_i] \geq 1 - \epsilon(\delta)$, where the probability is taken over the internal randomness of $\mathcal{A}^{\mathbf{y}}$.
2. The algorithm $\mathcal{A}$ makes at most r queries to the vector $\mathbf{y}$.

3 Our framework

We describe our framework which derives PoR schemes from a given LCC $\mathcal{C}$. In all our PoRs, the client's file is encoded as a codeword of $\mathcal{C}$ and uploaded to the server. We want to protect the client from an adversary able to introduce corruptions on the outsourced file. To do so, we need to describe an audit that probes a few symbols of the outsourced file and accepts if it thinks that the corruptions can all be corrected. Recall that, in the CC definition, an audit is considered secure if it only succeeds when the outsourced file is retrievable in full, without modifications. If we want to derive PoR schemes from an LCC $\mathcal{C}$ in CC, we thus need to do the following three things :

1. Give an extraction procedure that aims at retrieving the outsourced file while correcting any corruption encountered.

2. Characterize the configurations of corruptions that are uncorrectable by this extraction procedure.

3. Give an audit procedure that is able to detect those configurations of uncorrectable corruptions on the outsourced file.

It is essential that the extraction procedure be indistinguishable from a sequence of audits. Otherwise, a malicious server could answer audits with uncorrupted symbols to make it accept. Then, this malicious server could only send corrupted data during the extraction process to make it fail. Since a good PoR scheme must have low communication complexity, we want to exploit the locality of LCCs to design our audit procedure. The remark above means that our extraction procedure must be an iteration of the local correction algorithm of the LCC. This means that our schemes will try to locally correct any corruption encountered during the extraction. Thus, we need a way to identify those corruptions. Using the composability of the CC framework, we will place ourselves in a setting where adversaries can only introduce erasures on the outsourced file. We can design our PoR schemes with this assumption and we will need to construct an authenticated server to realize it later on. Our blueprint becomes :

1. Give an extraction procedure that aims at correcting erasures by using the local correctability of $\mathcal{C}$.

2. Characterize the configurations of erasures that are uncorrectable by this extraction procedure.

3. Our audit is the following : try to locally correct a random position of the outsourced file, if the correction is impossible return `reject`, else return `accept`.

In step 2, we identify the configurations of erasures that are unrecoverable when iterating the local correction of $\mathcal{C}$. We find a lower bound on the number of local correction queries that would fail if such a configuration of erasures existed. When instantiating our framework in sec. 5, we shall see that this problem is much easier than giving a lower bound on the minimal size of such a configuration of unrecoverable erasures. In the CC model of security for PoRs, the advantage of the adversary in breaking the security of the scheme is the probability that the audit accepts when the file is not retrievable. In our case, our audit consists in checking if a random local correction query succeeds. Our file is not retrievable if there exists a configuration of unrecoverable erasures. Thus, the lower bound we computed above is all we need to assess the security of the PoR. We give a complete proof when instantiating our framework, see th. 2 of sec. 5.

4 Our authentication protocol

In the rest of this work, we focus on schemes based on erasure capabilities of error correcting codes. Thus, we need a setting where the actions of adversaries only lead to introducing erasures, instead of errors, in the outsourced data. This is exactly what an authentic server-memory resource (**aSMR**) achieves since the adversary can only delete data or restore previously deleted data. Thus, we need a protocol that constructs an **aSMR** from a basic **SMR**.

In [1], Badertscher and Maurer present a protocol that constructs an **aSMR** using a MAC function, timestamps and a tree structure on the outsourced data. Their construction of the **aSMR** has the following features :

1. The **aSMR** of size n with alphabet Σ is constructed from an **SMR** of size $2n-1$, alphabet $\Sigma \times \mathbb{Z}_q \times \mathcal{T}$ and a local memory of constant size for the clients. $\mathcal{T}$ is the tag space of the MAC function used.

2. To read or write one memory cell on the **aSMR**, the protocol of [1] produces $O(\log n)$ `read` and `write` queries to the **SMR**.

Our work focuses on PoR schemes where clients upload a very large encoded file to an outsourced server. In this context, the logarithm of the size of the alphabet Σ is an order of magnitude smaller than the length of the MAC tags. The **aSMR** construction of [1] is not suited for this kind of application. Its issues are threefold. First, since the file size is huge, a factor of 2 in the storage overhead is a big problem. Second,

the $O(\log n)$ communication complexity for write operations is of no use to us since we will be working on encoded data and updating a codeword requires anyway to read a linear number of symbols. Third, the verification phase of PoRs often consists in probing as few symbols as possible to ensure that the outsourced file is retrievable in full. Having a $O(\log n)$ read communication complexity is a problem in this context.

With these observations, we now present a different protocol that constructs an **aSMR** with good features for our context :

1. The **aSMR** of size n with alphabet Σ is constructed using an **SMR** of size n, alphabet $\Sigma \times \mathcal{T}$ and a local memory of constant size for the clients.
2. A `write` request on our **aSMR** produces at most $2n-1$ `read` and `write` requests on the **SMR**.
3. A `read` request to the **aSMR** produces one `read` request to the **SMR**.

This way, we minimize the storage overhead and the communication complexity of read requests on the one hand. On the other hand, the increased communication complexity for write requests does not matter since we will be using codes to build our PoR scheme. Our new construction is described in theorem 1.

Theorem 1. *Let $k, n \in \mathbb{N}$ and let $\Sigma_1 := \Sigma \times \mathcal{T}$ for some alphabet Σ. The protocol $\mathsf{auth} := (\mathsf{init}_{\mathsf{auth}}, \mathsf{auth}_{RW}, \ldots, \mathsf{auth}_{RW})$ (with k copies of auth_{RW}) described in fig. 8 and 9 constructs the authentic SMR, say $\mathbf{aSMR}^k_{\Sigma,n}$, from $\mathbf{SMR}^k_{\Sigma_1,n}$ and a local memory $\mathbf{L}$ of constant size with respect to the simulator $\mathsf{sim}_{\mathsf{auth}}$ described in fig. 10 and the pair $(\mathsf{honSrv}, \mathsf{honSrv})$. More precisely, for all distinguisher $\mathbf{D}$, we have*

$$\Delta^{\mathbf{D}}(\mathsf{honSrv}^{\mathsf{S}}\mathsf{auth}_{\mathcal{P}}[\mathbf{L}, \mathbf{SMR}^k_{\Sigma_1,n}], \mathsf{honSrv}^{\mathsf{S}}\,\mathbf{aSMR}^k_{\Sigma,n}) = 0$$

$$\textit{and} \quad \Delta^{\mathbf{D}}(\mathsf{auth}_{\mathcal{P}}[\mathbf{L}, \mathbf{SMR}^k_{\Sigma_1,n}], \mathsf{sim}^{\mathsf{S}}_{\mathsf{auth}}\,\mathbf{aSMR}^k_{\Sigma,n}) \leq \Gamma^{\mathbf{DC}}(\mathbf{G}^{\mathsf{MAC}})$$

Proof. A description of the converters, simulator and a full proof can be found in app. E. □

5 Instantiation with high rate LCCs

5.1 Lifted Reed-Solomon Codes

We introduce a very interesting class of LCCs, namely the high rate lifted Reed-Solomon codes of Guo *et al.* [5]. We recall their construction in app. F.

In the following, let $\mathbb{F}_q$ be the finite field with q elements and m be a positive integer. The set of affine lines in $\mathbb{F}_q^m$ is denoted by $\mathcal{L}_m := \{(at+b)_{t\in\mathbb{F}_q} \mid a, b \in \mathbb{F}_q^m\}$. $\mathrm{RS}_q[q,d]$ is the q-ary Reed-Solomon code of length q and minimum distance $d = q-k+1$.

Definition 4 (Lifted Reed-Solomon Code [5])**.** Let $\mathbb{F}_q$ be a finite field. Let $d, m \in \mathbb{N}^*$. The m-lift of $\mathrm{RS}_q[q,d]$ is $\mathrm{Lift}_m(\mathrm{RS}_q[q,d]) := \{w \in \mathbb{F}_q^m \mid \forall \text{ line } \ell \subseteq \mathbb{F}_q^m, w_{|\ell} \in \mathrm{RS}_q[q,d]\}$.

As we are using an **aSMR**, codewords can only be affected by potential erasures. A codeword of the base Reed-Solomon code $\mathrm{RS}_q[q,d]$ is the vector of evaluations of a polynomial f of degree strictly less than $k = n-d+1$. Thus, if there are at most $d-1$ erasures, we can always recover the codeword *i.e.* the polynomial f by interpolating on $k > \deg f$ points. Therefore, if we want to correct a coordinate $x \in \mathbb{F}_q^m$ of the code $\mathrm{Lift}_m(\mathrm{RS}_q[q,d])$ we can pick a random line going through x and run the aforementioned local decoding algorithm.

5.2 The lifted RS PoR scheme

In this section, we use our PoR framework to design a secure and efficient PoR scheme using lifted Reed-Solomon erasure codes. We call this scheme *lifted RS PoR scheme* for short. We build our PoR for an **aSMR** and then use the composability of CC. Since this server is authenticated, we only have to deal with potential erasures instead of errors.

Using the blueprint of sec. 3, we need to do the following three things :

1. Give a global decoding algorithm for lifted Reed-Solomon codes using their local correctability.
2. Characterize the configurations of erasures that are unrecoverable by this algorithm.
3. Give an audit procedure that looks like a local correction and is able to detect those configurations of uncorrectable corruptions on the outsourced file.

Let us start with the global decoding algorithm. For the lifted Reed-Solomon code $\text{Lift}_m(\text{RS}_q[q, m])$, our global decoder works as follows. For each erasure, the decoding algorithm corrects it by finding, if it exists, a line going through the erasure and containing less than $d-1$ other erasures (using interpolation as presented in sec. 5.1). If one or more erasures have been corrected during this step, the algorithm tries to correct the remaining erasures using the same method. Indeed, since some erasures where corrected, there exists lines with less erasures than before. If, during one iteration, no erasures have been corrected, the algorithm stops and returns the vector. We give a pseudo-code description of this algorithm in fig. 3.

Input: The encoded file V with potential erasures
Output: The encoded file $\tilde{F}$.
repeat
  $E := \emptyset$
  for an erased position $x \in \mathbb{F}_q^m$ **do**
    if there exists a line $\ell \subseteq \mathbb{F}_q^m$ going through x with strictly less than d erasures. **then**
      Use the global decoder of $\text{RS}_q[q, d]$ on the restriction of the file to ℓ.
      We have corrected all the erasures on that line, x included.
      $E = E \cup \{x\}$
      Modify V accordingly.
until $E = \emptyset$
return V

Figure 3: Our global decoding algorithm for lifted Reed-Solomon codes.

We now study the fail cases of the global decoding algorithm. Let $\text{Lift}_m(\text{RS}_q[q, d])$ be a lifted Reed-Solomon code. For an erased position $s \in \mathbb{F}_q^m$ to be unrecoverable, it is necessary that each line going through s possesses at least d erasures. However, it is not sufficient. Indeed, suppose that there exists a line ℓ going through s with exactly d erasures. If there exists an erasure position s' on the line ℓ and a line ℓ' going through s' with at most $d-1$ erasures then the symbol erased at position s' can be recovered using the $\text{RS}_q[q, d]$ decoder. Since, s' lies on ℓ, this means that ℓ now contains only $d-1$ erasures and they all can be corrected, the one at s included.

In order to capture a set of unrecoverable erasures for our global decoding algorithm, we introduce the following property :

Definition 5. Let $\mathbb{F}_q$ be a finite field and m, d be positive integers. We say that a set $S \subseteq \mathbb{F}_q^m$ is a *d-cover* set if S verifies the following property :

$$\forall s \in S, \forall \text{ line } \ell \subseteq \mathbb{F}_q^m \text{ going through } s, |S \cap \ell| \geq d$$

Or equivalently, for all line $\ell \subseteq \mathbb{F}_q^m, |S \cap \ell| = 0$ or $|S \cap \ell| >= d$

Since the d-cover subsets of $\mathbb{F}_q^m$ represent the unrecoverable erasure patterns, we want to find an audit procedure that can detect their existence with high probability and low communication complexity. Our audit procedure is the following :

1. The client randomly chooses a line $\ell \subseteq \mathbb{F}_q^m$.
2. The client retrieves the restriction of the outsourced file on the chosen line.

3. If it contains d or more erasures, reject, if not, accept.

The next step is to determine the probability that this audit detects a set of unrecoverable erasures if one exists. Let $S \subseteq \mathbb{F}_q^m$ be a non-empty d-cover set. Then there exists $s \in S$ and for each line ℓ going through s, $|\ell \cap S| >= d$. We also know that for any line $\ell \subseteq \mathbb{F}_q^m$, either $|\ell \cap S| = 0$ or $|\ell \cap S| >= d$.

Recall that $L := (q^m - 1)/(q-1)$ is the the number of lines going through a point in $\mathbb{F}_q^m$ and that $q^{m-1}L$ is the total number of lines in $\mathbb{F}_q^m$. Let ℓ be the randomly chosen line for the audit and s be an element of S. We have :

$$\Pr[|\ell \cap S| \neq 0] = \frac{L}{q^{m-1}L} \cdot 1 + \left(1 - \frac{1}{q^{m-1}}\right) \cdot \Pr[|\ell \cap S| \neq 0 \mid s \notin \ell]$$

Let E be the event $\{|\ell \cap S| \neq 0 \mid s \notin \ell\}$. For each point $x \in \ell$, there is a unique line (xs) going through x and s. Since $s \in S$, this line contains at least d erased points in S, one being s. Since lines in $\mathbb{F}_q^m$ have q points, the probability that $x \in S$ is at least $(d-1)/(q-1)$. Moreover, if at least $q-d+1$ points of ℓ do not belong to S we immediately know that $\ell \cap S = \emptyset$ since, by definition of S, either $|\ell \cap S| = 0$ or $|\ell \cap S| \geq d$. Thus,

$$\begin{aligned} \Pr[E] &\geq 1 - \left(1 - \frac{d-1}{q-1}\right)^{q-d+1} \\ \text{Therefore, } \Pr[|\ell \cap S| \neq 0] &\geq 1 - \left(1 - \frac{1}{q^{m-1}}\right)\left(1 - \frac{d-1}{q-1}\right)^{q-d+1} \end{aligned} \tag{1}$$

The calculation we just made is essential. Indeed, since we supposed $S \neq \emptyset$, the event $\neg\{|\ell \cap S| \neq 0\}$ can be interpreted as 'the audit accepts although the file is not retrievable'. In the CC security model for PoR, this is exactly the advantage of the distinguisher, *i.e.* the security of the scheme. In other words, we just upper-bounded the security of our PoR scheme. We now formally prove the security of our PoR in the CC framework.

We quickly describe the converters $\mathsf{init}_{\mathsf{audit}}$ and $\mathsf{audit}_{\mathsf{lcc}}$. Both use the encoder and global decoder for lifted Reed-Solomon codes. On input $(\texttt{read}, i)$, both converters retrieve the whole memory using `read` requests, then they call the global decoder on the obtained word (corrupted values ϵ are replaced with erasures) and return the i-th symbol of the output if decoding succeeds. On input $(\texttt{write}, i, x)$, both converters retrieve the whole memory with read requests and decode it like before. If decoding succeeds, they replace the i-th symbol by x, encode the whole word and store it on the SMR using `write` requests.

On input `audit`, $\mathsf{audit}_{\mathsf{lcc}}$ chooses a random line $\ell \subseteq \mathbb{F}_q^m$ and retrieves the restriction of the outsourced file to this line through `read` requests. If the restriction contains d or more erasures, the converter returns `reject`. If not, it returns `accept`. This behavior is depicted in app. G.

Theorem 2. *Let $d, m \in \mathbb{N}$, $\mathbb{F}_q$ be a finite field. The protocol* $\mathsf{audit} := (\mathsf{init}_{\mathsf{audit}}, \mathsf{audit}_{\mathsf{lcc}}, \ldots, \mathsf{audit}_{\mathsf{lcc}})$ *(with k copies of* $\mathsf{audit}_{\mathsf{lcc}}$*) for the lifted Reed-Solomon code* $\mathrm{Lift}_m(\mathrm{RS}_q[q,d])$ *of dimension ℓ constructs the auditable and authentic SMR, say* $\mathbf{aSMR}_{\Sigma,\ell}^{k,\mathsf{audit}}$*, from* $\mathbf{aSMR}_{\Sigma,q^m}^{k}$*, with respect to the simulator* $\mathsf{sim}_{\mathsf{audit}}$ *and the pair (*honSrv, honSrv*). More precisely, for all distinguisher* $\mathbf{D}$ *making at most r audits, we have*

$$\Delta^{\mathbf{D}}(\mathsf{honSrv}^{\mathsf{S}}\mathsf{audit}_{\mathcal{P}}\,\mathbf{aSMR}_{\Sigma,q^m}^{k}, \mathsf{honSrv}^{\mathsf{S}}\,\mathbf{aSMR}_{\Sigma,\ell}^{k,\mathsf{audit}}) = 0$$

$$\textit{and} \quad \Delta^{\mathbf{D}}(\mathsf{audit}_{\mathcal{P}}\,\mathbf{aSMR}_{\Sigma,q^m}^{k}, \mathsf{sim}_{\mathsf{audit}}^{\mathsf{S}}\,\mathbf{aSMR}_{\Sigma,\ell}^{k,\mathsf{audit}}) \leq r \cdot \left(1 - \frac{1}{q^{m-1}}\right)\left(1 - \frac{d-1}{q-1}\right)^{q-d+1}$$

Proof. A proof is given in app. H. □

5.3 The graph code PoR scheme

We give another instantiation of our framework using the graph codes of Tanner [18]. We briefly recall how these codes are constructed.

Let $G = (V, E)$ be a q-regular graph on n vertices. For a vertex $v \in V$, let $\Gamma(v)$ be the set of vertices adjacent to v. Let $\mathbb{F}$ be a finite field and let $\mathcal{C}_0 \subseteq \mathbb{F}^q$ be a linear code, called the *inner* code. Fix an arbitrary order on the edges incident to each vertex of G and let $\Gamma_i(v)$ be the i-th neighbor of v. A Tanner code is defined as the set of all labelings of the edges of G that respect the inner code $\mathcal{C}_0$. Formally,

Definition 6 (Tanner code)**.** Let $G = (V, E)$ be a q-regular graph on n vertices and let $\mathcal{C}_0 \subseteq \mathbb{F}^q$ be a linear code. The Tanner code $\mathcal{C}(G, \mathcal{C}_0) \subseteq \mathbb{F}^E$ is a linear code of length $nq/2$, so that for $c \in \mathbb{F}^E$, $c \in \mathcal{C}(G, \mathcal{C}_0)$ if and only if, for all $v \in V$,

$$\left(c_{(v,\Gamma_1(v))}, \ldots, c_{(v,\Gamma_q(v))}\right) \in \mathcal{C}_0$$

One can easily check, by counting constraints, that if $\mathcal{C}_0$ has rate R, then $\mathcal{C}(G, \mathcal{C}_0)$ has rate at least $2R-1$. These codes possess some sort of local correction. Indeed, if one wants to correct an edge e incident to a vertex v, one can retrieve the vector $(c_{(v,\Gamma_1(v))}, \ldots, c_{(v,\Gamma_q(v))})$ of labels of edges incident to v and correct it using the decoder of $\mathcal{C}_0$.

In the following, let d be the minimal distance of the inner code. Again, using the composability of the CC framework, we only have to deal with potential erasures. Following our framework of sec. 3, we start by sketching our global decoder. In the following, we say that an edge is *erased* when the label of that edge is erased. Similarly, we say that we *correct* an edge if we correct the label of that edge.

Assume that we want to correct an erasure on an edge e incident to a vertex v. If v is incident to less than $d-1$ erased edges, we can use the erasure decoding of $\mathcal{C}_0$ to correct all the edges incident to v, e included. Otherwise, v is incident to $k > d-1$ erased edges. Pick an erased edge incident to v. This edge is also incident to a vertex $v' \neq v$. If v' is incident to less than $d-1$ erased edges, we can correct them all and v is now incident to $k-1$ erased edges. If $k-1 \leq d-1$ we can correct the edge e. Else, we iterate the process on v and its neighbors.

Now, we have to characterize the configurations of erased edges that are unrecoverable for our decoding algorithm. We claim that these unrecoverable configurations correspond to subgraphs of G of minimal degree d. Indeed, these are the graph analogues of the d-cover sets for lifted Reed-Solomon codes. We prove our claim : suppose that the subgraph formed by the unrecoverable edges possesses a vertex v incident to less than $d-1$ unrecoverable edges. Then, by iterating the local decoding algorithm, we can recover the other edges incident to v so that only these unrecoverable edges remain erased. This being done, since there are less than $d-1$ erased edges incident to v and since the minimal distance of the inner code is d, we can correct all the edges incident to v using the decoder of the inner code. This is in contradiction with these edges being unrecoverable.

Finally, the audit consists in randomly choosing a vertex v and retrieve the vector $w := (c_{(v,\Gamma_1(v))}, \ldots, c_{(v,\Gamma_q(v))})$ of labeling of edges incident to v. If w contains d or more erasures, the audit rejects. Else, it accepts.

The security of the audit depends on the graph G and the minimal distance d of the inner code $\mathcal{C}_0$. The bigger the minimal subgraphs of G with minimal degree d are, the better the security of the PoR will be. Indeed, let s be the minimal size (number of vertices) of a subgraph of G with minimal degree d. For a configuration of unrecoverable erasures to exist, we thus need at least s vertices of G with at least d erased edges. Recall that our audit chooses a random vertex of G and accepts if and only if this vertex is incident to less than $d-1$ erased edges. Thus, the probability that our audit accepts when there exists an unrecoverable set of erased edges is less than $1 - s/|V|$. In our framework, this is exactly the advantage of the adversary in breaking the security of our PoR. A similar proof to the one of th. 2 yields the following theorem :

Theorem 3. *Let $G = (V, E)$ be a q-regular graph with $|V| := n$ and let $\mathcal{C}_0 \subseteq \mathbb{F}^q$ be a linear code with minimal distance d and rate R. Let s be the minimal size, number of vertices, of a subgraph of G with minimal degree d. The protocol* $\mathsf{audit} := (\mathsf{init}_{\mathsf{audit}}, \mathsf{audit}_{\mathsf{graph}}, \ldots, \mathsf{audit}_{\mathsf{graph}})$ *(with k copies of* $\mathsf{audit}_{\mathsf{graph}}$*) for a Tanner code $\mathcal{C}(G, \mathcal{C}_0)$ of length $nq/2$ and rate at least $2R-1$ that :*

1. *Starts by encoding the file and uploads it to the server.*
2. *On an* `audit` *request, chooses a random vertex $v \in V$ and accepts if and only if v is incident to less than $d-1$ erased edges.*

3. Extracts the file using the algorithm sketched above.

constructs the auditable and authentic SMR, say $\mathbf{aSMR}^{k,\mathsf{audit}}_{\mathbb{F},(2R-1)nq/2}$*, from* $\mathbf{aSMR}^{k}_{\mathbb{F},nq/2}$*, with respect to the simulator* $\mathsf{sim}_{\mathsf{audit}}$ *and the pair (*honSrv*,* honSrv*). More precisely, for all distinguisher* $\mathbf{D}$ *making at most* r *audits, we have*

$$\Delta^{\mathbf{D}}(\mathsf{honSrv}^{\mathsf{S}}\mathsf{audit}_{\mathcal{P}}\,\mathbf{aSMR}^{k}_{\mathbb{F},nq/2}, \mathsf{honSrv}^{\mathsf{S}}\,\mathbf{aSMR}^{k,\mathsf{audit}}_{\mathbb{F},(2R-1)nq/2}) = 0$$

$$\textit{and}\quad \Delta^{\mathbf{D}}(\mathsf{audit}_{\mathcal{P}}\,\mathbf{aSMR}^{k}_{\mathbb{F},nq/2}, \mathsf{sim}^{\mathsf{S}}_{\mathsf{audit}}\,\mathbf{aSMR}^{k,\mathsf{audit}}_{\mathbb{F},(2R-1)nq/2}) \leq r \cdot \left(1 - \frac{s}{n}\right)$$

6 Parameters

The impact of the choices of various lifted Reed-Solomon codes on the parameters of our lifted RS PoR scheme are highlighted in fig. 5. The grey line gives a choice of parameters with a storage overhead of 13.9% and total communication of 0.01% of the file size. Increasing the length q of the base code decreases the storage overhead and increasing the lifting parameter m vastly increases the size of the file stored. A summary of the exact values of the parameters of our PoR scheme can be found in fig. 4.

We now give a comparison between our parameters and the ones of Lavauzelle and Levy-dit-Vehel [12]. First, in both schemes, the client's file is encoded using a lifted Reed-Solomon code and the audit consists in probing the restriction of this codeword to a random affine line. In our case, we authenticate the data using our MAC based authentication protocol (see sec. 4) whereas [12] binds data to a specific location by using an encryption scheme. Let κ be the computational security parameter of both scheme and Σ be the alphabet of the code. Our scheme stores a code symbol along with a MAC tag, that is $\kappa + \log|\Sigma|$ bits, in each memory location of the server whereas [12] stores a ciphertext, that is κ bits, in each memory location. Since $\log|\Sigma| \ll \kappa$ (we have $\kappa = 128$ and $|\Sigma| = q$ in fig. 5), our scheme and the one of [12] have very close storage overhead and communication complexity.

A major advantage of our scheme is that our audit produces way less "false positives" than the audit of [12]. In the case of PoRs, a false positive occurs when an audit rejects while the file is still retrievable. In other words, the client thinks that he lost his file forever, but it is still retrievable in full. The number of false positive audits has no influence on the security of the PoR but, in practice, it is a situation that we absolutely wish to avoid. The audit of [12] rejects if the restriction of the file to an affine line does not belong to the base Reed-Solomon code. In other words, if there is at least one corruption on the line probed by the audit, it deems the file unretrievable. If the adversary introduces at least one erasure on every line of the space, the audit would always reject independently of the correction capability (*i.e.* the minimal distance) of the code. Using our framework and our authentication protocol, we are able to fix this problem. Indeed, our audit deems the file unretrievable only if the probed line contains at least d erasures, where d is the minimal distance of the base Reed-Solomon code. This means that we decrease the number of false positive audits to a minimum, making our scheme much more reliable and usable in practice.

One natural line of work is now to evaluate the efficiency of Tanner codes PoR according to different choices of inner codes and graphs.

	Exact value	Asymptotics
C. storage overhead	κ	$\mathcal{O}(1)$
S. storage overhead	$(\frac{1}{R} - 1)\lvert F\rvert + q^m\kappa$	$\mathcal{O}(\lvert F\rvert)$
C. $\rightarrow$ S.	$2m \log q$	$\mathcal{O}(\lvert F\rvert)$
S. $\rightarrow$ C.	$q(\kappa + \log q)$	$\mathcal{O}(\lvert F\rvert^{1/m})$

Figure 4: The exact parameters of our scheme. $|F|$ denotes the file size in bits, κ the security parameter of the MAC, q the field size and $m \geq 2$ the lifting parameter. We have $Rq^m \log q = |F|$.

PoR param.			Results					
m	q	d	$\|F\|$ (bits)	$\frac{1}{R}-1$	comm. C. $\to$ S. (bits)	comm. S. $\to$ C. (bits)	comm./$\|F\|$	Statistical Security
2	256	32	255003	1.056	32	2048	0.0081	2^{-42}
	512		1446533	0.631	36	4608	0.0032	2^{-43}
	1024		7441987	0.409	40	10240	0.0013	2^{-44}
	2048		36072982	0.279	44	22528	0.0006	2^{-44}
	4096		168474135	0.195	48	49152	0.0003	2^{-44}
	8192		765948403	0.139	52	106496	0.0001	2^{-44}
	1024	64	6389859	0.641	40	10240	0.0016	2^{-88}
	2048		32605896	0.415	44	22528	0.0007	2^{-89}
	4096		157041023	0.282	48	49152	0.0003	2^{-90}
	8192		728834780	0.197	52	106496	0.0001	2^{-90}

Figure 5: Different choices of lifted Reed-Solomon codes for our PoR scheme.

References

[1] Christian Badertscher and Ueli Maurer. Composable and robust outsourced storage. In *Topics in Cryptology - CT-RSA 2018 - The Cryptographers' Track at the RSA Conference 2018, San Francisco, CA, USA, April 16-20, 2018, Proceedings*, pages 354–373, 2018. doi:10.1007/978-3-319-76953-0_19.

[2] Kevin D. Bowers, Ari Juels, and Alina Oprea. Proofs of retrievability: Theory and implementation. In *Proceedings of the 2009 ACM Workshop on Cloud Computing Security*, CCSW '09, pages 43–54, New York, NY, USA, 2009. ACM. doi:10.1145/1655008.1655015.

[3] R. Canetti. Universally composable security: a new paradigm for cryptographic protocols. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*, page nil, - 2001. URL: https://doi.org/10.1109/sfcs.2001.959888.

[4] Yevgeniy Dodis, Salil Vadhan, and Daniel Wichs. Proofs of retrievability via hardness amplification. In *Proceedings of the 6th Theory of Cryptography Conference on Theory of Cryptography*, TCC '09, pages 109–127, Berlin, Heidelberg, 2009. Springer-Verlag. doi:10.1007/978-3-642-00457-5_8.

[5] Alan Guo, Swastik Kopparty, and Madhu Sudan. New affine-invariant codes from lifting. In *Proceedings of the 4th Conference on Innovations in Theoretical Computer Science*, ITCS '13, pages 529–540, New York, NY, USA, 2013. ACM. doi:10.1145/2422436.2422494.

[6] Brett Hemenway, Rafail Ostrovsky, and Mary Wootters. Local correctability of expander codes. In Fedor V. Fomin, Rūsiņš Freivalds, Marta Kwiatkowska, and David Peleg, editors, *Automata, Languages, and Programming*, pages 540–551, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[7] Daniel Jost and Ueli Maurer. *Overcoming Impossibility Results in Composable Security Using Interval-Wise Guarantees*, pages 33–62. Advances in Cryptology - CRYPTO 2020. Springer International Publishing, 2020.

[8] Daniel Jost, Ueli Maurer, and Marta Mularczyk. *A Unified and Composable Take on Ratcheting*, pages 180–210. Theory of Cryptography. Springer International Publishing, 2019.

[9] Ari Juels and Burton S. Kaliski, Jr. Pors: Proofs of retrievability for large files. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 584–597, New York, NY, USA, 2007. ACM. doi:10.1145/1315245.1315317.

[10] Jonathan Katz and Luca Trevisan. On the efficiency of local decoding procedures for error-correcting codes. In *Proceedings of the Thirty-second Annual ACM Symposium on Theory of Computing*, STOC '00, pages 80–86, New York, NY, USA, 2000. ACM. doi:10.1145/335305.335315.

[11] Swastik Kopparty, Shubhangi Saraf, and Sergey Yekhanin. High-rate codes with sublinear-time decoding. In *Proceedings of the Forty-third Annual ACM Symposium on Theory of Computing*, STOC '11, pages 167–176, New York, NY, USA, 2011. ACM. doi:10.1145/1993636.1993660.

[12] Julien Lavauzelle and Françoise Levy-Dit-Vehel. New proofs of retrievability using locally decodable codes. In *International Symposium on Information Theory ISIT 2016*, pages 1809 – 1813, Barcelona, Spain, 2016. doi:10.1109/ISIT.2016.7541611.

[13] Ueli Maurer. *Constructive Cryptography - A New Paradigm for Security Definitions and Proofs*, pages 33–56. Theory of Security and Applications. Springer Berlin Heidelberg, 2012.

[14] Ueli Maurer and Renato Renner. Abstract cryptography. In *In Innovations In Computer Science*. Tsinghua University Press, 2011.

[15] Ueli Maurer and Renato Renner. From indifferentiability to constructive cryptography and back. In *Proceedings, Part I, of the 14th International Conference on Theory of Cryptography - Volume 9985*, page 3–24, Berlin, Heidelberg, 2016. Springer-Verlag.

[16] Maura B. Paterson, Douglas R. Stinson, and Jalaj Upadhyay. A coding theory foundation for the analysis of general unconditionally secure proof-of-retrievability schemes for cloud storage. *Journal of Mathematical Cryptology*, 7(3):183–216, 2013. URL: https://doi.org/10.1515/jmc-2013-5002, doi:doi:10.1515/jmc-2013-5002.

[17] Hovav Shacham and Brent Waters. Compact proofs of retrievability. In Josef Pieprzyk, editor, *Advances in Cryptology - ASIACRYPT 2008*, pages 90–107, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[18] R. Tanner. A recursive approach to low complexity codes. *IEEE Transactions on Information Theory*, 27(5):533–547, 1981. doi:10.1109/TIT.1981.1056404.

A Background on Constructive Cryptography

A.1 Resources, Converters and Distinguishers

A *resource* $\mathbf{R}$ is a system that interacts, in a black-box manner, at one or more of its *interfaces*, by receiving an input at a given interface and subsequently sending an output at the same interface. Do note that a resource only defines the observable behavior of a system and not how it is defined internally. We use the notation $[\mathbf{R}_1, \ldots, \mathbf{R}_k]$ to denote the parallel composition of resources. It corresponds to a new resource and, if $\mathbf{R}_1, \ldots, \mathbf{R}_k$ have disjoint interfaces sets, the interface set of the composed resource is the union of those.

In CC, *converters* are used to link resources and reprogram interfaces, thus expressing the local computations of the parties involved. A converter is plugged on a set of interfaces at the inside and provides a set of interfaces at the outside. When it receives an input at its outside interface, the converter uses a bounded number of queries to the inside interface before computing a value and outputting it at its outside interface.

A converter π connected to the interface set $\mathcal{I}$ of a resource $\mathbf{R}$ yields a new resource $\mathbf{R}' := \pi^{\mathcal{I}}\mathbf{R}$. The interfaces of $\mathbf{R}'$ inside the set $\mathcal{I}$ are the interfaces emulated by π. A protocol can be modelled as a tuple of converters with pairwise disjoint interface sets.

A *distinguisher* $\mathbf{D}$ is an environment that connects to all interfaces of a resource $\mathbf{R}$ and sends queries to them. At any point, the distinguisher can end its interaction by outputting a bit. Its advantage is defined as $\Delta^{\mathbf{D}}(\mathbf{R}, \mathbf{S}) := |\Pr[\mathbf{D}(\mathbf{R}) = 1] - \Pr[\mathbf{D}(\mathbf{S}) = 1]|$.

In this work, we make statements about resources with interface sets of the form $\mathcal{I} := \mathcal{P} \cup \{\mathsf{S}, \mathsf{W}\}$, where $\mathcal{P} := \{\mathsf{C}_0, \ldots, \mathsf{C}_k\}$ is the set of honest client interfaces. A protocol is a vector of converters $\pi :=$

$(\pi_{I_1}, \ldots, \pi_{I_{|\mathcal{P}|}})$ that specifies one converter for each interface $I \in \mathcal{P}$. The goal of this protocol is to construct a so-called ideal resource from an available real resource in presence of a potentially dishonest server S. The world interface W models the direct influence of a distinguisher on a resource.

A.2 Specifications and Relaxations

An important concept of CC is the one of *specifications*. Systems are grouped according to desired or assumed properties that are relevant to the user, while other properties are ignored on purpose. A specification $\mathcal{S}$ is a set of resources that have the same interface set and share some properties, for example confidentiality. In order to construct this set of confidential resources, one can use a specification of assumed resources $\mathcal{R}$ and a protocol π, and show that the specification $\pi\mathcal{R}$ satisfies confidentiality. Proving security is thus proving that $\pi\mathcal{R} \subseteq \mathcal{S}$, sometimes written as $\mathcal{R} \xrightarrow{\pi} \mathcal{S}$, and we say that the protocol π constructs the specification $\mathcal{S}$ from the specification $\mathcal{R}$. The composition property of the framework comes from the transitivity of inclusion. Formally, for specifications $\mathcal{R}, \mathcal{S}$ and $\mathcal{T}$ and protocols π for $\mathcal{R}$ and π' for $\mathcal{S}$, we have $\mathcal{R} \xrightarrow{\pi} \mathcal{S} \wedge \mathcal{S} \xrightarrow{\pi'} \mathcal{T} \Rightarrow \mathcal{R} \xrightarrow{\pi' \circ \pi} \mathcal{T}$.

We use the real-world/ideal-world paradigm, and often refer to $\pi\mathcal{R}$ and $\mathcal{S}$ as the real and ideal-world specifications respectively, to understand security statements. Those statements say that the real-world is "just as good" as the ideal one, meaning that it does not matter whether parties interact with an arbitrary element of $\pi\mathcal{R}$ or one of $\mathcal{S}$. This means that the guarantees of the ideal specification $\mathcal{S}$ also apply in the real world where an assumed resource is used together with the protocol.

In this work, we use *simulators*, *i.e.*, converters that translate behaviors of the real world to the ideal world, to make the achieved security guarantees obvious. For example, one can model confidential servers as a specification $\mathcal{S}$ that only leaks the data length, combined with an arbitrary simulator σ, and show that $\pi\mathcal{R} \subseteq \sigma\mathcal{S}$. It is then clear that the adversary cannot learn anything more that the data length.

In order to talk about computational assumptions, post-compromise security or other security notions, the CC framework relies on *relaxations* which are mappings from specifications to larger, and thus weaker, *relaxed specifications*. The idea of relaxation is that, if we are happy with constructing specification $\mathcal{S}$ in some context, then we are also happy with constructing its relaxed variant. One common example of this is computational security. Let ϵ be a function that maps distinguishers $\mathbf{D}$ to the winning probability, in $[0, 1]$, of a modified distinguisher $\mathbf{D}'$ (the reduction) on the underlying computational problem. Formally,

Definition 7. Let ϵ be a function that maps distinguishers to a value in $[0, 1]$. Then, for a resource $\mathbf{R}$, the reduction relaxation $\mathbf{R}^\epsilon$ is defined as $\mathbf{R}^\epsilon := \{\mathbf{S} \mid \forall \mathbf{D}, \Delta^{\mathbf{D}}(\mathbf{R}, \mathbf{S}) \leq \epsilon(\mathbf{D})\}$. This (in fact any) relaxation can be extended to a specification $\mathcal{R}$ by defining $\mathcal{R}^\epsilon := \cup_{\mathbf{R} \in \mathcal{R}} \mathbf{R}^\epsilon$.

B Basic Server Memory Resource

The basic SMR is described in fig. 6.

C The auditable and authentic SMR

The auditable and authentic SMR of [1] is described in fig. 7.

D Message Authentication Codes in CC

Our protocols will use message authentication codes (MAC). Thus, we recall the notation along with a description of the security condition for MAC given in [1]. We consider MAC functions f with message space $\mathcal{M}$, tag space $\mathcal{T}$ and key space $\mathcal{K}$ with its associated distribution. The security condition for MAC function f states that no efficient adversary can win the following game $\mathbf{G}^{\mathsf{MAC}}$ better than with negligible probability. In CC, games are represented as resources. In our case, the game $\mathbf{G}^{\mathsf{MAC}}$ chooses a secret key $sk \leftarrow\!\!\!\!\!\!\leftarrow \mathcal{K}$. Then,

Resource $\mathrm{SMR}^{k}_{\Sigma,n}$

Initialization Initialization

INIT, ACTIVE, INTRUSION $\leftarrow$ `false`
HIST $\leftarrow$ [ ]

Interface Interface C_0

Input: `init`
  if not INIT **then**
    for $i = 1$ to n **do**
      $\mathbb{M}[i] \leftarrow \lambda$
    HIST $\leftarrow$ HIST $||$ $(0, \texttt{init})$
    INIT $\leftarrow$ `true`

Input: $(\texttt{read}, i) \in [1, n]$
  if INIT **and not** ACTIVE **then**
    HIST $\leftarrow$ HIST $||$ $(0, \texttt{R}, i)$
    return $\mathbb{M}[i]$

Input: $(\texttt{write}, i, x) \in [1, n] \times \Sigma$
  if INIT **and not** ACTIVE **then**
    HIST $\leftarrow$ HIST $||$ $(0, \texttt{W}, i, x)$
    $\mathbb{M}[i] \leftarrow x$

Input: `initComplete`
  ACTIVE $\leftarrow$ `true`

Interface Interface S_H

Input: `getHist`
  return HIST

Input: $(\texttt{read}, i) \in [1, n]$
  return $\mathbb{M}[i]$

Interface Interface S_I

Input: $(\texttt{write}, i, x) \in [1, n] \times \Sigma$
  if INTRUSION **then**
    return $\mathbb{M}[i] \leftarrow x$

Interface Interface W

Input: `startWriteMode`
  if ACTIVE **then**
    INTRUSION $\leftarrow$ `true`

Input: `stopWriteMode`
  if ACTIVE **then**
    INTRUSION $\leftarrow$ `false`

Interface Interface $\mathsf{C}_t, t \in \{1, \ldots, k\}$

Input: $(\texttt{read}, i) \in [1, n]$
  if ACTIVE **and not** INTRUSION **then**
    HIST $\leftarrow$ HIST $||$ $(t, \texttt{R}, i)$
    return $\mathbb{M}[i]$

Input: $(\texttt{write}, i, x) \in [1, n] \times \Sigma$
  if ACTIVE **and not** INTRUSION **then**
    HIST $\leftarrow$ HIST $||$ $(t, \texttt{W}, i, x)$
    $\mathbb{M}[i] \leftarrow x$

Figure 6: Description of the basic server-memory resource

Resource aSMR$^{k,\mathsf{audit}}_{\Sigma,n}$

Interfaces Interface $\mathsf{C}_r, r \in 1, \ldots, k$

Input: $(\texttt{write}, i, x) \in [n] \times \Sigma$

Defined as in **aSMR** except the version number *ctr* has been removed.

Input: audit

if ACTIVE **and not** INTRUSION **then**
  output `auditReq` to S_H
  Let $d \in \{\text{allow}, \text{abort}\}$ be the result
  if $d = \text{allow}$ **then**
    $\mathbb{M}' \leftarrow [\,]$
    for $i = 1$ to n **do**
      if $\exists k, x, t : \text{HIST}[k] = (t, \texttt{W}, i, x)$ **then**
        $k_0 \leftarrow \max\{k \mid \exists t, x : \text{HIST}[k] = (t, \texttt{W}, i, x)\}$
        Parse $\text{HIST}[k_0]$ as $(t, \texttt{W}, i, x_0)$
        $\mathbb{M}'[i] \leftarrow x_0$
      else
        $\mathbb{M}'[i] \leftarrow \lambda$
    if $\mathbb{M}' = \mathbb{M}$ **then**
      return `accept`
    else
      return `reject`
  else
    return `reject`

Interfaces Interface S_I

Input: $(\texttt{restore}, i) \in [n]$

if INTRUSION **then**
  if $\exists k, x, t : \text{HIST}[k] = (t, \texttt{W}, i, x)$ **then**
    $k_0 \leftarrow \max\{k \mid \exists t, x : \text{HIST}[k] = (t, \texttt{W}, i, x)\}$
    Parse $\text{HIST}[k_0]$ as $(t, \texttt{W}, i, x_0)$
    $\mathbb{M}[i] \leftarrow x_0$
  else
    $\mathbb{M}[i] \leftarrow \lambda$

Figure 7: Description of the auditable and authentic SMR of [1] (only the differences with our **aSMR** are shown)

it acts as a signing oracle by receiving messages $m \in \mathcal{M}$ at its interface and responding with $f_{sk}(m)$. At any point, the adversary can make a forging attempt by providing a message m' and a tag t' to the game. The game is won if and only if $f_{sk}(m') = t'$ and m' has never been signed by the game before. The probability of adversary $\mathbf{A}$ to win the game is denoted by $\Gamma^{\mathbf{A}}(\mathbf{G}^{\mathsf{MAC}})$.

E Protocol and proof for our aSMR construction

In the following, let n be the size of the **SMR**, $f_{sk}(\cdot)$ be a MAC function with tag space $\mathcal{T}$ and Σ be a finite alphabet. We recall how to model MAC in CC in app. D. The clients will also have read and write access to a local memory resource denoted by $\mathbf{L}$. The protocol starts with the clients choosing a secret key sk for the MAC function, setting a version number ctr to 0 and storing both of them in their local memory $\mathbf{L}$. The main idea is the following : if the i-th cell is supposed to store the data $x \in \Sigma$, the protocol will store the tuple $(x, f_{sk}(x, ctr, i)) \in \Sigma \times \mathcal{T}$ instead. Do note that the version number ctr is incremented with every write request. This also means that every valid tag needs to be updated with every write request. Intuitively, this protocol prevents the adversary from :

1. Replacing the data x with $y \neq x$ since this would make the tag invalid.
2. Moving the data stored in location i to location $j \neq i$ since this would make the tag invalid.
3. Replaying an older value since the version numbers would not match and the tag would thus be invalid.

The protocol is formally depicted in figure 9, as a converter auth_{RW} to be plugged at the clients interface. The generation of cryptographic keys and the initialization of the local and outsourced memories are presented in the initialization converter $\mathsf{init}_{\mathsf{auth}}$, in figure 8.

Converter $\mathsf{init}_{\mathsf{auth}}$

Interface Interface `out`

Input: `init`
  $sk \twoheadleftarrow \mathcal{K}, ctr \leftarrow 0$
  output $(\texttt{write}, 1, sk)$ at interface `in` of $\mathbf{L}$
  output $(\texttt{write}, 2, ctr)$ at interface `in` of $\mathbf{L}$
  output `init` at interface C_0 of **SMR**
  for $i = 1$ **to** n **do**
    $v \leftarrow (\lambda, f_{sk}(\lambda, i, ctr))$
    output $(\texttt{write}, i, v)$ at interface C_0 of **SMR**

Input: $(\texttt{read}, i)$ or $(\texttt{write}, i, x)$
  Defined in the same way as for auth_{RW} in figure 9

Input: `initComplete`
  output `initComplete` at interface C_0 of **SMR**

Figure 8: The initialization protocol $\mathsf{init}_{\mathsf{auth}}$ for the construction of our **aSMR**.

Proof. We adapt the proof of [1] to our protocol and **aSMR** resource.

The first condition, called *availability*, makes sure that the protocol correctly implements the required functionality even when the adversary is not present. This rules out absurd constructions : for example,

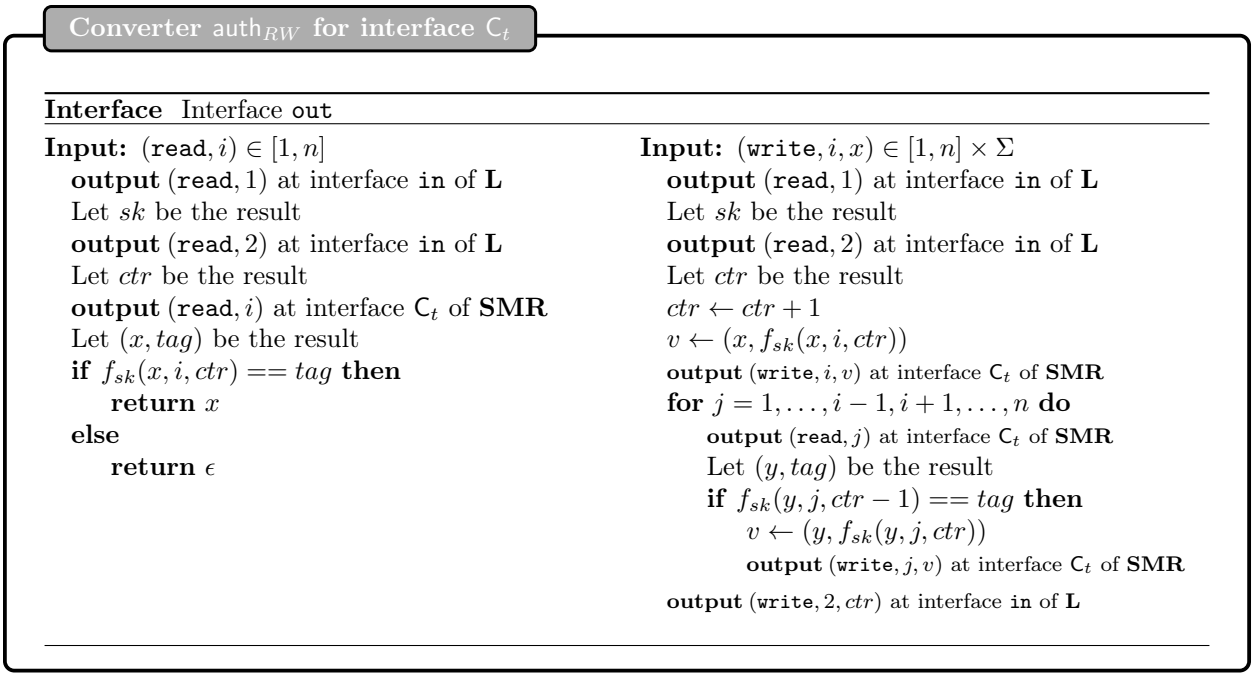

Figure 9: The protocol auth_{RW} for the construction of our **aSMR**.

in the outsourced storage setting, one could provide confidentiality to outsourced files by never outsourcing them in the first place. The protocol would be secure since the server would not be able to learn anything about the file but it would not meet the availability condition. To prove the availability of our protocol, we introduce the special converter honSrv which, when plugged in interface S, prevents the server from interfering with the client protocol and generating queries at its interface.

We briefly recall the second condition, called *security*, illustrated in figure 1. Recall that we use the real world-ideal world paradigm and prove that whatever the adversary can do on the real resource equipped with the protocol, it could as well do on the ideal resource using a simulator. Since the ideal resource is secure by definition, there are no successful attacks possible on the real resource equipped with the protocol.

The correctness condition is obvious for the auth protocol so we only prove the security condition. We analyze the behavior of the real and the ideal systems on every possible input at their interfaces.

Upon $\mathtt{init}, \mathtt{initComplete}$ **at interface** C_0**:** Upon the $\mathtt{init}$ query, the real system $\mathsf{auth}_{\mathcal{P}}[\mathbf{L}, \mathbf{SMR}^k_{\Sigma_1,n}]$ samples a new MAC key and initializes the version number ctr with the value 0. Then, it writes the value $(\lambda, f_{sk}(\lambda, i, ctr))$ at location i, for $i \in [1, n]$ and λ a fixed value in Σ. This adds $n + 1$ entries to the history which reads $(0, \mathtt{init}) \,||\, (0, \mathtt{W}, 1, f_{sk}(\lambda, 1, 0)) \,||\, \ldots (0, \mathtt{W}, n, f_{sk}(\lambda, n, 0))$.
In the ideal system $\mathsf{sim}^{\mathsf{S}}_{\mathsf{auth}}\, \mathbf{aSMR}^k_{\Sigma,n}$, the query initializes the memory with the value λ. The simulator samples a MAC key sk and initializes the version number ctr to 0. Then, it initializes its simulated history H_{sim} with $n + 1$ entries just like above.

Finally, on entry $\mathtt{initComplete}$ both systems deactivate interface C_0 and the other client interfaces become available.

Upon $(\mathtt{read}, i)$ **at interface** C_k**:** On this query, both protocols $\mathsf{init}_{\mathsf{auth}}$ (if $k = 0$) and auth_{RW} (if $k > 0$) read the i-th memory cell. The history is thus increased by the value $(k, \mathtt{R}, i)$. Then, the protocols check the validity of the tag. If they succeed, the last value written to this location x_i is returned. Otherwise, ϵ is returned.
In the ideal system, the **aSMR** directly returns the last value written in location i if this cell's content has not been deleted. Otherwise, ϵ is returned. The simulator emulates this view by simulating the memory of

Simulator $\mathsf{sim}_{\mathsf{auth}}$

Initialization Initialization

$sk \twoheadleftarrow \mathcal{K}, ctr \leftarrow 0$
$\mathbb{M}_{sim} \leftarrow (\lambda, f_{sk}(\lambda, 1, ctr)) \,||\, \ldots \,||\, (\lambda, f_{sk}(\lambda, n, ctr))$
$H_{sim} \leftarrow (0, \texttt{init}) \,||\, (0, \texttt{W}, 1, (\lambda, f_{sk}(\lambda, 1, ctr))) \,||\, \ldots \,||\, (0, \texttt{W}, n, (\lambda, f_{sk}(\lambda, n, ctr)))$
$pos \leftarrow |H_{sim}| + 1$

Interface Interface S_H

Input: `getHist`
UpdateLog
return H_{sim}
Input: $(\texttt{read}, i) \in [1, n]$
UpdateLog
return $\mathbb{M}_{sim}[i]$

Interface Interface S_I (Intrusion = `true`)

Input: $(\texttt{write}, i, (v, tag)) \in [1, n] \times (\Sigma \times \mathcal{T})$
UpdateLog
$\mathbb{M}_{sim}[i] \leftarrow (v, tag)$
Determine the last entry in H_{sim} that wrote value (v', tag') to location i.
if $v == v'$ and $tag == tag'$ **then**
  output $(\texttt{restore}, i)$ at interface S_I of **aSMR**
else
  output $(\texttt{delete}, i)$ at interface S_I of **aSMR**

procedure UpdateLog
  output `getHist` at interface `in` of **aSMR**
  Let Hist be the returned value
  for $j = pos$ to $|\text{Hist}|$ **do**
    if $\text{Hist}[j] = (k, \texttt{R}, i)$ **then**
      $H_{sim} \leftarrow H_{sim} \,||\, (k, \texttt{R}, i)$
    else if $\text{Hist}[j] = (k, \texttt{W}, i, x, ctr')$ **then**
      if $ctr' > ctr$ **then**
        $ctr \leftarrow ctr'$
        $\mathbb{M}_{sim}[i] \leftarrow (x, f_{sk}(x, i, ctr))$
        $H_{sim} \leftarrow H_{sim} \,||\, (k, \texttt{W}, i, (x, f_{sk}(x, i, ctr)))$
        for $\ell = 1$ **to** n, $\ell \neq i$ **do**
          $H_{sim} \leftarrow H_{sim} \,||\, (k, \texttt{R}, \ell)$
          $y, tag \leftarrow \mathbb{M}_{sim}[\ell]$
          if $tag == f_{sk}(y, \ell, ctr - 1)$ **then**
            $\mathbb{M}_{sim}[\ell] \leftarrow (y, f_{sk}(y, \ell, ctr))$
            $H_{sim} \leftarrow H_{sim} \,||\, (k, \texttt{W}, \ell, (y, f_{sk}(y, \ell, ctr)))$
  $pos \leftarrow |\text{Hist}| + 1$

Figure 10: The simulator of the construction of our **aSMR**.

the real world and sending `delete` (resp. `restore`) requests when the adversary writes values that would fail (resp. pass) the real world check. If this simulation is perfect, the behavior of the ideal system will be the same as the ideal one. We will discuss this when we analyze the `write` requests at interface S_I. Additionally, the next time the simulator is activated, it will update its simulated history H_{sim} using its procedure UpdateLog. If the read request $(k, \mathtt{R}, i)$ is the next entry in the history Hist of the **aSMR**, the simulator increases its simulated history with the value $(k, \mathtt{R}, i)$ which perfectly matches the real world behavior.

Upon $(\mathtt{write}, i, x)$ **at interface** C_k**:** On a write request, the protocols start by incrementing the version number ctr and writing the value x, together with its tag t, to location i. The value $(\mathtt{write}, i, (x, t))$ is thus appended to the history. Then, they read the content of each other location (in ascending order) and check their tag. If it is correct, a new tag is computed to account for the version number increase, and the value is written back together with the new tag. For $j = 1, \ldots, i-1, i+1, \ldots, n$, the history is increased by $(k, \mathtt{R}, j) \,||\, (k, \mathtt{W}, j, (x_j, tag_j))$ if the check succeeds and by $(k, \mathtt{R}, j)$ if the check fails.
In the ideal world, on a $(\mathtt{write}, i, x)$ request, the i-th memory cell is updated with the value x and $(k, \mathtt{W}, i, x, ctr)$ is appended to the history of **aSMR**, where ctr is the current version number. On the entry $(k, \mathtt{W}, i, x, ctr)$ of the history, UpdateLog will increase the simulated history and update its simulated memory with the values listed above, using the version number ctr and its simulated key sk to check and produce the appropriate tags. Thus, the simulated history and memory perfectly match the ones of the real world.

Upon `getHist` **at interface** S_H **:** In the real system, the output is the history of **SMR**. By the above analysis, an inductive argument shows that in the ideal system, the simulated history H_{sim}, which is returned upon this query, perfectly emulates the real-world one.

Upon $(\mathtt{write}, i, (x, tag))$ **at interface** S_I**:** In the real world, an adversarial write request is a simple replacement of the memory cell i of **SMR**. If (x, tag) corresponds to the last honest value written to this cell, then this cell might become valid again[3]. This is perfectly simulated in the ideal-world since the simulator can update its simulated memory and parse the history to check if (x, tag) is indeed the last honest value written to cell i. If it is the case, the simulator sends a $(\mathtt{restore}, i)$ request to **aSMR** which makes the cell valid again if it should be.
Now, let's study the case where the pair (x, tag) is not the same as the last honest one written to this cell. In the real world, the content of the memory cell i is just replaced. In the ideal world, the simulator sends a $(\mathtt{delete}, i)$ request to **aSMR** and assigns the value (x, tag) to the i-th cell of its simulated memory. If the pair (x, tag) fails the tag check, the simulation is perfect since the content of the i-th cell is deemed invalid in both worlds, making subsequent read requests return ϵ. However the bad event, denoted by BAD, occurs if the pair (x, tag) passes the verification (recall that this pair differs from the last honest value written to cell i). Indeed, in the real world, the check would succeed and the value x would be returned on subsequent read requests. Meanwhile, in the ideal world, the value of cell i would be deleted and ϵ would be returned on subsequent read requests. Hence, the real and ideal systems are identical until event BAD occurs.

Upon $(\mathtt{read}, i)$ **at interface** S_H**:** In the real system, this query returns the value at location i of **SMR**. In the ideal system, the value at location i of the simulated memory is returned instead. The simulator updates its simulated memory on each activation. As we discussed above, the simulated memory perfectly emulates the real one in all cases. The behavior of both worlds are thus identical.

We conclude that the real and ideal systems are identical until the BAD event occurs. The occurrence of BAD implies a successful forgery against the MAC function f_{sk}. We use the same reduction **C** as in [1] from a distinguisher **D** to an adversary $\mathbf{A} := \mathbf{DC}$ against the game[4] $\mathbf{G}^{\mathsf{MAC}}$. **C** simulates the real system, but evaluates the MAC function using oracle queries to $\mathbf{G}^{\mathsf{MAC}}$. If **D** issues a write query at interface S_I that provokes event BAD, **C** issues this value as a forgery to the game. Hence, we conclude by noting that

[3] if the version number has not increased
[4] defined in app.D

$$\begin{aligned}\Delta^{\mathbf{D}}(\mathsf{auth}_{\mathcal{P}}[\mathbf{L}, \mathbf{SMR}^k_{\Sigma_1,n}], \mathsf{sim}_{\mathsf{auth}^{\mathsf{S}}}\, \mathbf{aSMR}^k_{\Sigma,n}) &\leq \Pr^{\mathbf{D}(\mathsf{auth}_{\mathcal{P}}[\mathbf{L},\mathbf{SMR}^k_{\Sigma_1,n}])}[\mathsf{BAD}] \\ &\leq \Gamma^{\mathbf{DC}}(\mathbf{G}^{\mathsf{MAC}})\end{aligned}$$

□

F Lifted Reed-Solomon Codes

Let $\mathbb{F}_q$ be the finite field with q elements and S be any finite set. A function $f : S \to \mathbb{F}_q$ can be given by its vector of evaluations $(f(x))_{x\in S}$. With this notation, we see that $\{f : \mathbb{F}_q \to \mathbb{F}_q \mid \deg f < k\}$ is the q-ary Reed-Solomon code of length $n = q$ and minimum distance $d = q - k + 1$, denoted by $\mathrm{RS}_q[q, d]$.

Definition 8 (Affine-invariant code)**.** Let $\mathbb{F}_Q$ be an extension of $\mathbb{F}_q$. A code $\mathcal{C} \subseteq \{f : \mathbb{F}_Q^t \to \mathbb{F}_q\}$ is said to be affine-invariant if, for all affine permutations $T : \mathbb{F}_Q^t \to \mathbb{F}_Q^t$, we have $f \circ T \in \mathcal{C}$.

Guo *et al.* [5] then build a LCC in the following way :

Definition 9 (Affine Lifting [5])**.** Let $\mathcal{C} \subseteq \{f : \mathbb{F}_Q \to \mathbb{F}_q\}$ be an affine-invariant code. The m-th affine lift of $\mathcal{C}$ is the affine-invariant code :

$$\mathrm{Lift}_m(\mathcal{C}) = \{g : \mathbb{F}_Q^m \to \mathbb{F}_q \mid \forall \text{ affine injections } T : \mathbb{F}_Q \to \mathbb{F}_Q^m,\ g \circ T \in \mathcal{C}\}$$

We will denote the set of affine lines in $\mathbb{F}_q^m$ by $\mathcal{L}_m := \{(at + b)_{t\in\mathbb{F}_q} \mid a, b \in \mathbb{F}_q^m\}$.

In our case, we can choose a Reed-Solomon code $\mathcal{C} := \mathrm{RS}_q[q, d]$ and then lift it to a subset $\mathrm{Lift}_m(\mathcal{C})$ of $\{g : \mathbb{F}_q^m \to \mathbb{F}_q\}$, such that each restriction of functions in $\mathrm{Lift}_m(\mathcal{C})$ to an affine line in $\mathcal{L}_m$ belongs to $\mathrm{RS}_q[q, d]$. It is clear that this lifted code is locally correctable : to locally correct a codeword $c \in \mathrm{Lift}_m(\mathrm{RS}_q[q, d])$, one just restricts c to an affine line of $\mathcal{L}_m$ and uses the decoding algorithm of the base Reed-Solomon code to recover some symbols. Formally, we consider the following code :

Definition 10 (Lifted Reed-Solomon Code [5])**.** Let $\mathbb{F}_q$ be a finite field. Let $d, m \in \mathbb{N}^*$. The m-lift of the Reed-Solomon code $\mathrm{RS}_q[q, d]$ is the following code :

$$\mathrm{Lift}_m(\mathrm{RS}_q[q, d]) := \{w \in \mathbb{F}_q^m \mid \forall \text{ line } \ell \subseteq \mathbb{F}_q^m, w_{|\ell} \in \mathrm{RS}_q[q, d]\}$$

G Audit protocol of the auditable SMR

The audit protocol of the auditable **aSMR** is depicted in fig. 11.

H Proof of theorem 2

Proof. We prove the security condition. We only compare the behaviors of the audit of the real system (the **aSMR** with the protocol) and of the ideal one (the $\mathbf{aSMR}^{\mathsf{audit}}$ with the simulator). The reader can refer to [1] for a full proof. We describe the simulator $\mathsf{sim}_{\mathsf{auth}}$. It maintains a simulated memory, emulating the real world memory, using the history of the ideal resource. On $(\texttt{delete}, i)$, the simulator replaces the i-th entry of its simulated memory by ϵ. On $(\texttt{restore}, i)$, the simulator restores the content of the i-th entry of its simulated memory to the last value written here. The simulator maintains a simulated history using the ideal history of the $\mathbf{aSMR}^{\mathsf{audit}}$.

If, after a `delete` request, the set of corrupted locations of the simulated memory contains a d-cover subset of $\mathbb{F}_q^m$, the simulator deletes the whole ideal memory by sending `delete` requests to $\mathbf{aSMR}^{\mathsf{audit}}$. Similarly, if after a `restore` request, the set of corrupted locations of the simulated memory does not contain a d-cover subset of $\mathbb{F}_q^m$, the simulator restores the whole ideal memory by sending `restore` requests to $\mathbf{aSMR}^{\mathsf{audit}}$.

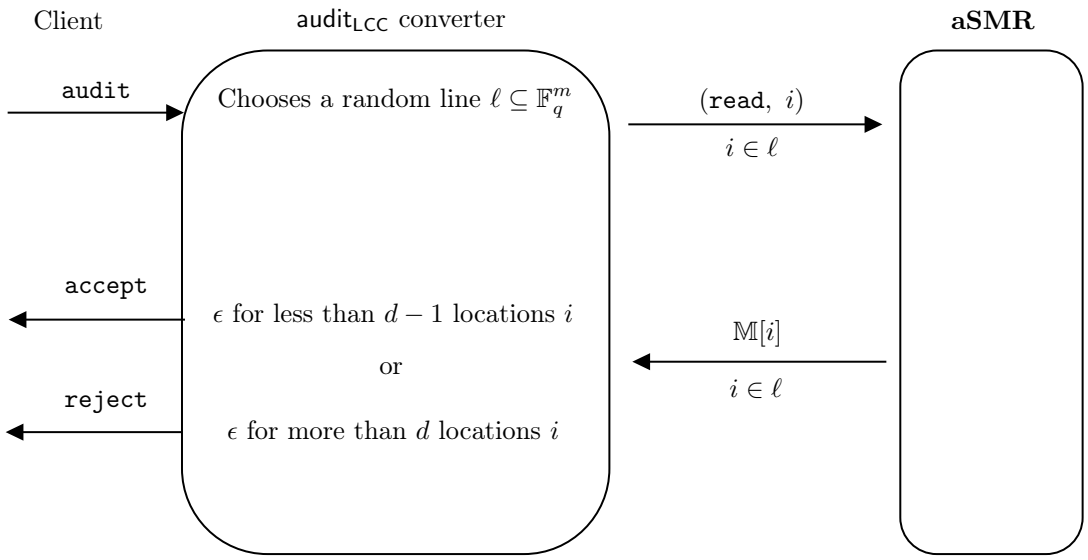

Figure 11: The audit mechanism for lifted Reed-Solomon codes (in the `allow` case)

On an `audit` request, the simulator chooses a random line $\ell \subseteq \mathbb{F}_q^m$, adds the entries $(\texttt{read}, i)$ for $i \in \ell$ to its simulated history. Then, if the restriction of its simulated memory to this line contains strictly less than d corrupted cells, the simulator sends `accept` to $\mathbf{aSMR}^{\mathsf{audit}}$. Else, it sends `reject`.

Upon `auditReq` at interface S_H : Recall that d-cover sets are the sets of unrecoverable erasures for our global decoder of lifted Reed-Solomon codes. Suppose that a subset of the corrupted cells forms a d-cover set. In order to run the audit, the converter chooses a random line $\ell \subseteq \mathbb{F}_q^m$, retrieves the restriction of the memory to this line through `read` requests and adds the corresponding entries to its simulated history. We showed, see equation 1, that the probability that this restriction contains strictly less than d erasures, *i.e.*, that the audit is successful, is less than

$$\left(1 - \frac{1}{q^{m-1}}\right)\left(1 - \frac{d-1}{q-1}\right)^{q-d+1}$$

The simulation is perfect unless the following BAD event occurs : *having simulated a real audit, the simulator answers* `allow` *(audit should succeed) whereas a d-cover subset of corrupted cells exists.* In that case, the simulator has chosen a restriction of the memory a line ℓ that contains strictly less than d corrupted cells, and has written the corresponding `read` requests to its simulated history. Note the the distinguisher has access to the simulated history. Then, the simulator outputs `allow` to the ideal resource, that runs the ideal audit. Since there exists a d-cover set of corrupted memory cells, the file is unretrievable so the ideal audit fails and the client receives `reject`. The distinguisher thus observes the following incoherence : `reject` is output while the (simulated) history contains the trace of a valid audit. The simulator concludes that it is interacting with the ideal system. We give a detailed explanation of the BAD event in app. I.

To sum up, the only observable difference from a distinguisher point of view lies in the audit procedure. The overall distinguishing probability is thus the one of distinguishing a real audit from a simulated one. As we saw, if the distinguisher runs r audits, this probability is less than $r \cdot (1-1/q^{m-1}) \cdot (1-(d-1)/(q-1))^{q-d+1}$, yielding the aforementioned result. □

I Detailed description of the BAD event

We here detail the interactions between the audit requests, the simulator, the distinguisher and the ideal resource, during an audit. It is very important to notice that the simulator is only connected to the interfaces

S_I and S_H of the server, and has no interaction with the clients. As `audit` is a functionality available at clients' interfaces, it follows that an audit request is not directly treated by the simulator. As described in fig.7, the following steps are done when the ideal resource $\mathbf{aSMR\_LCC}^{\mathsf{audit}}$ receives an `audit` request at one of its client interfaces C_j :

1. the resource sends `auditReq` at interface S_H.
2. via its interface S_H, the simulator either responds `allow` or `abort`.
3. if `allow`, the audit is run and the resource sends back to the client either `accept` or `reject`, depending on whether the ideal audit has succeeded or failed.
4. if `abort`, the ideal resource always sends `reject` back to the client.

Thus, in the proof, the simulator cannot directly control the outcome of the audit, in the sense that it cannot decide whether the ideal resource would send `accept` or `reject` back to the client. On the other hand, as it receives the `auditReq` request at interface S_H, the simulator can choose to answer `abort` or `allow` to the ideal resource. Let us now examine how the simulator behaves in both cases :

1. The simulator answers `abort` :
   It has received an audit request via `auditReq`. It then runs a simulation of a real audit (the one of the protocol) on its simulated memory: it chooses a set of t addresses, adds the corresponding `read` requests to its simulated history, and tests whether the values stored at those addresses (in its simulated memory) are valid ($\neq \epsilon$) and up-to-date. In this case, this test fails : at least one value is invalid, and the simulator is convinced that the real audit has to fail. It thus decides to answer the ideal resource `abort`. Consequently, the ideal resource does not run an ideal audit, but rather answers the client `reject`. The client is in fact controlled by the distinguisher in the ideal setting. Looking at the simulated history, it (the distinguisher) believes to interact with the real resource.
2. The simulator answers `allow`:
   It has received an audit request via `auditReq`. It then runs a simulation of a real audit (the one of the protocol) on its simulated memory: it chooses a set of t addresses, adds the corresponding `read` requests to its simulated history, and tests whether the values stored at those addresses (in its simulated memory) are valid ($\neq \epsilon$) and up-to-date. In this case, this test succeeds, all values are valid, and the simulator is convinced of the success of the real audit, as it simulated it with success. It thus sends `allow` to the ideal resource. The subtlety lies here : the ideal resource receives `allow` but it does not imply that it will send `accept` back to the client. It will run its ideal audit, and send the outcome of this audit to the client. The point is, when the simulator sends its answer (here `allow`) to the ideal resource, it already has simulated the real audit (here with success), and written the corresponding entries in its simulated history. Being not connected to the client's interface, the simulator does not "see" the result of the ideal audit (`accept` or `reject`) sent by the ideal resource to the client. Thus it cannot *a posteriori* modify its audit simulation to comply with the response of the ideal resource. The distinguisher, being connected to server and client's interfaces, has access to the ideal audit response and, by comparing it to the simulated history, can see the incoherence; (the trace of the audit in the simulated history corresponds to one which should fail). This incoherence never happens in the real resource, thus the distinguisher has distinguished between the two systems. Using equation 1, with probability $\leq (1 - 1/q^{m-1}) \cdot (1 - (d-1)/(q-1))^{q-d+1}$ the simulator makes a wrong simulation.