Dynamic Proofs of Retrievability from Chameleon-Hashes

Stefan Rass
Institute of Applied Informatics, Alpen-Adria Universitaet Klagenfurt,
Universitaetsstrasse 65-67, 9020 Klagenfurt, Austria

Keywords: Cloud Storage, Proofs of Retrievability, Data Availability, Security.

Abstract: Proofs of retrievability (POR) are interactive protocols that allow a verifier to check the consistent existence and availability of data residing at a potentially untrusted storage provider, e.g., a cloud. While most POR protocols strictly refer to static files, i.e., content that is read-only, dynamic PORs shall achieve the same security guarantees (existence, consistency and the possibility to retrieve the data) for content that is subject to an unlimited number of (legitimate) modifications. This work discusses how to construct such a dynamic proof of retrievability from chameleon hashes (trapdoor commitments). Like standard POR constructions, the presented scheme is sentinel-based and does audit queries via spot checking mechanism. Unlike previous schemes, however, a-posteriori insertions of new sentinels throughout the lifetime of the file is supported. This novel feature is apparently absent in any other POR scheme in the literature. Moreover, the system is designed for compatibility with XML structured data files.

1 INTRODUCTION

Proofs of retrievability (POR) have been introduced by (Juels and Kaliski, 2007) and (Lillibridge et al., 2003) as a tool to verify the existence and consistency of a remotely stored file. Having outsourced the file to a remote storage server implies that the verifier is no longer in possession of the actual data, yet uses a POR to verify that the stored information is still available and intact. The main challenge for a POR is to achieve this much more efficiently than the trivial approach of downloading the whole file. With the rise of cloud computing services, especially cloud storage, PORs have received lot of interest over the last years. Most POR protocols are designed to work with static files, i.e., the file structure and contents are assumed to remain unchanged over the lifetime of the file and any number of POR executions. Of much more practical interest are POR protocols that allow for changes (updates) to the stored file. These have evolved into their own line of research, called *dynamic* proofs of retreivability. While the construction of static POR protocols is rather straightforward, most known dynamic POR variants are relatively complex and come with strongly extended security models. This work shows a construction that naturally fits dynamic proofs of retrievability into the same security framework that applies for static PORs.

The terminology of the POR framework is strongly aligned to the vocabulary of interactive proof systems: we have the *verifier* $\mathcal{V}$, being the file owner who has given the data to a server for storage. The proof of retrievability is carried out between the verifier and the server, called the *prover* in this context. This prover is as well the potential adversary. The ”proof” is established by specifying a *knowledge extraction* algorithm, which unlike its abstract sibling in the zero-knowledge paradigm, has a quite simple physical interpretation for a POR: it is precisely the algorithm that ”downloads” the data whose existence has been assured a-priori by the interactive part of the POR (challenge-response cycles).

1.1 Related Work

Besides static and dynamic POR variants, related protocols can broadly be classified into *bounded-* and *unbounded-use schemes*, where the former allows only a limited (large) number of verifications over the lifetime of the file, as opposed to the latter. Bounded use protocols are sometimes called *keyless* schemes (e.g., (Juels and Kaliski, 2007)), where unbounded use schemes are also known as *keyed* (e.g., (Shacham and Waters, 2008; Xu and Chang, 2012), who in addition also provide protocols with public verifiability). The work of (Paterson et al., 2012) establishes a coding-theoretic foundation for static proofs of retrievability that unifies keyed and keyless schemes on

Rass S..
Dynamic Proofs of Retrievability from Chameleon-Hashes.
DOI: 10.5220/0004505102960304
In *Proceedings of the 10th International Conference on Security and Cryptography* (SECRYPT-2013), pages 296-304
ISBN: 978-989-8565-73-0

common grounds of error correcting encoding. In fact, it is shown that under the general framework of a challenge-response protocol which makes up part of every POR, error-correcting codes can be defined from a given POR. Conversely, such codes appear as a major building block of many known POR constructions, and even induce parts of the adversary model, if the attacker is considered as a noisy channel (Bowers et al., 2009b). It must be emphasized that error-correcting encoding appears more than advisable in order to cope with noisy channels, even though those are not part of the security and adversary model considered here. The construction in this work will not explicitly rest on any particular error-correcting code (ECC), besides applying an ECC for file storage and to be consistent with the standard definition of a POR.

The POR construction described in the following will be computationally secure. Unconditionally secure schemes for static files have been given recently (Dodis et al., 2009). Dynamic proofs of retrievability have been studied in (Zheng and Xu, 2011) for the first time, and subsequently in (Cash et al., 2012; Chen and Curtmola, 2012). The last reference adds the requirement of robustness, which demands recovery abilities from arbitrary amounts of corruptions within the data. This is traditionally achieved by forward error-correcting codes. Such best-practice security precautions are considered as implicitly done in the upcoming protocols, thus details are omitted for the sake of compactness. Most closely related to this work is (Wang et al., 2011), which as well employs Merkle-trees to update the file contents, but uses elliptic curve cryptographic primitives to do this, which is not required here.

Another closely related yet slightly weaker notion is *provable data possession* (Ateniese et al., 2007), which like POR comes in static and dynamic variants (Ateniese et al., 2008; Erway et al., 2009). However, and as recognized in the last reference, proving the *possession* property is weaker than proving the *retrievability* property, due to the extraction algorithm that a POR protocol prescribes but a PDP protocol does not need (although many PDP protocols do have a knowledge extractor prescribed implicitly by their security model definitions). Other related notions include proofs of storage (Ateniese et al., 2009) and proofs of ownership (POW) (Halevi et al., 2011). The latter may be viewed as a ”reverse” direction of a POR, where it is the verifier who ought to show the server that a file has originally been in his possession. For that reason, the security guarantees achieved by a POW are weaker than those of a POR.

Finally, it is worth noting that PORs have become a valuable building block in various recent cloud storage architecture proposals. See (Bowers et al., 2009a; Resch and Plank, 2011; Stefanov et al., 2012) to get started.

1.2 Contributions

Two mainstream constructions for a POR are known: using spot-checks or using homomorphy (cf. (Liu and Chen, 2011)). In the first variant, the verifier embeds sentinels in the file that he will later on challenge to verify the integrity of the file. The POR details mostly determine how to create and hide the sentinels in the file, so that the prover cannot precompute correct responses in advance. An example scheme in this class is (Juels and Kaliski, 2007), and those schemes are mostly bounded-use. The second line of construction uses homomorphic primitives (signatures, authenticators, etc.) to have the prover process the entire file content in order to correctly respond to a given challenge. Such schemes often use cryptographic keys for the processing, and are thus often unbounded-use. An example from this class is (Shacham and Waters, 2008).

The contribution in this work is the design of a scheme that falls into neither of these classes. The construction is essentially sentinel-based, but due to the dynamic update support lets us introduce new fresh sentinels over the lifetime of the file, hence a-posterior increase the number of possible challenges. It is therefore referred to as *quasi-bounded use* (although it is not entirely keyless). Moreover, the scheme is most straightforwardly suitable for XML file storage, and unlike other dynamic POR constructions, can align its data structure to the given file, rather than the other way around (as usual for dynamic POR).

The Construction in Brief: as in most POR schemes, the file owner (verifier) embeds sentinel data blocks in the file whose values are stored for subsequent verification by spot checking. The idea of the proposed scheme is to do these spot checks via requesting hash-values from the file host (prover), whilst allowing the blocks to be modified without altering the hash-values. This requires the verifier's ability to find hash-collisions, and hence the use of chameleon-hashes. Combining the latter with a conventional Merkle-hashtree construction then essentially creates a dynamic POR, with the unusual capability of allowing for a-posteriori sentinel embedding while the file resides at the storage provider already.

2 DEFINITIONS

A function $negl(t)$ is called *negligible*, if $negl(t) < 1/|p(t)|$ for every polynomial p and sufficiently large t. Concerning probabilities, we say that a value v is *overwhelming*, if $1-v$ is negligible. The notation $x\|y$ denotes an encoding of two strings (or general data items) x, y into a single string, from which a unique extraction of x and y is possible (with additional error-correction if needed). For a partitioning of a file F into blocks as $F = x_1\|\cdots\|x_n$, we refer to a single block as a *record* (in alignment with database terminology).

2.1 Structure of a POR

The structure of a POR, as used throughout this work, is a slight extension (and simplification) of the original POR definition of (Juels and Kaliski, 2007). The changes concern mostly the addition of the `update` procedure, and the omission of details on error-correcting encoding (justifications follow inline).

`Setup`: this algorithm takes a security parameter $t \in \mathbb{N}$ as input and initializes all cryptographic engines (hash-functions, encryptions) by outputting the respective public and secret parameters.

`Encode`: this algorithm takes a file $F = x_1\|\cdots\|x_n$ and encodes in a way that enables subsequent challenge-response verification cycles towards a proof of retrievability. The process at some stage involves error-correcting encoding to cover for channel noise (in (Bowers et al., 2009b), the adversary itself is viewed as a noisy channel, thus making the encoding the central duty of a POR protocol. However, this channel noise model may be questioned to precisely capture an active attacker that essentially does not act randomly). Error-correcting encoding is assumed to happen at the verifier's and/or prover's side, and further details on this stage are omitted (although this aspect is briefly revisited later). The output of `Encode` consists of two data items F^*, β, where F^* is the encoded file submitted to the prover for storage, and β comprises all information locally stored at the verifier's premises.

`Challenge`: this algorithm takes the current verifier's information β and outputs a challenge c_i and an expected response r_i.

`Verify`: the algorithm $\mathcal{V}_{\mathsf{verify}}$ checks a given challenge c against a response r. If successful, it outputs 1, and zero otherwise.

`Update`: this algorithm takes a record index i and new record data $\widetilde{x}_i$. It interacts with the prover to replace the existing record x_i with the new record $\widetilde{x}_i$, and outputs an updated version β' of the current verifier state β.

`Extract`: this algorithm takes the verifier's data β to compute a sequence of challenges $c_1, \ldots, c_n$, from whose respective responses $r_1, \ldots, r_n$ the file $F^{*\prime}$ can be reconstructed (downloaded). This part of a POR serves two purposes: first (and obviously), we must have some way of accessing the full lot of stored data from the prover. Second, and inspired by the construction of interactive proof systems, `extract` serves as a proof of knowledge for the prover to demonstrate the possession of the file. Notice that this function may as well execute update queries.

2.2 Chameleon Hashes

A *chameleon hash* (a.k.a. *trapdoor commitment*) acts as a normal hash-function, but allows for efficient construction of collisions if some secret trapdoor information is known. The structure will only be outlined and illustrated by an example. Full-fledged definitions and security proofs are available in (Ateniese and de Medeiros, 2005).

A chameleon-hash (in a simplified setting) consists of the following algorithms:

`KeyGen`: a probabilistic algorithm that takes a security parameter t and outputs a public/secret key-pair (pk, sk).

`Hashing`: a deterministic algorithm CH that uses the public-key pk to map a string $x \in \{0,1\}^*$ and an auxiliary random value r to a hash $CH_{pk}(m,r) \in \{0,1\}^\ell$ of fixed length ℓ (determined by the security parameter t).

`Forge`: a deterministic algorithm that takes the secret key sk, a pre-image (x,r) and its hash-value $CH_{pk}(x,r)$ to produce a second pre-image (y,s) such that $CH_{pk}(x,r) = CH_{pk}(y,s)$.

In a full-fledged definition (see (Ateniese and de Medeiros, 2005)), the construction of collisions is referred to as *universal forgery*, as opposed to the additional requirement of *instance forgery*, in which case we would be given two pre-images and ought to compute a third one with the same hash. Moreover, all of the above algorithms would additionally take some auxiliary inputs. This technical degree of freedom is not required in the following.

Security of a chameleon hash usually concerns collision-resistance but as well semantic security, message-hiding and key-exposure freeness.

The interested reader may consult (Ateniese and de Medeiros, 2005) for details, since the only property needed in the following is collision-resistance. For any probabilistic algorithm $\mathcal{A}$, a hash is said to be *collision-resistant*, if the likelihood of $\mathcal{A}$ to output a second pre-image upon given (x,r) and hash-value h is negligible in the security parameter. Formally, this conditional probability is denoted as

$$\mathbf{Succ}_2^{CH}(\mathcal{A}) := \Pr[CH_{pk}(y,s) = CH_{pk}(x,r) \mid (y,s) \leftarrow \mathcal{A}(x,r,pk)],$$

where the explicit dependence on $\mathcal{A}$ is omitted whenever this is clear from the context.

An example construction has been given in (Ateniese and de Medeiros, 2005):

KeyGen: Pick two large primes p,q such that $p = u \cdot q + 1$, and select a generator element g of the subgroup of squares of order q. Pick a random secret key $sk \in \{1,2,\ldots,q-1\}$ and define the public-key to be $pk = g^{sk} \text{MOD } p$. Choose a pre-image resistant cryptographic hash function $H : \{0,1\}^* \to \{0,1\}^\ell$ with $\ell \geq \lceil \log_2 p \rceil$.

Hashing: Choose two random values $\rho, \delta \in \mathbb{Z}_q$ and compute $e := H(m \| \rho)$ and define the Chameleon hash as

$$CH_{pk}(m,\rho,\delta) := \rho - (pk^e g^\delta \bmod p) \bmod q.$$

Forge: Let $C = CH_{pk}(m,\rho,\delta)$ be the known output for which we seek another pre-image. Pick an arbitrary value $m' \neq m$ and a random number $k \in \{1,2,\ldots,q-1\}$. Compute the values $\rho' = C + (g^k \bmod p) \bmod q$, $e' = H(m' \| \rho')$ and $\delta' \equiv k - e' \cdot sk \pmod q$. The sought collision is found at (m', ρ', δ'), since

$$\begin{aligned} CH_{pk}(m',\rho',\delta') &= \rho' - (pk^{e'} g^{\delta'} \bmod p) \bmod q \\ &= C + \left(g^k \bmod p\right) - \left(g^{sk \cdot e'} g^{\delta'} \bmod p\right) \bmod q \\ &= C = CH_{pk}(m,\rho,\delta). \end{aligned}$$

To ease notation, let us henceforth omit the explicit mentioning of auxiliary randomizers along with the hash input, and write $CH_{pk}(m)$ as a shorthand of $CH_{pk}(m,\rho,\delta)$, whenever the randomizers themselves are of no particular interest.

It is essential for this example hash function, as well as for the protocol presented in section 3, that parts of the pre-image constructed by `Forge` can be chosen freely. This makes the use of randomizers along with the hash input inevitable.

2.3 Merkle-Hashtrees

Merkle-hashtrees are a widely studied and standard hashing construction. It is worthwhile to briefly review the idea here, to draw attention to the particular fact that only $O(\log n)$ hashes are required to update a given hash, if one out of n blocks of the data is replaced. This is important in the following.

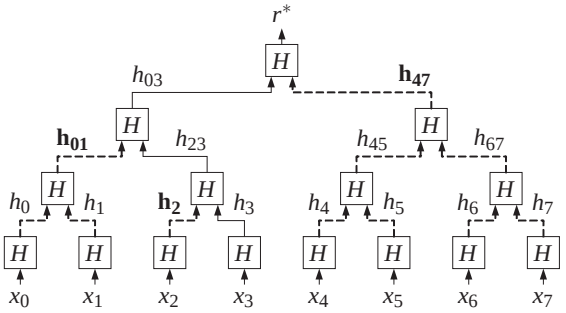

Figure 1: Merkle-tree example.

Assume the data F to be partitioned into n records as $F = x_1 \| \cdots \| x_n$. For brevity, let us assume that n is a power of two. Hashing is done by assigning the n blocks to a binary tree of height $O(\log n)$, where each inner node u is assigned the hash-value $H(v\|w)$, whenever v, w are child nodes of v (we associate the name of a node with its attached data item for simplicity). The root-hash is then computed recursively, starting from the leaf nodes that have the records $x_1, \ldots, x_n$ attached to them. Now, suppose that a single record x_i is replaced by $\widetilde{x_i}$. Then, updating the root-hash only requires updating the hashes along the path from $\widetilde{x_i}$ to the root. For that matter, we require the hashes of all sibling nodes along the path nodes, which gives a total of $O(\log n)$ hashes for a consistent change to the data and hash value. Figure 1 illustrates this for the case of eight records. The labels h_{ij} denote hashes ranging over sets of blocks $x_i, x_{i+1}, \ldots, x_j$. Assuming that we update x_3, we need only the values h_2, h_{01} and h_{47} (shown bold) along the path from x_3 up to the root, to update the overall hash r^*.

2.4 Adversary and Security Model

The original game-based security model and definition of (Juels and Kaliski, 2007) will be extended, since the POR construction will explicitly support modifications to the stored file. The adversary $\mathcal{A}$ is composed from two probabilistic algorithms $\mathcal{A}_{\text{setup}}$ and $\mathcal{A}_{\text{resp}}$. Algorithm $\mathcal{A}_{\text{setup}}$ interacts with an honest verifier $\mathcal{V}$ to initialize the POR system, and set up an archive storing a file F^* in first place. To this end, it is allowed to get challenges and updates from $\mathcal{V}$. The output of this phase is an archive F^* (held by the prover) and public parameters for the POR protocol. In the second phase, $\mathcal{A}_{\text{resp}}$ (as an oracle) responds to further challenges and updates issued by the verifier, before $\mathcal{V}$ finishes the experiment by extracting the file. We consider an attack as successful, if $\mathcal{V}$

extracts a file $F \neq F^*$. This model is formalized via two experiments, taking a security parameter t for the setup, and the system parameters α for the challenge-response phase.

Oracles for the verifier's functions challenge, update and verify are denoted as $\mathcal{V}_{\mathsf{chal}}$, $\mathcal{V}_{\mathsf{upd}}$ and $\mathcal{V}_{\mathsf{verify}}$. Oracle access to all of the verifier's functions is abbreviated as $\mathcal{A}^{\mathcal{V}}$. The symbol $\leftarrow_R$ denotes a uniformly random draw.

Experiment $\mathbf{Exp}^{\mathcal{A}}_{\mathsf{setup}}(t)$	Experiment $\mathbf{Exp}^{\mathcal{A}}_{\mathsf{chal}}(F^*,\alpha)$
$\kappa \leftarrow \mathsf{KeyGen}(t)$	action $\leftarrow_R \{\mathsf{chal},\mathsf{upd}\}$
$(F^*,\alpha) \leftarrow \mathcal{A}^{\mathcal{V}}_{\mathsf{setup}}$	$c \leftarrow \mathcal{V}_{\mathsf{action}}(\alpha)$
give α to $\mathcal{V}$	$r \leftarrow \mathcal{A}_{\mathsf{resp}}(F^*,\alpha)$
	output $\mathcal{V}_{\mathsf{verify}}(r,\alpha)$

Following the security model of (Juels and Kaliski, 2007), a POR is considered as secure, if any adversary succeeding in $\mathbf{Exp}^{\mathcal{A}}_{\mathsf{chal}}(F^*,\alpha)$ with overwhelming probability ($\geq 1-\zeta$) cannot trick the verifier into extracting something else than F^*. The success rate in $\mathbf{Exp}^{\mathcal{A}}_{\mathsf{chal}}(F^*,\alpha)$ is denoted as

$$\mathbf{Succ}^{\mathcal{A}}_{\mathsf{chal}}(F^*,\alpha) := \Pr\left[\mathbf{Exp}^{\mathcal{A}}_{\mathsf{chal}}(F^*,\alpha)=1\right].$$

Now, the *security game* is the following: the adversary $\mathcal{A}$ is assumed to host the file F^*, created during an execution of $\mathbf{Exp}^{\mathcal{A}}_{\mathsf{setup}}(t)$. The verifier $\mathcal{V}$ is given oracle access to $\mathcal{A}_{resp}$ and attempts to extract the file. The attacker wins if $\mathcal{V}$ extracts $F \neq F^*$. The probability for this not to happen is denoted as

$$\mathbf{Succ}^{\mathcal{A}}_{\mathsf{extract}}(F^*,\alpha) := \Pr\left[F = F^* | F \leftarrow \mathtt{extract}^{\mathcal{A}_{\mathsf{resp}}}(\alpha)\right].$$

Definition 2.1. *We call a POR* (ρ,λ)*-valid, if for some value* ζ *negligible in the security parameter* t*,*

$$\Pr\left[\begin{array}{c|l} \boldsymbol{Succ}^{\mathcal{A}}_{chal}(F^*,\alpha) \geq \lambda, & (F^*,\alpha) \leftarrow \boldsymbol{Exp}^{\mathcal{A}}_{\mathsf{setup}}(t), \\ \boldsymbol{Succ}^{\mathcal{A}}_{extract}(F^*,\alpha) < 1-\zeta & F \leftarrow \mathtt{extract}^{\mathcal{A}_{resp}}(\alpha) \end{array}\right] \leq \rho.$$

Intuitively, we seek a large value of λ and a small value of ρ. In that case, with a large likelihood $1-\rho$, either the file can be extracted with overwhelming probability, or the attacker is discovered by virtue of the challenge-response cycles.

3 THE CONSTRUCTION

The idea is closely related to how sanitizable signatures are designed; using Merkle-hashtrees and chameleon-hashes to construct a sentinel-based proof of retrievability. Like the Juels-Kasiski scheme, the protocol uses sentinels for spot checking, but unlike this previous proposal, those *are not embedded* in the file. Assume that the file is organized in a binary hash-tree, with leafs corresponding to data chunks, hereafter called *records*. Let the i-th such record be denoted by x_i, so that the file is $F = x_1\|x_2\|\ldots\|x_n$. For simplicity, let us assume that n is a power of two (to have the tree full) and think of the file F as an ordered set of records. Moreover, assume that the verifier has selected a (secret) subset $S \subseteq \{1,2,\ldots,n\}$ for subsequent POR-challenges.

Encoding. Assume that the file is encoded in an error-correcting fashion (see, e.g., (Juels and Kaliski, 2007) or (Bowers et al., 2009b) for detailed justifications), yielding a sequence of blocks, which we index by $i \in S$ again. Notice that the ECC is applied separately to each partition $x_1,\ldots,x_n$ of the file, in order to avoid invalidating parts of a code-word via a legitimate update operation. The `encode` algorithm chooses a challenge value c_i for each $i \in S$ and computes the root hash along the tree with the i-th record being concatenated with c_i, i.e., it hashes $x_i\|c_i$ in place of x_i to compute the expected correct response r_i for the challenge c_i on record $i \in S$. He stores the list of all r_i locally, along with the root-hash $r^* = CH_{pk}(F)$ of the original file F. The file F is then given (as is) to the prover (notice that no explicit sentinel information is embodied, as all verification data is stored locally at the verifier's side).

Challenges. The `challenge` algorithm picks a random record index (not necessarily one from S; reasons will follow below) and submits the challenge (i,c_i) to the prover (where c_i is random if $i \notin S$). The prover responds by re-computing the hash tree using the modified leaf $x_i\|c_i$, and returns the data record x_i and the hash-values of all sibling node's along the path from x_i up to the root.

Challenges on the same record cannot be used more than once, in order to prevent the server from learning correct responses to a particular record. However, for the sake of detecting a corruption more reliably, challenges should be repeated on different records during the same audit, i.e., POR execution. Extensions towards multiple queries on the same record are discussed in section 5.4.

Verifications. The `verify` algorithm uses the prover's provided hash-values to recompute the root hash r'_i and accepts if either $i \notin S$, or if $i \in S$ and the prover responds with $r'_i = r_i$.

Updates. Observe that we cannot straightforwardly replace a record x_i by $\widetilde{x}_i$, as this would instantly in-

validate all locally stored responses. Here comes the chameleon hash into play: first, the client queries the prover by running `challenge` to submit the pair (i,λ), where i is the record-index to be updated and λ is the value to be concatenated. If $i \in S$, then $\lambda = c_i$ (the known challenge), otherwise λ can be chosen randomly. The prover's response will consist of the "old" data item x_i, and additional verification information (if a record in S is updated, then $\mathcal{V}$ can do a verification, or otherwise skip this intermediate step). Then, in order not to invalidate other locally stored responses, the client uses his secret key *sk* to compute a collisions

$$CH_{pk}(x_i) = CH_{pk}(\widetilde{x}_i),$$

for the chameleon hash, so that all known root hashes r_j for all $j \in S$ remain intact. Here, let us assume that the collision $\widetilde{x}_i$ embodies the updated record contents, along with properly constructed auxiliary randomizers attached inside $\widetilde{x}_i$ to enforce the hash-collision (the example chameleon hash of section 2.2 permits this).

Embedding new sentinels: In case that the new record $\widetilde{x}_i$ shall be challenged subsequently, the verifier concatenates another fresh challenge value $\widetilde{c}_i$ to $\widetilde{x}_i$, and computes the new root hash $\widetilde{r}_i$ (by virtue of the verification information obtained previously for x_i) as the correct response to a potential future challenge. All of this happens locally (so the prover does not know about the existence of this new sentinel). Notice that the prover, although it knows that the hash-values for the old and new record are identical, cannot abandon the update, as the client may in future query exactly this modified record.

The scheme is thus called *quasi bounded-use*, as challenges that were consumed by `challenge` can be refreshed by `update`.

Extraction. The extract algorithm simply requests and error-corrects all records from the prover, and verifies the hash of the file in its current state against the locally stored root hash $r^* = CH_{pk}(F)$. In case of an adversary that does not respond deterministically (i.e., a probabilistic attacker), the same technique as in (Juels and Kaliski, 2007) can be applied: we first use the error-correcting encoding to correct as many errors as possible. If this recovery fails, then a block is requested multiple times, and a majority decoding is done. The analysis as done in (Juels and Kaliski, 2007) applies here as well, thus making the majority decoding work correctly, if a fraction strictly greater than $1/2$ can be retrieved correctly.

4 SECURITY AND EFFICIENCY

Unlike a security proof by reduction, the argument will not rest on an algorithm that breaks some cryptographic primitive using a breaking algorithm for the here presented scheme. Instead, the proof of theorem 4.1 is "direct".

Theorem 4.1. *The POR construction given in section 3 is* $(\rho, 1-|S|/|F|)$*-valid for* ρ *being negligible in the security parameter* t.

Proof. Define the events

$$A := \left\{ \mathbf{Succ}^{\mathcal{A}}_{\mathsf{extract}}(F^*, \alpha) < 1-\zeta \right\}$$

and

$$B := \left\{ \mathbf{Succ}^{\mathcal{A}}_{\mathsf{chal}}(F^*, \alpha) \geq \lambda \right\},$$

both of which are conditional on $[(F^*,\alpha) \leftarrow \mathbf{Exp}^{\mathcal{A}}_{\mathsf{setup}}(t)] \wedge [F \leftarrow \mathtt{extract}^{\mathcal{A}_{\mathsf{resp}}}(\alpha)]$. We show that the probability of $\neg A \cup \neg B$ is overwhelming ($\geq 1-\zeta$), so that the likelihood $\Pr[A \cap B]$ is negligible (less than ρ). We have $\Pr[\neg A \cup \neg B] = \Pr[\neg A] + \Pr[\neg B] - \Pr[\neg A \cap \neg B]$. The event $\neg A$ happens if and only if the verifier retrieves $F = F^*$ with overwhelming probability. By construction, however, `extract` checks the hash $CH_{pk}(F^*)$ against the known root hash $r^* = CH_{pk}(F)$. The event of acceptance upon $CH_{pk}(F) = CH_{pk}(F^*)$ for a corrupted file $F^* \neq F$ is nothing else than a hash-collision, whose occurrence is only negligibly probable for a cryptographic hash (as well as a Chameleon-hash, based on a collision-resistant hash). It follows that $\Pr[\neg A] \geq 1 - \mathrm{negl}(t)$.

Concerning the event $\neg B$, the attacker can in any case correctly respond to a fraction of at most $\lambda = 1-|S|/|F|$ challenges (as the prover has no expected responses stored for these blocks). So for this λ, we have $\Pr[\neg B] = 0$.

By Sklår's theorem, $\Pr[\neg A \cap \neg B]$ is expressible as $\Pr[\neg A \cap \neg B] = C(\Pr[\neg A], \Pr[\neg B])$ for some copula-function $C(x,y)$ that satisfies the upper Fréchet-Hoeffding bound $C(x,y) \leq \min\{x,y\}$. Hence, $\Pr[\neg A \cap \neg B] = 0$ because $\Pr[\neg B] = 0$ (intuitively and less technically, the intersection of two sets cannot be larger than either of the two). The proof is complete, since $\Pr[\neg A \cup \neg B] \geq 1 - negl(t) + 0 - 0$ and thus $\Pr[A \cap B] = 1 - \Pr[\neg A \cup \neg B] \leq \mathrm{negl}(t)$. □

Concerning *efficiency*, the file storage requirements are increasing with the number of updates. Measuring the performance in absolute values (via an implementation) is subject of currently ongoing efforts (along with theoretical improvements as sketched in the conclusion section below). Initially, the file is stored as is, so that no overhead is needed

Table 1: Complexity (excluding efforts for error correction).

Operation	computational cost for the verifier	prover
Encode	$O(n\log n)$	$O(n\log n)$
Challenge	$O(1)$	–
Response	–	$O(\log n)$
Verify	$O(\log n)$	–
Update	$O(1)$	$O(1)$
new sentinel	$O(\log n)$	–
Extract	$O(n\log n)$	$O(n\log n)$

if the randomizer for the chameleon hash is computed from the data itself (via a pseudorandom function for example), unless explicitly stored with the file record. However, the nature of the chameleon hash implies that after k updates, we have a total lot of $O(|F|+k)$ bits stored at the verifier's side.

For a response to a challenge or an update, we transmit all hashes along all sibling nodes on the path before submitting the new data. This comes to $O(\log n)$ bits for n records in the file and a binary hash-tree (generalizations are discussed in the next section).

The computational burden is determined by the number of chameleon-hashes to be computed. Precisely, for a file with n records, the costs are listed in table 1. `Extract` is here the most expensive operation for the verifier, since $\mathcal{V}$ after having downloaded the file via a sequence of n challenges recomputes the whole hash-tree. The cost on both sides is thus $O(n\log n)$.

Freshness. Notice that although the chameleon hash of an old and new record is the same, the provider cannot simply refrain from updating the record, as he must expect future queries on exactly this updated record. In that case, if the old record x has been queried with challenge c, then the new record will be queried with some challenge $\widetilde{c}$, yielding $CH_{pk}(x\|c) \neq CH_{pk}(x\|\widetilde{c})$, unless this is a hash-collision by coincidence.

An Example Parametrization. The chameleon-hash used in this work is basically a variation of Nyberg-Rueppel, which in turn is closely related to the ElGamal signature scheme. Consequently, the same security recommendations as for ElGamal apply to the parametrization of the chameleon hash (see (Ateniese and de Medeiros, 2005) and (Menezes et al., 1997) for comments). So, for the example, let a hash-value and challenge have 256 bits each.

Suppose that we store a 2GB file, made up of $n = 2^{27}$ blocks (e.g., unicode characters with 4 bytes). Suppose that we wish to run one audit per day over the next five years, without embedding new sentinels. Then we ought to design the protocol to handle $5 \times 365 = 1\,825$ verifications. If each audit consists of 1 000 challenges, then there are 1 825 000 sentinels with 2×256 bits (for the challenge-response pair) to be stored at the client side. This makes a total of roughly 116.8 MBytes for the client (approximately 5.4% of the total file size). The likelihood for a *single* challenge to detect a corruption is thus only 5.4%. However, making a 1 000 challenges per audit, the likelihood to discover a corruption quickly approaches 1.

5 EXTENSIONS

Several extensions to the scheme are imaginable and partially straightforward.

5.1 Saving Random Coins

Observe that except for the leaf-level where the Chameleon-hash is required, any standard cryptographic hash-algorithm can be used for the inner nodes in the Merkle-tree, so to save random coins that would be required otherwise.

5.2 Application to XML Files

As being inherently tree-structured, the Merkle-hashtree can be generalized to ℓ-ary trees in the obvious way, so that the scheme remains unchanged except for trivial modifications. However, the computational cost all grow by the branching factor (the maximal count of children of an inner node) ℓ of the tree.

5.3 Insertions and Deletions

Those are slightly more tricky and basically come at the same cost as for these operations to be performed on a humble array. More precisely, to *insert* a record at a given position i, we may apply `update` to all $n-i$ successor records to shift them one place apart, so that the new record can be inserted at the chosen position. The removal of a record at position i can be done in the same fashion or we mark the record as removed by replacing the data with its hash-value (this would correspond to a sanitization or redaction in editable signature terminology). In case that the hash-tree is already full, it must be recomputed. Alternatives are offered by ℓ-ary trees with designated free spaces in between to insert new records, or if the data is organized in a skip-list rather than a tree.

5.4 Multiple Queries on a Record

A simple way to avoid the prover learning what records have been queried is to challenge a whole set $S' \subset F$ of records at a time, where $S' \cap S \neq \emptyset$. Any data referring to a record in S' for which no stored response is expected can be abandoned. In this way, the prover is left with residual uncertainty about what record has actually been queried. A more elegant possibility is offered by private-information retrieval (PIR; see (Gasarch, 2004) for a survey), yet the additional computational and communication overhead must be assured not to outweigh the cost for an entire download via `extract`.

5.5 Fairness

An interesting additional security requirement in dynamic PORs has been introduced in (Zheng and Xu, 2011), called *fairness*. In brief, this requires that an honest prover cannot be accused successfully by a malicious verifier to have modified the stored file. Similar notions appear in the context of sanitizable signatures (signer- and sanitizier accountability). However, we can keep the model and security definitions much simpler if we require all challenges and update requests to be digitally signed by the verifier, including the originally submitted file via `encode`. Arguments like the previous ones can then be settled at the court by the prover showing the entire history of updates and the original file signature. This essentially relies on a versioning system that a good storage should maintain anyway. Note that the signature can indeed remain intact without needing the verifier's secret signature key, since the construction can be extended to fit into standard sanitizable signature schemes. This direction is left open for future research.

6 CONCLUSIONS

This work presented a simple and partially generic construction of dynamic proofs of retrievability from chameleon-hashes (trapdoor commitments). The proposed scheme is simple and most naturally used with XML structured data that is stored at an untrusted external server, e.g., a cloud storage provider. Unlike standard proofs of retrievability schemes, the construction in this work is neither bounded nor unbounded use, but allows for the introduction of new sentinels for future integrity spot checks. This feature seemingly does not exist in any so-far existing proof of retrievability scheme. In its present form, the protocol is designed to allow for changes to the file, but not to the structure as such, which is an interesting open question for future research. Especially so, since structural changes are so-far not supported by any known POR protocol.

ACKNOWLEDGEMENTS

I thank the anonymous reviewers for their careful reading, valuable comments and useful suggestions.

REFERENCES

Ateniese, G., Burns, R., Curtmola, R., Herring, J., Kissner, L., Peterson, Z., and Song, D. (2007). Provable data possession at untrusted stores. In *Proceedings of the 14th ACM conference on Computer and communications security*, CCS '07, pages 598–609, New York, NY, USA. ACM.

Ateniese, G. and de Medeiros, B. (2005). On the key exposure problem in chameleon hashes. In *Proceedings of the 4th international conference on Security in Communication Networks*, SCN'04, pages 165–179, Berlin, Heidelberg. Springer.

Ateniese, G., Di Pietro, R., Mancini, L. V., and Tsudik, G. (2008). Scalable and efficient provable data possession. In *Proceedings of the 4th international conference on Security and privacy in communication netowrks*, SecureComm '08, pages 9:1–9:10, New York, NY, USA. ACM.

Ateniese, G., Kamara, S., and Katz, J. (2009). Proofs of storage from homomorphic identification protocols. In *Proceedings of the 15th International Conference on the Theory and Application of Cryptology and Information Security: Advances in Cryptology*, ASIACRYPT '09, pages 319–333, Berlin, Heidelberg. Springer-Verlag.

Bowers, K. D., Juels, A., and Oprea, A. (2009a). HAIL: a high-availability and integrity layer for cloud storage. In *ACM Conference on Computer and Communications Security*, pages 187–198.

Bowers, K. D., Juels, A., and Oprea, A. (2009b). 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. ACM. full version available from ePrint, report 2008/175; http://eprint.iacr.org.

Cash, D., Küpçü, A., and Wichs, D. (2012). Dynamic proofs of retrievability via oblivious RAM. In *IACR Cryptology ePrint Archive*. Report 2012/550.

Chen, B. and Curtmola, R. (2012). Robust dynamic provable data possession. In *ICDCS Workshops*, pages 515–525. IEEE Computer Society.

Dodis, Y., Vadhan, S., and Wichs, D. (2009). Proofs of retrievability via hardness amplification. In *Proceedings of the 6th Conference on Theory of Cryptogra-*

phy, TCC '09, pages 109–127, Berlin, Heidelberg. Springer-Verlag.

Erway, C., Küpçü, A., Papamanthou, C., and Tamassia, R. (2009). Dynamic provable data possession. In *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, pages 213–222, New York, NY, USA. ACM.

Gasarch, W. (2004). A survey on private information retrieval. *Bulletin of the EATCS*, 82:72–107.

Halevi, S., Harnik, D., Pinkas, B., and Shulman-Peleg, A. (2011). Proofs of ownership in remote storage systems. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 491–500, New York, NY, USA. ACM.

Juels, A. and Kaliski, B. S. J. (2007). PORs: Proofs of Retrievability for Large Files. In *ACM Conference on Computer and Communications Security, CCS 2007*, pages 584–597. ACM.

Lillibridge, M., Elnikety, S., Birrell, A., Burrows, M., and Isard, M. (2003). A cooperative internet backup scheme. In *Proceedings of the USENIX Annual Technical Conference*, ATEC '03, pages 29–41, Berkeley, CA, USA. USENIX Association.

Liu, S. and Chen, K. (2011). Homomorphic linear authentication schemes for proofs of retrievability. In *Proceedings of the 2011 Third International Conference on Intelligent Networking and Collaborative Systems*, INCOS '11, pages 258–262, Washington, DC, USA. IEEE Computer Society.

Menezes, A., van Oorschot, P. C., and Vanstone, S. (1997). *Handbook of applied Cryptography*. CRC Press LLC.

Paterson, M. B., Stinson, D. R., and Upadhyay, J. (2012). A coding theory foundation for the analysis of general unconditionally secure proof-of-retrievability schemes for cloud storage. *CoRR*, abs/1210.7756.

Resch, J. K. and Plank, J. S. (2011). AONT-RS: blending security and performance in dispersed storage systems. In *Proceedings of the 9th USENIX conference on File and storage technologies*, FAST'11, pages 14–14, Berkeley, CA, USA. USENIX Association.

Shacham, H. and Waters, B. (2008). Compact Proofs of Retrievability. In *Advances in Cryptology - ASIACRYPT 2008*, volume 5350 of *LNCS*, pages 90–107. Springer.

Stefanov, E., van Dijk, M., Juels, A., and Oprea, A. (2012). Iris: a scalable cloud file system with efficient integrity checks. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, pages 229–238, New York, NY, USA. ACM.

Wang, Q., Wang, C., Ren, K., Lou, W., and Li, J. (2011). Enabling public auditability and data dynamics for storage security in cloud computing. *IEEE Transactions on Parallel and Distributed Systems*, 22(5):847–859.

Xu, J. and Chang, E.-C. (2012). Towards efficient proofs of retrievability. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '12, pages 79–80, New York, NY, USA. ACM.

Zheng, Q. and Xu, S. (2011). Fair and dynamic proofs of retrievability. In *Proceedings of the first ACM conference on Data and application security and privacy*, CODASPY '11, pages 237–248, New York, NY, USA. ACM.