Scalable and Efficient Provable Data Possession

Giuseppe Ateniese[a], Roberto Di Pietro[b], Luigi V. Mancini[c], and Gene Tsudik[d]

Abstract

Storage outsourcing is a rising trend which prompts a number of interesting security issues, many of which have been extensively investigated in the past. However, Provable Data Possession (PDP) is a topic that has only recently appeared in the research literature. The main issue is how to frequently, efficiently and securely verify that a storage server is faithfully storing its client's (potentially very large) outsourced data. The storage server is assumed to be untrusted in terms of both security and reliability. (In other words, it might maliciously or accidentally erase hosted data; it might also relegate it to slow or off-line storage.) The problem is exacerbated by the client being a small computing device with limited resources. Prior work has addressed this problem using either public key cryptography or requiring the client to outsource its data in encrypted form.

In this paper, we construct a highly efficient and provably secure PDP technique based entirely on symmetric key cryptography, while not requiring any bulk encryption. Also, in contrast with its predecessors, our PDP technique allows outsourcing of dynamic data, i.e, it efficiently supports operations, such as block modification, deletion and append.

1 Introduction

In recent years, the concept of third-party data warehousing and, more generally, data outsourcing has become quite popular. Outsourcing of data essentially means that the data owner (client) moves its data to a third-party provider (server) which is supposed to – presumably for a fee – faithfully store the data and make it available to the owner (and perhaps others) on demand. Appealing features of outsourcing include reduced costs from savings in storage, maintenance and personnel as well as increased availability and transparent up-keep of data.

A number of security-related research issues in data outsourcing have been studied in the past decade. Early work concentrated on data authentication and integrity, i.e., how to efficiently and securely ensure that the server returns correct and complete results in response to its clients' queries [14, 21]. Later research focused on outsourcing encrypted data (placing even less trust in the server) and associated difficult problems mainly having to do with efficient querying over encrypted domain [18, 27, 10, 17].

More recently, however, the problem of *Provable Data Possession* (PDP) –is also sometimes referred to as *Proof of Data Retrivability* (POR)– has popped up in the research literature. The central goal in PDP is to allow a client to efficiently, frequently and securely verify that a server – who purportedly stores client's potentially very large amount of data – is not cheating the client. In this context, *cheating* means that the server might delete some of the data or it might not store all data in fast storage, e.g., place it on CDs or

[1]The Johns Hopkins University. E-mail: ateniese@cs.jhu.edu
[2]Università di Roma Tre. E-mail: dipietro@mat.uniroma3.it
[3]Università di Roma "La Sapienza". E-mail: manicini@di.uniroma1.it
[4]University of California, Irvine. E-mail: gts@ics.uci.edu

other tertiary off-line media. It is important to note that a storage server might not be malicious; instead, it might be simply unreliable and lose or inadvertently corrupt hosted data. An effective PDP technique must be equally applicable to malicious and unreliable servers. The problem is further complicated by the fact that the client might be a small device (e.g., a PDA or a cell-phone) with limited CPU, battery power and communication facilities. Hence, the need to minimize bandwidth and local computation overhead for the client in performing each verification.

Two recent results PDP [3] and POR [19] have highlighted the importance of the problem and suggested two very different approaches. The first [3] is a public-key-based technique allowing any verifier (not just the client) to query the server and obtain an interactive proof of data possession. This property is called *public verifiability*. The interaction can be repeated any number of times, each time resulting in a fresh proof. The POR scheme [19] uses special blocks (called *sentinels*) hidden among other blocks in the data. During the verification phase, the client asks for randomly picked sentinels and checks whether they are intact. If the server modifies or deletes parts of the data, then sentinels would also be affected with a certain probability. However, sentinels should be indistinguishable from other regular blocks; this implies that blocks must be encrypted. Thus, unlike the PDP scheme in [3], POR cannot be used for public databases, such as libraries, repositories, or archives. In other words, its use is limited to confidential data. In addition, the number of queries is limited and fixed a priori. This is because sentinels, and their position within the database, must be revealed to the server at each query – a revealed sentinel cannot be reused.

Motivation: As pointed out in [1], data generation is currently outpacing storage availability, hence, there will be more and more need to outsource data.

A particularly motivational application for PDP is distributed data store systems. Consider, for example, the Freenet network [11], or the Cooperative Internet Backup Scheme [20]. Such systems rely and thrive on free storage. One reason to cheat in such systems would be if storage space had real monetary cost attached to it. Moreover, in a distributed backup system, a user who grants usage of some of its own space to store other users' backup data is normally granted usage of a proportional amount of space somewhere else in the network, for his own backup. Hence, a user might cheat to obtain better redundancy for his own data. Furthermore, users will only place trust in such a system as long as the storage can be consistently relied upon, which is difficult in the event of massive cheating. A PDP scheme could act as a powerful deterrent to cheating, thus increasing trust in the system and helping spread its popularity and usage. Finally, note that same considerations can be applied to alternative service models, such as peer-to-peer data archiving [12], where new forms of assurance of data integrity and data accessibility are needed. (Though simple replication offers one path to higher-assurance data archiving, it also involves unnecessary, and sometimes unsustainable, overhead.)

Another application of PDP schemes is in the context of censorship-resistance. Note that, on the one hand, there are situations where a duly authorized third party (censor) may need to modify a document in some controlled and limited fashion [4]. On the other hand, there are situations where the content owner has not granted the third party the rights to perform modifications. In this case, there are some well-known solutions, such as [28, 2] and [29]. But, they either require data replication or on-demand computation of a function (e.g., a hash) over entire outsourced data. Though the latter might seem doable, it does not scalable to petabytes and terabytes of data. In contrast, a well-designed PDP scheme would be, at the same time, secure and scalable/efficient.

We note that our motivation above is identical to that used to justify two prominent recent results [3, 19]. A potentially interesting new angle for motivating secure and efficient PDP techniques is the outsourcing of **personal** digital content. (For example, consider the rapidly increasing popularity of personal content

outsourcing services, such as *.MAC*, *GMAIL*, *PICASSA* and *OFOTO*.) Consider the following scenario: Alice wants to outsource her life-long collection of digital content (e.g. photos, videos, art) to a third party, giving read access to her friends and family. The combined content is precious to Alice and, whether or not the outsourcing service is free, Alice wants to make sure that her data is faithfully stored and readily available. To verify data possesion, Alice could use a resource-constrained personal device, e.g., a PDA or a cell-phone. In this realistic setting, our two design requirements – (1) outsourcing data in cleartext and (2) bandiwdth and computation efficiency – are very important.

Contributions: This paper's contribution is two-fold:

1. *Efficiency and Security:* the proposed PDP scheme, as [19], relies only on efficient symmetric-key operations in both setup (performed once) and verification phases. However, our scheme is more efficient than POR as it requires no bulk encryption of outsourced data and no data expansion due to additional sentinel blocks —see Section 5 for details. Our scheme provides probabilistic assurance of the untampered data being stored on the server with the exact probability fixed at setup, as in [3, 19]. Furthermore, our scheme is provably secure in the random oracle model (ROM).

2. *Dynamic Data Support:* unlike both prior results [3, 19], the new scheme supports secure and efficient dynamic operations on outsourced data blocks, including: modification, deletion and append.[1] Supporting such operations is an important step toward practicality, since many envisaged application are not limited to data warehousing, i.e., dynamic operations need to be provided.

Roadmap: The next section introduces our approach to provable data possession and discusses its effectiveness and security; Section 3 extends and enhances it to support dynamic outsourced data (i.e., operations such as block update, append and delete). Then, Section 4 discusses some features of our proposal, followed by Section 5 which overviews related work. Finally, Section 6 includes a discussion of our results and outlines avenues for future work.

2 Proposed PDP Scheme

In this section we describe the proposed scheme. It consists of two phases: *setup* and *verification* (also called *challenge* in the literature).

2.1 Notation

- D – outsourced data. We assume that D can be represented as a single contiguous file of d equal-sized blocks: $D[1], \ldots, D[d]$. The actual bit-length of a block is not germane to the scheme.
- $\mathcal{OWN}$ – the owner of the data.
- $\mathcal{SRV}$ – the server, i.e., the entity that stores outsourced data on owner's behalf.
- $H(\cdot)$ – cryptographic hash function. In practice, we use standard hash functions, such as SHA-1, SHA-2, etc.

[1] However, these operations involve slightly higher computational and bandwidth costs with respect to our basic scheme.

- $AE_{key}(\cdot)$ – an authenticated encryption scheme, that provides both privacy and authenticity. In practice, privacy and authenticity is achieved by encrypting the message first and then computing a Message Authentication Code (MAC) on the result. However, a less expensive alternative is to use a mode of operation for the cipher that provides authenticity in addition to privacy in a single pass, such as OCB, XCBC, IAPM, [24].

- $AE^{-1}_{key}(\cdot)$ – decryption operation for the scheme introduced above.

- $f_{key}(\cdot)$ – pseudo-random function (PRF) indexed on some (usually secret) *key*. In practice, a "good" block cipher acts as a PRF, in particular we could use AES where keys, input blocks, and outputs are all of 128 bits (AES is a good pseudo-random permutation). Other even more practical constructions of PRFs deployed in standards use "good" MAC functions, such as HMAC [6].

- $g_{key}(\cdot)$ – pseudo-random permutation (PRP) indexed under *key*. AES is assumed to be a good PRP. To restrict the PRP output to sequence numbers in a certain range, one could use the techniques proposed by Black and Rogaway [8].

2.2 General Idea

Our scheme is based entirely on symmetric-key cryptography. The main idea is that, before outsourcing, $\mathcal{OWN}$ pre-computes a certain number of short possession verification tokens, each token covering some set of data blocks. The actual data is then handed over to $\mathcal{SRV}$. Subsequently, when $\mathcal{OWN}$ wants to obtain a proof of data possession, it challenges $\mathcal{SRV}$ with a set of random-looking block indices. In turn, $\mathcal{SRV}$ must compute a short integrity check over the specified blocks (corresponding to the indices) and return it to $\mathcal{OWN}$. For the proof to hold, the returned integrity check must match the corresponding value pre-computed by $\mathcal{OWN}$. However, in our scheme $\mathcal{OWN}$ has the choice of either keeping the pre-computed tokens locally or outsourcing them – in encrypted form – to $\mathcal{SRV}$. Notably, in the latter case, $\mathcal{OWN}$'s storage overhead is constant regardless of the size of the outsourced data. Our scheme is also very efficient in terms of computation and bandwidth.

2.3 Setup phase

We start with a database D divided into d blocks. We want to be able to challenge the storage server t times. We use a pseudo-random function, f, and a pseudo-random permutation g defined as:

$$f : \{0,1\}^c \times \{0,1\}^k \longrightarrow \{0,1\}^L$$

and

$$g : \{0,1\}^l \times \{0,1\}^L \longrightarrow \{0,1\}^l.$$

In our case $l = \log d$ since we use g to permute indices. The output of f is used to generate the key for g and $c = \log t$. We note that both f and g can be built out of standard block ciphers, such as AES. In this case $L = 128$. We use the PRF f with two master secret keys W and Z, both of k bits. The key W is used to generate session permutation keys while Z is used to generate challenge nonces.

During the Setup phase, the owner $\mathcal{OWN}$ generates in advance t possible random challenges and the corresponding answers. These answers are called *tokens*. To produce the i^{th} token, the owner generates a set of r indices as follows:

1. First, generate a permutation key $k_i = f_W(i)$ and a challenge nonce $c_i = f_Z(i)$.
2. Then, compute the set of indices $\{I_j \in [1,\ldots,d] \mid 1 \leq j \leq r\}$, where $I_j = g_{k_i}(j)$.
3. Finally, compute the token $v_i = H(\, c_i,\, D[I_1],\, \ldots\, ,\, D[I_r]\,)$.

Basically, each token v_i is the answer we expect to receive from the storage server whenever we challenge it on the randomly selected data blocks $D[I_1],\, \ldots,\, D[I_r]$. The challenge nonce c_i is needed to prevent potential pre-computations performed by the storage server. Notice that each token is the output of a cryptographic hash function so its size is small.

Once all tokens are computed, the owner outsources the entire set to the server, along with the file D, by encrypting each token with an authenticated encryption function[2]. The setup phase is shown in detail in Algorithm 1.

Algorithm 1: Setup phase

```
begin
    Choose parameters c, l, k, L and functions f, g;
    Choose the number t of tokens;
    Choose the number r of indices per verification;
    Generate randomly master keys W, Z, K ∈ {0,1}^k.
    for (i ← 1 to t) do
        begin Round i
1           Generate k_i = f_W(i) and c_i = f_Z(i)
2           Compute v_i = H( c_i , D[g_{k_i}(1)] , ... , D[g_{k_i}(r)] )
3           Compute v'_i = AE_K( i , v_i)
        end
    Send to SRV: (D, {[i, v'_i] for 1 ≤ i ≤ t})
end
```

2.4 Verification Phase

To perform the i-th proof of possession verification, $\mathcal{OWN}$ begins by re-generating the i-th token key k_i as in step 1 of Algorithm 1. Note that $\mathcal{OWN}$ only needs to store the master keys W, Z, and K, plus the current token index i. It also re-computes c_i as above. Then, $\mathcal{OWN}$ sends $\mathcal{SRV}$ both k_i and c_i (step 2 in Algorithm 2). Having received the message from $\mathcal{OWN}$, $\mathcal{SRV}$ computes:

$$z = H(\, c_i\, ,\, D[g_{k_i}(1)]\, ,\, \ldots\, ,\, D[g_{k_i}(r)]\,)$$

$\mathcal{SRV}$ then retrieves v'_i and returns $[z,\, v'_i]$ to $\mathcal{OWN}$ who, in turn, computes $v = AE_K^{-1}(v'_i)$ and checks whether $v = (i,z)$. If the check succeeds, $\mathcal{OWN}$ assumes that $\mathcal{SRV}$ is storing all of D with a certain probability.

We point out that there is almost no cost for $\mathcal{OWN}$ to perform a verification. It only needs to re-generate the appropriate $[k_i, c_i]$ pair (two PRF-s invocations) and perform one decryption in order to check the reply from $\mathcal{SRV}$. Furthermore, the bandwidth consumed by the verification phase is constant (in both step 2 and 4 of Algorithm 2). This represents truly minimal overhead. The computation cost for $\mathcal{SRV}$, though slightly higher (r PRP-s on short inputs, and one hash), is still very reasonable.

[2]Note that these values could be stored by the owner itself if the number of queries is not too large. This would require the owner to store just a short hash value per query.

Algorithm 2: Verification phase

begin *Challenge i*

$\mathcal{OWN}$ computes $k_i = f_W(i)$ and $c_i = f_Z(i)$

$\mathcal{OWN}$ sends $\{k_i, c_i\}$ to $\mathcal{SRV}$

$\mathcal{SRV}$ computes $z = H(\ c_i\ ,\ D[g_{k_i}(1)]\ ,\ \ldots\ ,\ D[g_{k_i}(r)]\ )$

$\mathcal{SRV}$ sends $\{z, v_i'\}$ to $\mathcal{OWN}$

$\mathcal{OWN}$ extracts v from v_i'. If decryption fails or $v \neq (i\ ,\ z)$ then REJECT.

end

2.5 Detection Probability

We now consider the probability P_{esc} that $\mathcal{OWN}$ manages to successfully complete one run of our verification protocol while $\mathcal{SRV}$ has either deleted, or tampered with, m data blocks. Note that $\mathcal{SRV}$ avoids detection only if none among all the r data blocks involved in the i-th token verification process, are deleted or modified. Thus, we have that:

$$P_{esc} = \left(1 - \frac{m}{d}\right)^r \tag{1}$$

For example, if the fraction of corrupted blocks (m/d) is 1% and $r = 512$, then the probability of avoiding detection is below 0.6%. This is in line with the detection probability expressed in [3].

2.6 Security Analysis

We follow the security definitions in [3, 19] and in particular the one in [3] which is presented in the form of a security game. In practice we need to prove that our protocol is a proof of knowledge of the queried blocks, i.e., if the adversary passes the verification phase then we can extract the queried blocks. The most generic (extraction-based) definition that does not depend on the specifics of the protocol is introduced in [19]. Another formal definition for the related concept of sub-linear authentication is provided in [22].

We have a challenger that plays a security game (experiment) with the adversary $\mathcal{A}$. There is a Setup phase where $\mathcal{A}$ is allowed to select data blocks $D[i]$ for $1 \leq i \leq d$. In the Verification phase the challenger selects a nonce and r random indices and sends them to $\mathcal{A}$. Then the adversary generates a proof of possession P for the blocks queried by the challenger.

If P passes the verification then the adversary won the game. As in [3], we say that our scheme is a Provable Data Possession scheme if for any probabilistic polynomial-time adversary $\mathcal{A}$, the probability that $\mathcal{A}$ wins the PDP security game on a set of blocks is negligibly close to the probability that the challenger can extract those blocks. In this type of proof of knowledge, the challenger can employ a "knowledge extractor" to extract the blocks from $\mathcal{A}$ via several queries. We show the following:

Theorem. *Our scheme is a secure Provable Data Possession scheme in the random oracle model.*

At first, it may seem that we do not need to model the hash function as a random oracle and that we might just need its collision-resistance property (assuming we are using a "good" standard hash function and not some contrived hash scheme). However, the security definition for PDP is an extraction-type definition since we are required to extract the blocks that were challenged. So we need to use the random oracle model but not really in a fundamental way. We just need the ability to see the inputs to the random oracle but we are not using its programmability feature (i.e., we are not "cooking-up" values as outputs of the random

oracle). We need to see the inputs so that we can extract the queried data blocks. Since we are using just a random oracle our proof does not rely on any cryptographic assumption but it is information-theoretic.

Proof. We are assuming that $AE_K(\cdot)$ is an ideal authenticated encryption. This implies that, given $AE_K(X)$, the adversary cannot see or alter X, thus we can assume that X is stored directly by the challenger (i.e., there is no need for the adversary to send X to the challenger and we can remove $AE_K(X)$ from the picture).

More formally, our game is equivalent to the following:

- A simulator $\mathcal{S}$ sets up a PDP system and choses its security parameters.
- The adversary $\mathcal{A}$ selects values $x_1, \ldots, x_n$ and sends them to the simulator $\mathcal{S}$
- The adversary can query the random oracle at any point in time. For each input to the random oracle, the simulator replies with a random value and stores the input and the corresponding output in a table (to be consistent with his replies).
- At the challenge phase the simulator challenges $\mathcal{A}$ on the $i-th$ value, x_i and sends a random value c_i to $\mathcal{A}$.
- $\mathcal{A}$ replies with a string P.

Note that, in our original game, the value x_i corresponds to the ordered sequence of r concatenated blocks which are queried during the $i-th$ challenge. In addition, the simulator can query any x_i only once.

Clearly if the adversary wins the game $(P = H(c_i, x_i))$ with non-negligeable probability the simulator $\mathcal{S}$ extracted x_i with non-negligeable probability. This is because the best the adversary can do to impede the extraction of x_i is to guess the correct output of the random oracle or to find a collision (a random oracle is trivially collision-resistant). Both these events happen with negligible probability.

3 Supporting Dynamic Outsourced Data

Thus far, we assumed that D represents static or warehoused data. Although this is the case for many envisaged settings, such as libraries, there are many potential scenarios where outsourced data is dynamic. This leads us to consider various data block operations (e.g., update, delete, append and insert) and the implications upon our scheme which stem from supporting each operation.

One obvious and trivial solution to all dynamic operations, is (for each operation) for $\mathcal{OWN}$ to download from $\mathcal{SRV}$ the entire outsourced data D and to re-run the setup phase. This would clearly be secure but highly inefficient. We now show how to amend the basic scheme to *efficiently* handle dynamic operations on outsourced data.

3.1 Block Update

We assume that $\mathcal{OWN}$ needs to modify the n-th data block which is currently stored on $\mathcal{SRV}$, from its current value $D[n]$ to a new version, denoted $D'[n]$.

In order to amend the remaining verification tokens, $\mathcal{OWN}$ needs to factor out every occurrence of $D[n]$ and replace it with $D'[n]$. However, one subtle aspect is that $\mathcal{OWN}$ cannot disclose to $\mathcal{SRV}$ which (if any) verification tokens include the n-th block. The reason is simple: if $\mathcal{SRV}$ knows which tokens include the n-th block, it can simply discard $D[n]$ if and when it knows that no remaining tokens include it. Another way to see the same problem is to consider the probability of a given block being included in any verification

token: since there are r blocks in each token and t tokens, the probability of a given block being included in any token is $\frac{r*t}{d}$. Thus, if this probability is small, a single block update is unlikely to involve any verification token update. If $\mathcal{SRV}$ is allowed to see when a block update causes no changes in verification tokens, it can safely erase the new block since it knows (with absolute certainty) that it will not be demanded in future verifications.

Based on the above discussion, we require $\mathcal{OWN}$ to modify all **remaining** verification tokens. We also need to amend the token structure as follows (refer to line 2 in Alg. 1 and line 3 in Alg. 2):

$$
\begin{aligned}
&\textbf{from:} && \\
&\quad v_i &=\ & H(\, c_i \,,\, D[g_{k_i}(1)] \,,\, \ldots \,,\, D[g_{k_i}(r)] \,) \\
&\quad v'_i &=\ & AE_K(\, i \,,\, v_i) \\
&\textbf{to:} && \\
&\quad v_i &=\ & H(c_i,\, 1,\, D[g_{k_i}(1)]) \,\oplus\, \ldots \,\oplus\, H(c_i,\, r,\, D[g_{k_i}(r)]) \\
&\quad v'_i &=\ & AE_K(\, ctr \,, i \,,\, v_i) \ \text{ where } ctr \text{ is an integer counter}
\end{aligned}
\tag{2}
$$

We use ctr to keep track of the latest version of the token v_i (in particular, to avoid replay attacks) and to change the encryption of unmodified tokens. Alternatively, we could avoid storing ctr by just replacing the key K of the authenticated encryption with a new fresh key.

The modifications described above are needed to allow the owner to efficiently perform the token update. The basic scheme does not allow it since, in it, a token is computed by serially hashing r selected blocks. Hence, it is difficult to factor out any given block and replace it by a new version.[3]

One subtle but important change in the computation of v_i above is that now each block is hashed along with an index. This was not necessary in the original scheme since blocks were ordered before being hashed together. Now, since blocks are hashed individually, we need to take into account the possibility of two identical blocks (with the same content but different indices) that are selected as part of the same verification token. If we computed each block hash as: $H(c_i,\ D[g_{k_i}(j)])$, instead of: $H(c_i,\ j,\ D[g_{k_i}(j)])$, then two blocks with identical content would produce the same hash and would cancel each other out due to the exclusive-or operation. Including an unique index in each block hash addresses this issue. We note that our XOR construction is basically the same as the XOR MAC defined by Bellare, et al. [7]. In the random oracle model $H(c_i,\ldots)$ is a PRF under the key c_i. However, we do not need a randomized block as in XMACR [7] or a counter as in XMACC [7] since we use the key c_i only once (it changes at each challenge).

The update operation is illustrated in Algorithm 3. As shown at line 7, $\mathcal{OWN}$ modifies all tokens regardless of whether they contain the n-th block. Hence, $\mathcal{SRV}$ cannot discern tokens covering the n-th block. However, the crucial part of the algorithm is line 6 where $\mathcal{OWN}$ simultaneously replaces the hash of the block's old version $H(c_i, j,\ D[n])$ with the new one: $H(c_i, j,\ D'[n])$. This is possible due to the basic properties of the $\bigoplus$ operator.

We now consider the costs. The bandwidth consumed is minimal given our security goals: since $\mathcal{OWN}$ does not store verification tokens, it needs to obtain them in line 2. Consequently, and for the same reason, it also needs to upload new tokens to $\mathcal{SRV}$ at line 8. Computation costs for $\mathcal{OWN}$ are similar to those of the setup phase except that no hashing is involved. There is no computation cost for $\mathcal{SRV}$. As far as storage, $\mathcal{OWN}$ must have enough to accommodate all tokens, as in the setup phase.

[3] Of course, it can be done trivially by asking $\mathcal{SRV}$ to supply all blocks covered by a given token; but, this would leak information and make the scheme insecure.

Algorithm 3: Update

% Assume that block $D[n]$ is being modified to $D'[n]$
% and, assume that no tokens have been used up thus far.
begin
1 Send to $\mathcal{SRV}$: UPDATE REQUEST
2 Receive from $\mathcal{SRV}$: $\{[i, v_i'] \mid 1 \leq i \leq t\})$
3 $ctr = ctr + 1$
for $(i \leftarrow 1$ **to** $t)$ **do**
4 $z' = AE_K^{-1}(v_i')$
% if decryption fails, exit
% if z' is not prefixed by $(ctr-1), i$, exit
5 extract v_i from z'
$k_i = f_W(i)$
$c_i = f_Z(i)$
for $(j \leftarrow 1$ **to** $r)$ **do**
if $(g_{k_i}(j) == n)$ **then**
6 $v_i = v_i \oplus H(c_i, j, D[n]) \oplus H(c_i, j, D'[n])$
7 $v_i' = AE_K(ctr, i, v_i)$
8 Send to $\mathcal{SRV}$: $\{n, D'[n], \{[i, v_i'] \mid 1 \leq i \leq t\}\}$
end

Security Analysis. The security of the Update algorithm in the random oracle model follows directly from the proof of the static case. The only difference is in the way we compute v_i: Rather than hashing the concatenation of blocks, the adversary hashes each single block and XOR resulting outputs. This does not change the ability of the simulator to extract the blocks queried during the challenge. Indeed, as in [7], we include a unique index in the hash (from 1 to r) that "forces" the adversary to query the random oracle on each single block indicated by the challenge. In addition, the collision-resistance property of the XOR-ed hashes comes directly from [7] so we omit it here.

3.2 Block Deletion

After being outsourced, certain data blocks might need to be deleted. Especially if the outsourced storage functions on a pay-per-block basis, $\mathcal{OWN}$ has an incentive to minimize costs. Supporting block deletion turns out to be very similar to the update operation. We maintain the same verification token structure and almost the same algorithm as in the previous section. The only modification is in line 6 (of Algorithm 3):

from:
$$v_i = v_i \oplus H(c_i, j, D[n]) \oplus H(c_i, j, D'[n])$$
to:
$$v_i = v_i \oplus H(c_i, j, D[n])$$

This small change results in the deleted block being removed from all verification tokens where it has occurred. Moreover, as in the update case, all tokens are modified such that $\mathcal{SRV}$ remains oblivious as far

as the inclusion of the deleted block. Note that $\mathcal{SRV}$ during the execution of the verification phase, will ignore an index $g_{k_i}(j)$, if it happes to be an index of the deleted block — see line 3 of Algorithm 2. The costs associated to the delete operation are identical to those of the update.

3.3 Batching Updates and Deletions

Although the above demonstrates that the proposed technique can accommodate certain operations on outsourced data, it is clear that the cost of updating all remaining verification tokens for a single block update or deletion is not negligible for $\mathcal{OWN}$. Fortunately, it is both easy and natural to batch multiple operations. Specifically, any number of block updates and deletes can be performed at the cost of a single update or delete. To do this, we need to modify the **for-loop** in Algorithm 3 to take care of both deletions and updates at the same time. Since the modifications are straight-forward, we do not illustrate them in a separate algorithm.

3.4 Single-Block Append

In some cases, the owner might want to increase the size of the outsourced database. We anticipate that the most common operation is block append, e.g., if the outsourced data represents a log. Clearly, the cost of re-running the setup phase for each single block append would be prohibitive.

We obviously can not easily increase the range of the PRP $g_{k_i}(\cdot)$ or other parameters since these are hardcoded into the scheme – we pre-compute all answers to all challenges. However, rather than considering a single contiguous file of d equal-sized blocks, we could consider a logical bi-dimensional structure of the outsourced data, and append a new block to one of the original blocks $D[1],\ldots,D[d]$ in a round-robin fashion. In other words, assume that $\mathcal{OWN}$ has the outsourced data $D[1],\ldots,D[d]$, and that it wants to append the blocks $D[d+1],\ldots,D[d+k]$. Then a logical matrix is built by column as follows:

$$\begin{aligned}
D'[1] &= D[1]\,, \quad D[d+1] \\
D'[2] &= D[2]\,, \quad D[d+2] \\
&\ldots \\
D'[k] &= D[k]\,, \quad D[d+k] \\
&\ldots \\
D'[d] &= D[d]
\end{aligned}$$

For the index j in the i-th challenge, the server will have to include in the computation of the XOR-ed hashes v_i any blocks linked to $D[g_{k_i}(j)]$, i.e., the entire row $g_{k_i}(j)$ in the logical matrix above. In particular, $\mathcal{SRV}$ will include:

$$H(c_i,\ j,\ D[g_{k_i}(j)])\ \oplus\ H(c_i,\ d+j,\ D[g_{k_i}(j)+d])\ \oplus \ldots \oplus\ H(c_i,\ \alpha d+j,\ D[g_{k_i}(j)+\alpha d]),$$

where α is the length of the row $g_{k_i}(j)$ of the logical matrix. Clearly, $\mathcal{OWN}$ will have to update the XOR-ed hashes in a similar fashion by invoking the Update algorithm whenever a certain block must be appended.

The advantage of this solution is that we can just run the Update operation to append blocks so we can even batch several appends. Notice also that the probability for any block to be selected during a challenge is still $1/d$ (this is because intuitively we consider d rows of several blocks combined together). The drawback is that the storage server will have to access more blocks per query and this may become increasingly expensive for $\mathcal{SRV}$ if an excessive number of blocks is appended to the database.

3.5 Bulk Append

Another variant of the append operation is *bulk append* which takes place whenever $\mathcal{OWN}$ needs to upload a large number of blocks at one time. There are many practical scenarios where such operations are common. For example, in a typical business environment, a company might accummulate electronic documents (e.g., sales receipts) for a fixed term and might be unable to outsource them immediately for tax and instant availability reasons. However, as the fiscal year rolls over, the accummulated batch would need to be uploaded to the server in bulk.

To support the bulk append operation, we sketch out a slightly different protocol version. In this context, we require for $\mathcal{OWN}$ to anticipate the maximum size (in blocks) for its data. Let d_{max} denote this value. However, in the beginning, $\mathcal{OWN}$ only outsources d blocks where $d < d_{max}$.

The idea behind bulk append is simple: $\mathcal{OWN}$ budgets for the maximum anticipated data size, including (and later verifying) not r but $r_{max} = \lceil r * (d_{max}/d) \rceil$ blocks per verification token. However, in the beginning, when only d blocks exist, $\mathcal{OWN}$, as part of setup, computes each token v_i in a slightly different fashion (the computation of v'_i remains unchanged):

$$v_i = \bigoplus_{j=1}^{r_{max}} Q_j \quad \text{where:} \quad Q_j = \begin{cases} H(c_i,\ j,\ D[g_{k_i}(j)]) & \text{if} \quad g_{k_i}(j) \leq d \\ 0^{|g_{k_i}()|} & \text{otherwise} \end{cases}$$

The important detail in this formula is that, whenever the block index function $g_{k_i}(j) > d$ (i.e., it exceeds the current number of outsourced blocks), $\mathcal{OWN}$ simply skips it.[4] Note that, since $r_{max} = r * (d_{max}/d)$, we know that, on average, r indices will fall into the range of existing d blocks. Since both $\mathcal{OWN}$ and $\mathcal{SRV}$ agree on the number of currently outsourced blocks, $\mathcal{SRV}$ will follow exactly the same procedure when re-computing each token upon $\mathcal{OWN}$'s verification request.

Now, when $\mathcal{OWN}$ is ready to append a number (say, d) of new blocks, it first fetches all unused tokens from the server, as in the block update or delete operations described above. Then, $\mathcal{OWN}$ re-computes each unused token by updating the counter (as in Equation 2). The only difference is that now, $\mathcal{OWN}$ factors in (does not skip) $H(c_i,\ j,\ D[g_{k_i}(j)])$ whenever $g_{k_i}(j)$ falls in the $]d, 2d]$ interval. (It continues to skip, however, all block indices exceeding $2d$.)

It is easy to see that this approach works but only with sufficiently large append batches; small batches would involve high overhead, as each append requires $\mathcal{OWN}$ to fetch all unused tokens, which is a bandwidth-intensive activity.

4 Discussion

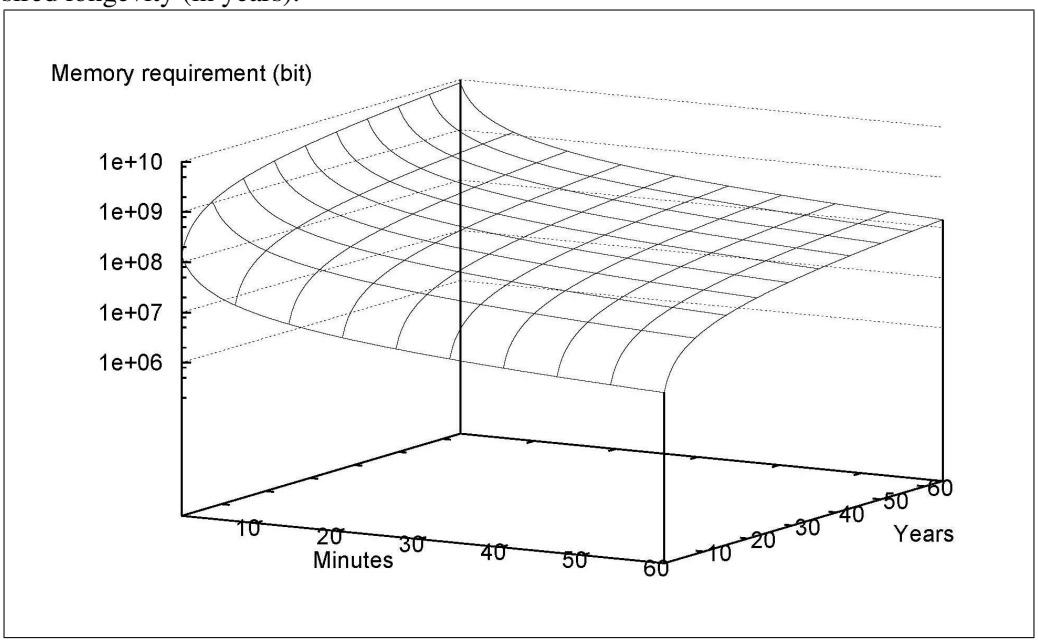
In this section we discuss some features of the proposed scheme.

4.1 Bandwidth-Storage Tradeoff

Our basic scheme, as presented in Section 2, involves $\mathcal{OWN}$ outsourcing to $\mathcal{SRV}$ not only data D, but also a collection of encrypted verification tokens: $\{[i, v'_i]$ for $1 \leq i \leq t\}$. An important advantage of this method is that $\mathcal{OWN}$ remains virtually stateless, i.e., it only maintains the "master-key" K. The storage burden is shifted entirely to $\mathcal{SRV}$. The main disadvantage of this approach is that dynamic operations described in Section 3 require $\mathcal{SRV}$ to send all unused tokens back to $\mathcal{OWN}$ resulting in the bandwidth overhead of

[4] It actually uses a string of zeros in place of each $g_{k_i}(j)$ that exceeds d.

Figure 1: Token Storage Overhead (in bits): X-axis shows verification frequency (in minutes) and Y-axis shows desired longevity (in years).



$(t \times r \times |AE_{key}|)$ bits. Of course, $\mathcal{OWN}$ must subsequently send all updated tokens back to $\mathcal{SRV}$; however, as discussed earlier, this is unavoidable in order to preserve security.

One simple way to reduce bandwidth overhead is for $\mathcal{OWN}$ to store some (or all) verification tokens locally. Then, $\mathcal{OWN}$ simply checks them against the responses received from $\mathcal{SRV}$. We stress that each token is a hash function computed on a set of blocks, thus $\mathcal{OWN}$ will have to store, in the worst case, a single hash per challenge. Another benefit of this variant is that it obviates the need for encryption (i.e., AE_{key} is no longer needed) for tokens stored locally. Also, considering that $\mathcal{SRV}$ is a busy entity storing data for a multitude of owners, storing tokens at $\mathcal{OWN}$ has a further advantage of simplifying update, append, delete and other dynamic operations by cutting down on the number of messages.

4.2 Limited Number of Verifications

One potentially glaring drawback of our scheme is the pre-fixed (at setup time) number of verifications t. For $\mathcal{OWN}$ to increase this number would require to re-running setup which, in turn, requires obtaining D from $\mathcal{SRV}$ in its entirety. However, we argue that, in practice, this is a not an issue. Assume that $\mathcal{OWN}$ wants to periodically (every M minutes) obtain a proof of possession and wants the tokens to last Y years. The number of verification tokens required by our scheme is thus: $(Y \times 365 \times 24 \times 60)/M$. The graph in Figure 1 shows the storage requirements (for verification tokens) for a range of Y and M values. The graph clearly illustrates that this overhead is quite small. For example, assuming $AE_K(\cdot)$ block size of 256 bits, with less than 128 Mbits (16 Mbytes), outsourced data can be verified each hour for 64 years, or, equivalently, every 15 minutes for 16 years.

Furthermore, the number of tokens is independent from the number of data blocks. Thus, whenever outsourced data is extremely large, extra storage required by our approach is negligible. For instance, the

Web Capture project of the Library of Congress has stored, as of May 2007, about 70 Terabytes of data.[5] As shown above, checking this content every 15 minutes for the next 16 years would require only 1 Mbyte of extra storage per year, which arguably can be even stored directly at $\mathcal{OWN}$.

4.3 Computation Overhead

Another important variable is the computation overhead of the setup phase in the basic scheme. Recall that at setup $\mathcal{OWN}$ needs to compute t hashes, each over a string of size $r * |b|$ size, where $|b|$ is the block size. (In the modified scheme, the overhead is similar: $t \times r$ hashes over roughly one block each.)

To assess the overhead, we assume that to process a single byte, a hash function (e.g., SHA), requires just 20 machine cycles [5], which is a conservative assumption. Furthermore, we assume the following realistic parameters:

- $\mathcal{OWN}$ outsources 2^{37} bytes of data, i.e., the content of a typical hard drive – 128-GB.
- Each data block is 4-KB ($|b| = 2^{12}$) and $d = 2^{37}/2^{12} = 2^{25}$.
- $\mathcal{OWN}$ needs to perform one verification daily for the next 32 years, i.e., $t = 32 \times 365 = 11,680$.
- $\mathcal{OWN}$ requires the detection probability of 99% with 1% of the blocks being missing or corrupted; hence $r = 2^9$ (see Equation 1).

Based on the above, each hash, computed over roughly $2^{21} = 2^{12} \times 2^9$ bytes, would take – on a low-end computing device with a 1-GHz CPU, less than 0.04 seconds [5]. The total number of hashes is $11,680$. Thus, setup time is about $11,680 \times 0.04 = 467$ seconds, which is less than 8 minutes. Note that setup also involves $t \times r$ PRP, $2t$ PRF, as well as t AE_K invocations. However, these operations are performed over short inputs (e.g., 128 bits) and their costs are negligible with respect to those of hashing.

5 Related Work

Initial solutions to the PDP problem were provided by Deswarte *et al.* [13] and Filho *et al.* [15]. Both use RSA-based hash functions to hash the entire file at every challenge. This is clearly prohibitive for the server whenever the file is large.

Similarly, Sebe *et al.* [26] give a protocol for remote file integrity checking, based on the Diffie-Hellman problem in composite-order groups. However, the server must access the entire file and in addition the client is forced to store several bits per file block, so storage at the client is linear with respect to the number of file blocks as opposed to constant.

Schwarz and Miller [25] propose a scheme that allows a client to verify storage integrity of data across multiple servers. However, even in this case, the server must access a linear number of file blocks per challenge.

Golle *et al.* [16] first proposed the concept of *enforcement of storage complexity* and provided efficient schemes. Unfortunately the guarantee they provide is weaker than the one provided by PDP schemes since it only ensures that the server is storing something at least as large as the original file but not necessarily the file itself.

[5] See `http://www.loc.gov/webcapture/faq.html`

Provable data possession is a form of memory checking [9, 23] and in particular it is related to the concept of *sub-linear authentication* introduced by Naor and Rothblum [23]. However, schemes that provide sub-linear authentication are quite inefficient.

Compared to the PDP scheme in [3], our scheme is significantly more efficient in both setup and verification phases since it relies only on symmetric-key cryptography. The scheme in [3] allows unlimited verifications and offers public verifiability (which we do not achieve). However, we showed that limiting the number of challenges is not a concern in practice. In addition, our scheme supports dynamic operations, while the PDP scheme in [3] works only for static databases.

Compared to the POR scheme [19], our scheme provides better performance on the client side, requires much less storage space, and uses less bandwidth (sizes of challenges and responses in our scheme is a small constant, less than the size of a single block). To appreciate the difference in storage overhead, consider that, POR needs to store s sentinels per verification, where each sentinel is one data block in size (hence, a total of $s \times t$ sentinels). In contrast, our scheme needs a single encrypted value (256 bits) per verification. Note that, in POR, for a detection probability of around 93%, where at most 1% of the blocks have been corrupted, the suggested value s is on the order of one thousand [19]. Like [3], POR is limited to static data (in fact, [19] considers supporting dynamic data an open problem). In summary, we provide more features than POR by consuming considerably less resources.

To further compare the two schemes we assess their respective tamper detection probabilities. In [19], the outsourced data is composed of d blocks and there are s sentinels. We compute the probability that, m being the number of corrupted blocks, the system consumes all sentinels and corruption is undetected. Similarly, to evaluate our scheme, for the same amount of data, we assume t tokens, each based on r verifiers.

For our scheme, the probability of no corrupted block being detected after all t tokens are consumed is:

$$\frac{\binom{d-\eta}{m}}{\binom{d}{m}} \tag{3}$$

Note that η accounts for the number of different blocks used by at least one token. Indeed, to avoid detection, $\mathcal{SRV}$ cannot corrupt any block used to compute any token. It can be shown that, for a suitably large d, for every t, it holds that: $\eta \geq min\{tr/2, d/2\}$ with overwhelming probability. Hence, the upper bound for Equation 3, when $tr/2 \leq d/2$, is given by:

$$\frac{\binom{d-\frac{tr}{2}}{m}}{\binom{d}{m}} \tag{4}$$

In [19], $\mathcal{SRV}$ can avoid detection if it does not corrupt any *sentinel* block. Since a total of $s \times t$ sentinels are added to the outsourced data, the probability of avoiding detection is:

$$\frac{\binom{d}{m}}{\binom{d+(s\times t)}{m}} \tag{5}$$

The above equations show that, in our scheme, for a given t, increasing r also increases the tamper-detection probability. However, r has no effect on the total number of tokens, i.e., does not influence storage overhead. As for POR, for a given t, increasing tamper detection probability requires increasing in s, which, in turn, increases storage overhead.

6 Conclusions

We developed and presented a step-by-step design of a very light-weight and provably secure PDP scheme. It surpasses prior work on several counts, including storage, bandwidth and computation overheads as well as the support for dynamic operations. However, since it is based upon symmetric key cryptography, it is unsuitable for public (third-party) verification. A natural solution to this would be a hybrid scheme combining elements of [3] and our scheme.

To summarize, the work described in this paper represents an important step forward towards practical PDP techniques. We expect that the salient features of our scheme (very low cost and support for dynamic outsourced data) make it attractive for realistic applications.

References

[1] The expanding digital universe: A forecast of worldwide information growth through 2010.—sponsored by emc. Technical report, 2007.

[2] J. Aspnes, J. Feigenbaum, A. Yampolskiy, and S. Zhong. Towards a theory of data entanglement. In *Proc. of Euro. Symp. on Research in Computer Security*, 2004.

[3] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song. Provable data possession at untrusted stores. In *ACM CCS'07, Full paper available on e-print (2007/202)*, 2007.

[4] G. Ateniese, D. Chou, B. de Medeiros, and G. Tsudik. Sanitizable signatures. In *ESORICS'05*, 2005.

[5] B. Preneel, et al. NESSIE D21 - performance of optimized implementations of the nessie primitives.

[6] M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. In *CRYPTO'96*, 1996.

[7] M. Bellare, R. Guérin, and P. Rogaway. XOR MACs: New methods for message authentication using finite pseudorandom functions. In *CRYPTO'95*, 1995.

[8] J. Black and P. Rogaway. Ciphers with arbitrary finite domains. In *CT-RSA'02*, 2002.

[9] M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Naor. Checking the correctness of memories. In *Proc. of the FOCS '95*, 1995.

[10] D. Boneh, G. di Crescenzo, R. Ostrovsky, and G. Persiano. Public key encryption with key-word search. In *EUROCRYPT'04*, 2004.

[11] I. Clarke. FREENET – the free network project. Technical report.

[12] B. Cooper and H. Garcia-Molina. Peer-to-peer data trading to preserve information. *ACM ToIS*, 20(2):133–170, 2002.

[13] Y. Deswarte, J.-J. Quisquater, and A. Saidane. Remote integrity checking. In *Proc. of Conference on Integrity and Internal Control in Information Systems (IICIS'03)*, November 2003.

[14] P. Devanbu, M. Gertz, C. Martel, and S. Stubblebine. Authentic third-party data publication. In *IFIP DBSec'03, also in Journal of Computer Security, Vol. 11, No. 3, pages 291-314, 2003*, 2003.

[15] D. L. G. Filho and P. S. L. M. Baretto. Demonstrating data possession and uncheatable data transfer. IACR ePrint archive, 2006. Report 2006/150, `http://eprint.iacr.org/2006/150`.

[16] P. Golle, S. Jarecki, and I. Mironov. Cryptographic primitives enforcing communication and storage complexity. In *Financial Cryptography*, pages 120–135, 2002.

[17] P. Golle, J. Staddon, and B. Waters. Secure conjunctive keyword search over encrypted data. In *ACNS'04*, 2004.

[18] H. Hacigümüs, B. Iyer, C. Li, and S. Mehrotra. Executing sql over encrypted data in the database-service-provider model. In *ACM SIGMOD'02*, 2002.

[19] A. Juels and B. Kaliski. PORs: Proofs of retrievability for large files. In *ACM CCS'07, Full paper available on e-print (2007/243)*, 2007.

[20] M. Lillibridge, S. Elnikety, A. Birrell, M. Burrows, and M. Isard. A cooperative internet backup scheme. In *USENIX'03*, 2003.

[21] E. Mykletun, M. Narasimha, and G. Tsudik. Authentication and integrity in outsourced databases. In *ISOC NDSS'04*, 2004.

[22] M. Naor and G. Rothblum. The complexity of online memory checking. In *FOCS'05*, 2005.

[23] M. Naor and G. N. Rothblum. The complexity of online memory checking. In *Proc. of FOCS*, 2005. Full version appears as ePrint Archive Report 2006/091.

[24] P. Rogaway, M. Bellare, and J. Black. OCB: A block-cipher mode of operation for efficient authenticated encryption. *ACM TISSEC*, 6(3), 2003.

[25] T. S. J. Schwarz and E. L. Miller. Store, forget, and check: Using algebraic signatures to check remotely administered storage. In *Proceedings of ICDCS '06*. IEEE Computer Society, 2006.

[26] F. Sebe, A. Martinez-Balleste, Y. Deswarte, J. Domingo-Ferrer, and J.-J. Quisquater. Time-bounded remote file integrity checking. Technical Report 04429, LAAS, July 2004.

[27] D. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *IEEE S&P'00*, 2000.

[28] M. Waldman and D. Mazières. TANGLER: a censorship-resistant publishing system based on document entanglements. In *ACM CCS'01*, 2001.

[29] M. Waldman, A. Rubin, and L. Cranor. PUBLIUS: a robust, tamper-evident, censorship-resistant web publishing system. In *USENIX SEC'00*, 2000.