Provable Data Possession at Untrusted Stores[*]

Giuseppe Ateniese[†] Randal Burns[†] Reza Curtmola[†]
Joseph Herring[†] Lea Kissner [‡] Zachary Peterson[†] Dawn Song [§]

Abstract

We introduce a model for *provable data possession* (PDP) that allows a client that has stored data at an untrusted server to verify that the server possesses the original data without retrieving it. The model generates probabilistic proofs of possession by sampling random sets of blocks from the server, which drastically reduces I/O costs. The client maintains a constant amount of metadata to verify the proof. The challenge/response protocol transmits a small, constant amount of data, which minimizes network communication. Thus, the PDP model for remote data checking supports large data sets in widely-distributed storage systems.

We present two provably-secure PDP schemes that are more efficient than previous solutions, even when compared with schemes that achieve weaker guarantees. In particular, the overhead at the server is low (or even constant), as opposed to linear in the size of the data. Experiments using our implementation verify the practicality of PDP and reveal that the performance of PDP is bounded by disk I/O and not by cryptographic computation.

1 Introduction

Verifying the authenticity of data has emerged as a critical issue in storing data on untrusted servers. It arises in peer-to-peer storage systems [28, 34], network file systems [29, 25], long-term archives [31], web-service object stores [45], and database systems [30]. Such systems prevent storage servers from misrepresenting or modifying data by providing authenticity checks when accessing data.

However, archival storage requires guarantees about the authenticity of data on storage, namely that storage servers possess data. It is insufficient to detect that data have been modified or deleted when accessing the data, because it may be too late to recover lost or damaged data. Archival storage servers retain tremendous amounts of data, little of which are accessed. They also hold data for long periods of time during which there may be exposure to data loss from administration errors as the physical implementation of storage evolves, *e.g.*, backup and restore, data migration to new systems, and changing memberships in peer-to-peer systems.

Archival network storage presents unique performance demands. Given that file data are large and are stored at remote sites, accessing an entire file is expensive in I/O costs to the storage server and in transmitting the file across a network. Reading an entire archive, even periodically,

[*]This is the full version of the paper that appears in Proceedings of the 14th ACM Conference on Computer and Communications Security(CCS 2007)

[†]Department of Computer Science, Johns Hopkins University – {`ateniese, randal, crix`}`@cs.jhu.edu`, `jrh@jhu.edu, zachary@cs.jhu.edu`

[‡]Google, Inc. – `leak@cs.cmu.edu`

[§]University of California Berkeley/Carnegie Mellon University – `dawnsong@cs.berkeley.edu` greatly limits the scalability of network stores. (The growth in storage capacity has far outstripped the growth in storage access times and bandwidth [43]). Furthermore, I/O incurred to establish data possession interferes with on-demand bandwidth to store and retrieve data. We conclude that clients need to be able to verify that a server has retained file data *without retrieving the data from the server* and *without having the server access the entire file.*

Previous solutions do not meet these requirements for proving data possession. Some schemes [19] provide a weaker guarantee by enforcing *storage complexity*: The server has to store an amount of data at least as large as the client's data, but not necessarily the same exact data. Moreover, all previous techniques require the server to access the entire file, which is not feasible when dealing with large amounts of data.

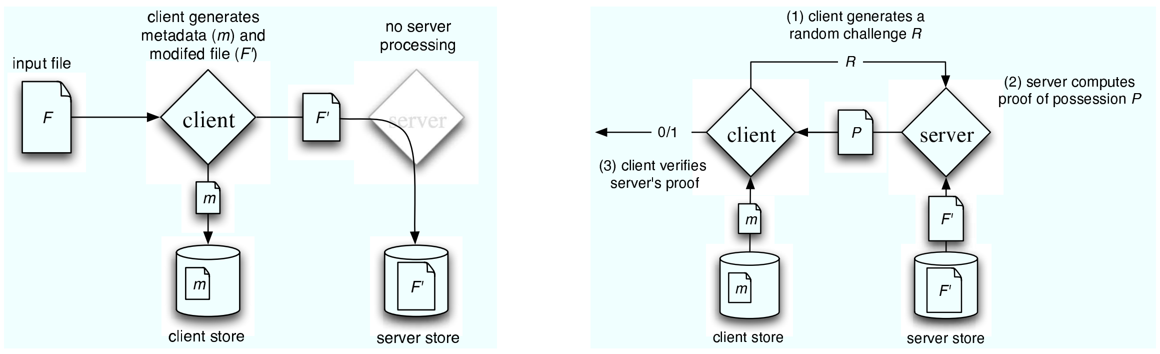
We define a model for provable data possession (PDP) that provides probabilistic proof that a third party stores a file. The model is unique in that it allows the server to access small portions of the file in generating the proof; all other techniques must access the entire file. Within this model, we give the first provably-secure scheme for remote data checking. The client stores a small O(1) amount of metadata to verify the server's proof. Also, the scheme uses O(1) bandwidth[1]. The challenge and the response are each slightly more than 1 Kilobit. We also present a more efficient version of this scheme that proves data possession using a single modular exponentiation at the server, even though it provides a weaker guarantee.

Both schemes use *homomorphic verifiable tags.* Because of the homomorphic property, tags computed for multiple file blocks can be combined into a single value. The client pre-computes tags for each block of a file and then stores the file and its tags with a server. At a later time, the client can verify that the server possesses the file by generating a random challenge against a randomly selected set of file blocks. Using the queried blocks and their corresponding tags, the server generates a proof of possession. The client is thus convinced of data possession, without actually having to retrieve file blocks.

The efficient PDP scheme is the fundamental construct underlying an archival introspection system that we are developing for the long-term preservation of Astronomy data. We are taking possession of multi-terabyte Astronomy databases at a University library in order to preserve the information long after the research projects and instruments used to collect the data are gone. The database will be replicated at multiple sites. Sites include resource-sharing partners that exchange storage capacity to achieve reliability and scale. As such, the system is subject to freeloading in which partners attempt to use storage resources and contribute none of their own [19]. The location and physical implementation of these replicas are managed independently by each partner and will evolve over time. Partners may even outsource storage to third-party storage server providers [22]. Efficient PDP schemes will ensure that the computational requirements of remote data checking do not unduly burden the remote storage sites.

We implemented our more efficient scheme (E-PDP) and two other remote data checking protocols and evaluated their performance. Experiments show that probabilistic possession guarantees make it practical to verify possession of large data sets. With sampling, E-PDP verifies a 64MB file in about 0.4 seconds as compared to 1.8 seconds without sampling. Further, I/O bounds the performance of E-PDP; it generates proofs as quickly as the disk produces data. Finally, E-PDP is 185 times faster than the previous secure protocol on 768 KB files.

[1]Storage overhead and network overhead are constant in the size of the file, but depend on the chosen security parameter.


input file
F
client generates metadata (m) and modifed file (F')
client
F'
no server processing
server
m
m
client store
F'
server store
(1) client generates a random challenge R
R
(2) server computes proof of possession P
0/1
client
P
server
(3) client verifies server's proof
m
F'
m
client store
F'
server store

(a) Pre-process and store (b) Verify server possession

Figure 1: Protocol for provable data possession.

Contributions. In this paper we:

- formally define protocols for provable data possession (PDP) that provide probabilistic proof that a third party stores a file.
- introduce the first provably-secure and practical PDP schemes that guarantee data possession.
- implement one of our PDP schemes and show experimentally that probabilistic possession guarantees make it practical to verify possession of large data sets.

Our PDP schemes provide several features such as *public verifiability* and *data format independence*, which are relevant in practical deployments (more details on this in the remarks of Section 4.3). Moreover, our schemes put no restriction on the number of times the client can challenge the server to prove data possession.

Paper Organization. The rest of the paper is organized as follows. In Section 2, we describe a framework for provable data possession, emphasizing the features and parameters that are relevant for PDP. Section 3 overviews related work. In Section 4, we introduce homomorphic verifiable tags, followed by definitions for PDP schemes and then we give our constructions (S-PDP and E-PDP). We support our theoretical claims with experiments that show the practicality of our schemes in Section 5 and conclude in Section 6.

2 Provable Data Possession (PDP)

We describe a framework for provable data possession. This provides background for related work and for the specific description of our schemes. A PDP protocol (Fig. 1) checks that an outsourced storage site retains a file, which consists of a collection of n blocks. The client C (data owner) pre-processes the file, generating a piece of metadata that is stored locally, transmits the file to the server S, and may delete its local copy. The server stores the file and responds to challenges issued by the client. Storage at the server is in $\Omega(n)$ and storage at the client is in $O(1)$, conforming to our notion of an outsourced storage relationship.

As part of pre-processing, the client may alter the file to be stored at the server. The client may expand the file or include additional metadata to be stored at the server. Before deleting its local copy of the file, the client may execute a data possession challenge to make sure the server has successfully stored the file. Clients may encrypt a file prior to out-sourcing the storage. For our purposes, encryption is an orthogonal issue; the "file" may consist of encrypted data and our metadata does not include encryption keys.

	[19]	[19]-Wagner (MHT-SC)	[16, 18] (B-PDP)	[40]	[39]	S-PDP	E-PDP
Data possession	No	No	Yes	Yes	Yes[*]	Yes	Yes
Supports sampling	No	No	No	No	No[†]	Yes	Yes
Type of guarantee	deterministic				probabilistic / deterministic		probabilistic
Server block access	$O(n)$	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$
Server computation	$O(n)$	$O(1)$	$O(n)$	$O(1)$	$O(n)$	$O(1)$	$O(1)$
Client computation	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Communication	$O(1)$	$O(\log n)$	$O(1)$	$O(1)$	$O(n)$	$O(1)$	$O(1)$
Client storage	$O(1)$	$O(1)$	$O(1)$	$O(n)$	$O(1)$	$O(1)$	$O(1)$

Table 1: Features and parameters (per challenge) of various PDP schemes when the server misbehaves by deleting a fraction of an n-block file (*e.g.*, 1% of n). The server and client computation is expressed as the total cost of performing modular exponentiation operations. For simplicity, the security parameter is not included as a factor for the relevant costs.
[*] *No security proof is given for this scheme, so assurance of data possession is not confirmed.*
[†] *The client can ask proof for select symbols inside a block, but cannot sample across blocks.*

At a later time, the client issues a challenge to the server to establish that the server has retained the file. The client requests that the server compute a function of the stored file, which it sends back to the client. Using its local metadata, the client verifies the response.

Threat model. The server S must answer challenges from the client C; failure to do so represents a data loss. However, the server is not trusted: Even though the file is totally or partially missing, the server may try to convince the client that it possesses the file. The server's motivation for misbehavior can be diverse and includes reclaiming storage by discarding data that has not been or is rarely accessed (for monetary reasons), or hiding a data loss incident (due to management errors, hardware failure, compromise by outside or inside attacks etc). The goal of a PDP scheme that achieves probabilistic proof of data possession is to detect server misbehavior when the server has deleted a fraction of the file.

Requirements and Parameters. The important performance parameters of a PDP scheme include:

- *Computation complexity:* The computational cost to pre-process a file (at C), to generate a proof of possession (at S) and to verify such a proof (at C);
- *Block access complexity:* The number of file blocks accessed to generate a proof of possession (at S);
- *Communication complexity:* The amount of data transferred (between C and S).

For a scalable solution, the amount of computation and block accesses at the server should be minimized, because the server may be involved in concurrent interactions with many clients. *We stress that in order to minimize bandwidth, an efficient* PDP*scheme cannot consist of retrieving entire file blocks.* While relevant, the computation complexity at the client is of less importance, even though our schemes minimize that as well.

To meet these performance goals, our PDP schemes sample the server's storage, accessing a random subset of blocks. In doing so, the PDP schemes provide a probabilistic guarantee of possession; a deterministic guarantee cannot be provided without accessing all blocks. In fact, as a special case of our PDP scheme, the client may ask proof for all the file blocks, making the data possession guarantee deterministic. Sampling proves data possession with high probability based on accessing few blocks in the file, which radically alters the performance of proving data possession. Interestingly, when the server deletes a fraction of the file, the client can detect server misbehavior with high probability by asking proof for a constant amount of blocks, independently of the total number of file blocks. As an example, for a file with $n = 10,000$ blocks, if S has deleted 1% of the blocks, then C can detect server misbehavior with probability greater than 99% by asking proof of possession for only 460 randomly selected blocks (representing 4.6% of n). For more details see Section 5.1.

We list the features of our PDP schemes (S-PDP and E-PDP) in Table 1. We also include a comparison of related techniques [19], [16, 18](B-PDP), [40] and [39]. The scheme [19]-Wagner (MHT-SC) refers to the variant suggested by David Wagner [19], based on Merkle hash trees. Both schemes in [19] do not provide a data possession guarantee, but only enforce storage complexity. Indeed, as noted by Golle *et al.* [19], the server could pass the verification procedure by using (and storing) a different file, which is at least as big as the original one. We emphasize that PDP schemes that offer an inherently deterministic guarantee by accessing all the blocks of the file ([19, 16, 18]) cannot offer *both* sampling across blocks and constant storage on the client; fundamental changes would be required in these schemes in order to avoid storing $O(n)$ metadata on the client. The scheme in [39] allows the client to request proof of possession of segments from each file block. However, the server's response includes one "signature" per each block, which makes its size linear with respect to the number of blocks.

Our PDP schemes provide additional features such as *public verifiability* and *data format independence*, which extend the applicability of PDP in practical deployments (more details on this in the remarks of Section 4.3). For example, the advantages of having public verifiability are akin to those of public key over symmetric key cryptography. These features also motivated the design of our schemes.

A "strawman" solution. We notice that if one is willing to allow the server to send entire file blocks back to the client as proof of possession, then a simple solution is possible. Initially, the client computes a message authentication code (MAC) for each file block and stores the MACs at the server. Later, to obtain a probabilistic proof of data possession, the client retrieves a number of randomly selected file blocks and their corresponding MACs. However, this solution requires an arbitrarily large amount of bandwidth, which is linear with respect to the number of queried blocks. In contrast, our solutions have constant network communication.

3 Related Work

Deswarte *et al.* [16] and Filho *et al.* [18] provide techniques to verify that a remote server stores a file using RSA-based hash functions. Unlike other hash-based approaches, it allows a client to perform multiple challenges using the same metadata. In this protocol, communication and client storage complexity are both $O(1)$. The limitation of the algorithm lies in the computational complexity at the server, which must exponentiate the entire file, accessing all of the file's blocks. Further, RSA over the entire file is extremely slow — 20 seconds per Megabyte for 1024-bit keys on a 3.0 GHz processor [18]. In fact, these limitations led us to study algorithms that allowed for sub-file access (sampling). We implement this protocol for comparison with our PDP scheme and refer to it as B-PDP (basic PDP). A description of B-PDP is provided in Appendix B. Shah *et al.*

[41] use a similar technique for third-party auditing of data stored at online service providers and put forth some of the challenges associated with auditing online storage services.

Schwarz and Miller [39] propose a scheme that allows a client to verify the storage of m/n erasure-coded data across multiple sites even if sites collude. The data possession guarantee is achieved using a special construct, called an "algebraic signature": A function that fingerprints a block and has the property that the signature of the parity block equals the parity of the signatures of the data blocks. The parameters of the scheme limit its applicability: The file access and computation complexity at the server and the communication complexity are all linear in the number of file blocks (n) per challenge. Additionally, the security of the scheme is not proven and remains in question.

Sebe *et al.* [40] give a protocol for remote file integrity checking, based on the Diffie-Hellman problem in $\mathbb{Z}_N$. The client has to store N bits per block, where N is the size of an RSA modulus, so the total storage on the client is $O(n)$ (which does not conform to our notion of an outsourced storage relationship). Indeed, the authors state that this solution only makes sense if the size of a block is much larger than N. Moreover, the protocol requires the server to access the entire file.

Homomorphic hash functions have been used in *source authentication*, *e.g.*, to verify data transfer in a pay-for-bandwidth content distribution network. Krohn *et al.* [27] provide the best example and give a good review of other applications of homomorphic hashing. They use homomorphism to compose multiple blocks inputs into a single value. However, this protocol can only compose specific subsets of blocks, based on erasure coding. Source authentication techniques do not apply to provable data possession.

Related to provable data possession is the enforcement of storage complexity, which shows that a server retains *an amount of information at least as large as the file* received from the client; the server does not necessarily retain the original file. To the best of our knowledge, Golle *et al.* [19] were the first to propose a scheme that enforces storage complexity, based on a new assumption (n-Power Decisional Diffie-Hellman). Their scheme considers a different setting that involves three parties and provides two additional features (binding and concealing) that resemble commitment schemes. Golle *et al.* also briefly mention a scheme suggested by David Wagner, based on Merkle hash trees, which lowers the computational requirements for the server at the expense of increased communication. We implement Wagner's suggestion for comparison with our PDP scheme and refer to it as MHT-SC. A description of MHT-SC is provided in Appendix B.

Oprea *et al.* [38] propose a scheme based on tweakable block ciphers that allows a client to detect the modification of data blocks by an untrusted server. The scheme does not require additional storage at the server and if the client's data has low entropy then the client only needs to keep a relatively low amount of state. However, verification requires the entire file to be retrieved, which means that the server file access and communication complexity are both linear with the file size per challenge. The scheme is targeted for data retrieval. It is impractical for verifying data possession.

Memory checking protocols [11, 36] verify that all reads and writes to a remote memory behave identically to reads and writes to a local memory. PDP is a restricted form of memory checking in that memory checking verifies every read and write generated from a program at any granularity. Due to this restriction, memory checking is much more difficult and expensive than proving data possession. Related to PDP, Naor and Rothblum introduced the problem of *sub-linear authentication* [36], verifying that a file stored on a remote server has not been significantly corrupted. They show that the existence of one-way functions is an essential condition for efficient online checking.

Similar to PDP, Juels and Kaliski introduce the notion of *proof of retrievability* (POR) [24], which allows a server to convince a client that it can retrieve a file that was previously stored at the server. The main POR scheme uses disguised blocks (called sentinels) hidden among regular file blocks in order to detect data modification by the server. Although comparable in scope with PDP, their POR scheme can only be applied to encrypted files and can handle only a limited number of queries, which has to be fixed a priori. In contrast, our PDP scheme can be applied to (large) public databases (*i.e.*, digital libraries, astronomy/medical/legal repositories, archives, etc.) other than encrypted ones. In addition, much like public-key signatures, our scheme allows anyone to verify data possession, not just the data owner (*public verifiability*).

An alternative to checking remote storage is to make data resistant to undetectable deletion through entanglement [44, 2], which encodes data to create dependencies among unrelated data throughout the storage system. Thus, deleting any data reveals itself as it deletes other unrelated data throughout the system.

Our homomorphic verifiable tags are related with the concept of homomorphic signatures introduced by Johnson *et al.* [23]. As pointed out by the same authors, fully-additive homomorphic signature schemes cannot be secure; our homomorphic verifiable tags are able to achieve a form of message homomorphism under addition because of their special structure that uses one-time indices.

The PDP solutions that we present are the first schemes that securely prove the possession of data on an untrusted server and are computationally efficient, *i.e.*, require a constant number of modular exponentiations and have constant I/O complexity. This makes our PDP solutions the first schemes suitable for implementation in large data systems.

4 Provable Data Possession Schemes

4.1 Preliminaries

The client C wants to store on the server S a file $\mathtt{F}$ which is a finite ordered collection of n blocks: $\mathtt{F} = (m_1, \ldots, m_n)$.
We denote the output x of an algorithm $\mathcal{A}$ by $x \leftarrow \mathcal{A}$.

Homomorphic Verifiable Tags (HVTs). We introduce the concept of a homomorphic verifiable tag that will be used as a building block for our PDP schemes.

Given a message m (corresponding to a file block), we denote by $\mathtt{T}_m$ its homomorphic verifiable tag. The tags will be stored on the server together with the file $\mathtt{F}$. Homomorphic verifiable tags act as verification metadata for the file blocks and, besides being unforgeable, they also have the following properties:

– *Blockless verification*: Using HVTs the server can construct a proof that allows the client to verify if the server possesses certain file blocks, even when the client does not have access to the actual file blocks.
– *Homomorphic tags*: Given two values $\mathtt{T}_{m_i}$ and $\mathtt{T}_{m_j}$, anyone can combine them into a value $\mathtt{T}_{m_i+m_j}$ corresponding to the sum of the messages $m_i + m_j$.

In our construction, an HVT is a pair of values $(\mathtt{T}_{i,m}, \mathtt{W}_i)$, where $\mathtt{W}_i$ is a random value obtained from an index i and $\mathtt{T}_{i,m}$ is stored at the server. The index i can be seen as a *one-time* index because it is never reused for computing tags (a simple way to ensure that every tag uses a different index i is to use a global counter for i). The random value $\mathtt{W}_i$ is generated by applying a pseudo-random function on the index i, which ensures that $\mathtt{W}_i$ is different and unpredictable each time a tag is computed. HVTs and their corresponding proofs have a fixed constant size and are (much) smaller than the actual file blocks.

We emphasize that techniques based on *aggregate signatures* [12], *multi-signatures* [32, 37], batch RSA [17], batch verification of RSA [21, 3], condensed RSA [35], etc. would all fail to provide *blockless verification*, which is needed by our PDP scheme. Indeed, the client has to have the ability to verify the tags on *specific* file blocks even though he *does not* possess any of those blocks.

4.2 Definitions

We start with the precise definition of a provable data possession scheme, followed by a security definition that captures the data possession property.

Definition 4.1 (Provable Data Possession Scheme (PDP)) *A PDP scheme is a collection of four polynomial-time algorithms* ($\mathsf{KeyGen}, \mathsf{TagBlock}, \mathsf{GenProof}, \mathsf{CheckProof}$) *such that:*

$\mathsf{KeyGen}(1^k) \rightarrow (\mathrm{pk}, \mathrm{sk})$ *is a probabilistic key generation algorithm that is run by the client to setup the scheme. It takes a security parameter k as input, and returns a pair of matching public and secret keys* $(\mathrm{pk}, \mathrm{sk})$.

$\mathsf{TagBlock}(\mathrm{sk}, m) \rightarrow \mathtt{T}_m$ *is a (possibly probabilistic) algorithm run by the client to generate the verification metadata. It takes as inputs a secret key* sk *and a file block m, and returns the verification metadata $\mathtt{T}_m$.*

$\mathsf{GenProof}(\mathrm{pk}, \mathtt{F}, \mathsf{chal}, \Sigma) \rightarrow \mathcal{V}$ *is run by the server in order to generate a proof of possession. It takes as inputs a public key* pk*, an ordered collection $\mathtt{F}$ of blocks, a challenge* chal *and an ordered collection Σ which is the verification metadata corresponding to the blocks in $\mathtt{F}$. It returns a proof of possession $\mathcal{V}$ for the blocks in $\mathtt{F}$ that are determined by the challenge* chal.

$\mathsf{CheckProof}(\mathrm{pk}, \mathrm{sk}, \mathsf{chal}, \mathcal{V}) \rightarrow \{\text{“}success\text{”}, \text{“}failure\text{”}\}$ *is run by the client in order to validate a proof of possession. It takes as inputs a public key* pk*, a secret key* sk*, a challenge* chal *and a proof of possession $\mathcal{V}$. It returns whether $\mathcal{V}$ is a correct proof of possession for the blocks determined by* chal.

A PDP system can be constructed from a PDP scheme in two phases, Setup and Challenge:

Setup: The client C is in possession of the file $\mathtt{F}$ and runs $(\mathrm{pk}, \mathrm{sk}) \leftarrow \mathsf{KeyGen}(1^k)$, followed by $\mathtt{T}_{m_i} \leftarrow \mathsf{TagBlock}(\mathrm{sk}, m_i)$ for all $1 \leq i \leq n$. C stores the pair $(\mathrm{sk}, \mathrm{pk})$. C then sends $\mathrm{pk}, \mathtt{F}$ and $\Sigma = (\mathtt{T}_{m_1}, \ldots, \mathtt{T}_{m_n})$ to S for storage and deletes $\mathtt{F}$ and Σ from its local storage.

Challenge: C generates a challenge chal that, among other things, indicates the specific blocks for which C wants a proof of possession. C then sends chal to S. S runs $\mathcal{V} \leftarrow \mathsf{GenProof}(\mathrm{pk}, \mathtt{F}, \mathsf{chal}, \Sigma)$ and sends to C the proof of possession $\mathcal{V}$. Finally, C can check the validity of the proof $\mathcal{V}$ by running $\mathsf{CheckProof}(\mathrm{pk}, \mathrm{sk}, \mathsf{chal}, \mathcal{V})$.

In the Setup phase, C computes tags for each file block and stores them together with the file at S. In the Challenge phase, C requests proof of possession for a subset of the blocks in $\mathtt{F}$. This phase can be executed an unlimited number of times in order to ascertain whether S still possesses the selected blocks.

We state the security for a PDP system using a game that captures the data possession property. Intuitively, the Data Possession Game captures that an adversary cannot successfully construct a valid proof without possessing all the blocks corresponding to a given challenge, unless it guesses all the missing blocks.

Data Possession Game:

- **Setup**: The challenger runs $(\mathrm{pk}, \mathrm{sk}) \leftarrow \mathsf{KeyGen}(1^k)$, sends pk to the adversary and keeps sk secret.
- **Query**: The adversary makes tagging queries adaptively: It selects a block m_1 and sends it to the challenger. The challenger computes the verification metadata $\mathtt{T}_{m_1} \leftarrow \mathsf{TagBlock}(\mathrm{sk}, m_1)$ and sends it back to the adversary. The adversary continues to query the challenger for the verification metadata $\mathtt{T}_{m_2}, \ldots, \mathtt{T}_{m_n}$ on the blocks of its choice $m_2, \ldots, m_n$. As a general rule, the challenger generates $\mathtt{T}_{m_j}$ for some $1 \leq j \leq n$, by computing $\mathtt{T}_{m_j} \leftarrow \mathsf{TagBlock}(\mathrm{sk}, m_j)$. The adversary then stores all the blocks as an ordered collection $\mathtt{F} = (m_1, \ldots, m_n)$, together with the corresponding verification metadata $\mathtt{T}_{m_1}, \ldots, \mathtt{T}_{m_n}$. At any point, the adversary can check the validity of the tags received.
- **Challenge**: The challenger generates a challenge chal and requests the adversary to provide a proof of possession for the blocks $m_{i_1}, \ldots, m_{i_c}$ determined by chal, where $1 \leq i_j \leq n, 1 \leq j \leq c, 1 \leq c \leq n$.
- **Forge**: The adversary computes a proof of possession $\mathcal{V}$ for the blocks indicated by chal and returns $\mathcal{V}$.

If $\mathsf{CheckProof}(\mathrm{pk}, \mathrm{sk}, \mathsf{chal}, \mathcal{V}) =$ "*success*", then the adversary has won the Data Possession Game.

Definition 4.2 *A PDP system* $(\mathsf{Setup}, \mathsf{Challenge})$ *built on a PDP scheme* $(\mathsf{KeyGen}, \mathsf{TagBlock}, \mathsf{GenProof}, \mathsf{CheckProof})$ *guarantees data possession if for any (probabilistic polynomial-time) adversary $\mathcal{A}$ the probability that $\mathcal{A}$ wins the Data Possession Game on a set of file blocks is negligibly close to the probability that the challenger can extract those file blocks by means of a knowledge extractor $\mathcal{E}$.*

In our security definition, the notion of a knowledge extractor is similar with the standard one introduced in the context of proofs of knowledge [4]. If the adversary is able to win the Data Possession Game, then $\mathcal{E}$ can execute $\mathsf{GenProof}$ repeatedly until it extracts the selected blocks. On the other hand, if $\mathcal{E}$ cannot extract the blocks, then the adversary cannot win the game with more than negligible probability. We emphasize that, because of the public verifiability feature, we allow the adversary to check the validity of the tags during the Query phase in the Data Possession Game described above.

We refer the reader to [24] for a more generic and extraction-based security definition for POR and to [36] for the security definition of sub-linear authenticators.

4.3 Efficient and Secure PDP Schemes

In this section we present our PDP constructions: The first ($\mathsf{S\text{-}PDP}$) provides a strong data possession guarantee, while the second ($\mathsf{E\text{-}PDP}$) achieves better efficiency at the cost of weakening the data possession guarantee.

We start by introducing some additional notation used by the constructions. Let $p = 2p' + 1$ and $q = 2q' + 1$ be safe primes and let $N = pq$ be an RSA modulus. Let g be a generator of QR_N, the unique cyclic subgroup of $\mathbb{Z}_N^*$ of order $p'q'$ (*i.e.*, QR_N is the set of quadratic residues modulo N). We can obtain g as $g = a^2$, where $a \overset{R}{\leftarrow} \mathbb{Z}_N^*$ such that $\gcd(a \pm 1, N) = 1$. All exponentiations are performed modulo N, and for simplicity we sometimes omit writing it explicitly.

Let $h : \{0,1\}^* \rightarrow QR_N$ be a secure deterministic hash-and-encode function[2] that maps strings uniformly to QR_N.

The schemes are based on the KEA1 assumption which was introduced by Damgard in 1991 [13] and subsequently used by several others, most notably in [20, 5, 6, 26, 14]. In particular, Bellare and Palacio [5] provided a formulation of KEA1, that we follow and adapt to work in the RSA ring:

KEA1-r (**K**nowledge of **E**xponent **A**ssumption): For any adversary $\mathbf{A}$ that takes input (N, g, g^s) and returns (C, Y) such that $Y = C^s$, there exists an "extractor" $\bar{\mathbf{A}}$ which, given the same inputs as $\mathbf{A}$, returns x such that $g^x = C$.

Recently, KEA1 has been shown to hold in generic groups (*i.e.*, it is secure in the generic group model) by A. Dent [15] and independently by Abe and Fehr [1]. Later in this section, we also show an alternative strategy which does not rely on the KEA1-r assumption, at the cost of increased network communication.

S-PDP overview. We first give an overview of our provable data possession scheme that supports sampling. In the Setup phase, the client computes a homomorphic verifiable tag $(\mathtt{T}_{i,m_i}, \mathtt{W}_i)$ for each block m_i of the file. In order to maintain constant storage, the client generates the random values $\mathtt{W}_i$ using a pseudo-random function; thus, TagBlock has an extra parameter, i. Each value $\mathtt{T}_{i,m_i}$ includes information about the index i of the block m_i, in the form of a value $h(\mathtt{W}_i)$, as $\mathtt{W}_i$ is obtained from i by using a pseudo-random function[3]. This binds the tag on a block to that specific block, and prevents using the tag to obtain a proof for a different block. These tags are stored on the server together with the file $\mathtt{F}$. The extra storage at the server is the price to pay for allowing thin clients that only store a small, constant amount of data, regardless of the file size.

In the Challenge phase, the client asks the server for proof of possession of c file blocks whose indices are randomly chosen using a pseudo-random permutation keyed with a fresh randomly-chosen key for each challenge. This prevents the server from anticipating which blocks will be queried in each challenge. C also generates a fresh (random) challenge $g_s = g^s$ to ensure that S does not reuse any values from a previous Challenge phase. The server returns a proof of possession that consists of two values: $\mathtt{T}$ and ρ. $\mathtt{T}$ is obtained by combining into a single value the individual tags $\mathtt{T}_{i,m_i}$ corresponding to each of the requested blocks. ρ is obtained by raising the challenge g_s to a function of the requested blocks. The value $\mathtt{T}$ contains information about the indices of the blocks requested by the client (in the form of the $h(\mathtt{W}_i)$ values). C can remove all the $h(\mathtt{W}_i)$ values from $\mathtt{T}$ because it has both the key for the pseudo-random permutation (used to determine the indices of the requested blocks) and the key for the pseudo-random function (used to generate the values $\mathtt{W}_i$). C can then verify the validity of the server's proof by checking if a certain relation holds between $\mathtt{T}$ and ρ.

[2]h is computed by squaring the output of the full-domain hash function for the provably secure FDH signature scheme [7, 8] based on RSA. We refer the reader to [7] for ways to construct an FDH function out of regular hash functions, such as SHA-1. Alternatively, h can be the deterministic encoding function used in RSA-PSS[9].

[3] *"Hashing"* the output of a PRF is needed by our security proof to cover the case when public verifiability is enabled. The function $h(\cdot)$ is essential to prevent forgeries of tags and it is also an encoding function which maps strings into QR_N.

As previously defined, let f and w be pseudo-random functions, let π be a pseudo-random permutation and let H be a cryptographic hash function.

```
KeyGen$(1^k)$: Generate $\mathrm{pk} = (N, e, g)$ and $\mathrm{sk} = (N, d, v)$, such that $(N, e)$ and $(N, d)$ are matching
RSA public and secret keys, and $v \overset{R}{\leftarrow} \{0,1\}^\kappa$. Output $(\mathrm{pk}, \mathrm{sk})$.

TagBlock$(\mathrm{sk}, m, i)$:
1.  Let $(N, d, v) = \mathrm{sk}$. Generate $\mathtt{W}_i = w_v(i)$. Compute $\mathtt{T}_{i,m} = (h(\mathtt{W}_i) \cdot g^m)^d \bmod N$.
2.  Output $(\mathtt{T}_{i,m}, \mathtt{W}_i)$.

GenProof$(\mathrm{pk}, \mathtt{F} = (m_1, \ldots, m_n), \mathsf{chal}, \Sigma = (\mathtt{T}_{1,m_1}, \ldots, \mathtt{T}_{n,m_n}))$:
1.  Let $(N, e, g) = \mathrm{pk}$ and $(c, k_1, k_2, g_s) = \mathsf{chal}$.
    For $1 \le j \le c$:
      • compute the indices of the blocks for which the proof is generated: $i_j = \pi_{k_1}(j)$
      • compute coefficients: $a_j = f_{k_2}(j)$.
2.  Compute $\mathtt{T} = \mathtt{T}^{a_1}_{i_1,m_{i_1}} \cdot \ldots \cdot \mathtt{T}^{a_c}_{i_c,m_{i_c}} = (h(\mathtt{W}_{i_1})^{a_1} \cdot \ldots \cdot h(\mathtt{W}_{i_c})^{a_c} \cdot g^{a_1 m_{i_1} + \ldots + a_c m_{i_c}})^d \bmod N$.
    (note that $\mathtt{T}_{i_j,m_{i_j}}$ is the $i_j$-th value in $\Sigma$).
3.  Compute $\rho = H(g_s^{a_1 m_{i_1} + \ldots + a_c m_{i_c}} \bmod N)$.
4.  Output $\mathcal{V} = (\mathtt{T}, \rho)$.

CheckProof$(\mathrm{pk}, \mathrm{sk}, \mathsf{chal}, \mathcal{V})$:
1.  Let $(N, e, g) = \mathrm{pk}$, $v = \mathrm{sk}$, $(c, k_1, k_2, s) = \mathsf{chal}$ and $(\mathtt{T}, \rho) = \mathcal{V}$.
2.  Let $\tau = \mathtt{T}^e$. For $1 \le j \le c$:
      • compute $i_j = \pi_{k_1}(j)$, $\mathtt{W}_{i_j} = w_v(i_j)$, $a_j = f_{k_2}(j)$, and $\tau = \frac{\tau}{h(\mathtt{W}_{i_j})^{a_j}} \bmod N$
    (As a result, one should get $\tau = g^{a_1 m_{i_1} + \ldots + a_c m_{i_c}} \bmod N$)
3.  If $H(\tau^s \bmod N) = \rho$, then output "success". Otherwise output "failure".
```

We construct a PDP system from a PDP scheme in two phases, Setup and Challenge:

Setup: The client C runs $(\mathrm{pk}, \mathrm{sk}) \leftarrow \mathsf{KeyGen}(1^k)$, followed by $(\mathtt{T}_{i,m_i}, \mathtt{W}_i) \leftarrow \mathsf{TagBlock}(\mathrm{sk}, m_i, i)$ for all $1 \le i \le n$ (note that TagBlock has an extra parameter, i, since the random values $\mathtt{W}_i$ are generated using a PRF). C stores the pair $(\mathrm{sk}, \mathrm{pk})$. C then sends $\mathrm{pk}, \mathtt{F}$ and $\Sigma = (\mathtt{T}_{1,m_1}, \ldots, \mathtt{T}_{n,m_n})$ to S for storage and deletes $\mathtt{F}$ and Σ from its local storage.

Challenge: C requests proof of possession for c distinct blocks of the file $\mathtt{F}$ (with $1 \le c \le n$):

1. C generates the challenge $\mathsf{chal} = (c, k_1, k_2, g_s)$, where $k_1 \overset{R}{\leftarrow} \{0,1\}^\kappa$, $k_2 \overset{R}{\leftarrow} \{0,1\}^\kappa$, $g_s = g^s \bmod N$ and $s \overset{R}{\leftarrow} \mathbb{Z}_N^*$. C sends chal to S.
2. S runs $\mathcal{V} \leftarrow \mathsf{GenProof}(\mathrm{pk}, \mathtt{F}, \mathsf{chal}, \Sigma = (\mathtt{T}_{1,m_1}, \ldots, \mathtt{T}_{n,m_n}))$ and sends to C the proof of possession $\mathcal{V}$.
3. C sets $\mathsf{chal} = (c, k_1, k_2, s)$ and checks the validity of the proof $\mathcal{V}$ by running $\mathsf{CheckProof}(\mathrm{pk}, v, \mathsf{chal}, \mathcal{V})$.

Figure 2: S-PDP: a PDP scheme with strong data possession guarantee

S-PDP in detail. Let κ, ℓ be security parameters and let H be a cryptographic hash function. In addition, we make use of two pseudo-random functions (PRF) f, w, and a pseudo-random permutation (PRP) π with the following parameters:

- $f : \{0,1\}^\kappa \times \{0,1\}^{\log_2(n)} \to \{0,1\}^\ell$;
- $w : \{0,1\}^\kappa \times \{0,1\}^{\log_2(n)} \to \{0,1\}^\ell$;
- $\pi : \{0,1\}^\kappa \times \{0,1\}^{\log_2(n)} \to \{0,1\}^{\log_2(n)}$

We write $f_k(x)$ to denote f keyed with key k applied on input x. Our S-PDP scheme is described in Fig. 2. The purpose of including the the a_j coefficients in the values for ρ and $\mathtt{T}$ computed by S is to ensure that S possesses each one of the requested blocks. These coefficients are determined by a PRF keyed with a fresh randomly-chosen key for each challenge, which prevents S from storing combinations (*e.g.*, sums) of the original blocks instead of the original file blocks themselves. Also, we are able to maintain constant communication cost because tags on blocks can be combined into a single value.

In Appendix A, we prove:

Theorem 4.3 *Under the RSA and KEA1-r assumptions,* S-PDP *guarantees data possession in the random oracle model.*

Regarding efficiency, we remark that each challenge requires a small, constant amount of communication between C and S (the challenge and the response are each slightly more than 1 Kilobit). In terms of server block access, the demands are c accesses for S, while in terms of computation we have c exponentiations for both C and S. When S deletes a fraction of the file blocks, c is a relatively small, constant value, which gives the $O(1)$ parameters in Table 1 (for more details, see Section 5.1). Since the size of the file is $O(n)$, accommodating the additional tags does not change (asymptotically) the storage requirements for the server.

In our analysis we assume w.l.o.g. that the indices for the blocks picked by the client in a challenge are different. One way to achieve this is to implement π using the techniques proposed by Black and Rogaway [10]. In a practical deployment, our protocol can tolerate collisions of these indices. Moreover, notice that the client can dynamically append new blocks to the stored file after the Setup phase, without re-tagging the entire file.

Notice that the server may store the client's file $\mathtt{F}$ however it sees fit, as long as it is able to recover the file when answering a challenge. For example, it is allowed to compress $\mathtt{F}$ (*e.g.*, if all the blocks of $\mathtt{F}$ are identical, then the server may only store one full block and the information that all the blocks are equal). Alternatively, w.l.o.g., one could assume that $\mathtt{F}$ has been optimally compressed by the client and the size of $\mathtt{F}$ is equal to $\mathtt{F}$'s information entropy function.

A concrete example of using S-PDP. For a concrete example of using S-PDP, we consider a 1024-bit modulus N and a 4 GB file $\mathtt{F}$ which has $n = 1,000,000$ 4KB blocks. During Setup, C stores the file and the tags at S. The tags require additional storage of 128 MB. The client stores less than 2048 bits (N has 1024 bits and e, d, v have less than 1024 bits). During the Challenge phase, C and S use AES for π (used to select the random block indices i), HMAC for w (used to compute the random values $\mathtt{W}$), HMAC for f (used to determine the random coefficients a) and SHA1 for H. In a challenge, C sends to S four values which total 168 bytes (c has 4 bytes, k_1 has 16 bytes, k_2 has 20 bytes, g_s has 1024 bits). Assuming that S deletes at least 1% of $\mathtt{F}$, then C can detect server misbehavior with probability over 99% by asking proof for $c = 460$ randomly selected blocks. The server's response contains two values which total 148 bytes ($\mathtt{T}$ has 1024 bits, ρ has 20 bytes). We emphasize that the server's response to a challenge consists of a small, constant value; in particular, the server does not send back to the client any of the file blocks and not even their sum.

A more efficient scheme, with weaker guarantees (E-PDP). Our S-PDP scheme provides the guarantee that S possesses each one of the c blocks for which C requested proof of possession in a challenge. We now describe a more efficient variant of S-PDP, which we call E-PDP, that achieves better performance at the cost of offering weaker guarantees. E-PDP differs from S-PDP only in that all the coefficients a_j are equal to 1:

- In GenProof (steps 2 and 3) the server computes $\mathtt{T} = \mathtt{T}_{i_1,m_1} \cdot \ldots \cdot \mathtt{T}_{i_c,m_c}$ and $\rho = H(g_s^{m_{i_1}+\ldots+m_{i_c}} \bmod N)$.
- In CheckProof (step 2) the client computes $\tau = \dfrac{\mathtt{T}^e}{h(\mathtt{W}_{i_1}) \cdot \ldots \cdot h(\mathtt{W}_{i_c})} \bmod N$.

The E-PDP scheme reduces the computation on both the server and the the client to one exponentiation, as described in Table 1 (see Server Computation details in Section 5.2, the server computes ρ as one exponentiation to a value whose size in bits is slightly larger than $|m_i|$). E-PDP only guarantees possession of the sum of the blocks $m_{i_1} + \ldots + m_{i_c}$ and not necessarily possession of each one of the blocks for which the client requests proof of possession. However, we argue next that this is not a practical concern.

If the server pre-computes and stores the sums of all possible combinations of c blocks out of the n blocks ($\binom{n}{n-c}$ values), then the server could successfully pass any challenge. When using E-PDP, the client needs to choose the parameters of the scheme such that it is infeasible for the server to misbehave by pre-computing and storing all these sums. For example, if $n = 1000$ and $c = 101$, the server needs to pre-compute and store $\approx 10^{140}$ values and might be better off trying to factor N. The client can also reduce the server's ability to misbehave by choosing c as a prime integer and by using a different c value for each challenge.

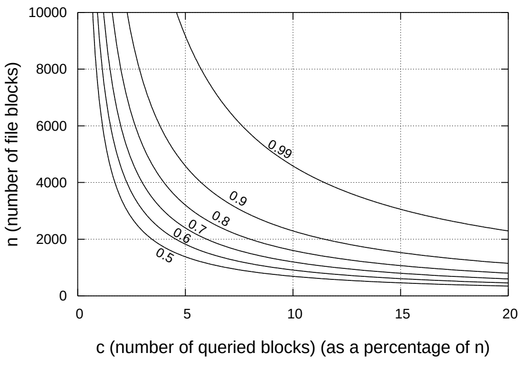
An alternative strategy. Instead of the value ρ, the server S could send the sum of the queried blocks as an integer (S does not know the order of QR_N) and let C verify the proof of possession using this value. Thus, we will not have to rely on the KEA1-r assumption. However, network communication will increase by a significant amount.

In addition, notice that any solution based on proofs of knowledge of the sum would require even more bandwidth than just sending the sum itself. This is, again, because S does not know the order of QR_N and would have to work with large integers.

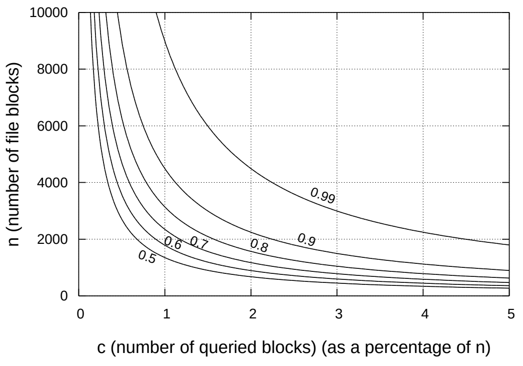
Remark 1 (Public Verifiability). Our PDP schemes provide *public verifiability*: After the initial Setup phase, the client (data owner) can publish N, e, g and v (the key for PRF w) so that anyone can challenge the server to verify data possession. The rest of the protocol remains unchanged.

Remark 2 (Prime-order Group Variant). Alternatively, our PDP schemes can be modified to work within a group of a publicly-known prime order q. In this case, however, file blocks (seen as integers) must be less than q, otherwise the server could simply store them reduced modulo q. In a prime-order setting, network communication is further reduced (particularly in the elliptic curve setting), but pre-processing becomes more expensive given the small size of the file blocks. In contrast, the RSA setting allows us to work with arbitrarily large file blocks. Most importantly, the prime-order group variant does not provide the public verifiability property.

Remark 3 (Data Format Independence). Our PDP schemes put no restriction on the format of the data, in particular files stored at the server do not have to be encrypted. This feature is very relevant since we anticipate that PDP schemes will have the biggest impact when used with large public repositories.



(a) $t = 1\%$ of n



(b) $t = 5\%$ of n

Figure 3: P_X, the probability of server misbehavior detection. We show P_X as a function of n (the number of file blocks) and c (the number of blocks queried by the client, shown as a percentage of n) for two values of t (the number of blocks deleted by the server). Note that each graph has a different scale.

Remark 4 (Multiple Files). We have described PDP schemes for the case when a client stores a single file $\mathtt{F}$ on the server. In the $\mathsf{TagBlock}$ algorithm, for each block m_i the client computes a tag over the tuple $(\mathtt{W}_i, m_i)$. We emphasize that the values $\mathtt{W}_i$ cannot be reused. Since $\mathtt{W}_i$ is obtained from i using a PRF, this implies that the indices i must be different across all tags. In other words, the client should not use the same index twice for computing tags. This condition holds because in our scheme an index i is simply the position of the block m_i in the file.

In order to store multiple files on the server, the client must ensure that indices used to compute tags are distinct not only across the tags corresponding to the blocks of each file, but also across the tags corresponding to the blocks of all files. One simple method to achieve this is to prepend the file's identifier to the index. For example, if the identifier of a file $\mathtt{F} = (m_1, \dots, m_n)$ is given by $\mathsf{id}(\mathtt{F})$, then for each block m_i, $1 \leq i \leq n$, C computes the tag $(\mathtt{T}_{\mathsf{id}(\mathtt{F})||i,m_i}, \mathtt{W}_{\mathsf{id}(\mathtt{F})||i}) \leftarrow \mathsf{TagBlock}(\mathrm{sk}, m_i, \mathsf{id}(\mathtt{F})||i)$. The uniqueness of indices is ensured under the assumption that each file has a unique identifier. Another simple way to ensure that indices are only used once is to use a global counter for the index, which is incremented by the client each time after a tag is computed.

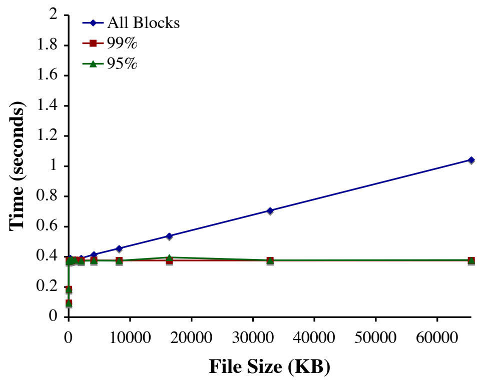
5 System Implementation and Performance Evaluation

5.1 Probabilistic Framework

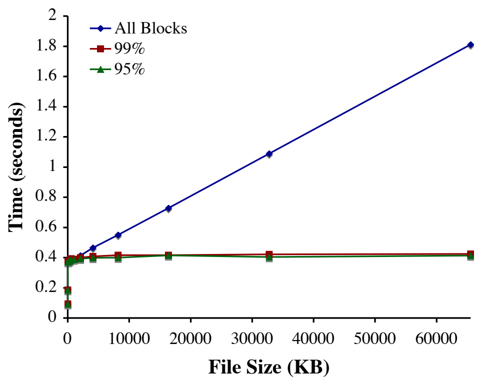
Our PDP schemes allow the server to prove possession of select blocks of $\mathtt{F}$. This "sampling" ability greatly reduces the workload on the server, while still achieving detection of server misbehavior with high probability. We now analyze the probabilistic guarantees offered by a scheme that supports block sampling.

Assume S deletes t blocks out of the n-block file $\mathtt{F}$. Let c be the number of different blocks for which C asks proof in a challenge. Let X be a discrete random variable that is defined to be the number of blocks chosen by C that match the blocks deleted by S. We compute P_X, the probability that at least one of the blocks picked by C matches one of the blocks deleted by S. We have:

$$P_X = P\{X \geq 1\} = 1 - P\{X = 0\} = 1 - \frac{n-t}{n} \cdot \frac{n-1-t}{n-1} \cdot \frac{n-2-t}{n-2} \cdot \ldots \cdot \frac{n-c+1-t}{n-c+1}.$$



(a) Server (in cache)



(b) Server (on disk)

Figure 4: Performance of sampling at multiple confidence levels.

Since $\frac{n-i-t}{n-i} \geq \frac{n-i-1-t}{n-i-1}$, it follows that:

$$1 - \left(\frac{n-t}{n}\right)^c \leq P_X \leq 1 - \left(\frac{n-c+1-t}{n-c+1}\right)^c.$$

P_X indicates the probability that, if S deletes t blocks of the file, then C will detect server misbehavior after a challenge in which it asks proof for c blocks. Fig. 3 plots P_X for different values of n, t, c. Interestingly, when t is a fraction of the file, C can detect server misbehavior with a certain probability by asking proof for a constant amount of blocks, independently of the total number of file blocks: *e.g.*, if $t = 1\%$ of n, then C asks for 460 blocks and 300 blocks in order to achieve P_X of at least 99% and 95%, respectively.

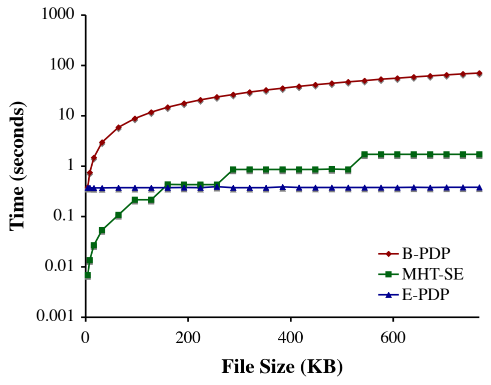
5.2 Implementation and Experimental Results

We measure the performance of E-PDP and the benefits of sampling based on our implementation of E-PDP in Linux. As a basis for comparison, we have also implemented the scheme of Deswarte *et al.* [16] and Filho *et al.* [18] (B-PDP), and the more efficient scheme in [19] (MHT-SC) suggested by David Wagner (these schemes are described in Appendix B).

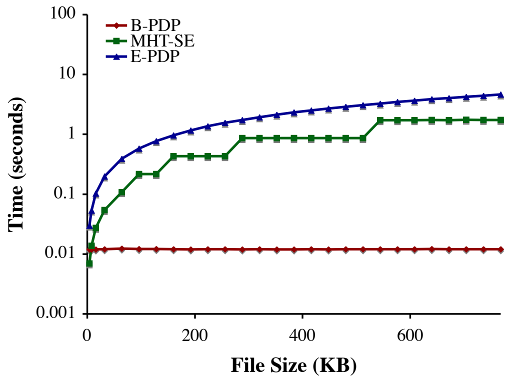
All experiments were conducted on an Intel 2.8 GHz Pentium IV system with a 512 KB cache, an 800 MHz EPCI bus, and 1024 MB of RAM. The system runs Red Hat Linux 9, kernel version 2.4.22. Algorithms use the crypto library of OpenSSL version 0.9.8b with a modulus N of size 1024 bits and files have 4KB blocks. Experiments that measure disk I/O performance do so by storing files on an ext3 file system on a Seagate Barracuda 7200.7 (ST380011A) 80GB Ultra ATA/100 drive. All experimental results represent the mean of 20 trials. Because results varied little across trials, we do not present confidence intervals.

Sampling. To quantify the performance benefits of sampling for E-PDP, we compare the client and server performance for detecting 1% missing or faulty data at 95% and 99% confidence (Fig. 4). These results are compared with using E-PDP over all blocks of the file at large file sizes, up to 64 MB. We measure both the computation time only (in memory) as well as the overall time (on disk), which includes I/O costs.

Examining all blocks uses time linear in the file size for files larger than 4MB. This is the



(a) Challenge time compared



(b) Pre-processing time compared

Figure 5: Computation performance.

point at which the computation becomes bound from either memory or disk throughput. Larger inputs amortize the cost of the single exponentiation required by E-PDP. This is also the point at which the performance of sampling diverges. The number of blocks needed to achieve the target confidence level governs performance.

For larger files, E-PDP generates data as fast as it can be accessed from memory and summed, because it only computes a single exponentiation. In E-PDP, the server generates $\sum_{i=1}^{c} m_i$, which it exponentiates. The maximum size of this quantity in bits is $|m_i| + \log_2(c)$; its maximum value is $c \cdot 2^{|m_i|}$. Thus, the cryptographic costs grows logarithmically in the file size. The linear cost of accessing all data blocks and computing the sum dominate this logarithmic growth.

Comparing results when data are on disk versus in cache shows that disk throughput bounds E-PDP's performance when accessing all blocks. With the exception of the first blocks of a file, I/O and the challenge computation occur in parallel. Thus, E-PDP generates proofs faster than the disk can deliver data: 1.0 second versus 1.8 seconds for a 64 MB file. Because I/O bounds performance, no protocol can outperform E-PDP by more than the startup costs. While faster, multiple-disk storage may remove the I/O bound today. Over time increases in processor speeds will exceed those of disk bandwidth and the I/O bound will hold.

Sampling breaks the linear scaling relationship between time to generate a proof of data possession and the size of the file. At 99% confidence, E-PDP can build a proof of possession for any file, up to 64 MB in size in about 0.4 seconds. Disk I/O incurs about 0.04 seconds of additional runtime for larger file sizes over the in-memory results. Sampling performance characterizes the benefits of E-PDP. Probabilistic guarantees make it practical to use public-key cryptography constructs to verify possession of very large data sets.

Server Computation. The next experiments look at the worst-case performance of generating a proof of possession, which is useful for planning purposes to allow the server to allocate enough resources. For E-PDP, this means sampling every block in the file, while for MHT-SC this means computing the entire hash tree. We compare the computation complexity of E-PDP with other algorithms, which do not support sampling. All schemes perform an equivalent number of disk and memory accesses.

In step 3 of the GenProof algorithm of S-PDP, S has two ways of computing ρ: Either sum the values $a_j m_{i_j}$ (as integers) and then exponentiate g_s to this sum or exponentiate g_s to each value

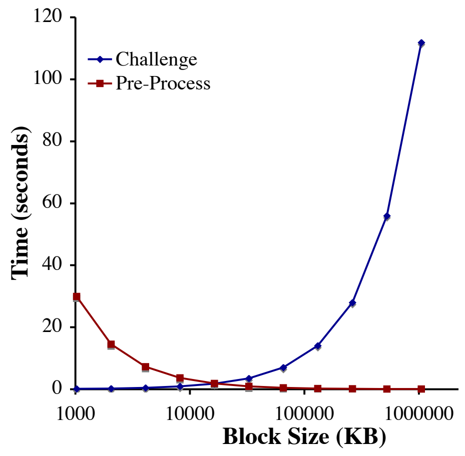


Figure 6: E-PDP pre-processing vs. challenge trade-offs with block size for a 1 GB file.

$a_j m_{i_j}$ and then multiply all values. We observed that the former choice takes considerable less time, as it only involves one exponentiation to a $(|m_i| + \ell + \log_2(c))$-bit number, as opposed to c exponentiations to a $(|m_i| + \ell)$-bit number (typically, $\ell = 160$).

Fig. 5(a) shows the computation time as a function of file size used at the server when computing a proof for B-PDP, MHT-SC and E-PDP. Note the logarithmic scale. Computation time includes the time to access the memory blocks that contain file data in cache. We restrict this experiment to files of 768 KB or less, because of the amount of time consumed by B-PDP.

E-PDP radically alters the complexity of data possession protocols and even outperforms protocols that provide weaker guarantees, specifically MHT-SC. For files of 768 KB, E-PDP is more than 185 times faster than B-PDP and more than 4.5 times as fast as MHT-SC. These performance ratios become arbitrarily large for larger file sizes. For B-PDP performance grows linearly with the file size, because it exponentiates the entire file. For MHT-SC, performance also grows linearly, but in disjoint clusters which represent the height of the Merkle-tree needed to represent a file of that size.

Pre-Processing. In preparing a file for outsourced storage, the client generates its local metadata. In this experiment, we measure the processor time for metadata generation only. This does not include the I/O time to load data to the client or store metadata to disk, nor does it include the time to transfer the file to the server. Fig. 5(b) shows the pre-processing time as a function of file size for B-PDP, MHT-SC and E-PDP.

E-PDP exhibits slower pre-processing performance. The costs grow linearly with the file size at 162 KB/s. E-PDP performs an exponentiation on every block of the file in order to create the per-block tags. For MHT-SC, preprocessing performance mirrors challenge performance, because both protocol steps perform the same computation. It generates data at about 433 KB/s on average.

The preprocessing performance of B-PDP differs from the challenge phase even though both steps compute the exact same signature. This is because the client has access to $\phi(N)$ and can reduce the file modulo $\phi(N)$ before exponentiating. In contrast, the security of the protocol depends on $\phi(N)$ being a secret that is unavailable to the server. The preprocessing costs comprise a single exponentiation and computing a modulus against the entire file.

E-PDP also exponentiates data that was reduced modulo $\phi(N)$ but does not reap the same speed up, because it must do so for every block. This creates a natural trade-off between preprocessing time and challenge time by varying the block size; *e.g.*, the protocol devolves to B-PDP for files of a single block. Fig. 6 shows this trade-off and indicates that the best balance occurs at natural file system and memory blocks sizes of 4-64 KB. We choose a block size of 4K in order to minimize the server's effort.

Given the efficiency of computing challenges, pre-processing represents the limiting performance factor for E-PDP. The rate at which clients can generate data to outsource bounds the overall system performance perceived by the client. However, there are several mitigating factors. (1) Outsourcing data is a one time task, as compared to challenging outsourced data, which will be done repeatedly. (2) The process is completely parallelizable. Each file can be processed independently at a different processor. A single file can be parallelized trivially if processors share key material.

6 Conclusion

We focused on the problem of verifying if an untrusted server stores a client's data. We introduced a model for provable data possession, in which it is desirable to minimize the file block accesses, the computation on the server, and the client-server communication. Our solutions for PDP fit this model: They incur a low (or even constant) overhead at the server and require a small, constant amount of communication per challenge. Key components of our schemes are the homomorphic verifiable tags. They allow to verify data possession without having access to the actual data file.

Experiments show that our schemes, which offer a probabilistic possession guarantee by sampling the server's storage, make it practical to verify possession of large data sets. Previous schemes that do not allow sampling are not practical when PDP is used to prove possession of large amounts of data. Our experiments show that such schemes also impose a significant I/O and computational burden on the server.

7 Acknowledgments

This work was supported in part by NSF awards CCF-0238305 and IIS-0456027 and by the IBM Corporation. We thank Gene Tsudik, Susan Hohenberger, Roberto Di Pietro, Luigi Mancini, Răzvan Musăloiu-E., Seny Kamara, and Brent Waters for their insightful comments.

References

[1] M. Abe and S. Fehr. Perfect NIZK with adaptive soundness. In *Proc. of Theory of Cryptography Conference (TCC '07)*, 2007. Full version available on Cryptology ePrint Archive, Report 2006/423.

[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] M. Bellare, J. Garay, and T. Rabin. Fast batch verification for modular exponentiation and digital signatures. In *Proc. of EUROCRYPT '98*, LNCS, pages 236–250, 1998.

[4] M. Bellare and O. Goldreich. On defining proofs of knowledge. In *Proc. of CRYPTO '92*, volume 740 of *LNCS*, pages 390–420, 1992.

[5] M. Bellare and A. Palacio. The knowledge-of-exponent assumptions and 3-round zero-knowledge protocols. In *Proc. of CRYPTO '04*, Lecture Notes in Computer Science, pages 273–289. Springer, 2004.

[6] M. Bellare and A. Palacio. Towards plaintext-aware public-key encryption without random oracles. In *Proc. of ASIACRYPT '04*, volume 3329 of *LNCS*, pages 48–62. Springer, 2004.

[7] M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *First Conference on Computer and Communications Security*, pages 62–73. ACM, 1993.

[8] M. Bellare and P. Rogaway. The exact security of digital signatures - How to sign with RSA and Rabin. In *EUROCRYPT*, pages 399–416, 1996.

[9] M. Bellare and P. Rogaway. PSS: Provably secure encoding method for digital signatures. IEEE P1363a: Provably secure signatures, 1998. `http://grouper.ieee.org/groups/1363/P1363a/PSSigs.html`.

[10] J. Black and P. Rogaway. Ciphers with arbitrary finite domains. In *Proc. of CT-RSA*, volume 2271 of *LNCS*, pages 114–130. Springer-Verlag, 2002.

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

[12] D. Boneh, C. Gentry, B. Lynn, and H. Shacham. Aggregate and verifiably encrypted signatures from bilinear maps. In *Proc. of EUROCRYPT '03*, volume 2656 of *LNCS*, pages 416–432. Springer, 2003.

[13] I. Damgård. Towards practical public key systems secure against chosen ciphertext attacks. In J. Feigenbaum, editor, *CRYPTO91*, volume 576, pages 445–456. Springer, 1992.

[14] A. W. Dent. The Cramer-Shoup encryption scheme is plaintext aware in the standard model. In *Proc. of EUROCRYPT '06*, volume 4004 of *LNCS*, pages 289–307. Springer, 2006.

[15] A. W. Dent. The hardness of the DHK problem in the generic group model. Cryptology ePrint Archive, Report 2006/156, 2006. `http://eprint.iacr.org/2006/156`.

[16] 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.

[17] A. Fiat. Batch RSA. In G. Brassard, editor, *Proc. CRYPTO 89*, pages 175–185. Springer-Verlag, 1990. Lecture Notes in Computer Science No. 435.

[18] 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`.

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

[20] S. Hada and T. Tanaka. On the existence of 3-round zero-knowledge protocols. In *Proc. of CRYPTO '98*, volume 1462 of *Lecture Notes in Computer Science*, pages 408–423. Springer, 1998.

[21] L. Harn. Batch verifying multiple RSA digital signatures. *Electronics Letters*, 34(12):1219–1220, 1998.

[22] R. Hasan, W. Yurcik, and S. Myagmar. The evolution of storage server providers: Techniques and challenges to outsourcing storage. In *Proc. of the Workshop on Storage Security and Survivability*, 2005.

[23] R. Johnson, D. Molnar, D. Song, and D. Wagner. Homomorphic signature schemes. In *Proc. of CT-RSA*, volume 2271 of *LNCS*, pages 244–262. Springer, 2002.

[24] A. Juels and B. S. Kaliski. PORs: Proofs of retrievability for large files. Technical Report 2007/243, IACR ePrint Cryptography Archive, 2007. `http://eprint.iacr.org/2003/216`.

[25] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus: Scalable secure file sharing on untrusted storage. In *Proc. of FAST*, 2003.

[26] H. Krawczyk. HMQV: A high-performance secure Diffie-Hellman protocol. In *Proc. of CRYPTO '05*, volume 3621 of *LNCS*, pages 546–566. Springer, 2005.

[27] M. N. Krohn, M. J. Freedman, and D. Mazières. On-the-fly verification of rateless erasure codes for efficient content distribution. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2004.

[28] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of ACM ASPLOS '00*. ACM, November 2000.

[29] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *Proceedings of the Symposium on Operating Systems Design and Implementation*, 2004.

[30] U. Maheshwari, R. Vingralek, and W. Shapiro. How to build a trusted database system on untrusted storage. In *Proc. of OSDI*, 2000.

[31] P. Maniatis, M. Roussopoulos, T. Giuli, D. Rosenthal, M. Baker, and Y. Muliadi. The LOCKSS peer-to-peer digital preservation system. *ACM Transactions on Computing Systems*, 23(1):2–50, 2005.

[32] S. Micali, K. Ohta, and L. Reyzin. Accountable-subgroup multisignatures: extended abstract. In *Proc of ACM CCS '01*, pages 245–254, 2001.

[33] G. L. Miller. Riemann's hypothesis and tests for primality. *JCSS*, 13(3):300–317, 1976.

[34] A. A. Muthitacharoen, R. Morris., T. M. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. In *Proceedings of 5th Symposium on Operating Systems Design and Implementation*, 2002.

[35] E. Mykletun, M. Narasimha, and G. Tsudik. Authentication and integrity in outsourced databases. In *Proceedings of NDSS*. The Internet Society, 2004.

[36] 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.

[37] T. Okamoto. A digital multisignature schema using bijective public-key cryptosystems. *ACM Transactions on Computer Systems*, 6(4):432–441, 1988.

[38] A. Oprea, M. K. Reiter, and K. Yang. Space-efficient block storage integrity. In *Proc. of NDSS '05*, 2005.

[39] 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.

[40] 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.

[41] M. Shah, M. Baker, J. C. Mogul, and R. Swaminathan. Auditing to keep online storage services honest. In *Proc. of HotOS XI*. Usenix, 2007.

[42] A. Shamir. On the generation of cryptographically strong pseudorandom sequences. *ACM Trans. Comput. Syst.*, 1(1):38–44, 1983.

[43] D. Thompson and J. Best. The future of magnetic data storage technology. *IBM Journal Research and Development*, 44(3):311–319, May 2000.

[44] M. Waldman and D. Mazières. Tangler: a censorship-resistant publishing system based on document entanglements. In *Proc. of CCS*, 2001.

[45] A. Y. Yumerefendi and J. Chase. Strong accountability for network storage. In *Proc. of FAST*, 2007.

A Proof of Theorem 4.3

Under the KEA1-r assumption, we reduce the security of our S-PDP scheme to the security of the RSA problem and the security of integer factoring. We assume there exists an adversary $\mathcal{B}$ that wins the Data Possession Game on a challenge picked by $\mathcal{A}$ and show that $\mathcal{A}$ will be able to extract the blocks determined by the challenge. If $\mathcal{B}$ can break the data possession guarantee of the S-PDP scheme, we show how to construct an adversary $\mathcal{A}$ that uses $\mathcal{B}$ in order to either break RSA or factor the product of two large primes.

For the RSA problem, $\mathcal{A}$ is given (N, e, y), with $y \overset{R}{\leftarrow} \mathbb{Z}_N^*$, and needs to find a value $b \equiv y^{1/e} \bmod N$. $\mathcal{A}$ will play the role of the challenger in the Data Possession Game and will interact with $\mathcal{B}$.

We first look at the case when in GenProof and CheckProof all the coefficients $a_1, \ldots, a_c$ are equal to 1. This corresponds to the case where the server proves it possesses the sum of the requested blocks. We then generalize the proof to the case where the coefficients are random and pairwise distinct, which corresponds to the case where the server proves it possesses each individual block.

$\mathcal{A}$ simulates a PDP environment for $\mathcal{B}$ as follows:

Setup: $\mathcal{A}$ computes $g = y^2 \bmod N$, sets the public key pk $= (N, e, g)$ and sends pk to $\mathcal{B}$. $\mathcal{A}$ publishes the description of the PRF w and also generates the corresponding secret key $v \overset{R}{\leftarrow} \{0,1\}^\kappa$.

Query: $\mathcal{B}$ makes tagging queries adaptively: $\mathcal{B}$ selects a block m_1 and is also allowed to select an index i_1. $\mathcal{B}$ sends m_1 and i_1 to $\mathcal{A}$. $\mathcal{A}$ generates $(\mathtt{T}_{i_1,m_1}, \mathtt{W}_{i_1})$ and sends it back to $\mathcal{B}$. $\mathcal{B}$ continues to query $\mathcal{A}$ for the tags $(\mathtt{T}_{i_2,m_2}, \mathtt{W}_{i_2}), \ldots, (\mathtt{T}_{i_n,m_n}, \mathtt{W}_{i_n})$ on the blocks $m_2, \ldots, m_n$ and indices $i_1, \ldots, i_n$ of its choice. The only restriction is that $\mathcal{B}$ cannot make tagging queries for two different blocks using the same index. Once $\mathcal{B}$ gets the tag $(\mathtt{T}_{i,m_i}, \mathtt{W}_i)$ on a block m_i, it may check the tag's validity by checking if the relation $(\mathtt{T}_{i,m_i})^e = h(\mathtt{W}_i) \cdot g^{m_i}$ holds.

When $\mathcal{B}$ makes tagging oracle queries and hash oracle queries, $\mathcal{A}$ will answer those queries itself:

- when $\mathcal{B}$ makes a tagging query for a block m and index i, with $1 \le i \le n$:
  - if a previous tagging query has been made for the same m and i, then $\mathcal{A}$ retrieves the recorded tuple $(m, i, r_i, \mathtt{W}_i)$ and returns $(\mathtt{T}_{i,m}, \mathtt{W}_i)$, where $\mathtt{T}_{i,m} = r_i$.
  - else, $\mathcal{A}$ picks $r_i \overset{R}{\leftarrow} QR_N$, computes $\mathtt{W}_i = w_v(i)$, records the tuple $(m, i, r_i, \mathtt{W}_i)$ and returns $(\mathtt{T}_{i,m}, \mathtt{W}_i)$, where $\mathtt{T}_{i,m} = r_i$.
- when $\mathcal{B}$ makes a hash query for a value $\mathtt{W}_i$:
  - if the value $\mathtt{W}_i$ is recorded as having been previously used for a tagging query, then $\mathcal{A}$ retrieves the corresponding tuple $(m, i, r_i, \mathtt{W}_i)$ and returns $h(\mathtt{W}_i) = r_i^e \cdot g^{-m_i} \bmod N$.
  - else, $\mathcal{A}$ picks at random a value from QR_N and returns it.

At the end of the **Query** phase $\mathcal{A}$ can reveal the PRF key v to $\mathcal{B}$ (this is to cover the case when public verifiability is enabled).

Challenge: $\mathcal{A}$ generates the challenge $\mathsf{chal} = (g_s, i_1, \ldots, i_c)$, where $g_s = g^s \bmod N$, $s \overset{R}{\leftarrow} \mathbb{Z}_N^*$ and $i_1, \ldots, i_c$ are the indices of the blocks for which $\mathcal{A}$ requests proof of possession (with $1 \le i_j \le n$, $1 \le j \le c, 1 \le c \le n$). $\mathcal{A}$ sends chal to $\mathcal{B}$.

Forge: $\mathcal{B}$ generates a proof $\mathcal{V} = (\mathtt{T}, \rho)$ about the blocks $m_{i_1}, \ldots, m_{i_c}$ determined by $i_1, \ldots, i_c$, where $\mathtt{T} = \mathtt{T}_{\{i_1,\ldots,i_c\}, m_{i_1}+\ldots+m_{i_c}}$. Note that $\mathcal{V}$ is a valid proof that passes $\mathsf{CheckProof}(\mathrm{pk}, \mathrm{sk}, \mathsf{chal}, \mathcal{V})$. $\mathcal{B}$ returns $\mathcal{V}$ to $\mathcal{A}$ and $\mathcal{A}$ checks the validity of $\mathcal{V}$. Let $M = m_{i_1} + \ldots + m_{i_c}$.

As H is a random oracle, with overwhelming probability we can extract the pre-image value ρ_p that $\mathcal{B}$ utilized to calculate ρ. (By the definition of a random oracle, $\mathcal{B}$ can guess a valid value of ρ with only negligible probability.)

$\mathcal{A}$ has given $\mathcal{B}$ both g, g^s and $\mathcal{B}$ has implicitly returned $\tau = \frac{\mathtt{T}^e}{\prod_{j=1}^{c} h(\mathtt{W}_{i_j})}, \rho_p$ by returning $\mathtt{T}, \rho$. Because $\tau^s = \rho_p$, by KEA-1r, $\mathcal{A}$ can utilize the extractor $\bar{\mathbf{B}}$ to extract a value M^* such that $g^{M^*} = \tau$.

If $M^* = M$, then $\mathcal{A}$ was able to successfully extract the correct message M. We analyze next the case when $M^* \ne M$. Note that M^* is the "full-domain" value utilized by this calculation. (If the extractor $\bar{\mathbf{B}}$ is able to extract a value $M' \ne M^*$ such that $g^{M'} = g^{M^*} \bmod N$, this will allow to compute a multiple of $\phi(N)$, from which the factorization of N can be efficiently computed [33].)

From $\tau = g^{M^*}$ we get $\mathtt{T}^e = \prod_{j=1}^{c} h(\mathtt{W}_{i_j}) \cdot g^{M^*}$, where clearly $g^{M^*} \ne g^M$, and thus:

$$\begin{aligned}
\mathtt{T} &= \left(\prod_{j=1}^{c} h(\mathtt{W}_{i_j}) \cdot g^{M^*} \right)^d \\
&= \left(\prod_{j=1}^{c} (r_{i_j}^e \cdot g^{-m_{i_j}}) \cdot g^{M^*} \right)^d \\
&= \prod_{j=1}^{c} r_{i_j} \cdot \left(g^{M^*-M} \right)^d
\end{aligned}$$

$\mathcal{A}$ computes:

$$z = \frac{\mathtt{T}}{\prod_{j=1}^{c} r_{i_j}} = \left(g^{M^*-M} \right)^d$$

We have $z^e = g^{M^*-M} = y^{2(M^*-M)}$. Let's assume that $gcd(e, 2(M^*-M)) = 1$ (in fact this always holds if e is prime and greater than $|M^*-M|$). Applying Shamir's "trick" [42], $\mathcal{A}$ uses the extended Euclidian algorithm to efficiently compute integers u and v such that $u \cdot e + v \cdot 2(M^* - M) = 1$ and outputs $y^{1/e} = y^u z^v$.

Note that the interactions of $\mathcal{A}$ with $\mathcal{B}$ are indistinguishable to $\mathcal{B}$ from interactions with an honest challenger in the Data Possession Game, as $\mathcal{A}$ chooses all parameters according to our protocol.

The proof generalizes to the case where the coefficients $a_1, \ldots, a_c$ are random and pairwise distinct. Indeed, in this case it is enough to apply the same simulation shown above and in addition to notice that at the end of the simulation $\mathcal{A}$ will be able to extract $\bar{M} = a_1 m_{i_1} + \ldots + a_c m_{i_c}$. We now have to show that our protocol constitutes a proof of knowledge of the blocks $m_{i_1}, \ldots, m_{i_c}$ when $a_1, \ldots, a_c$ are pairwise distinct. We show that a knowledge extractor $\mathcal{E}$ may extract the file blocks $m_{i_1}, \ldots, m_{i_c}$. Note that each time $\mathcal{E}$ runs the PDP protocol, he obtains a linear equation of the form $\bar{M} = a_1 m_{i_1} + \ldots + a_c m_{i_c}$. By choosing independent coefficients $a_1, \ldots, a_c$ in c executions of the protocol on the same blocks $m_{i_1}, \ldots, m_{i_c}$, $\mathcal{E}$ obtains c independent linear equations in the variables $m_{i_1}, \ldots, m_{i_c}$. $\mathcal{E}$ may then solve these equations to obtain the file blocks $m_{i_1}, \ldots, m_{i_c}$.

B Implemented PDP Schemes

As a basis of comparison, we have implemented the following two PDP schemes in addition to our E-PDP scheme:

Basic RSA-based PDP Scheme (B-PDP)[16, 18]. Let N be an RSA modulus and let $g \in \mathbb{Z}_N^*$.

Setup: C stores $a = g^{\mathtt{F}} \bmod N$. C sends $\mathtt{F}$ to S.

Challenge:

1. C generates a random value $r \overset{R}{\leftarrow} \mathbb{Z}_n^*$ and sends g^r to S.
2. S computes $b = (g^r)^{\mathtt{F}} \bmod N$ and sends b to C.
3. C computes a^r and checks if $b = a^r \bmod N$.

Merkle Hash Tree-based Storage Enforcing Scheme (MHT-SC)[19]. Let ψ be a one-way length-increasing transformation and let h be a cryptographic hash function. In a binary tree, we denote by $PATH(i)$ the set of nodes on the path between the root of the tree and the i-th leaf.

Setup: C applies the transformation ψ on the file $\mathtt{F}$ and obtains the expanded file $\mathtt{F}' = \psi(\mathtt{F})$. C stores h_{root} as the root of the Merkle hash tree[4] which is computed using h on the blocks of $\mathtt{F}'$.

Challenge:

1. C randomly picks the index i of a block $\mathtt{F}'_i$ of the expanded file $\mathtt{F}'$ and sends i to S.
2. Let H_i be the set of hashes corresponding to the nodes that "hang" off $PATH(i)$ in the Merkle hash tree determined by $\mathtt{F}'$. S sends H_i and $\mathtt{F}'_i$ back to C.
3. C uses H_i and $\mathtt{F}'_i$ to recompute the root of the tree and compares this value with the stored value for h_{root}.

[4]Merkle hash trees compute a parent node by applying a hash function on the concatenation of its children nodes.