SOK: A PRACTICAL COST COMPARISON AMONG PROVABLE DATA POSSESSION SCHEMES

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Alex Bartlett

May 2018

COMMITTEE MEMBERSHIP

TITLE:	SoK: A Practical Cost Comparison Among Provable Data Possession Schemes
AUTHOR:	Alex Bartlett
DATE SUBMITTED:	May 2018
COMMITTEE CHAIR:	Zachary N.J. Peterson, Ph.D. Associate Professor of Computer Science
COMMITTEE MEMBER:	Bruce DeBruhl, Ph.D. Assistant Professor of Computer Science
COMMITTEE MEMBER:	Phillip Nico, Ph.D. Professor of Computer Science

Abstract

SoK: A Practical Cost Comparison Among Provable Data Possession Schemes

Alex Bartlett

Provable Data Possession (PDP) schemes provide users with the ability to efficiently audit and verify the integrity of data stored with potentially unreliable third-parties, such as cloud storage service providers. While dozens of PDP schemes have been developed, no PDP schemes have been implemented with an existing cloud service. This work attempts to provide a starting point for the integration of PDP schemes with cloud storage service providers by providing a cost analysis of PDP schemes. This cost analysis is performed by implementing and analyzing five PDP schemes representative of the dozens of various PDP approaches. This paper provides analysis of the overhead and performance of each of these schemes to generate a comparable cost for each scheme using real-world cloud pricing models. Results show that the total cost of each scheme is comparable for smaller file sizes, but for larger files this cost can vary across schemes by an order of magnitude. Ultimately, the difference in cost between the simple MAC-based PDP scheme and the most "efficient" PDP scheme is negligible. While the MAC-PDP scheme may not be the most efficient, no other scheme improving upon its complexity can be implemented without the use of additional services or APIs leading to the conclusion that the simplest, storage only PDP scheme is the most practical to implement. Furthermore, the findings in this paper suggest that, in general, PDP schemes optimize on an inaccurate cost model and that future schemes should consider the existing economic realities of cloud services.

ACKNOWLEDGMENTS

Thanks to:

- Dr. Peterson for the advice and guidance throughout the school year
- Dr. DeBruhl and Dr. Nico for being on my committee
- My family for their support throughout my college career

TABLE OF CONTENTS

List of Tables

List of Figures

Chapter 1

INTRODUCTION

Numerous organizations around the world have begun to rely on 3rd party storage, such as clouds, to store excess data. 3rd party storage allows these organizations many benefits, the most prominent being saving storage locally. However, saving data with a 3rd party comes with an increased security risk. Organizations need to be able to securely verify that data they store on 3rd party servers remains in the possession of the server and accessible from the server. However, remotely verifying that the server fulfills its contractual obligations to store data can be challenging. Data stored on a 3rd party server that is infrequently accessed is prone to undetectable loss from inadvertent administration errors or malicious activity from the storage service provider. A malicious storage service provider may not disclose a data loss incident in order to preserve its reputation or delete storage that is infrequently used in order to resell the same storage space to a different party. Data loss incidents from popular cloud storage providers such as Amazon S3 or Dropbox emphasize the need for an efficient and reliable auditing mechanism to ensure that data stored on 3rd party platforms remains intact. This is where Provable Data Possession (PDP) schemes can be utilized.

PDP schemes provide probabilistic guarantees that data stored on a remote storage server has not been maliciously or accidentally altered. PDP schemes are designed to provide these probabilistic guarantees at a low cost to both the client and the server. To do this, PDP schemes must be able to provide the client with a proof of data possession from the server without needing the client to retrieve the entire file from the server and without needing the server to access the entire file. Numerous meth-

ods have been developed to achieve sufficient provable data possession while limiting costs, but despite years of research into the viability of PDP schemes there are no existing practical implementations for any commercial cloud service. This may be because prior research of PDP schemes has focused primarily on providing a high probability guarantee of data possession to the client and minimizing bandwidth between the client/server interactions without considering any technique to compare the real world efficiencies of PDP schemes. While probabilistic guarantees and bandwidth minimization are vital aspects of PDP schemes, without a viable example of real world costs of a PDP scheme (i.e. the time it takes to generate a challenge, verify a proof, tag the file, cost to store the tags, run an audit service, service audit requests, etc.) it is very difficult to know which PDP schemes work best in practice.

This work attempts to bridge the gap between research and implementation of PDP schemes. To do so, a taxonomy of PDP schemes was created. This taxonomy compiles research from over 30 different PDP schemes into a single classification table making it easy to identify PDP schemes with similar characteristics. Using this taxonomy, five different PDP schemes were identified that can be used to wholly represent various characteristics of the entirety of PDP schemes. These were then implemented as an open-source library, *libpdp*. *Libpdp* provides generic cost models based on mathematical formulas that express abstract models that can be used to determine future cost. These cost models allow for a real word cost analysis of any PDP scheme. A comparison and validation of the cost models is provided for each of the five implemented schemes in *libpdp*. This comparison provides an accurate distinction between the five implemented schemes (MAC-PDP, A-PDP, SE-PDP, CPOR, and MR-PDP) and allows a proper determination of which scheme functions best in a practical, real-world environment. The costs were evaluated in terms of recurring computational costs induced by running the PDP schemes.

Based on the cost models provided in *libpdp*, the five PDP schemes implemented were found to be most cost-comparable in terms of preprocessing, storing, and auditing. More complex PDP schemes that prioritize lower communication complexity at the cost greater preprocessing and storage costs were found to be significantly more expensive in practice. There was a high variance in total basis costs (up-front cost to tag and additional, cumulative costs for storing and auditing data) across the implemented schemes, especially with larger file sizes. Auditing a 1GB file for one year at one audit per hour is under $1 for MAC-PDP, A-PDP, MR-PDP, and CPOR, but that cost inflates to a range of $4,400 to $38,700 across schemes for a 1PB file. Auditing once per hour with each scheme was an arbitrary choice that allowed for comparisons between each implemented scheme. In practice, the user of a PDP scheme would perform audits at a rate based on how important the data is they are storing. Additionally, this paper claims that schemes which require server side processing do not fit with pre-existing cloud APIs leading to the conclusion that the simplest, storage-only scheme may be the most cost-efficient to practically implement. The findings of this paper suggest that all PDP schemes may be optimizing utilizing an inaccurate cost model and that PDP schemes developed in the future should consider existing realities of server-side implementations on cloud storage providers.

The rest of this paper is formatted as follows. In Chapter 2 there will be a background on relevant PDP specific information and notations that will help provide a foundation and understanding for the rest of this paper. In Chapter 3 relevant related work will be presented and used to give a background of the five chosen PDP schemes. Then, the taxonomy of PDP schemes will be introduced and discussed in Chapter 4 to provide context as to how the five implemented represent the entirety of PDP schemes. Next, the paper will cover the design and implementation of the five algorithms in *libpdp* in Chapter 5. Chapter 6 will cover and explain the Operation

Models used within *libpdp* while Chapter 7 will provide an evaluation and explanation of the results obtained from the implementation. Possible future work will be discussed in Chapter 8. Finally, Chapter 9 will provide concluding thoughts to the paper.

Chapter 2

BACKGROUND

Generically, PDP schemes can be broken up into two different phases: a setup phase and a verification phase. During the setup phase, the client will generate keys (private and/or public, depending on the scheme) and use those keys to tag their file. Typically, the file will be split up into blocks and each block will be tagged individually. The client will then send the file along with the tags to the storage service provider and delete their local copies. During the verification phase the client will typically select a random subset of blocks to generate a challenge for. The client will then create a challenge and then send the challenge to the prover. The prover uses the challenge and the file blocks to generate a proof of verification that it sends back to the client. The client then uses the proof of verification to verify that the server actually possesses the correct data, providing a probabilistic guarantee that the prover does or does not possess the challenged data.

To further explain PDP algorithms, this paper follows the notation of Juels and Kaliski [27] and Bower, Juels, and Oprea [12]. A file M can be divided into n blocks, $M = \langle m_1, m_2, ..., m_n \rangle$. P will denote the prover (storage provider), V denotes the verifier (client or third-party auditor), η denotes the file's identifier, and ω denotes local client state. A generic PDP scheme can be considered a five-tuple of algorithms, (Key-Gen, Tag, Challenge, Proof, Verify) which can be described as the following:

KeyGen$(1^k) \rightarrow (pk, sk)$ This algorithm is used by the client to generate random public and private keys by employing security parameter k.

Tag$(M; pk, sk, \omega) \rightarrow M^*_\eta$ This algorithm is used by the client to process a file and

produce verification tag data. It takes as input a public and private key pair (pk, sk) and file M. It generates a file ID η and returns M^*_η, the encoded file with verification tag data. It also updates the client state ω to include any locally held data such as the file ID, file size, number of blocks, etc. The data M^*_η can be stored remotely.

Challenge$(\eta; pk, sk, \omega) \rightarrow c$ This algorithm is used by the client to produce a challenge c. This challenge is sent to the prover during an audit.

Proof$(\eta, M^*_\eta, c; pk) \rightarrow p$ This algorithm is used by the prover to demonstrate proof of possession of specified file blocks as a response to challenge c. It takes as input the remote, encoded data M^*_η and challenge c, to generate proof p.

Verify$(c, p, \eta; pk, sk, \omega) \rightarrow b \in \{0, 1\}$ This algorithm is used by the client to validate the proof p. It takes as input the public and private key pair (pk, sk), challenge c and proof p. Upon successful validation it returns 1, else it returns 0.

2.1 Analyzed Schemes

This paper focuses on five specific PDP schemes: a basic Message Authentication Code (MAC) based scheme (MAC-PDP), the scheme developed by Ateniese, Burns, Curtmola, Herring, Kissner, Peterson and Song (A-PDP) [5], the scheme developed by Curtmola, Khan, Burns, and Ateniese (MR-PDP) [18], the scheme developed by Ateniese, Di Pietro, Mancini and Tsudik (SE-PDP) [6], and the scheme developed by Shacham and Waters (CPOR) [37].

These schemes were chosen due to their unique characteristics that can be used to represent all existing PDP schemes. MAC-PDP and SE-PDP are reliant on using symmetric key primitives; MAC-PDP is the simplest PDP scheme, it does not require any cryptographic computation from the cloud storage provider, whereas SE-PDP requires a degree of server-side computation. CPOR provides the public verifiability characteristic and optimizes for proof compactness. A-PDP was developed using public key primitives, requires computation from the cloud service provider, and also allows for public verifiability. Furthermore, all the schemes have various size and computational complexity of their tags, challenges, and proofs, which allowed this research to differentiate between storage and computation costs. These schemes and their characteristics are discussed further in Chapters 3 and 4.

2.2 Scheme Complexity

The complexity of each analyzed scheme is provided in Table 2.1. The block size, bs, is a function of file size and n, the number of file blocks, where bs = file size / n. MAC-PDP endures a relatively high communication complexity, which is offset by the simplicity in it's implementation. The remaining schemes, A-PDP, MR-PDP, CPOR, and SE-PDP were designed to optimize communication complexity at the cost of computational and storage complexity.

Table 2.1: Communication Complexity of Target Schemes

	Challenge	Proof
MAC-PDP	$\mathcal{O}(\ell \log(n))$	$\mathcal{O}(\ell(bs + k))$
A-PDP	$\mathcal{O}(\log(\ell + 2\kappa + \log(N))$	$\mathcal{O}(\log(N))$
MR-PDP	$\mathcal{O}(\log(\ell + 2\kappa + \log(N))$	$\mathcal{O}(\log(N))$
CPOR	$\mathcal{O}(\ell + (\log(n) + d))$	$\mathcal{O}(\log(p))$
SE-PDP	$\mathcal{O}(L)$	$\mathcal{O}(d + L)$

2.3 Comparison Criteria

In order to accurately make a comparison between PDP schemes, comparable criteria amongst all schemes must be selected. The following 3 concepts are criteria in which PDP schemes may be considered to be comparable:

2.3.1 Strength of Security.

For any given scheme, this is expressed as Pr[*forge*], the probability that a prover can manipulate the verifier to accept a forged proof as valid, i.e. when the proof was computed without the use of each challenged block by the verifier.

2.3.2 Strength of Audit.

For any given scheme, this is expressed as Pr[*audit*], the probability that a single audit will be validated even when k of n blocks have been deleted. For numerous schemes, this is a combinatorial argument based on the probability that the ℓ random challenge indices are among the k blocks deleted.

2.3.3 Efficiency of Recovery.

There exist select PDP schemes, often referred to as POR schemes (such as CPOR), that have the additional characteristic that the original file sent to 3rd party storage can be recovered even after some number of failed audits. For such a scheme, this is expressed as Pr[*recover*], the probability of retrieval after an ϵ fraction of audits have failed.

However, a strict comparison following these criteria is problematic for various reasons. For example, schemes rely on different primitives (e.g., full domain hash functions, authenticated encryption schemes, pseudorandom permutations and functions)

which makes parameter selection for a comparison between schemes with Pr[*forge*] difficult. Furthermore, schemes utilize these properties with different adversarial models with different arguments. Lastly, arguments for schemes have been expressed in asymptotic terms, which makes parameter deviation difficult, especially when arguments use bounds that may not be tight. Because the simple, combinatorial arguments used for Pr[*audit*] are typically the most reusable, this research emphasizes parameter selection for comparison with regards to Strength of Audit. Selection of this parameter most closely relates to understanding policy on how often an audit is performed. Because this research aims to derive the recurring cost of an audit, this was an obvious parameter to consider and analyze carefully.

Chapter 3

RELATED WORK

Each PDP scheme analyzed in this thesis [5], [18], [37], [6] is based on a previously developed and researched scheme. Each scheme is presented under the context of attempting to minimize bandwidth and providing the user with a high probability guarantee of data possession. This thesis further develops the feasibility of PDP schemes by implementing these schemes and providing a cost comparison, based on $\Pr[audit]$, between the schemes. This cost comparison can be used as a direct comparison between the schemes and allows us to determine whether any of these schemes are feasible and if so, which scheme is most feasible. The papers summarized in this section provide a description of how each implemented PDP scheme functions. Following the descriptions, a more concrete definition of each scheme is provided.

The first scheme implemented in *libpdp* is a naive MAC based scheme. It is the simplest of the 5 implemented PDP schemes. It works by having the user calculate a MAC for the data they want to store on a cloud or any other 3rd party device. The user sends the calculated MAC along with the data they want to store to the server; whenever the user wants to verify that the data they stored remains untouched the user simply needs to retrieve their file and corresponding MAC and calculate a new MAC on the retrieved data. After calculating the new MAC the user compares it to the old MAC retrieved from the server and if the two MACs are the same then the data is also the same. While this scheme accomplishes the goals of a PDP algorithm it is highly inefficient. The cost of retrieving the whole file and recalculating a MAC every single time the user wants to verify data possession is too high. As an optimization to the MAC-PDP scheme the user can split the data file into blocks and compute a

MAC for each block. Then, the user sends both the blocks and their corresponding MACs to the server and stores only the secret key used to calculate the MAC, sk. When the user wants to verify data possession the user requests a randomly selected subset of blocks and their corresponding MACs from the server. Using sk, the user re-computes the MACs of the received blocks and compares the new MAC to the old MAC. If the two MACs are the same, then the user is assured of the integrity of their stored data. The rationale behind the second implementation of MAC-PDP is it is much easier to verify a portion of the file as opposed to the entire file, which helps limit bandwidth.

The next scheme implemented in *libpdp* is A-PDP [5]. Ateniese et al. devised their PDP scheme on the basis of Homomorphic Verifiable Tags (HVTs). HVTs are used as verification metadata by the user where the user tags each block with an HVT and can be convinced of data possession by receiving a linear combination of file blocks and their corresponding HVTs. A-PDP also differentiates between public and private verifiability. With public verifiability anyone, not just the owner of the file, can challenge the remote server and verify the server's proof of possession as long as they own the file owner's public key. Ateniese, et al., introduced two main PDP schemes, S-PDP and E-PDP, that comprise A-PDP. Both schemes are RSA-based; in S-PDP the user splits the files into blocks and tags each block with an HVT. The user then sends the blocks and their respective tags to the server. If the client wants to verify data possession, the client asks the server for proof of possession of a randomly chosen subset of blocks stored on the server. The server then generates a proof of possession that consists of two values: T and ρ. T is a combination of the HVTs of each requested block while ρ is obtained by raising the challenge to a function of the requested blocks. The client can verify data possession by verifying that a certain relation holds between T and ρ. E-PDP works similarly, the difference being

that E-PDP only verifies the sum of the file blocks' HVTs and not necessarily each individual file block being challenged. This means that E-PDP runs more efficiently at the cost of being less secure than S-PDP.

MR-PDP [18] proposed by Curtmola et al. is a PDP scheme based off of A-PDP. MR-PDP builds off the algorithm used in A-PDP, but allows the client to create multiple replicas of the file to store at different servers. The rationale behind this is that if one of the client's files is altered for any reason the data is easily recoverable because it's stored at other servers. The functions used in A-PDP are modified to accommodate multiple files for MR-PDP. MR-PDP also implements a replica generation function that the client can use to create replicas of their data to send to other servers. The client only needs to generate one set of HVTs, just like in A-PDP, but to create a file replica the client masks the blocks of the original file by concatenating the original file's blocks with the output of a pseudo random function (PRF). After sending each replica of the file to different servers the client can verify data possession using the same process used in A-PDP.

SE-PDP [6] is the PDP scheme used to demonstrate the capabilities of Dynamic Provable Data Possession (DPDP). The aforementioned PDP schemes work with static or warehoused data, whereas SE-PDP, a DPDP scheme, allows the client to perform dynamic file operations such as update, delete, append, and insert on data stored by a 3rd party server. SE-PDP is based on a cryptographic hash function and symmetric key encryption. The general idea for verification of the scheme is similar to MAC-PDP. The client generates a number of short verification tokens before sending the data to the server; when the client wants to verify data possession the client challenges the server with a set of random block indices. The server then computes an integrity check of the requested indices and sends it to the client. The client then

verifies that the servers proof matches it's pre-computed tokens. Unlike MAC-PDP, A-PDP, MR-PDP, and CPOR, SE-PDP does not support unlimited auditing. In SE-PDP the number of audits is chosen in advance, based on the number of generated verification tokens.

CPOR [37] is a Proof of Retrievability (POR) scheme introduced by Shacham and Waters. POR schemes are intended to allow the client to verify that data they store with a 3rd party server is always retrievable or can be reconstructed by the client. The basic implementation of POR schemes allows the client to verify data by first encrypting the data then embedding disguised blocks (called sentinels) into the ciphertext. The purpose of the sentinels is to detect data modification by the server. In the verification phase the client requests for randomly selected sentinels and checks whether or not they are corrupted. If the data stored on the server is corrupted the sentinels are influenced with a high probability. CPOR builds on this by introducing Homomorphic Linear Authenticators (HLAs), which work similarly to HVTs. HLAs allow the server to aggregate the tags of individual file blocks which allows the server to generate a short tag as a response to the client's challenge as opposed to sending back each sentinel. Next, detailed descriptions of each scheme are provided.

3.1 MAC-PDP

The MAC-PDP scheme is defined following the description and notation from Shacham and Waters [37] and Riebel [35], adapted slightly for uniformity with the other schemes in this section [13].

Let f be a keyed PRF, as follows:

$$f : \{0,1\}^* \times K_{prf} \rightarrow \mathbb{Z}_p$$

KeyGen$(1^k) \rightarrow (pk, sk)$**.** Choose a random secret key for a hash-based MAC function $k_{mac} \xleftarrow{R} K_{prf}$. The secret key is $sk = \langle k_{mac} \rangle$ and public key is $pk = \perp$.

Tag$(M; pk, sk, \omega) \rightarrow M_\eta^*$ The file is split into n blocks, $M = \langle m_1, m_2, ..., m_n \rangle$. Choose a random file ID η, where $\eta \in \mathbb{Z}_p$. For each block m_i, $(1 \leq i \leq n)$, generate tag $\sigma_i = MAC_{k_{mac}}(\eta || m_i)$. The data stored remotely is $M_\eta^* = \langle M, \{\sigma_i\}_{1 \leq i \leq n} \rangle$.

Challenge$(\eta; pk, sk, \omega) \rightarrow c$ Choose a random ℓ-element subset $I \subseteq [1, n]$ of indices. Let c be the set $\{i\}_{i \in I}$.

Proof$(\eta, M_\eta^*, c; pk) \rightarrow p$ For each $i \in c$, return to the verifier $p = \{(m_i, \sigma_i)\}_{i \in c}$.

Verify$(c, p, \eta; pk, sk, \omega) \rightarrow b \in \{0,1\}$ For each $i \in c$, check if $\sigma_i \stackrel{?}{=} MAC_{k_{mac}}(\eta || m_i)$. If all l checks are correct then return $b = 1$, else return $b = 0$.

3.2 A-PDP/MR-PDP

The A-PDP scheme is defined following the description and notation from Ateniese [5], and similarly adapted slightly for uniformity with the other schemes in this section. MR-PDP shares functionality with A-PDP. The only difference is MR-PDP provides an additional GenerateReplica function, which is detailed below.

Let H be a cryptographic hash function, h be a full-domain hash function, f be

a PRF and π be a Pseudo-Random Permutation (PRP) as follows (where κ, ℓ, λ are security parameters i.e. the tunable parameters that affect how secure a computationally-secure algorithm is):

$$h : \{0,1\}^* \rightarrow QR_N \text{ (the set of quadratic residues modulo } N)$$
$$f : \{0,1\}^\kappa \times \{0,1\}^{\log_2(n)} \rightarrow \{0,1\}^\ell$$
$$\pi : \{0,1\}^\kappa \times \{0,1\}^{\log_2(n)} \rightarrow \{0,1\}^{\log_2(n)}$$

KeyGen$(1^k) \rightarrow (pk, sk)$ Choose safe primes p, q, where $p = 2p' + 1$ and $q = 2q' + 1$. Let $N = pq$. Let g be a generator of QR_N, the set of quadratic residues modulo N. Let $v \xleftarrow{R} \{0,1\}^\kappa$. The public key $pk = \langle N, g \rangle$ and the secret key $sk = \langle e, d, v \rangle$, such that e is a large secret prime with $ed = 1 \pmod{p'q'}$, $e > \lambda$, $d > \lambda$.

Tag$(M; pk, sk, \omega) \rightarrow M_\eta^*$ The file is split into n blocks, $M = \langle m_1, m_2, ..., m_n \rangle$. For each block m_i, compute $T_{i,m_i} = (h(W_i) \cdot g^{m_i})^d \bmod N$, where $W_i = v||i$. The data stored remotely is $M_\eta^* = \langle M, \{(T_{i,m_i}, W_i)\}_{1 \leq i \leq n} \rangle$.

GenerateReplica$(M; pk, sk, \omega) \rightarrow M_i^*$ This is an **MR-PDP** specific algorithm that generates n distinct replicas $\{M_i\}_{1 \leq i \leq n}, M_i = \{b_{i,1}, b_{i,2}, ..., b_{i,m}\}$, where b_i represents each block in the original file, using random masking as follows:

for $i = 1$ to n **do**

for $j = 1$ to m **do**

1. Compute a random value $r_{i,j} = f_x(i||j)$
2. Compute the replica's block $b_{i,j} = b_j + r_{i,j}$

Challenge$(\eta; pk, sk, \omega) \rightarrow c$ To audit ℓ blocks of M, generate challenge $c = \langle \ell, k_1, k_2, g_s \rangle$, where k_1 and k_2 are random κ-bit keys, and $g_s = g^s \bmod N$ for random $s \xleftarrow{R} \mathbb{Z}_N^*$.

Proof$(\eta, M_\eta^*, c; pk) \rightarrow p$ For $1 \leq j \leq \ell$, generate indices $i_j = \pi_{k_1}(j)$ and coefficients $a_j = f_{k_2}(j)$. Compute $T = T_{i_1,m_{i_1}}^{a_1} \cdot \ldots \cdot T_{i_\ell,m_{i_\ell}}^{a_\ell} = (h(W_{i_1})^{a_1} \cdot \ldots \cdot h(W_{i_\ell})^{a_\ell} \cdot g^{a_1 m_{i_1} + \ldots + a_\ell m_{i_\ell}})^d \bmod N$. Compute $\rho = H(g_s^{a_1 m_{i_1} + \ldots + a_\ell m_{i_\ell}} \bmod N)$. The proof is $p = \langle T, \rho \rangle$.

Verify$(c, p, \eta; pk, sk, \omega) \rightarrow b \in \{0, 1\}$**.** Let $\tau = T^e$. For $1 \leq j \leq \ell$, compute $i_j = \pi_{k_1}(j), W_{i_j} = v||i_j, a_j = f_{k_2}(j)$, and $\tau = \frac{\tau}{h(W_{i_j})^{a_j}} \bmod N$. If $H(\tau^s \bmod N) = \rho$ then return $b = 1$, else return $b = 0$.

3.3 CPOR

The CPOR scheme is defined following the description and notation from Shacham and Waters [37], adapted slightly for uniformity with the other schemes in Chapter 3.3.

Let f be a keyed PRF, as follows:

$$f : \{0, 1\}^* \times \mathcal{K}_{prf} \rightarrow \mathbb{Z}_p$$

KeyGen$(1^k) \rightarrow (pk, sk)$ Choose a random key $k_{enc} \xleftarrow{R} \mathcal{K}_{enc}$ for symmetric encryption scheme Enc, and a random HMAC key $k_{mac} \xleftarrow{R} \mathcal{K}_{mac}$. The secret key is $sk = \langle k_{enc}, k_{mac} \rangle$ and public key is $pk = \perp$.

Tag$(M; pk, sk, \omega) \rightarrow M_\eta^*$ Given the file M, split M into n blocks, each s sectors

long: $M = \langle m_{ij} \rangle_{\substack{1 \le i \le n \\ 1 \le j \le s}}$. Choose a PRF key $k_{prf} \xleftarrow{R} \mathcal{K}_{prf}$ and s random numbers $\alpha_1, ..., \alpha_s \xleftarrow{R} \mathbb{Z}_p$. Let $\tau_0 = \langle n || \mathsf{Enc}_{k_{enc}}(k_{prf} || \alpha_1 || \cdots || \alpha_s) \rangle$. The file tag is $\tau = \langle \tau_0 || \mathrm{MAC}_{k_{mac}}(\tau_0) \rangle$. For each $i, 1 \le i \le n$, compute

$$\sigma_i \leftarrow f_{k_{prf}}(i) + \sum_{j=1}^{s} \alpha_j m_{ij}$$

The data stored remotely is $M_\eta^* = \langle \{m_{ij}\}, \{\sigma_i\} \rangle$.

Challenge$(\eta; pk, sk, \omega) \to c$**.** Choose a random ℓ-element subset $I \subseteq [1, n]$. For each $i \in I$ choose random $v_i \xleftarrow{R} \mathbb{Z}_p$. Let c be the set $\{(i, v_i)\}_{i \in I}$.

Proof$(\eta, M_\eta^*, c; pk) \to p$ The prover parses c as $\{(i, v_i)\}$ and computes

$$\mu_j \leftarrow \sum_{(i,v_i) \in c} v_i m_{ij} \text{ for } 1 \le j \le s, \text{ and } \sigma \leftarrow \sum_{(i,v_i) \in c} v_i \sigma_i$$

The proof is $p = \langle \mu_k, \sigma \rangle_{1 \le k \le s}$.

Verify$(c, p, \eta; pk, sk, \omega) \to b \in \{0, 1\}$ Check

$$\sigma \stackrel{?}{=} \sum_{(i,v_i) \in c} v_i f_{k_{prf}}(i) + \sum_{j=1}^{s} \alpha_j \mu_j$$

If equal then return $b = 1$, else return $b = 0$.

3.4 SE-PDP

The SE-PDP scheme is defined following the description and notation from Ateniese, et al. [6], and similarly adapted slightly for uniformity with the other schemes in this

section.

Let t be the number of possible challenges, H be a cryptographic hash function, AE be an authenticated encryption scheme, f be a keyed PRF and π be a keyed PRP, defined as follows:

$$H : \{0,1\}^* \to \{0,1\}^d$$
$$f : \{0,1\}^k \times \{0,1\}^{\log(t)} \to \{0,1\}^L$$
$$\pi : \{0,1\}^L \times \{0,1\}^{\log(n)} \to \{0,1\}^{\log(n)}$$

KeyGen$(1^k) \to (pk, sk)$ Choose secret permutation key $W \xleftarrow{R} \{0,1\}^k$, master challenge nonce key $Z \xleftarrow{R} \{0,1\}^k$ and master encryption key $K \xleftarrow{R} \{0,1\}^k$. The secret key $sk = \langle W, Z, K \rangle$. The public key $pk = \perp$.

Tag$(M; pk, sk, \omega) \to M^*_\eta$ Divide message M into n blocks. Choose the number t of possible random challenges and the number ℓ of block indices per verification. For each $1 \leq i \leq t$, generate the i-th tag as:

Generate a permutation key $k_i = f_W(i)$ and nonce $c_i = f_Z(i)$.

Compute the set of indices $\{i_j \in [1,n] \mid 1 \leq j \leq \ell\}$ where $i_j = \pi_{k_i}(j)$.

Compute token $v_i = H(c_i, m_{i_1}, \ldots, m_{i_\ell})$.

Encrypt the token $\sigma_i \leftarrow \mathsf{AE}_K(i, v_i)$.

The data stored remotely is $M^*_\eta = \langle M, \{i, \sigma_i\} \rangle$.

Challenge$(\eta; pk, sk, \omega) \to c$ Generate the i-th challenge $c = \langle k_i, c_i \rangle$ by recomputing $k_i = f_W(i)$ and $c_i = f_Z(i)$.

Proof$(\eta, M^*_\eta, c; pk) \to p$ Compute $z = H\left(c_i, m_{i_1}, \ldots, m_{i_\ell}\right)$ where $i_j = \pi_{k_i}(j)$. The proof is $p = \langle z, \sigma_i \rangle$.

Verify$(c, p, \eta; pk, sk, \omega) \to b \in \{0, 1\}$ Compute $v = \mathsf{AE}_K^{-1}(\sigma_i)$. If $v \stackrel{?}{=} (i, z)$ then return $b = 1$, else return $b = 0$.

Chapter 4

PDP TAXONOMY

This section presents and explains the PDP taxonomy. The taxonomy was created by analyzing a multitude of research papers detailing schemes related to the PDP field. After a thorough search through various PDP schemes, characteristics were selected that were found across multiple schemes. The goal of the taxonomy was to compile all the research done on PDP schemes in order to select and implement a variety of PDP schemes that accurately covered each prevalent characteristic of PDP schemes, ensuring that the chosen schemes within *libpdp* are broadly representative of the current PDP approaches. Following the taxonomy, there will be an explanation of each characteristic chosen for the taxonomy as well as an explanation of schemes within the taxonomy. Explaining the schemes present within the taxonomy will illustrate redundancies between the implemented schemes in *libpdp* and similar schemes within the taxonomy. The chosen characteristics within the taxonomy were selected with PDP schemes in mind, however, not every scheme conforms to every characteristic within the taxonomy. As such, some information for certain characteristics chosen for PDP schemes was unavailable or not pertinent to specific schemes and these cases will be denoted with N/A within the taxonomy. The leftmost column of the taxonomy contains references to individual PDP schemes. Each subsequent column corresponds to a PDP characteristic found at the top of the taxonomy. An X within the taxonomy means that the PDP scheme supports that particular characteristic.

Taxonomy

	Public Auditing	Dynamic	Multiple File Copies	Erasure Codes	Primitives	Size of Proof	Block Sizes	Public/Private Key
[2]			*X*		RSA	20 bytes	8 KB	Public
[5]	*X*				RSA	20 bytes	20 KB	Public
[3]	*X*			*X*	RSA	20 bytes	16 KB	Both
[6]		*X*			SHA-256	*N/A*	4 KB	Private
[7]					RSA	*N/A*	*N/A*	Public
[8]	*X*		*X*		RSA	257 bits	4 KB	Public
[9]	*X*	*X*	*X*		SHA-256	257 bits	4 KB	Public
[11]				*X*	IP-ECC	*N/A*	*N/A*	Private
[15]		*X*		*X*	ORAM	*N/A*	*N/A*	Private
[16]		*X*		*X*	RSA	*N/A*	*N/A*	Public
[18]			*X*		RSA	20 bytes	20 KB	Public
[21]					SHA-256	N/A	N/A	Private
[22]				*X*	PoR Codes	N/A	N/A	Private
[23]		*X*			RSA	415KB	16KB	Public
[24]		*X*	*X*		SHA-256	O(1)	16KB	Public
[25]					SHA-256	N/A	512 bits	Public
[26]	*X*			*X*	SHA-256	20 bytes	20 KB	Public
[?]					PRP	N/A	128 bits	Private
[30]	*X*	*X*			SHA-256	N/A	4 KB	Public
[31]		*X*			SHA-256	128 bits	4 KB	Public
[32]	*X*	*X*			Diffie-Hellman	N/A	N/A	Public
[34]		*X*		*X*	CMBT	160 bits	File Size / 128	Public
[38]	*X*	*X*		*X*	Merkle Trees	N/A	4 KB	Public
[39]		*X*		*X*	SHA-256	5 KB	4 KB	Public
[40]	*X*				HAPS	N/A	2 KB	Public
[42]	*X*			*X*	HVTs	2 KB	N/A	Public
[43]	*X*			*X*	HVTs	N/A	N/A	Public
[44]	*X*				SHA-1	N/A	4 KB	Public
[45]		*X*			MACs	N/A	4 KB	Public
[46]		*X*			SHA-256	N/A	N/A	Public

Table 4.1: PDP Taxonomy

4.1 Public Auditing

Public Auditing, also known as Public Verification, refers to the ability for someone other than the owner of a file to securely verify data possession or retrievability. A-PDP [5] provides this feature by making the parameter, e , public as well as restricting file size. CPOR provides this option as well. This feature is also implemented in a variety of other PDP schemes as detailed below:

Outsourced Proofs of Retrievability [3] In this paper, they introduce the notion of outsourced proofs of retrievability (OPOR), in which users can task an external auditor to perform and verify POR with the cloud provider. In order to establish public auditing, the client must distribute his/her public key to their external auditor. Then, the auditor runs the algorithm detailed in [3] for the client. If the PoR verification fails, the external auditor immediately notifies the client. The process of distributing the public key for external auditing works very similarly in A-PDP and CPOR.

Integrity Verification Over Untrusted Cloud Servers [8] This paper presents an MR-PDP scheme very similar to the MR-PDP scheme implemented in *libpdp*. Their proposed scheme consists of five algorithms: KeyGen, CopyGen,TagGen, Prove, and Verify. The function of these algorithms is nearly exactly the same as *libpdp's* MR-PDP scheme with the exception of Verify. In [8], Verify can be run by an external auditor if the public key is shared, a feature already implemented by schemes within *libpdp*.

Provable Multicopy Dynamic Data Possession In Cloud Computing Systems [9] This paper is written by the same authors as the above scheme. The proposed scheme in this paper is fairly similar; the main difference is the scheme in this paper is dynamic. The scheme consists of 7 algorithms (KeyGen, CopyGen,

TagGen, PrepareUpdate, ExecUpdate, Prove, and Verify) which is typical of DPDP schemes. The additional algorithms, PrepareUpdate and ExecUpdate, are used to allow the client to modify their data. Similar algorithms are implemented in SE-PDP, the dynamic PDP scheme implemented in *libpdp*.

Efficient Simultaneous Robust Provable Data Possession [26] The intuition behind their protocol in general is that the server is required to prove the knowledge of a linear combination of file blocks (indicated by the challenge), where the coefficients are based on a value randomly chosen by the client in each protocol run. Along with this linear combination, the server aggregates the tags corresponding to the challenged file blocks, which enable verification by the client without having access to the actual file blocks [26]. This protocol is very similar to the scheme implemented in A-PDP [5]. Public auditing in both schemes behaves relatively the same, making this scheme [26] redundant.

Enabling Proof of Retrievability in Cloud Computing [30] This PoR scheme implements the use of 2 separate servers. Particularly, one server is for auditing on behalf of the client, and the other for storage of the client's data. The client is relieved from the computation of the tags for files, which is moved and outsourced to the cloud audit server. Furthermore, the cloud audit server also plays the role of auditing for the files remotely stored in the cloud storage server [30]. The use of a server to play the role of auditor makes this scheme unique, but the execution of public auditing with this scheme is generic. The server responsible for auditing simply verifies data retrievability similarly to any external auditor.

Public Data Integrity Verification for Secure Cloud Storage [32] The implementation of this scheme is very similar to the implementation of A-PDP. The user generates keys and then splits the file up into blocks and tags each

block. To challenge, the client generates a random subset of blocks and requests a proof of data possession from the server. Using generated metadata, the client can verify the server's proof, just like A-PDP. Furthermore, the public auditing in both schemes is performed similarly making [32] redundant.

Practical Dynamic Proofs of Retrievability [38] This PoR scheme works similarly to [9] with the main difference being that it does not generate file replicas. Both schemes allow for public auditing on data that can be dynamically updated. However, the public auditing implemented in this scheme is not drastically different from the public auditing implemented within A-PDP and CPOR and is therefore considered redundant.

Panda: Public Auditing for Shared Data in the Cloud [40] One of the primary functions of Panda is to account for different scenarios involving shared data. With shared data, if a user in a group needs to be denied access, the user should no longer have access to the shared data. Therefore, although the content of shared data is not changed during user revocation, the blocks, which were previously signed by the revoked user, still need to be re-signed by an existing user in the group. As a result, the integrity of the entire data set can still be verified with the public keys of existing users only [40]. This scheme uses HVTs to allow public verification, an idea that is already implemented by A-PDP.

Proofs of Retrievability with Public Verifiability in the Cloud [42] In this scheme the data owner first breaks an erasure coded file into n blocks and generates an authentication tag for each block. All data blocks and tags are outsourced to the server; when a client wants to retrieve data from the server, he generates a challenge message and sends it to the server. The server generates a proof of correctness based on the challenge message, the public key

and the previously stored tags, then returns the response to the client. Client runs a verification algorithm upon receiving the response [42]. In this scheme, any client with access to the public key can run this, not just the data owner. Many concepts in this scheme (erasure codes, public verification, challenge/response implementations) are already implemented by schemes present in *libpdp* (namely, A-PDP and CPOR).

Secure Public Cloud Storage Auditing with Deduplication [43] This scheme combines PDP characteristics with a PoW (Proof of Ownership) scheme to improve storage efficiency as well as making sure the data is secure. PoW schemes employ deduplication of common files, meaning instead of storing multiple copies of the same file for many different users, a server will only store one copy of a file, which is shared by many different users. As far as PDP characteristics are concerned, this paper primarily explores the concept of public auditing, which they achieve in a similar manner to A-PDP and CPOR.

Public Integrity Auditing for Dynamic Data Sharing [44] Like many of the above schemes that support public auditing, this scheme functions similarly to A-PDP: the client generate keys, splits the file into blocks and tags each block. Then client sends the blocks and tags to the servers, deletes their local copies and performs challenges to the server and verification of proofs similar to the A-PDP process. Furthermore, this scheme implements public verification in a generic way, making it unnecessary to implement in *libpdp*.

4.2 Dynamic

The dynamic characteristic of PDP schemes refers to a scheme's ability to allow the client to dynamically update data stored with a 3rd party server. This allows the client to append data to existing stored data or delete existing stored data while

maintaining secure proofs of data possession or retrievability. SE-PDP is the scheme within *libpdp* that supports this feature. Schemes that implement both public auditing and dynamic features were explained above and therefore will not be re-explained in this section.

Dynamic PoR via Oblivious Ram [15] This paper implements a Dynamic PoR scheme utilizing Oblivious RAM (ORAM). The dynamic characteristics of this scheme are similar to those of SE-PDP. Furthermore, recent works on ORAM have shown that even the fastest known constructions incur a large bandwidth overhead in practice [38] making this scheme impractical to implement in *libpdp*.

Robust Dynamic PDP [16] This paper attempts to add robustness to a DPDP scheme. Robust means that the auditing scheme incorporates mechanisms for mitigating arbitrary amounts of data corruption [16]. Their protocol can be constructed in four phases: Setup, Challenge, Update, and Retrieve. SE-PDP's update algorithms are implemented similarly to the Update phase implemented in [16], making this scheme redundant.

Dynamic Provable Data Possession [23] This scheme is based on a rank-based authenticated skip list, in which, only the relative indexes of blocks are used, so it can efficiently support dynamism. The proof for a block is computed using values in the search path from that block up to the root of the skip list. Since a skip list is a tree-like structure that has probabilistic balancing guarantees, the proofs will have O(log n) complexity with high probability [23]. This scheme implements it's dynamic portions differently than SE-PDP, but the result is ultimately the same as both schemes allow the client to dynamically update data stored with 3rd party servers.

Replicated Dynamic Provable Data Possession [24] This scheme describes 3 parties: the client, the cloud, and the organizer which is a server that commu-

nicates with all other servers on the cloud. Their scheme utilizes both replication and distribution (copies and stores the original copy of the stored file as well as distributing copies to other servers). The CSP is transparent to the client through the organizer server. This scheme implements the same 7 functions that many DPDP schemes use (KeyGen, CopyGen, TagGen, PrepareUpdate, ExecUpdate, Prove, and Verify). Its heavily based on the above DPDP scheme (it uses rank-based authenticated skip lists) but also introduces distributions of replicas managed by the organizer that also gives this scheme the benefits of MR-PDP. Ultimately, the dynamic and multi-replica attributes of this scheme are already implemented by schemes present within *libpdp*.

Improved Dynamic PDP [31] This scheme presents many implementations already present within *libpdp*. In their scheme, they divide the file into blocks, generate a tag for each block, compute a hash value for each tag, and use tags to ensure the integrity of the file blocks. Their update function works similarly to SE-PDP, and the overall scheme works similarly to A-PDP.

Dynamic PDP with log(N) Complexity [34] In this paper, they extend the static PoR scheme to a dynamic scenario. That is, the client can perform update operations, e.g., insertion, deletion and modification. After each update, the client can still detect the data losses even if the server tries to hide them. They develop a new version of authenticated data structure based on a B+ tree and a merkle hash tree. They call it the Cloud Merkle B+ tree (CMBT) [34]. Exact implementation details of the CMBT are provided within [34] but the basis of their scheme functions similarly to any dynamic scheme. While [34] and SE-PDP implement their dynamic functions differently, the results are the same.

Iris: A Scalable Cloud File System [39] One of the key innovations in Iris is

the design of a sparse randomized erasure code over the file-system data and metadata. The new erasure code is specifically crafted to hide the code parity structure (typically revealed by other codes during file updates) and be resilient against a potentially adversarial cloud. It enables recovery when corruptions are detected through auditing [39]. However, this paper does not provide significantly unique implementations of concepts already implemented in libpdp. It's dynamic features behave like SE-PDP's.

Efficient Dynamic PDP [45] This paper implements similar DPDP ideas, but uses what they coin as a Balanced Update Tree. The size of the tree doesnt depend on the amount of data stored in the cloud, but rather the number of updates (modifications, insertions, and deletions) to the remote blocks. The distinctive feature of their scheme: all dynamic operations are not followed by an integrity check, which saves computation and communication cost. Instead, data is verified in the standard challenge/response format. Their format also allows revision control and support for multiple users sharing the same data. Challenge/Response is achieved via public key encryption [45]. The implemented dynamic operations in this scheme are similar to SE-PDP and their verification process is implemented in various schemes within *libpdp*.

Fair and Dynamic PoR [46] This scheme introduces a new property, called fairness, which they claim is necessary and also inherent to the setting of dynamic data because, without ensuring it, a dishonest client could legitimately accuse an honest cloud storage server of manipulating its data [46]. Like the aforementioned DPDP schemes, their implementation of dynamic operations does not deviate sharply from those of SE-PDP.

While each DPDP scheme contains unique elements, ultimately the dynamic operations of each scheme were not different enough from those present within

SE-PDP to justify adding an additional dynamic scheme to *libpdp*.

4.3 Multiple File Copies

Multiple File Copies refers to a PDP scheme's ability to allow the client to generate replicas of their data stored with a 3rd party and store the replicated file copies with other servers. Should any sort of data corruption occur, the stored replicas ensure that the client's data is still retrievable. MR-PDP is the scheme implemented within *libpdp* that supports this feature. As explained before, schemes that contain both multi-replica and one of Public Auditing or Dynamic operations are mentioned above and therefore will not be mentioned in this section.

Mirror: Enabling PoR in the Cloud [2] This paper presents an issue with multi-replica PDP: large bandwidth on the user, and the user can take advantage of reduced costs from the cloud by saying they're storing replicas when they're really not (cloud cant tell because all the files are encrypted). Mirror is a tunable replication scheme. Mirror shifts the burden of constructing replicas to the cloud provider itself [2]. It implements the concept of MR-PDP, but the cloud creates the replicas instead of the user. The PoR aspects of their scheme are based on those used in CPOR, making them extremely similar. The primary difference between this scheme and [18], the MR-PDP scheme implemented in *libpdp*, is that in this scheme the cloud is expected to generate and store replicas. However, this would be difficult to implement with current cloud APIs and require extra services to implement the additional client/server interactions.

Additional MR-PDP schemes mentioned above do not dramatically alter the implementation used in [18] making additional MR-PDP schemes unnecessary to implement within *libpdp*.

4.4 Erasure Codes

Erasure coding is a form of data protection where data is split into fragments, expanded and encoded with redundant data pieces and stored across various different locations. It is a process prevalent in PoR schemes to ensure data retrievability. CPOR is the scheme implemented within *libpdp* that employs this characteristic.

Hail: High Ability and Integrity Layer [11] This paper presents HAIL; it is used as an upgrade to POR schemes. They make use of PORs as building blocks by which storage resources can be tested and reallocated when failures are detected [11]. For the dispersal code in HAIL, they propose a new cryptographic primitive that they call an integrity-protected error-correcting code (IP-ECC). Their IP-ECC construction draws together PRFs, ECCs, and universal hash functions (UHFs) into a single primitive. This primitive is an error-correcting code that is, at the same time, a corruption-resilient MAC on the underlying message. The additional storage overhead is minimal, basically just one extra codeword symbol [11]. Rather than replicating the stored file across servers, they instead distribute it using an error-correcting (or erasure) code. The PoR schemes presented in this paper do not differ greatly from CPOR, and their use of erasure codes are similar.

PoR via Hardness Amplification [22] Introduces their own primitive, PoR Codes, as the foundation of their PoR scheme: A PoR code consists of three procedures: Init, Read, and Resp. The function Init specifies the initial encoding of the original client file F into the server file F $=$ Init(F) which is stored on the server. The functions Read and Resp are used to specify a challenge-response audit protocol. The client sends a random challenge e which consists of two parts e = (e1, e2). The first part of the challenge identifies a set of t indices (i1, . . ., i_t) = Read(e1), which correspond to t locations in F′ that the

server should read to compute its response [22]. Their use of erasure codes is to ensure data retrievability, just like that of CPOR.

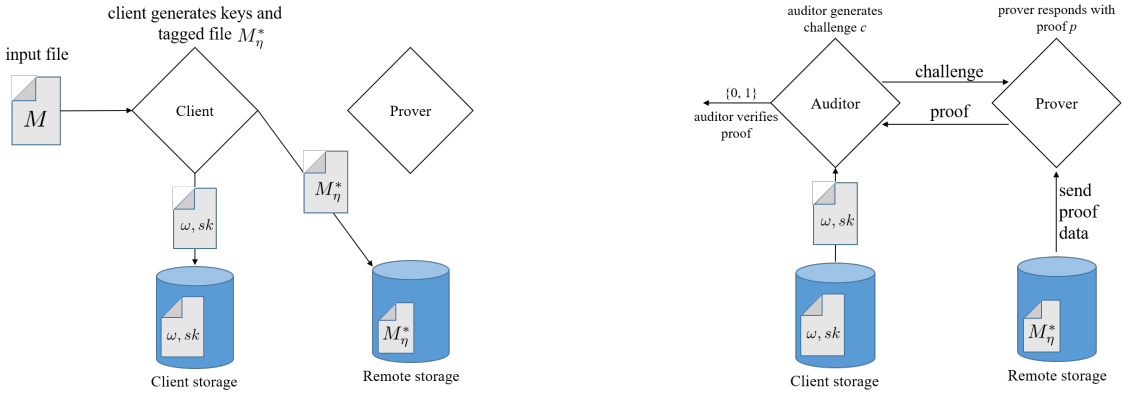
4.5 Primitives/Block Sizes/Proof Sizes/Keys

Various primitives are used throughout PDP schemes. The schemes implemented within *libpdp* implement a variety of different primitives. They also utilize different block/proof sizes with public key schemes as well as private key schemes. MAC-PDP and SE-PDP are built using symmetric key primitives: MAC-PDP is perhaps the simplest PDP scheme, requiring no (cryptographic) computation to be performed by the cloud provider, whereas the SE-PDP scheme requires some amount of server-side computation. CPOR adds public verifiability and optimizes for proof compactness. A-PDP relies upon public key primitives, requires server-side computation, and can provide public verifiability. MR-PDP builds off of A-PDP by allowing the client to generate and store file replicas. Beyond their underlying primitives, all the schemes differ in the size and computational complexity of their tags, challenges, and proofs, allowing us to differentiate storage versus computation costs [32].

Chapter 5

EXPERIMENTAL DESIGN

The performed PDP experiments can be divided into two phases: a set-up phase and an audit phase. In the set-up phase, the client generates keys (pk, sk), generates a tagged file, M_η^*, and sends M_η^* to remote storage (see Figure 5.1(a)). For the audit phase, the client generates a challenge c and sends it to the prover; the prover responds with a proof p, which is sent to the client; the client verifies the proof and indicates success or failure (see Figure 5.1(b)).



((a)) Set-up phase of PDP protocol

((b)) Audit phase of PDP protocol

Figure 5.1: Set-up and audit phases of PDP experiment. Adapted from [5]: A-PDP.

5.1 Measurements and Costs

It is important to define what system costs are measured in each of the experiments performed in *libpdp*. The operations included in *libpdp*'s measurements are included in Figure 5.2. Costs associated with transfer time and service latency are ignored, while a particular focus is put on significant, recurring computational costs. This is because various environments and deployments would suffer these costs differently, so it is hard to argue these costs as minimal or inherent.

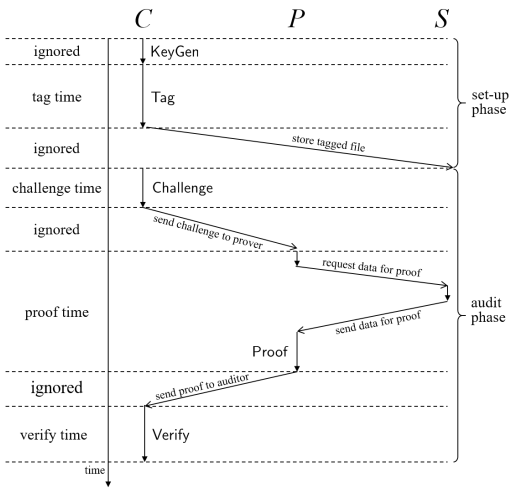


Figure 5.2: Timing Measurement Definitions

In the set-up phase the cost of generating keys (pk, sk) is ignored. During tagging, *libpdp* ignores the cost of sending the file and tag data M_η^* to the storage server S. In the audit phase, *libpdp* ignores the transfer time involved in sending the challenge to prover P and in returning the proof to client C. For proof generation, however, *libpdp* includes the time associated with retrieving challenge blocks from local or remote storage, including this as part of the proof time. We believe the cost associated with parsing the challenge, retrieving the data required for the proof, and the cost of generating the proof itself are intimately related, so these are combined in our measurement.

5.2 Implementation

Our benchmark test is a single-threaded application written in C using the *libpdp* library [33], an open-source C library providing implementations for MAC-PDP, A-PDP, MR-PDP, CPOR, and SE-PDP. In all experiments, the benchmark application is run on Amazon Web Services, including Amazon Elastic Cloud (EC2) and the Amazon Simple Storage Service (S3). The client, auditor, and prover are each run on the same EC2 instance: a c3.xlarge instance, running 64-bit Ubuntu Server 14.04

LTS using HVM virtualization. In other environments, these three parties might be separate hosts or owned by separate organizations (i.e. tagging performed by the data owner, and auditing performed by a third-party). Because it was chosen to define tag, challenge, and verify timing measurements, the properties of the network connecting these parties are irrelevant to the measurements and so these parties are run on the same host. For each of the implemented schemes, there are two types of benchmarks: using local data storage and using remote data storage. For local storage experiments, M_η^* is stored at the EC2 instance's local storage. For the remote storage experiments, M_η^* is stored to an Amazon S3 bucket.

Table 5.1: Default Benchmark Parameters

MAC-PDP	$\ell = 460$, $k_{mac} = 20$ bytes
A-PDP	$\ell = 460$, $N = 1024$ bits, PRP $k_1 = 16$ bytes, PRF $k_2 = 20$ bytes
MR-PDP	$\ell = 460$, $N = 1024$ bits, PRP $k_1 = 16$ bytes, PRF $k_2 = 20$ bytes
CPOR	$\ell = 460$, $k_{enc} = 32$ bytes, $k_{prf} = 20$, $k_{mac} = 20$ bytes, $\lambda = 80$, $p = 80$ bits, sector size = 9 bytes
SE-PDP	$\ell = 460$, AE $K = 16$ bytes, PRP $W, Z = 16$ bytes, PRF $k_i = 20$ bytes, $t = 1$

Experiments are run sequentially, each time doubling block size or file size for a particular scheme. Pre-experiment trials in which the order of experiments are randomized demonstrated no discernible impact to our results; thus, it is strongly believed that the trials are independent and order of test execution had no impact to the results. Each experiment is performed using pre-generated, random input file data. Every experiment is repeated three times (graphs in Section 6 show raw data from all three iterations). The default parameters (guided by each scheme's author recommendations) used for each scheme is provided in Table **??**.

Chapter 6

OPERATION MODELS

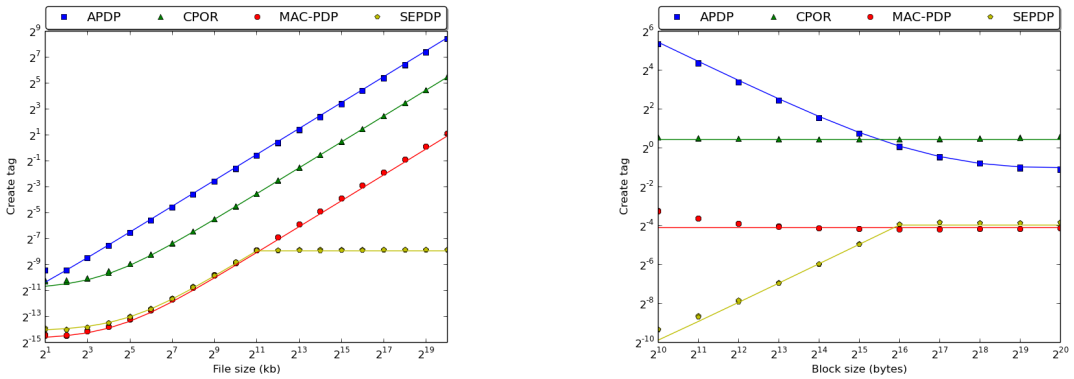
Timing data is analyzed and collected for each of the five major PDP operations: KeyGen, Tag, Challenge, Proof and Verify. For each model, the following notation is employed:

- bs, block size in bytes
- fs, file size in bytes
- ss, sector size in bytes
- $c_0, c_1, \ldots$, model-specific constants.

For all schemes, fs/bs yields the number of blocks in the file M, and bs/ss yields the number of sectors per block.All model-specific constants are derived experimentally using least-squares approximation. In each experiment, there is a point where the file size and block size are such that the total number of blocks falls below the default number of challenges selected for an audit. At this point, fewer computations are performed, resulting in faster algorithm times. Otherwise, all schemes approach some threshold where proof cost becomes constant [13]. Because MR-PDP utilizes the approach used by A-PDP, much of it's timing data is exactly the same. The only notable difference is MR-PDP endures higher overhead in certain situations due to the generation of file replicas. Therefore, it can be assumed that the results shown for A-PDP in the following data figures also apply to MR-PDP. Situations where A-PDP performs differently than MR-PDP will be noted and clarified.

6.1 Tag File

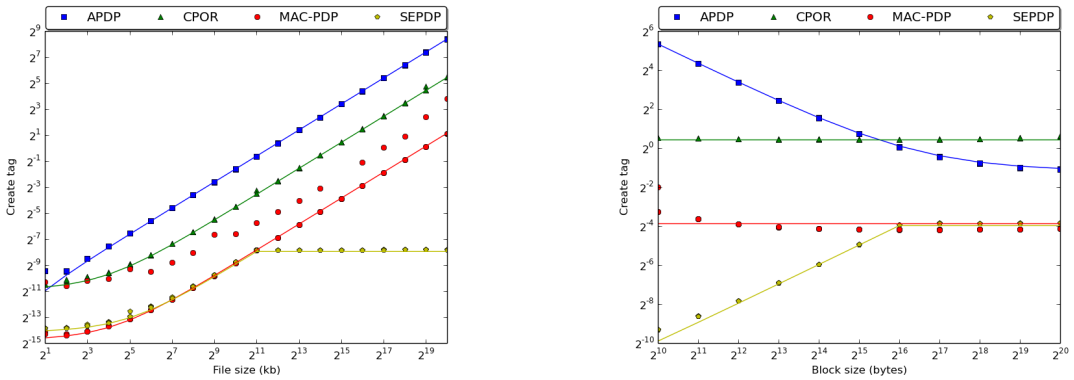
In the experiment, there is no theoretical difference between running the Tag algorithm with local data or S3 data. The measurements also bear this out, showing comparable performance.



((a)) File size vs. tag time

((b)) Block size vs. tag time

Figure 6.1: File and block size vs. tag time for local data experiments.



((a)) File size vs. tag time

((b)) Block size vs. tag time

Figure 6.2: File and block size vs. tag time for S3 data experiments.

6.1.1 MAC-PDP

When block size is held constant and file size increases, tag time increases linearly (see Figs 6.1(a) and 6.2(a)). When file size remains constant and block size varies,

tag time is nearly constant (see Figs 6.1(b) and 6.2(b)). This aligns with expectation: MAC-PDP generates tags via a hash-based MAC and the hash algorithm generates a digest through repeated operations on fixed-size blocks. Thus, the operation time should be proportional to the size of the input. These trends are summarized in Eq 6.1, expressing this simple linear model.

$$c_0 + c_1 \cdot fs \tag{6.1}$$

6.1.2 A-PDP/MR-PDP

When block size is held constant and file size increases, tag time increases linearly (see Figs 6.1(a) and 6.2(a)). When file size is held constant and block size increases, tag time decreases proportionally (see Figs 6.1(b) and 6.2(b)). This is because A-PDP and MR-PDP generate tags through modular exponentiation on every block. As file size grows, there are more blocks to tag, resulting in increased execution time. As block size increases, there are fewer blocks to tag. These trends are summarized in Eq 6.2, expressing tag time as proportional to the file size and inversely proportional to the block size.

$$c_0 + c_1 \cdot fs/bs + c_2 \cdot bs + c_3 \cdot fs \tag{6.2}$$

6.1.3 CPOR

When block size is constant and file size increases, tag time increases linearly (see Figs 6.1(a) and 6.2(a)). When file size is constant and block size increases, tag time is constant (see Figs 6.1(b) and 6.2(b)). In CPOR, tag costs are dominated by per-sector modular multiplication and addition through the use of HLAs. The change in block size has little effect on the overall cost of generating HLAs so the algorithm times remain nearly constant despite changing block sizes. These trends are summarized in

Eq 6.3, expressing tag time as proportional to file size and inversely proportional to sector size.

$$c_0 + c_1 \cdot fs + c_2 \cdot fs/ss \tag{6.3}$$

6.1.4 SE-PDP

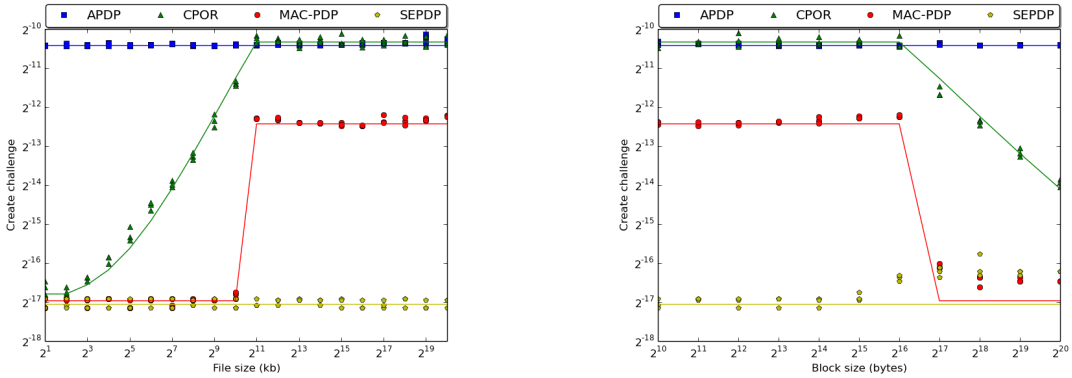
When block size is held constant and file size increases, tag time increases linearly up to a point, after which tag time remains constant (see Figs 6.1(a) and 6.2(a)). When file size is held constant and block size increases, tag time increases linearly until the point at which it becomes constant (see Figs 6.1(b) and 6.2(b)). This is because SE-PDP generates tokens by calculating the hash of a specified number of blocks. The tag time, then, is proportional to the number of bytes processed, determined by the number of blocks per token and the block size. The number of blocks per token is defined by the default security parameter ℓ. These trends are summarized in Eq 6.4, expressing tag time as proportional to total bytes processed.

$$(c_0 + c_1 \cdot \min((\min(fs/bs, \ell) \cdot bs, fs)) \cdot t \tag{6.4}$$

Above, $\min((\min(fs/bs, \ell) \cdot bs, fs)$ expressed the number of bytes processed per token. When $fs/bs < r$, the entire file is processed to generate tokens.

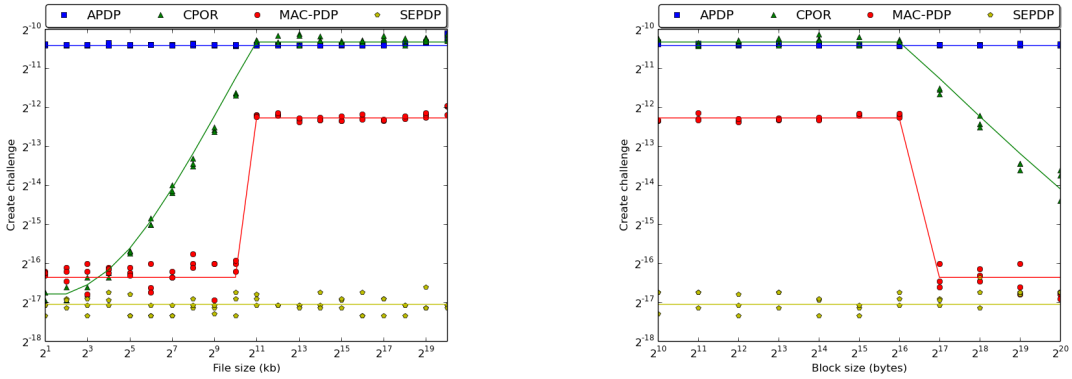
6.2 Generate Challenge

In the performed experiments, there is no theoretical difference between running the Challenge algorithm with local data or using AWS S3. The measurements verify this statement.



((a)) File size vs. challenge time ((b)) Block size vs. challenge time

Figure 6.3: File and block size vs. generate challenge time for local experiments.



((a)) File size vs. challenge time ((b)) Block size vs. challenge time

Figure 6.4: File and block size vs. generate challenge time for S3 data experiments.

6.2.1 MAC-PDP

When block size is held constant and file size increases, challenge time is constant until the point after which it switches to a slower constant cost (see Figs 6.3(a) and 6.4(a)). When file size is held constant and block size increases, similar bi-modal constant trends exist (see Figs 6.3(b) and 6.4(b)). This is explained in terms of the challenge indices ℓ and the number of blocks fs/bs. When there are fewer total blocks than ℓ, then all indices are used during the challenge. However, when there are more blocks than ℓ, then the challenge indices must be randomly selected, which requires

more complex logic than the logic required to simply select all indices, resulting in slower challenge speeds. These trends are summarized in Eq 6.5.

$$\begin{aligned} \lceil fs/bs \rceil &< \ell : c_0 \\ \lceil fs/bs \rceil &\geq \ell : c_1 \end{aligned} \tag{6.5}$$

6.2.2 A-PDP/MR-PDP

The generate challenge operation is constant time regardless of file or block size (see Figs 6.3 and 6.4). This is because A-PDP and MR-PDP challenges are independent of the file or block size. These trends are summarized in Eq 6.6, expressing challenge time as constant.

$$c_0 \tag{6.6}$$

6.2.3 CPOR

When block size is held constant and file size increases, the generate challenge time increases to the point after which it becomes constant. When the file size is held constant and the block size increases, the challenge time is constant time to a point after which it decreases proportionally to the number of blocks (see Figs 6.3(b) and 6.4(b)). This is explained in terms of CPOR generating a random ℓ-element set for the challenge. As file size increases, the size of this set increases, until the number of blocks exceeds ℓ. Similarly, when the block size increases, there may be fewer than ℓ blocks available. These trends are summarized in Eq 6.7, expressing challenge time as either constant or proportional to the number of blocks.

$$\begin{aligned} \lceil fs/bs \rceil &< \ell : c_1 + c_2 \cdot fs/bs \\ \lceil fs/bs \rceil &\geq \ell : c_0 \end{aligned} \tag{6.7}$$

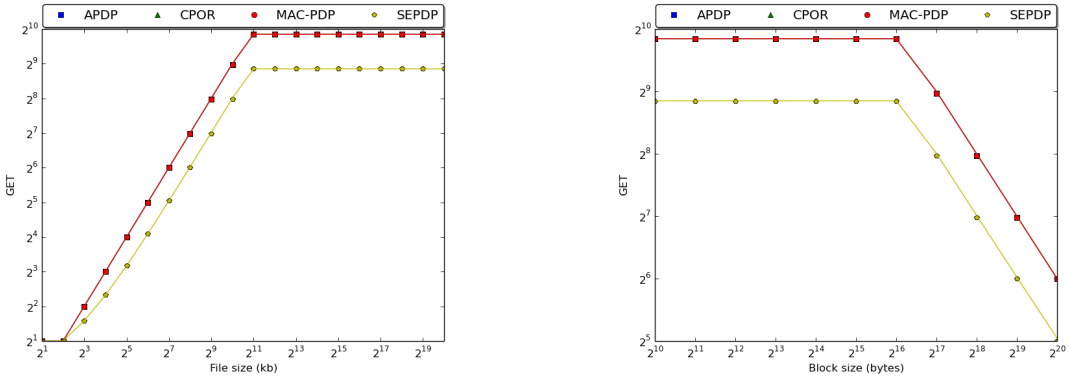
6.2.4 SE-PDP

When block size is held constant and file size increases, generate challenge runs in constant time (see Figs 6.3(a) and 6.4(a)). As the file size is held constant, challenge runs in constant time up to a point, after which the time is almost twice as slow (see Figs 6.3(b) and 6.4(b)). The former trend is explained in terms of SE-PDP recomputing k_i and c_i for the i-th challenge, neither of which is affected by the file size. Unfortunately, the latter trend seems to be an anomaly and is difficult to explain. Nothing in the algorithm design suggests that block size should affect the run time, and the anomaly is likely an artifact of implementation. These trends are summarized by Eq 6.8, expressing challenge time as constant.

$$c_0 \tag{6.8}$$

6.3 Generate Proof

Unsurprisingly, due to what is included in proof costs (see Fig 5.2), there is a noticeable difference in performance of proof costs when comparing local data storage and remote storage. These two scenarios are analyzed separately. For experiments interacting with S3, it is observed that when block size is held constant and file size increases, proof time increases linearly up to the point where the number of blocks exceeds ℓ, after which the proof time is constant (see Fig 6.7(a)). When file size is held constant and block size increases, proof time is nearly constant up to the point where ℓ exceeds the number of blocks, after which proof time decreases (see Fig 6.7(b)). This is because each `GET` from S3 takes significantly more time than generating the proof itself (see Fig 6.6). Thus, the number of `GET`s dominates this trend. For MAC-PDP, A-PDP, and CPOR the server executes one `GET` for each challenge block and



((a)) File size vs. GETs ((b)) Block size vs. GETs

Figure 6.5: File and block size vs. number of GETs from S3.

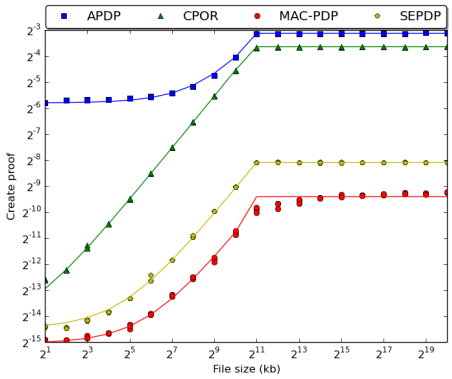
one GET for each corresponding tag (see Fig 6.5), summarized in Eq 6.9.

$$2 \cdot \min(fs/bs, \ell) \tag{6.9}$$

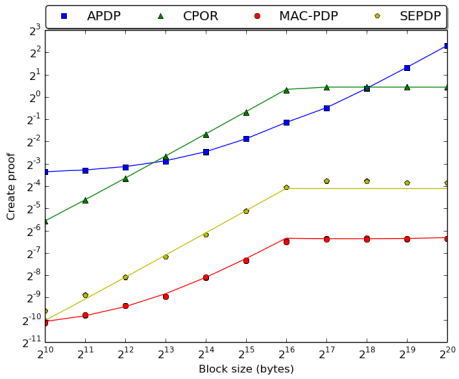
For SE-PDP, there is one GET for each challenged block, but only one GET for the token corresponding to the i-th challenge (see Fig 6.5). This is summarized in Eq 6.10, which express the number of GETs as one more than the total number of blocks or one more than ℓ, whichever is less.

In general, for local data experiments, it is observed that when block size is held constant and file size increases, the proof time increases linearly up to the point where the number of blocks exceeds the number of challenge blocks, after which proof time is nearly constant. When the file size is held constant and the block size increases, proof time increases linearly up to the point where ℓ exceeds the number of blocks, after which the proof time is constant. Each scheme reflects this same general trend, with their own model-specific constants.

$$\min(fs/bs, \ell) + 1 \tag{6.10}$$

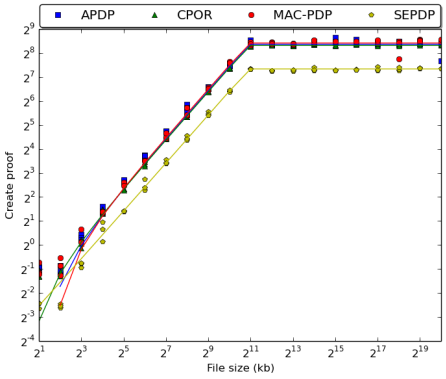


((a)) File size vs. proof time

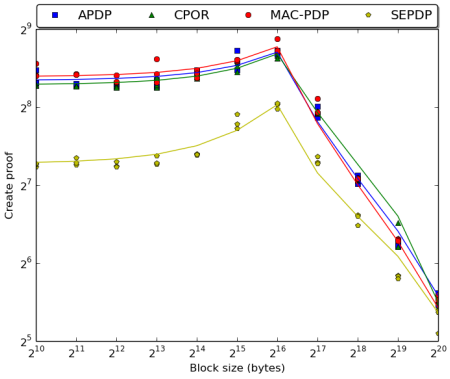


((b)) Block size vs. proof time

Figure 6.6: File and block size vs. generate proof time for local data experiments.



((a)) File size vs. proof time



((b)) Block size vs. proof time

Figure 6.7: File and block size vs. generate proof time for S3 data experiments.

6.3.1 MAC-PDP

For local data experiments, the same general trends described earlier are observed (see Figs 6.6(a) and 6.6(b)). This is because in MAC-PDP, the proof is dependent on the total number of bytes hashed. These trends are summarized in Eq 6.11, expressing proof time as proportional to the total number of blocks, file size and block size.

$$\begin{aligned} \lceil fs/bs \rceil < \ell &: c_0 + c_1 \cdot fs/bs + c_2 \cdot bs + c_3 \cdot fs \\ \lceil fs/bs \rceil \geq \ell &: c_4 + c_5 \cdot bs \end{aligned} \tag{6.11}$$

6.3.2 A-PDP/MR-PDP

For local data experiments, the same general trends described earlier are observed (see Figs 6.6(a) and 6.6(b)). This is because A-PDP and MR-PDP generates proofs through modular exponentiation of ℓ message blocks. Thus proof time depends on both the number of challenge blocks as well as the size of each block. These trends can be summarized in an identical way as in Eq 6.11, with its own model-specific constants.

6.3.3 CPOR

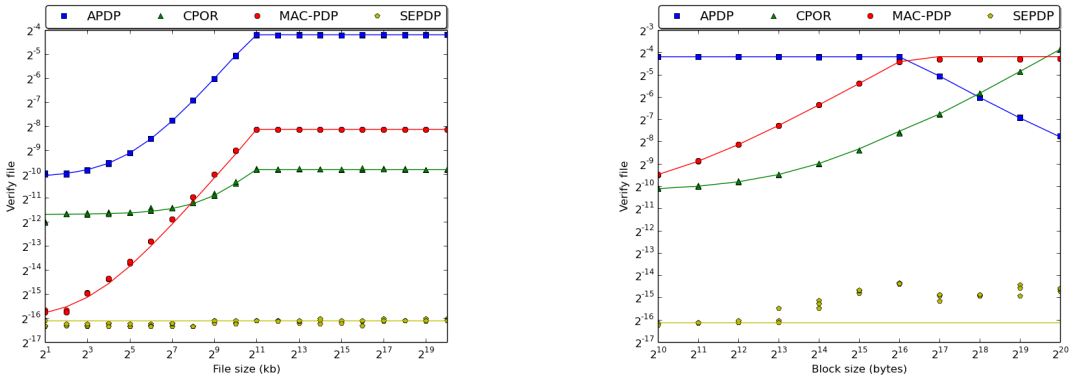
For local data experiments, the same general trends described earlier are observed (see Figs 6.6(a) and 6.6(b)). This is explained in terms of CPOR generating the proof by computing μ_j and σ for each of the indices in the challenge set. Additionally, μ_j involves modular multiplication of all sectors of each challenge block. Therefore, proof time increases with the indices in the challenge set, as well as when the block size increases. These trends can be summarized in an identical way as in Eq 6.11, with its own model-specific constants.

6.3.4 SE-PDP

For local data experiments, the same general trends described earlier are observed (see Figs 6.6(a) and 6.6(b)). This is explained in terms of SE-PDP generating the proof by computing the hash of all message blocks for a particular token. Proof time is proportional, then, to the number of bytes hashed, which is proportional to the number of challenge blocks and block size. These trends can be summarized in an identical way as in Eq 6.11, with its own model-specific constants.

6.4 Verify Proof

In our experiments, there is no theoretical difference between running the Verify algorithm with local data or using AWS S3. The measurements verify this statement.



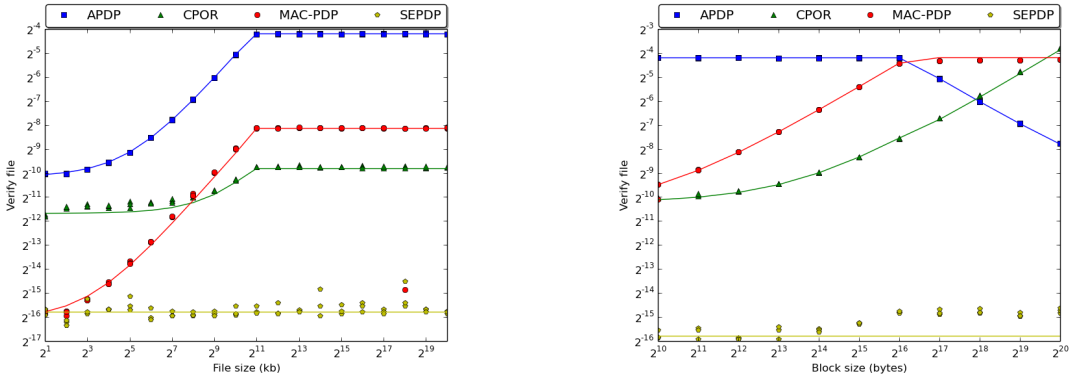
((a)) File size vs. verify time

((b)) Block size vs. verify time

Figure 6.8: File and block size vs. verify proof time for local data experiments.

6.4.1 MAC-PDP

It is observed that when block size is held constant and file size increases, verify time increases linearly up to the point where the number of blocks exceeds ℓ, after which it remains constant (see Figs 6.8(a) and 6.9(a)). When file size is held constant and block size increases, verify time increases linearly up to the point where ℓ exceeds the total



((a)) File size vs. verify time

((b)) Block size vs. verify time

Figure 6.9: File and block size vs. verify proof time for S3 data experiments.

number of blocks, after which it remains constant (see Figs 6.8(b) and 6.9(b)). This is because MAC-PDP verifies a proof by hashing each index in the challenge. Therefore, verify time is dependent on the total bytes hashed. These trends are summarized in Eq 6.12, expressing proof time as proportional to file size or proportional to block size.

$$\begin{aligned} \lceil fs/bs \rceil < \ell &: c_0 + c_1 \cdot fs \\ \lceil fs/bs \rceil \geq \ell &: c_2 + c_3 \cdot bs \end{aligned} \tag{6.12}$$

6.4.2 A-PDP/MR-PDP

It is observed that when block size is held constant and file size increases, verify time increases linearly up to the point where the number of blocks exceeds ℓ, after which it remains constant (see Figs 6.8(a) and 6.9(a)). When file size is held constant and block size increases, verify time remains constant up to the point where ℓ exceeds the total number of blocks, after which it decreases linearly (see Figs 6.8b and 6.9b). This is explained in terms of A-PDP and MR-PDP verifying proofs by generating τ

and comparing the hash of τ with ρ. Since τ is computed by generating ℓ hashes, time will be proportional to the total number of blocks challenged. These trends are summarized in Eq 6.13, expressing verify time as constant or proportional to the total number of blocks.

$$\begin{aligned} \lceil fs/bs \rceil < \ell &: c_1 + c_2 \cdot fs/bs \\ \lceil fs/bs \rceil \geq \ell &: c_0 \end{aligned} \tag{6.13}$$

6.4.3 CPOR

As in the other schemes, it is observed that when block size is held constant and file size increases, verify time increases linearly up to the point where the total number of blocks exceeds ℓ, after which it remains constant (see Figs 6.8(a) and 6.9(a)). When file size is held constant and block size increases, verify time increases linearly (see Figs 6.8(b) and 6.9(b)). This is explained in terms of CPOR verifying the proof by summing $\alpha_j \mu_j$ for all sectors of each block being challenged. As file size grows, the number of sectors for each challenge increases. As blocksize grows, the number of sectors per block increases. We summarize these trends in Eq 6.14, expressing verify time as proportional to the number of blocks, file size and block size, or proportional to just the block size.

$$\begin{aligned} \lceil fs/bs \rceil < \ell &: c_0 + c_1 \cdot fs/bs + c_2 \cdot bs + c_3 \cdot fs \\ \lceil fs/bs \rceil \geq \ell &: c_4 + c_5 \cdot bs \end{aligned} \tag{6.14}$$

6.4.4 SE-PDP

It is observed that when block size is held constant and file size increases, verify time remains constant (see Figs 6.8(a) and 6.9(a)). When the file size is held constant and

the block size increases, the verify time remains constant up to a point, after which the verify time runs anomolously slow (see Figs 6.8(b) and 6.9(b)). Generally, these SE-PDP trends reflect the cost of decrypting σ_i and comparing it with the proof. This decryption time is not dependent on file size or block size, and we believe that slow-down anomaly is an artifact of implementation and not an inherent feature of the scheme. These trends are summarized in Eq 6.15, expressing the verify time as nearly constant.

$$c_0 \tag{6.15}$$

Chapter 7

OPERATION COST MODELS

Costs are broken into three categories for analysis: (1) the upfront cost to tag, (2) the recurring cost to store and (3) the regular cost to audit. SE-PDP is not depicted on the cost graphs because, whereas MAC-PDP, A-PDP, MR-PDP, and CPOR all support an unlimited number of audits once the file is tagged, the number of audits for SE-PDP is chosen in advance. Thus a total cost graph for SE-PDP will depend on the desired frequency of audits before a file needs to be re-tagged. Additionally, the base cost of MR-PDP mirrors that of A-PDP (base cost meaning the use of no replicas). The cost of MR-PDP is directly proportional to the number of file replicas.

It is noted that costs in the results should be thought of as *minimal* costs. Some costs that reasonable implementations may incur are not measured, such as costs associated with waiting for a response from the prover, or the wake-up costs for the prover when it receives a request (see Fig 5.2). Measuring these costs would reflect network latency and implementation-specific details we do not believe to be strongly related to PDP. Also, in implementations where multiple audits may be performed for clients simultaneously, modeling 'wakeup' or 'downtime' costs would be misleading. Thus the basis costs do not reflect all possible actual costs, but can be used to accurately compare cost among schemes [13].

We chose to implement our benchmark tests on AWS; however, there are several alternatives with comparable pricing schemes and storage options—Microsoft Azure Blob storage, and Google Cloud Storage all have similar services and pricing schemes. The AWS S3 storage pricing scheme is shown in Table 7.1, current as of Sept. 2016 [1].

Table 7.1: AWS S3 Standard Storage Pricing

	Cost / GB
First 1 TB / month	$0.0300
Next 49 TB / month	$0.0295
Next 450 TB / month	$0.0290
Next 500 TB / month	$0.0285
Next 4000 TB / month	$0.0280
Over 5000 TB / month	$0.0275

Table 7.2: Comparison of Cloud Providers Remote Storage Limitations

	Max object size	Max `PUT` size	Max metadata size
Amazon S3	5 TB	5 GB	2 KB
Microsoft Azure	195 GB	64 MB	8 KB
Google Cloud Storage	5 TB	5 TB	unspecified
Rackspace	5 GB	5 GB	4 KB

7.1 Tag Costs

Tag costs consist of the cost to generate the tag and the `PUT` costs associated with uploading the file to storage (see Fig 7.1). These costs were calculated based on the cost to generate and store tags using Amazon EC2. In general, the more complex the tag, the longer it takes to create, which directly correlates to the cost create and store the tag. Therefore, these costs resemble the trends observed for timing costs associated with generating a tag (see Fig 6.1(a)), with A-PDP and MR-PDP being the most expensive, followed by CPOR, and MAC-PDP. The approximate basis costs to tag a file range from a fraction of a cent to $3 for a 1 GB file; $0.13 to $20 for a 1 TB file; and $135 to $20,400 for a 1 PB file.

7.2 Storage Costs

The storage cost of each scheme is calculated (see Fig 7.2) based on their corresponding tag sizes (see Table 7.3). Since A-PDP and MR-PDP have the largest tag size, they have the highest storage cost. MAC-PDP and CPOR have almost the same tag

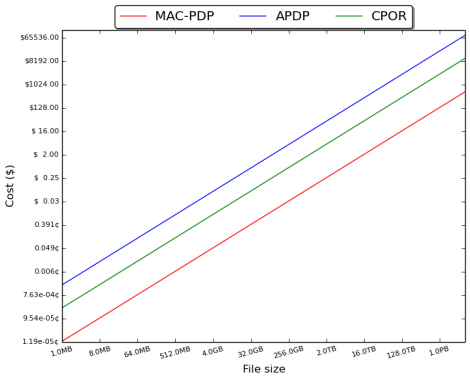


Figure 7.1: Cost to tag, based on tag algorithms and AWS EC2 pricing

size and, therefore, very similar storage costs.

It is appealing to consider the option of storing tags as metadata, attempting to get object tags "for free," however all the reviewed storage providers included metadata as part of the overall file size. Additionally, at the time of publication, AWS S3 limits metadata storage to 2KB. The maximum file sizes tags can be stored as metadata on AWS S3 are shown in Table 7.4.

7.3 Audit Costs

The total audit cost (see Fig 7.3) is calculated by determining the number of `GET`s and computational cost to generate a challenge, generate a proof and verify the proof. Since proof time is significantly larger than the challenge or verify times (compare Fig 6.7 with Figs 6.4 and 6.9), it is not surprising to find that proof time dictates audit cost trends. Additionally, differences in proof times observable in local data experiments

Table 7.3: Tag File Overhead and Tag Size

	Tag file overhead	Tag size
A-PDP	4.864%	204
MR-PDP	4.864%	204
MAC-PDP	0.477%	20
CPOR	0.429%	18

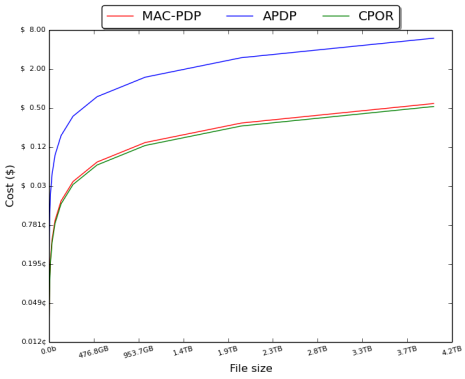


Figure 7.2: Cost to store tag, based on scheme tag overhead and AWS S3 pricing

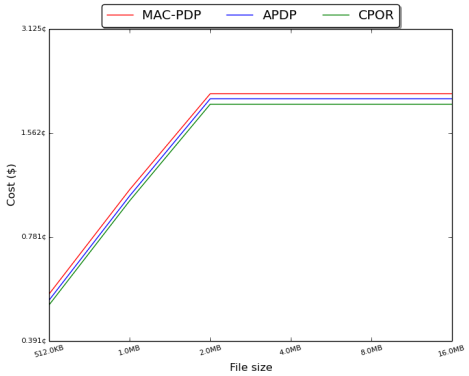
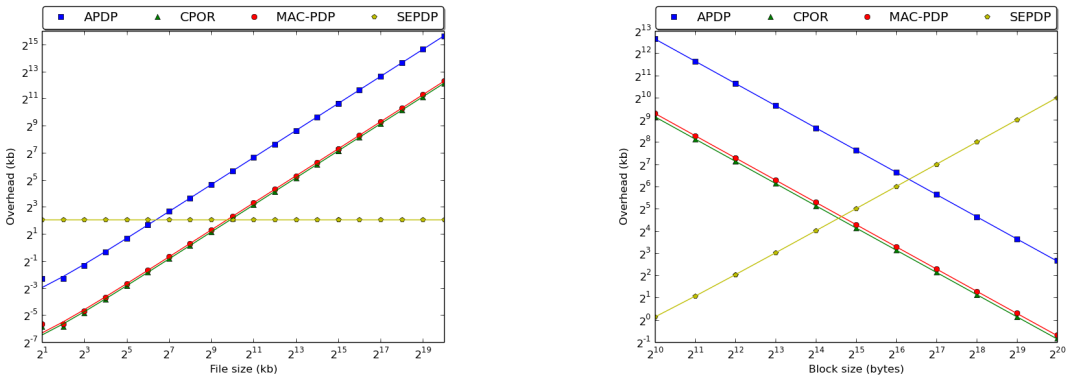


Figure 7.3: Cost to audit, based on audit cost models and AWS EC2 and S3 pricing



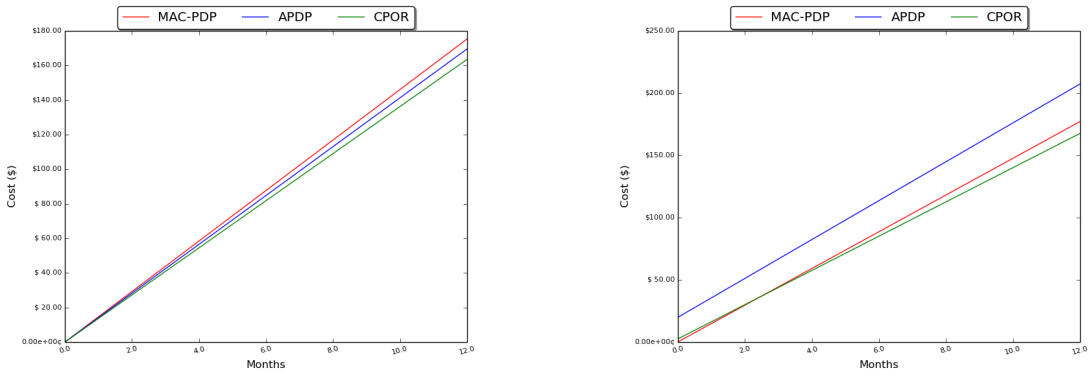
((a)) File size vs. overhead

((b)) Block size vs. overhead

Figure 7.4: File and block size vs. tag file overhead.

Table 7.4: Maximum file sizes at which tags can be stored as metadata on AWS S3

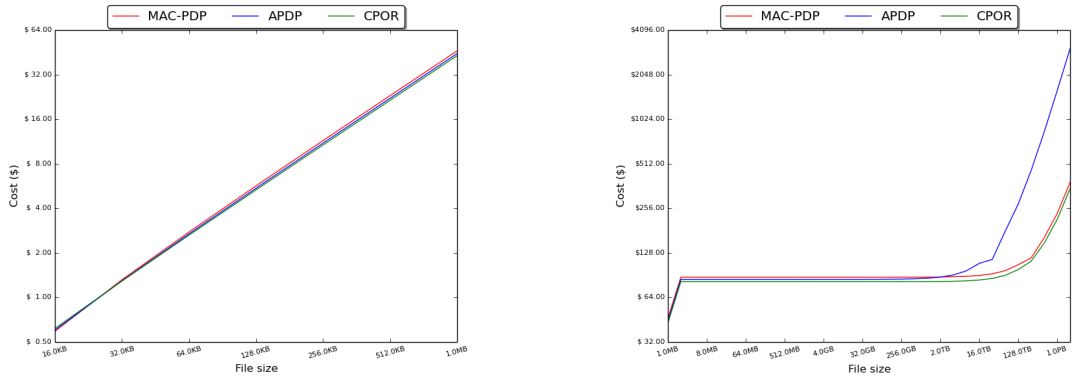
	File size
MAC-PDP	428 kb
A-PDP	41 kb
MR-PDP	41 kb
CPOR	476 kb
SE-PDP	0 kb



((a)) Tag, storage, and audit costs for 1 GB file

((b)) Tag, storage, and audit costs for 1 TB file

Figure 7.5: Cumulative tag, storage, and audit costs for one audit per hour.



((a)) File size vs. storage and audit costs

((b)) File size vs. storage and audit costs

Figure 7.6: File size vs. storage and audit costs for files at one audit per hour for one month.

(see Fig 6.6) nearly disappear in S3 experiments due to the relatively larger times required to communicate with S3 and transfer proof data. As a consequence of

the communication time common to all schemes, audit costs are nearly identical for MAC-PDP, A-PDP, MR-PDP, and CPOR. It is worth noting that the audit cost for SE-PDP is approximately half that of the three other schemes, since SE-PDP employs fewer `GET`s than the other schemes.

7.4 Total Cost Models

It is observed that the monthly cost to store and audit once per hour is nearly identical for all schemes until the storage costs begin to dominate at larger file sizes, after which A-PDP and MR-PDP become much more expensive than MAC-PDP and CPOR (see Fig 7.6). Since audit costs are nearly identical for all three schemes, the tag and storage costs have the most significant impact on the total cost of each scheme. Figs 7.5(a) and 7.5(b) show the up-front cost to tag and cumulative cost storing and auditing a 1 GB and 1 TB file, respectively, at one audit per hour each month. For the 1 GB file, the tag and storage costs are less significant and the slightly higher audit cost of MAC-PDP can be observed at one year of audits; however, the high tag and storage costs of the 1 TB file dominate, resulting in a higher cost for the A-PDP and MR-PDP schemes. The following are approximate basis costs incorporating up-front cost to tag and cumulative cost storing and auditing at one audit per hour for one year: $160 to $175 for a 1 GB file; $170 to $230 for a 1 TB file; and $2,000 to $38,700 for a 1 PB file.

MAC-PDP and CPOR generally performed better than A-PDP and MR-PDP, especially for large file sizes, leading to the conclusion that both MAC-PDP and CPOR would be more ideal for a practical implementation. SE-PDP provides benefits in terms of the lack of `GETs` executed by the server because SE-PDP requires a `GET` for each challenged block, but only one corresponding token as opposed to each corresponding tag. However, MAC-PDP is the only implemented scheme that requires

no server-side computation, meaning it should work without needing to alter any existing cloud APIs. Due to its relatively low cost and storage only nature, MAC-PDP seems to be the most practical scheme to implement.

Chapter 8

FUTURE WORK

Future work for this thesis can focus on further implementation of PDP schemes within *libpdp*. Currently, there are only 5 schemes implemented within *libpdp* and while those 5 schemes are representative of PDP schemes more schemes would provide additional cost analysis. Furthermore, *libpdp* could be used as a tool for future implementations of PDP schemes. Future implementations could use *libpdp's* cost analysis as a justification for the strength of their scheme.

The benchmark tests covered a limited number and type of PDP implementations, future studies could create benchmarks that incorporate erasure codes, or dynamic data, among other variants. The experiments performed in this thesis ignored costs associated with transfer time and service latency, focusing instead on computational costs. Follow-on work could separate the client, auditor, and prover in order to measure the communication costs between each entity. Furthermore, follow-on work could compare costs choosing different security parameters. In the experiments performed by this thesis, we selected security parameters designed to normalize comparison in terms of the strength of audit. Future work could select parameters to facilitate scheme comparison in terms of other properties, such as strength of security and efficiency of recovery.

Additional work could include attempting a practical implementation of MAC-PDP or similar scheme this thesis found more cost effective with commercial cloud services. While the cost analysis in this thesis is a useful tool for identifying which schemes could afford to be practically implemented, no practical implementation has been performed as of yet.

Chapter 9

CONCLUSION

In this thesis, generic cost models were developed as well as an open-source library implementing five, representative PDP schemes that can be used to estimate lower-bounds for economic costs. This thesis showed that for small files all schemes, regardless of their complexity, exhibit an affordable basis cost, ranging from $0.13 to $20 per year for a 1TB object. However, once the file size becomes sufficiently large, prices can range from $135 to $20,400 per year for a 1 PB object depending on the chosen scheme. The most significant contributors to this gap are the costs associated with preprocessing and storage in schemes using asymmetric key operations. Schemes that utilize symmetric key primitives (e.g. MAC-PDP and CPOR) are comparable in cost, whereas the cost of schemes utilizing public-key primitives (e.g. A-PDP and MR-PDP) becomes significantly more expensive for data sets of realistic size. Furthermore, there exists the possibility for significant cost savings for schemes that optimize for compactness and minimize the number of `GETs` performed by the server in exchange for supporting only a finite number of audits (e.g. SE-PDP).

The models used in *libpdp* are optimistic—they ignore some costs incurred by realistic, hypothetical deployments of PDP audit services for existing cloud infrastructures. Specifically, the cost models in *libpdp* ignore many costs associated with utilizing a third-party auditor. However, even with this cost ignored, schemes that utilize third-party auditors, such as CPOR, provide no practical cost advantage over the simple MAC-PDP scheme. Furthermore, CPOR is a scheme that relies on server-side computation, whereas MAC-PDP is a storage only scheme. The implementation of MAC-PDP would not require the server to provide any extra services or APIs. Because MAC-PDP requires no server-side computation, we conclude that this scheme would

be most practical to implement and that future PDP schemes should be developed with realistic cloud limitations in mind.

Bibliography

[1] Amazon Web Services. Cloud storage pricing: Amazon Simple Storage services (S3).

[2] F. Armknecht, L. Barman, J.-M. Bohli, and G. O. Karame. Mirror: Enabling proofs of data replication and retrievability in the cloud. In *Proceedings of the USENIX Security Symposium*, 2016.

[3] F. Armknecht, J.-M. Bohli, G. O. Karame, Z. Liu, and C. A. Reuter. Outsourced proofs of retrievability. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2014.

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

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

[6] G. Ateniese, R. Di Pietro, L. V. Mancini, and G. Tsudik. Scalable and efficient provable data possession. In *Proceedings of the 4th International Conference on Security and Privacy in Communication Networks*, page 9, 2008.

[7] G. Ateniese, S. Kamara, and J. Katz. Proofs of storage from homomorphic identification protocols. In M. Matsui, editor, *Advances in Cryptology – ASIACRYPT 2009*, pages 319–333, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[8] A. F. Barsoum and M. A. Hasan. Integrity verification of multiple data copies over untrusted cloud servers. In *Proceedings of the IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2012.

[9] A. F. Barsoum and M. A. Hasan. Provable multicopy dynamic data possession in cloud computing systems. *IEEE Transactions on Information Forensics and Security*, 10(3):485–497, 2015.

[10] J. Bentley and B. Floyd. Programming pearls: a sample of brilliance. *Communications of the ACM*, 30(9):754–757, 1987.

[11] K. D. Bowers, A. Juels, and A. Oprea. Hail: a high-availability and integrity layer for cloud storage. In *Proceedings of the ACM conference on Computer and communications security*, 2009.

[12] K. D. Bowers, A. Juels, and A. Oprea. Proofs of retrievability: Theory and implementation. In *Proceedings of the 2009 ACM Workshop on Cloud Computing Security*, pages 43–54, New York, NY, USA, 2009. ACM.

[13] Bremer. Cost comparison among provable data possession schemes. Master's thesis, Naval Postgraduate School, Monterey, 2016.

[14] B. Butler. What broke Amazon's cloud. *Network World*, 23 September 2015. Available: http://www.networkworld.com/article/2985554/cloud-computing/what-brought-down-amazon-s-cloud.html.

[15] D. Cash, A. Kupcu, and D. Wichs. Dynamic proofs of retrievability via oblivious ram. In T. Johansson and P. Q. Nguyen, editors, *Advances in Cryptology – EUROCRYPT 2013*, pages 279–295, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[16] B. Chen and R. Curtmola. Robust dynamic provable data possession. In *32nd International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 515–525, June 2012.

[17] B. Chen, R. Curtmola, G. Ateniese, and R. Burns. Remote data checking for network coding-based distributed storage systems. In *Proceedings of the 2010 ACM Workshop on Cloud Computing Security Workshop*, pages 31–42, 2010.

[18] R. Curtmola, O. Khan, R. Burns, and G. Ateniese. Mr-pdp: Multiple-replica provable data possession. In *2008 The 28th International Conference on Distributed Computing Systems*, pages 411–420, June 2008.

[19] D. Dash, V. Kantere, and A. Ailamaki. An economic model for self-tuned cloud caching. In *Proceedings of the IEEE International Conference on Data Engineering*, 2009.

[20] Y. Deswarte, J.-J. Quisquater, and A. Saidane. Remote integrity checking. In S. Jajodia and L. Strous, editors, *IFIP TC11/WG11.5 Sixth Working Conference on Integrity and Internal Control in Information Systems (IICIS)*, pages 1–11, 2004.

[21] R. Di Pietro and A. Sorniotti. Boosting efficiency and security in proof of ownership for deduplication. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security*, pages 81–82. ACM, 2012.

[22] Y. Dodis, S. Vadhan, and D. Wichs. Proofs of retrievability via hardness amplification. In *Proceedings of the Theory of Cryptography Conference*, 2009.

[23] C. Erway, A. Kupcu, C. Papamanthou, and R. Tamassia. Dynamic provable data possession. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, pages 213–222, New York, NY, USA, 2009. ACM.

[24] M. Etemad and A. Kupcu. Transparent, distributed, and replicated dynamic provable data possession. In *Proceedings of the 11th International Conference on Applied Cryptography and Network Security*, pages 1–18, Berlin, Heidelberg, 2013. Springer-Verlag.

[25] S. Halevi, D. Harnik, B. Pinkas, and A. Shulman-Peleg. Proofs of ownership in remote storage systems. In *Proceedings of the ACM conference on Computer and communications security*, 2011.

[26] C. Hanser and D. Slamanig. Efficient simultaneous privately and publicly verifiable robust provable data possession from elliptic curves. In *Security and Cryptography (SECRYPT), 2013 International Conference on*, pages 1–12, July 2013.

[27] A. Juels and B. S. Kaliski, Jr. PORs: Proofs of retrievability for large files. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, pages 584–597, New York, NY, USA, 2007. ACM.

[28] R. S. Kumar and A. Saxena. Data integrity proofs in cloud storage. In *Communication Systems and Networks (COMSNETS), Third International Conference on*, pages 1–4, Jan 2011.

[29] J. Li, M. Krohn, D. Mazieeres, and D. Shasha. Secure untrusted data repository (sundr). In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, pages 9–9, Berkeley, CA, USA, 2004. USENIX Association.

[30] J. Li, X. Tan, X. Chen, D. S. Wong, and F. Xhafa. Opor: Enabling proof of retrievability in cloud computing with resource-constrained devices. *Cloud Computing, IEEE Transactions on*, 3(2):195–205, 2015.

[31] F. Liu, D. Gu, and H. Lu. An improved dynamic provable data possession model. In *Cloud Computing and Intelligence Systems (CCIS), IEEE International Conference on*, pages 290–295, Sept 2011.

[32] H. Liu, P. Zhang, and J. Liu. Public data integrity verification for secure cloud storage. *Journal of Networks*, 8(2), 2013.

[33] M. Gondree and Z. Peterson. libpdp, a library for proofs of data possession.

[34] Z. Mo, Y. Zhou, and S. Chen. A dynamic proof of retrievability (PoR) scheme with big-oh (logn) complexity. In *Communications (ICC), IEEE International Conference on*, pages 912–916. IEEE, 2012.

[35] K. C. Riebel-Charity. Developing a library for proofs of data possession in charm. Master's thesis, Naval Postgraduate School, Monterey, 2013.

[36] T. S. J. Schwarz and E. L. Miller. Store, forget, and check: Using algebraic signatures to check remotely administered storage. In *Proceedings of the 26th IEEE International Conference on Distributed Computing Systems*, pages 12–33, Washington, DC, USA, 2006. IEEE Computer Society.

[37] H. Shacham and B. Waters. Compact proofs of retrievability. In J. Pieprzyk, editor, *Advances in Cryptology - ASIACRYPT 2008*, pages 90–107, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[38] E. Shi, E. Stefanov, and C. Papamanthou. Practical dynamic proofs of retrievability. In *Proceedings of the ACM SIGSAC conference on Computer & Communications Security*, 2013.

[39] E. Stefanov, M. van Dijk, A. Juels, and A. Oprea. Iris: A scalable cloud file system with efficient integrity checks. In *Proceedings of the ACM Computer Security Applications Conference*, 2012.

[40] B. Wang, B. Li, and H. Li. Panda: Public auditing for shared data with efficient user revocation in the cloud. *IEEE Transactions on Services Computing*, 8(1):92–106, Jan 2015.

[41] Y. Wang, Q. Wu, D. S. Wong, B. Qin, S. S. M. Chow, Z. Liu, and X. Tan. Securely outsourcing exponentiations with single untrusted program for cloud storage. In M. Kutylowski and J. Vaidya, editors, *19th European Symposium on Research in Computer Security (ESORICS)*, pages 326–343, Cham, 2014. Springer International Publishing.

[42] J. Yuan and S. Yu. Proofs of retrievability with public verifiability and constant communication cost in cloud. In *Proceedings of the 2013 international workshop on Security in cloud computing*, pages 19–26. ACM, 2013.

[43] J. Yuan and S. Yu. Secure and constant cost public cloud storage auditing with deduplication. In *Communications and Network Security (CNS), IEEE Conference on*, pages 145–153, Oct 2013.

[44] J. Yuan and S. Yu. Public integrity auditing for dynamic data sharing with multiuser modification. *IEEE Transactions on Information Forensics and Security*, 10(8):1717–1726, Aug 2015.

[45] Y. Zhang and M. Blanton. Efficient dynamic provable possession of remote data via balanced update trees. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, pages 183–194, New York, NY, USA, 2013. ACM.

[46] Q. Zheng and S. Xu. Fair and dynamic proofs of retrievability. In *Proceedings of the first ACM conference on Data and application security and privacy*, pages 237–248. ACM, 2011.