Efficient dynamic provable data possession with public verifiability and data privacy

Clementine Gritti
University of Wollongong, cjpg967@uowmail.edu.au

Willy Susilo
University of Wollongong, wsusilo@uow.edu.au

Thomas Plantard
University of Wollongong, thomaspl@uow.edu.au

We present a Dynamic Provable Data Possession (PDP) system with Public Verifiability and Data Privacy. Three entities are involved: a client who is the owner of the data to be stored, a server that stores the data and a Third Party Auditor (TPA) who may be required when the client wants to check the integrity of its data stored on the server. The system is publicly verifiable with the possible help of the TPA who acts on behalf of the client. The system exhibits data dynamicity at block level allowing data insertion, deletion and modification to be performed. Finally, the system is secure at the untrusted server and data private. We present a practical PDP system by adopting asymmetric pairings to gain efficiency and reduce the group exponentiation and pairing operations. In our scheme, no exponentiation and only three pairings are required during the proof of data possession check, which clearly outperforms all the existing schemes in the literature. Furthermore, our protocol supports proof of data possession on as many data blocks as possible at no extra cost.

Efficient Dynamic Provable Data Possession with Public Verifiability and Data Privacy

Clémentine Gritti, Willy Susilo and Thomas Plantard

Centre for Computer and Information Security Research
School of Computing and Information Technology
University of Wollongong, Australia
`cjpg967@uowmail.edu.au`, `{wsusilo,thomaspl}@uow.edu.au`

Abstract. We present a Dynamic Provable Data Possession (PDP) system with Public Verifiability and Data Privacy. Three entities are involved: a client who is the owner of the data to be stored, a server that stores the data and a Third Party Auditor (TPA) who may be required when the client wants to check the integrity of its data stored on the server. The system is publicly verifiable with the possible help of the TPA who acts on behalf of the client. The system exhibits data dynamicity at block level allowing data insertion, deletion and modification to be performed. Finally, the system is secure at the untrusted server and data private. We present a *practical* PDP system by adopting asymmetric pairings to gain efficiency and reduce the group exponentiation and pairing operations. In our scheme, no exponentiation and only three pairings are required during the proof of data possession check, which clearly outperforms all the existing schemes in the literature. Furthermore, our protocol supports proof of data possession on as many data blocks as possible at no extra cost.

Keywords: Provable Data Possession, Practicality, Data Operations, Public Verifiability, Data Integrity, Data Privacy.

1 Introduction

One of the most essential issue in storing data at an untrusted server is the ability to check the integrity of the data. Data owner has to ensure that the server really possesses the claimed stored data. Numerous proof-of-storage solutions have been proposed such as Proofs of Retrievability systems [1, 2] and Provable Data Possessions systems [3, 4]. In the latter system, the client is able to check that a server has stored its data without retrieving them from the server and without letting the server to access the entire data file. Both systems should satisfy the main property of *efficiency* in terms of computational and communication complexities and the storage overhead on the server's side should be as small as possible. The properties of unbounded uses on the number of proof-of-storage interactions and statelessness of the client are required to obtain systems with public verifiability, in which anyone can verify the integrity of the stored data

[5,6]. More recently, an idea emerged as delegating the data integrity check to a Third Party Auditor (TPA) [7,8]. More precisely, the client retains its data on an untrusted server and asks a trusted TPA to verify the authenticity of the stored data. This concept can be seen as a particular case of public verifiability. At the same time, another idea arised as dynamically updating the stored data [9–11]. In other words, the client is able to insert, delete and modify its stored data blocks and the server should then update these blocks on its side.

It is widely acknowledged that a storage service is susceptible to attacks or failures and leads to possible non-retrievable losses of the client's stored data. A solution is to construct a system that offers an efficient, frequent and secure data integrity check process to the client. Nevertheless, the frequency of data integrity verification and the percentage of checked data are often limited because of the computational and communication costs on both server's and client's sides, although these two properties are really essential for storage service.

1.1 Our Contributions

In this work, we provide a Dynamic Provable Data Possession (PDP) system with Public Verifiability and Data Privacy. There are three entities in the system: a client who is the owner of the data to be stored, a server that stores the data (e.g. a cloud), and a semi-honest Third Party Auditor (TPA) who can be required when the client wants to check the integrity of its data stored on the server. The client gets a large amount of data that it wants to store on the server without retaining a local copy. The server gets an important storage space and computation resources and supplies services for the client. The system is public verifiable, meaning that anyone is enabled to check the integrity of the data, not only the TPA on behalf of the client or the client itself. However, the TPA can be requested to judge whether the data integrity is maintained by checking the proof of data possession. We stress that the client may be able to perform integrity checking by itself; however, it could be limited in resources and therefore it may be neceesary to ask to the TPA (such as when the client is a mobile phone). Since this often happens naturally in practice, we only consider the case of the TPA acting on behalf of the client.

The system is also data dynamic at the block level supporting three operations: data insertion, deletion and modification. Finally, the system is secure at the untrusted server, meaning that a server cannot successfully generate a correct proof of data possession without storing all the file blocks, and data private, meaning that the TPA learns nothing about the data of the client from all available information.

Our scheme outperforms the existing schemes in the literature in terms of practicality. The first refinement is a better efficiency due the use of asymmetric pairings. The second amelioration is a decrease of the number of group exponentiation and pairing computations. In particular, the TPA needs to compute no exponentiation and only three pairings in order to verify the proof of data possession generated by the server. This implies that the latter can be requested by the client through the TPA to create the proof on any percentage of the stored data, without any computational constraints. The result of these improvements is clear in terms of performance evaluation.

1.2 Related Work

Ateniese et al. [3] first defined the notion of Provable Data Possession (PDP), which allows a client to verify the integrity of its data stored at an untrusted server without retrieving the entire file. Their scheme is designed for static data and used public key-based homomorphic tags for auditing the data file. Nevertheless, the precomputation of the tags imposes heavy computation overhead that can be expensive for entire file. Subsequently, Ateniese et al. [4] constructed scalable and efficient schemes using symmetric keys in order to improve the efficiency of verification. This results in lower overhead than their previous scheme. The scheme partially supports dynamic data operations (block updates, deletions and appends to the stored file); however, it is not publicly verifiable and is limited in number of verification requests.

Thereafter, several works were done following the models given in [3, 4]. Wang et al. [5] combined a BLS-based homomorphic authenticator with a Merkle hash tree to achieve a public auditing protocol with fully dynamic data. Hao et al. [6] designed a dynamic public auditing system based on RSA. However, they did not provide any proof of security. Their scheme is shown not to be secure in [12]. Erway et al. [9] proposed a fully dynamic PDP scheme based on rank-based authenticated dictionary. Unfortunately, their system is very inefficient. Zhu et al. [11] used index-hash tables to support fully dynamic data and constructed a zero-knowledge PDP. Zhu et al. [13] created a dynamic audit service based on fragment structure, random sampling and index-hash table that supports timely anomaly detection. Wang et al. [14] proposed a system to ensure the correctness of users' data stored on multiple servers by requiring homomorphic tokens and erasure codes in the auditing process. Le and Markopoulou [15] constructed an efficient dynamic remote data integrity checking scheme based on a homomorphic MAC scheme and CPA-secure encryption scheme and specifically designed for network coding based storage cloud. Wang et al. [16] gave a flexible distributed storage integrity auditing protocol utilizing the homomorphic token and the distributed erasure-coded data. Subsequently, Wang et al. [7] designed a privacy-preserving protocol, Oruta, that allows public auditing on shared data stored in the cloud. The scheme allows public auditing and identity privacy but fails to support large groups and traceability. In a parallel work, Wang et al. [8] presented a privacy-preserving auditing system, Knox, for data stored in the cloud and shared among a large number of users in the group. The scheme allows identity privacy, large users' number and traceability but is only for private auditing. Nevertheless, Yu et al. [17] demonstrated that the protocols in [16, 7, 8] are subject to active adversary attacks.

1.3 Paper Organization

In the next Section, we review the definitions and notations that will be used throughout this paper. Additionally, we also provide the definition of dynamic PDP scheme with public verifiability and data privacy, along with its security models. In Section 3, we present our scheme. In Section 4, we present the corresponding security proofs. In Section 5, we discuss about the computation and communication costs and we evaluate and compare the performance of our scheme with other existing ones. Finally, we conclude the paper in Section 6.

2 Definitions

We use the Homomorphic Verifiable Tags (HVT) [3] to build verification metadata linked to data blocks (defined in Appendix A). To construct our scheme, we use bilinear maps (defined in Appendix A for compleness).

Throughout this paper, we write $i \in [a, b]$ to describe that i can take all the values in the interval of reals between a and b. Let $i \in]a, b]$ (resp. $[a, b[$) mean that i can take all the values in the interval of reals between a and b, but a excluded (resp. b excluded). Let $i = 1, \cdots, n$ mean that i can take all the values in $\mathbb{N} \cap [1, n]$, where $\mathbb{N}$ is the set of naturals $\{1, 2, \cdots\}$. $\mathbb{Z}$ denotes the set of the integers $\{\cdots, -2, -1, 0, 1, 2, \cdots\}$, $\mathbb{Z}_p$ denotes the set $\{0, \cdots, p-1\}$ and $\mathbb{Z}_p^*$ denotes the set of positive integers smaller than p and relatively prime with p. $\mathbb{Q}$ denotes the set of the rationals. We let $|I|$ denote the cardinality of the set I. Let $||$ denote the symbol of concatenation, e.g. $m = m_1||m_2$ is the file made of the concatenation of the two blocks m_1 and m_2. We let $m =\perp$ mean that m does not take any value. Let $|m|$ denote the bit size of the element m.

2.1 DPDP Scheme

The following definition of the scheme follows the ones from [3] and [9]. A Dynamic Provable Data Possession scheme with Public Verifiability and Data Privacy $\mathit{\Pi} = (\textbf{KeyGen}, \textbf{Tag- Gen}, \textbf{PerfOp}, \textbf{CheckOp}, \textbf{Gen- Proof}, \textbf{CheckProof})$ is as follows:

$\textbf{KeyGen}(\lambda) \to (pk, sk)$. The probabilistic key generation algorithm is run by the client to setup the scheme. It takes as input the security parameter λ, and outputs a pair of public and secret keys (pk, sk).

$\textbf{TagGen}(pk, sk, m) \to T_m$. The (possibly) probabilistic tag generation algorithm is run by the client to generate the verification metadata. It takes as inputs the public key pk, the secret key sk and a file block m, and outputs a verification metadata T_m. Then, the client stores all the file blocks m in an ordered collection $\mathbb{F}$ and the corresponding verification metadata T_m in an ordered collection $\mathbb{E}$. It forwards these two collections to the server and deletes them from its local storage.

$\textbf{PerfOp}(pk, \mathbb{F}, \mathbb{E}, info) \to (\mathbb{F}', \mathbb{E}', \nu')$. This algorithm is run by the server in response to a data operation requested by the client. It takes as inputs the public key pk, the previous collection $\mathbb{F}$ of all the file blocks, the previous collection $\mathbb{E}$ of all the verification metadata, and the data operation details $info$ given by the client. The element $info$ specifies the operation to be performed: it can be either insertion or deletion or modification, along with other information like the rank where the operation has to be performed, the file block and the corresponding metadata that are looked at. It outputs the updated verification metadata collection $\mathbb{F}'$, the updated verification metadata collection $\mathbb{E}'$, and the related updating proof ν'. The server sends ν' to the TPA. We give more information about the data operation process below.

CheckOp$(pk, \nu') \rightarrow \{$"success", "failure"$\}$. This algorithm is run by the TPA on behalf of the client to verify the server's behavior during the data operation (insertion, deletion or modification). It takes as inputs the public key pk and the updating proof ν' sent by the server. It outputs "success" if ν' is a correct updating proof; otherwise it outputs "failure". We assume that the answer is then forwarded to the client. We omit this part of the process.

GenProof$(pk, F, chal, \varSigma) \rightarrow \nu$. This algorithm is run by the server in order to generate a proof of data possession. It takes as inputs the public key pk, an ordered collection $F \subset \mathbb{F}$ of blocks, a challenge $chal$ and an ordered collection $\varSigma \subset \mathbb{E}$ which are the verification metadata corresponding to the blocks in F. It outputs a proof of data possession ν for the blocks in F that are determined by the challenge $chal$.

We assume that a first challenge $chal_C$ is generated by the client and forwarded to the TPA. Then, the TPA generates a challenge $chal$ from $chal_C$ and sends it to the server. In particular, if the client wants to check the integrity of its data without the help of the TPA, then $chal_C = chal$. We omit the process done by the client at this point.

CheckProof$(pk, chal, \nu) \rightarrow \{$"success", "failure"$\}$. This algorithm is run by the TPA in order to validate the proof of data possession. It takes as inputs the public key pk, the challenge $chal$ and the proof of data possession ν. It outputs "success" if ν is a correct proof of data possession for the blocks determined by $chal$; otherwise it outputs "failure". We assume that the answer is then forwarded to the client. We omit this part of the process.

We require that a Dynamic Provable Data Possession scheme with Public Verifiability and Data Privacy $\varPi$ is *correct* if for $(pk, sk) \leftarrow$ **KeyGen**(λ), for $T_m \leftarrow$ **TagGen**(pk, sk, m), for $(\mathbb{F}', \mathbb{E}', \nu') \leftarrow$ **PerfOp**$(pk, \mathbb{F}, \mathbb{E}, info)$, for $\nu \leftarrow$ **GenProof**$(pk, F, chal, \varSigma)$, then "success" $\leftarrow$ **CheckOp**(pk, ν') and "success" $\leftarrow$ **CheckProof**$(pk, chal, \nu)$. We now give more details about the data operations that can be performed.

PerfOp$(pk, \mathbb{F}, \mathbb{E}, info = (\text{insertion}, \frac{2i+1}{2}, m_{\frac{2i+1}{2}}, T_{m_{\frac{2i+1}{2}}})) \rightarrow (\mathbb{F}', \mathbb{E}', \nu')$. This algorithm is run by the server in response to a data insertion requested by the client. It takes as inputs the public key pk, the previous collection $\mathbb{F}$ of all the file blocks, the previous collection $\mathbb{E}$ of all the verification metadata, the type "insertion" of the data operation to be performed, the index $\frac{2i+1}{2}$ denoting the rank where the data operation is performed (in the ordered collections $\mathbb{F}$ and $\mathbb{E}$), the file block $m_{\frac{2i+1}{2}}$ to be inserted, and the corresponding verification metadata $T_{m_{\frac{2i+1}{2}}}$ to be inserted, for $i = 0, \cdots, n$. More precisely, $m_{\frac{2i+1}{2}}$ is inserted between the existing blocks m_i and m_{i+1} and $T_{m_{\frac{2i+1}{2}}}$ is inserted between the existing verification metadata T_{m_i} and $T_{m_{i+1}}$, for $i = 1, \cdots, n-1$. For $i = 0$, $m_{\frac{1}{2}}$ is appended before m_1 and $T_{m_{\frac{1}{2}}}$ is appended before T_{m_1}. For $i = n$, $m_{\frac{2n+1}{2}}$ is appended after m_n and $T_{m_{\frac{2n+1}{2}}}$ is appended after T_{m_n}. Finally, it outputs the updated file block collection $\mathbb{F}'$ containing $m_{\frac{2i+1}{2}}$, the updated verification metadata collection $\mathbb{E}'$ containing $T_{m_{\frac{2i+1}{2}}}$, and the related updating proof ν'. The server sends ν' to the TPA.

PerfOp$(pk, \mathbb{F}, \mathbb{E}, info = (\text{deletion}, i)) \rightarrow (\mathbb{F}', \mathbb{E}', \nu')$. This algorithm is run by the server in response to a data deletion requested by the client. It takes as inputs the public key pk, the previous collection $\mathbb{F}$ of all the file blocks, the previous collection $\mathbb{E}$ of all the verification metadata, the type "deletion" of the data operation to be performed, and the index i denoting the rank where the data operation is performed (in the ordered collections $\mathbb{F}$ and $\mathbb{E}$). The server deletes the existing file block m_i, and the corresponding verification metadata T_{m_i}, for $i = 1, \cdots, n$. More precisely, m_i is deleted, giving that m_{i-1} is followed by m_{i+1} and T_{m_i} is deleted, giving that $T_{m_{i-1}}$ is followed by $T_{m_{i+1}}$, for $i = 2, \cdots, n-1$. For $i = 1$, m_1 is removed, giving that the file now begins from m_2, and T_{m_1} is removed, giving that the collection of verification metadata now begins from T_{m_2}. For $i = n$, m_n is removed, giving that the file now ends at m_{n-1}, and T_{m_n} is removed, giving that the collection of verification metadata now ends at $T_{m_{n-1}}$. Finally, it outputs the updated file block collection $\mathbb{F}'$ that does not contain m_i anymore, the updated verification metadata collection $\mathbb{E}'$ that does not contain T_{m_i} anymore, and the related updating proof ν'. The server sends ν' to the TPA. The deletion operation stops when the number of blocks is equal to 0.

PerfOp$(pk, \mathbb{F}, \mathbb{E}, info = (\text{modification}, i, m'_i, T_{m'_i})) \rightarrow (\mathbb{F}', \mathbb{E}', \nu')$. This algorithm is run by the server in response to a data modification requested by the client. It takes as inputs the public key pk, the previous collection $\mathbb{F}$ of all the file blocks, the previous collection $\mathbb{E}$ of all the verification metadata, the type "modification" of the data operation to be performed, the index i denoting the rank where the data operation is performed (in the ordered collections $\mathbb{F}$ and $\mathbb{E}$), the file block m'_i which replaces the existing block m_i, and the corresponding verification metadata $T_{m'_i}$ which replaces the existing verification metadata T_{m_i}, for $i = 1, \cdots, n$. We assume that the file block m'_i and the corresponding verification metadata $T_{m'_i}$ were provided by the client to the server, such that $T_{m'_i}$ was correctly computed by running the algorithm **TagGen**. It outputs the updated verification metadata collection $\mathbb{F}'$ replacing m_i by m'_i, the updated verification metadata collection $\mathbb{E}'$ replacing T_{m_i} by $T_{m'_i}$, and the related updating proof ν'. The server sends ν' to the TPA. We allow the client to make full re-write updates, meaning that all the file blocks $m_1, \cdots, m_n$ are replaced by $m'_1, \cdots, m'_n$ and all the verification metadata $T_{m_1}, \cdots, T_{m_n}$ are replaced by $T_{m'_1}, \cdots, T_{m'_n}$.

Remarks. About the proof of data possession: The set of data blocks (following a certain percentage of blocks; e.g. 90%) that are checked to be correctly stored are chosen by the TPA on behalf of the client. The server has to generate a proof of data possession based on this set. We notice that sometimes in the literature [6, 12], the TPA just sends a challenge *chal* without specifying which blocks have to be checked, which leads to the fact that the server must generate a proof of data possession based on all the stored data blocks, at the cost of the communication overhead.

About the data operations: We assume that the frequency of checking the integrity of the data is much higher than the frequency of performing data operations. To generate an updating proof, no challenge is required, meaning that the updating proof is based only the recently updated file block and the corresponding verification metadata. Therefore, one can think that this proof is not strong enough, however we suppose that the TPA on behalf of the client regularly asks to the server to check the integrity of the data by generating a challenge that can include the file blocks recently updated. Moreover, when the server is generating the updating proof ν', it can include an element $info' \in \{\text{insertion}, \text{deletion}, \text{modification}\}$ in this proof to enable the TPA to know which operation was performed. A solution to check that the server has correctly updated the collection $\mathbb{F}'$ of the file blocks and the collection $\mathcal{E}'$ of the verification metadata after operation is to order the data into a Merkle hash tree [5] or rank-based authenticated skip lists [9].

2.2 Security Models

Security against the server. The below-mentioned definition of the scheme follows the ones from [3] and [9]. We consider a Dynamic Provable Data Possession scheme with Public Verifiability and Data Privacy Π = (**KeyGen**, **TagGen**, **PerfOp**, **CheckOp**, **GenProof**, **CheckProof**). Let a data possession game between a challenger $\mathcal{C}$ and an adversary $\mathcal{A}$ be as follows:

KeyGen. $(pk, sk) \leftarrow \mathbf{KeyGen}(\lambda)$ is run by $\mathcal{C}$. The element pk is given to $\mathcal{A}$.

Adaptive queries. $\mathcal{A}$ makes adaptive queries through the intermediary of two oracles. The adversary is given access to a tag generation oracle $\mathcal{O}_{TG}$ as follows. $\mathcal{A}$ chooses a first block m_1 and forwards it the challenger. $\mathcal{C}$ computes the corresponding verification metadata $T_{m_1} \leftarrow \mathbf{TagGen}(pk, sk, m_1)$ and gives it to the adversary. The adversary keeps on the same queries process with $\mathcal{C}$ for the verification metadata $T_{m_2} \leftarrow \mathbf{TagGen}(pk, sk, m_2), \cdots, T_{m_n} \leftarrow \mathbf{TagGen}$ (pk, sk, m_n), where the blocks $m_2, \cdots, m_n$ are chosen by $\mathcal{A}$. Then, the adversary creates an ordered collection $\mathbb{F} = \{m_1, \cdots, m_n\}$ of file blocks along with an ordered collection $\mathbb{E} = \{T_{m_1}, \cdots, T_{m_n}\}$ of the corresponding verification metadata.

Thereafter, the adversary is given access to a data operation performance oracle $\mathcal{O}_{DOP}$ as follows. $\mathcal{A}$ submits to the challenger a block m_i, for $i = 1, \cdots, n$, and the corresponding value $info_i$ about the data operation that the adversary wants to perform. The adversary runs the algorithm **PerfOp** and outputs a new file blocks ordered collection $\mathbb{F}'$, a new metadata ordered collection $\mathbb{E}'$, and the corresponding updating proof ν'. $\mathcal{C}$ checks the value ν' by running the algorithm **CheckOp**(pk, ν') and gives back the resulting answer belonging to {"success", "failure"} to the adversary. If the answer is "failure", then the challenger aborts; otherwise, it proceeds. The above interaction between $\mathcal{A}$ and $\mathcal{C}$ can be repeated.

Setup. The adversary submits file blocks m_i^* along with the corresponding values $info_i^*$, for $i \in \mathcal{I} \subseteq]0, n+1[\cap \mathbb{Q}$. Adaptive queries are again generated by the adversary, such that the first $info_i^*$ specifies a full re-write update (this corresponds to the first time that the client sends a file to the server). The challenger verifies the data operations.

Challenge. The final version of the blocks $m_i \in \mathcal{I}$ is considered such that these blocks were created according to the data operations requested by the adversary, and verified and accepted by the challenger in the previous step. The challenger sets $\mathbb{F} = \{m_i\}_{i \in \mathcal{I}}$ of these file blocks and $\mathbb{E} = \{T_{m_i}\}_{i \in \mathcal{I}}$ of the corresponding verification metadata. $\mathcal{C}$ then takes an ordered collection $F = \{m_{i_1}, \cdots, m_{i_k}\} \subset \mathbb{F}$ and the corresponding verification metadata ordered collection $\Sigma = \{T_{m_{i_1}}, \cdots, T_{m_{i_k}}\} \subset \mathbb{E}$, for $i_j \in \mathcal{I}$, $j = 1, \cdots, k$. It generates a resulting challenge *chal* for F and Σ and forwards it to $\mathcal{A}$.

Forge. The adversary generates a proof of data possession ν on *chal*. Then, the challenger runs **CheckProof**$(pk, chal, \nu)$ and gives the answer belonging to {"success", "failure"} to $\mathcal{A}$. If the answer is "success" then the adversary wins.

The Dynamic Provable Data Possession scheme with Public Verifiability and Data Privacy Π = (**KeyGen**, **TagGen**, **PerfOp**, **CheckOp**, **GenProof**, **CheckProof**) is said to be secure if for any probabilistic polynomial-time (PPT) adversary $\mathcal{A}$ who can win the above data possession game with non-negligible probability, then the challenger $\mathcal{C}$ can extract at least the challenged parts of the file by resetting and challenging the adversary polynomially many times by means of a knowledge extractor $\mathcal{E}$.

Privacy against the TPA. The below-mentioned definition of the scheme follows the one from [12]. We consider a Dynamic Provable Data Possession scheme with Public Verifiability and Data Privacy Π = (**KeyGen**, **TagGen**, **PerfOp**, **CheckOp**, **GenProof**, **CheckProof**). Let a data privacy game between a challenger $\mathcal{C}$ and an adversary $\mathcal{A}$ be as follows:

KeyGen. $(pk, sk) \leftarrow$ **KeyGen**(λ) is run by $\mathcal{C}$. The element *pk* is given to $\mathcal{A}$.

Queries. $\mathcal{A}$ gives to the challenger two files $m_0 = m_{0,1}|| \cdots ||m_{0,n}$ and $m_1 = m_{1,1}|| \cdots ||m_{1,n}$ of equal length. $\mathcal{C}$ randomly selects a bit $b \in_R \{0, 1\}$, computes $T_{m_{b,i}} \leftarrow$ **TagGen**$(pk, sk, m_{b,i})$ for $i = 1, \cdots, n$ and gives them to $\mathcal{A}$. Then, the adversary creates an ordered collection $\mathbb{F} = \{m_{b,1}, \cdots, m_{b,n}\}$ of file blocks along with an ordered collection $\mathbb{E} = \{T_{m_{b,1}}, \cdots, T_{m_{b,n}}\}$ of the corresponding verification metadata.

Challenge. The adversary forwards *chal* to $\mathcal{C}$.

Generation of the Proof. The challenger outputs a proof of data possession $\nu^* \leftarrow$ **GenProof**$(pk, F, chal, \Sigma)$ for the blocks in F that are determined by the challenge *chal*, where $F = \{m_{b,i_1}, \cdots, m_{b,i_k}\} \subset \mathbb{F}$ is an ordered collection of blocks and $\Sigma = \{T_{m_{b,i_1}}, \cdots, T_{m_{b,i_k}}\} \subset \mathbb{E}$ is an ordered collectection of the verification metadata corresponding to the blocks in F, for $1 \leq i_j \leq n$, $1 \leq j \leq k$ and $1 \leq k \leq n$.

Guess. The adversary returns a bit b'. $\mathcal{A}$ wins if $b' = b$.

The Dynamic Provable Data Possession scheme with Public Verifiability and Data Privacy $\Pi =$ (**KeyGen**, **TagGen**, **PerfOp**, **CheckOp**, **GenProof**, **CheckProof**) is said to be data private if there is no probabilistic polynomial-time (PPT) adversary $\mathcal{A}$ who can win the above data privacy game with non-negligible advantage equal to $|Pr[b' = b] - \frac{1}{2}|$.

3 Our DPDP Construction

The file to be stored is split into n blocks, and each block is split into s sectors. We let each block and sector be elements of $\mathbb{Z}_p$ for some large prime p. For instance, let the file be b bits long. Then, the file is split into $n = \lceil b/s \cdot \log(p) \rceil$ blocks. The aforementioned intuition comes from [2]. Suppose that the blocks contain $s \geq 1$ elements of $\mathbb{Z}_p$. Therefore, a tradeoff exists between the storage overhead and the communtication overhead. More precisely, the communication complexity rises as $s + 1$ elements of $\mathbb{Z}_p$. Finally, a larger value of s yields less storage overhead at cost of a high communication. Moreover, p should be λ bits long, where λ is the security parameter such that $n >> \lambda$.

KeyGen$(\lambda) \rightarrow (pk, sk)$. Let **GroupGen**$(\lambda)$ be an algorithm that, on input the security parameter λ, generates the cyclic groups $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_T$ of prime order $p = p(\lambda)$ with bilinear map $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$. Let g_1 and g_2 be generators of $\mathbb{G}_1$ and $\mathbb{G}_2$ respectively. Then, the client randomly chooses s elements $h_1, \cdots, h_s \in_R \mathbb{G}_1$. Moreover, it selects at random $a \in_R \mathbb{Z}_p$ and sets its public key $pk = (p, \mathbb{G}_1, \mathbb{G}_2, e, g_1, g_2, h_1, \cdots, h_s, g_2^a)$ and its secret key $sk = a$.

TagGen$(pk, sk, m) \rightarrow T_m$. A file m is split into n blocks m_i, for $i = 1, \cdots, n$. Each block m_i is then split into s sectors $m_{i,j} \in \mathbb{Z}_p$, for $j = 1, \cdots, s$. We suppose that $|m| = b$ and $n = \lceil b/s \cdot \log(p) \rceil$. Therefore, the file m can be seen a $n \times s$ matrix with elements denoted as $m_{i,j}$. The client computes the verification metadata $T_{m_i} = (\prod_{j=1}^{s} h_j^{m_{i,j}})^{-sk} = (\prod_{j=1}^{s} h_j^{m_{i,j}})^{-a} = (\prod_{j=1}^{s} h_j^{-a \cdot m_{i,j}})$ for $i = 1, \cdots, n$. Then, it sets $T_m = (T_{m_1}, \cdots, T_{m_n}) \in \mathbb{G}_1^n$.

Then, the client stores all the file blocks m in an ordered collection $\mathbb{F}$ and the corresponding verification metadata T_m in an ordered collection $\mathbb{E}$. It forwards these two collections to the server and deletes them from its local storage.

PerfOp$(pk, \mathbb{F}, \mathbb{E}, info =$ (insertion, $\frac{2i+1}{2}$, $m_{\frac{2i+1}{2}}, T_{m_{\frac{2i+1}{2}}})) \rightarrow (\mathbb{F}', \mathbb{E}', \nu')$. After receiving the elements $\frac{2i+1}{2}$, $m_{\frac{2i+1}{2}}$ and $T_{m_{\frac{2i+1}{2}}}$ from the client, for $i = 0, \cdots, n$, the server prepares the updating proof as follows. It first selects at random $u_1, \cdots, u_s \in_R \mathbb{Z}_p$ and computes $U_1 = h_1^{u_1}, \cdots, U_s = h_s^{u_s}$. It also chooses at random $w_{\frac{2i+1}{2}} \in_R \mathbb{Z}_p$ and sets $c_j = m_{\frac{2i+1}{2},j} \cdot w_{\frac{2i+1}{2}} + u_j \in \mathbb{Z}_p$ for $j = 1, \cdots, s$, then $C_j = h_j^{c_j}$ for $j = 1, \cdots, s$, and $d = T_{m_{\frac{2i+1}{2}}}^{w_{\frac{2i+1}{2}}}$. Finally, it returns $\nu' = (U_1, \cdots, U_s, C_1, \cdots, C_s, d) \in \mathbb{G}_1^{2s+1}$ to the TPA.

PerfOp$(pk, \mathbb{F}, \mathbb{E}, info = (\text{deletion}, i)) \rightarrow (\mathbb{F}', \mathbb{E}', \nu')$. After receiving an index $i = 1, \cdots, n$ from the client, the server prepares the updating proof as follows. It first selects at random $u_1, \cdots, u_s \in_R \mathbb{Z}_p$ and computes $U_1 = h_1^{u_1}, \cdots, U_s = h_s^{u_s}$. It also chooses at random $w_i \in_R \mathbb{Z}_p$ and sets $c_j = m_{i,j} \cdot w_i + u_j \in \mathbb{Z}_p$ for $j = 1, \cdots, s$, then $C_j = h_j^{c_j}$ for $j = 1, \cdots, s$, and $d = T_{m_i}^{w_i}$, where m_i and T_{m_i} are the existing file block and verification metadata to be deleted respectively. Finally, it returns $\nu' = (U_1, \cdots, U_s, C_1, \cdots, C_s, d) \in \mathbb{G}_1^{2s+1}$ to the TPA.

PerfOp$(pk, \mathbb{F}, \mathbb{E}, info = (\text{modification}, i, m'_i, T_{m'_i})) \rightarrow (\mathbb{F}', \mathbb{E}', \nu')$. After receiving the elements i, m'_i and $T_{m'_i}$ from the client, the server prepares the updating proof as follows. It first selects at random $u_1, \cdots, u_s \in_R \mathbb{Z}_p$ and computes $U_1 = h_1^{u_1}, \cdots, U_s = h_s^{u_s}$. It also chooses at random $w_i \in_R \mathbb{Z}_p$ and sets $c_j = m'_{i,j} \cdot w_i + u_j \in \mathbb{Z}_p$ for $j = 1, \cdots, s$, then $C_j = h_j^{c_j}$ for $j = 1, \cdots, s$, and $d = T_{m'_i}^{w_i}$. Finally, it returns $\nu' = (U_1, \cdots, U_s, C_1, \cdots, C_s, d) \in \mathbb{G}_1^{2s+1}$ to the TPA.

CheckOp$(pk, \nu') \rightarrow \{$"success", "failure"$\}$. The TPA has to check whether the following equation holds:

$$e(d, g_2^a) \cdot e(\prod_{j=1}^{s} U_j, g_2) \stackrel{?}{=} e(\prod_{j=1}^{s} C_j, g_2) \tag{1}$$

If Eq. 1 holds, then the TPA returns "success" to the client; otherwise. it returns "failure" to the client.

GenProof$(pk, F, chal, \Sigma) \rightarrow \nu$. After receiving a challenge $chal_C$ from the client, the TPA prepares a challenge $chal$ to send to the server as follows. First, it chooses a subset $I \subseteq]0, n+1[\mathbb{Q}$, randomly chooses $|I|$ elements $v_i \in_R \mathbb{Z}_p$ and sets $chal = \{(i, v_i)\}_{i \in I}$. Second, after receiving the challenge $chal$ which indicates the specific blocks for which the client, through the TPA, wants a proof of data possession, the server sets the ordered collection $F = \{m_i\}_{i \in I} \subset \mathbb{F}$ of blocks and an ordered collection $\Sigma = \{T_{m_i}\}_{i \in I} \subset \mathbb{E}$ which are the verification metadata corresponding to the blocks in F. It then selects at random $r_1, \cdots, r_s \in_R \mathbb{Z}_p$ and computes $R_1 = h_1^{r_1}, \cdots, R_s = h_s^{r_s}$. It also sets $b_j = \sum_{(i,v_i) \in chal} m_{i,j} \cdot v_i + r_j \in \mathbb{Z}_p$ for $j = 1, \cdots, s$, then $B_j = h_j^{b_j}$ for $j = 1, \cdots, s$, and $c = \prod_{(i,v_i) \in chal} T_{m_i}^{v_i}$. Finally, it returns $\nu = (R_1, \cdots, R_s, B_1, \cdots, B_s, c) \in \mathbb{G}_1^{2s+1}$ to the TPA.

CheckProof$(pk, chal, \nu) \rightarrow \{$"success", "failure"$\}$. The TPA has to check whether the following equation holds:

$$e(c, g_2^a) \cdot e(\prod_{j=1}^{s} R_j, g_2) \stackrel{?}{=} e(\prod_{j=1}^{s} B_j, g_2) \tag{2}$$

If Eq. 2 holds, then the TPA returns "success" to the client; otherwise. it returns "failure" to the client.

Correctness. If all the algorithms are correctly generated, then the above scheme is correct. For the updating proof, we have:

$$e(d, g_2^a) \cdot e(\prod_{j=1}^{s} U_j, g_2) = e(T_{m_i}^{w_i}, g_2^a) \cdot e(\prod_{j=1}^{s} h_j^{u_j}, g_2) = e(\prod_{j=1}^{s} h_j^{m_{i,j} \cdot w_i + u_j}, g_2)$$
$$= e(\prod_{j=1}^{s} h_j^{c_j}, g_2) = e(\prod_{j=1}^{s} C_j, g_2)$$

For the proof of data possession, we have:

$$e(c, g_2^a) \cdot e(\prod_{j=1}^{s} R_j, g_2) = e\left(\prod_{\substack{(i,v_i) \\ \in chal}} T_{m_i}^{v_i}, g_2^a\right) \cdot e(\prod_{j=1}^{s} h_j^{r_j}, g_2) = e(\prod_{j=1}^{s} h_j^{\sum\limits_{\substack{(i,v_i) \\ \in chal}} m_{i,j} \cdot v_i + r_j}, g_2)$$
$$= e(\prod_{j=1}^{s} h_j^{b_j}, g_2) = e(\prod_{j=1}^{s} B_j, g_2)$$

Remarks.

About the verification metadata: The size of the verification metadata T_m is small in comparison to the size of the data blocks m. Additionally, the verification metadata protect the integrity of the blocks. Indeed, the server generates the proofs of data possession that certify the verification metadata instead of the data blocks themselves.

About the proof of data possession: The element $b_j = \sum_{(i,v_i) \in chal} m_{i,j} \cdot v_i + r_j$, for $j = 1, \cdots, s$, has size approximately equal to the size of a single block.

4 Security Proofs

Discrete Logarithm (DL) Problem. Let $\mathbb{G}_1$ be a multiplicative cyclic group of prime order $p = p(\lambda)$ (where λ is the security parameter). The DL problem is as follows: for $a \in \mathbb{Z}_p$, given $g_1, g_1^a \in \mathbb{G}_1$, output a. The DL problem holds in $\mathbb{G}_1$ if no t-time algorithm has advantage at least ε in solving the DL problem in $\mathbb{G}_1$.

Security against the Server. For any probabilistic polynomial-time (PPT) adversary $\mathcal{A}$ who wins the game, there is a challenger $\mathcal{C}$ that interacts with the adversary $\mathcal{A}$ as follows.

KeyGen. $\mathcal{C}$ runs **GroupGen**$(\lambda) \rightarrow (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e)$ and selects two generators g_1 and g_2 of $\mathbb{G}_1$ and $\mathbb{G}_2$ respectively. Then, it randomly chooses s elements $h_1, \cdots, h_s \in_R \mathbb{G}_1$ and an element $a \in_R \mathbb{Z}_p$. It sets the public key $pk = (p, \mathbb{G}_1, \mathbb{G}_2, e, g_1, g_2, h_1, \cdots, h_s, g_2^a)$ and forwards it to $\mathcal{A}$. It sets the secret key $sk = a$ and keeps it.

Adaptive queries. $\mathcal{A}$ has access to the tag generation oracle $\mathcal{O}_{TG}$ as follows. It first adaptively selects blocks m_i, for $i = 1, \cdots, n$. $\mathcal{C}$ splits each block m_i, for $i = 1, \cdots, n$ into s sectors $m_{i,j}$. Then, it computes $T_{m_i} = (\prod_{j=1}^{s} h_j^{m_{i,j}})^{-sk} =$

$(\prod_{j=1}^{s} h_j^{m_{i,j}})^{-a}$, for $i = 1, \cdots, n$, and gives them to $\mathcal{A}$. The adversary sets an ordered collection $\mathbb{F} = \{m_1, \cdots, m_n\}$ of blocks and an ordered collection $\mathbb{E} = \{T_{m_1}, \cdots, T_{m_n}\}$ which are the verification metadata corresponding to the blocks in $\mathbb{F}$. $\mathcal{A}$ has access to the data operation performance oracle $\mathcal{O}_{DOP}$ as follows. Repeatedly, the adversary selects a block m_l and the corresponding element $info_l$ and forwards them to the challenger. l denotes the rank where $\mathcal{A}$ wants the data operation to be performed; l is equal to $\frac{2i+1}{2}$ for an insertion and to i for a deletion or a modification. Moreover, $m_l =\perp$ in the case of a deletion, since only the rank is needed to perform this kind of operation. Then, $\mathcal{A}$ outputs a new file blocks ordered collection $\mathbb{F}'$ (containing the updated version of the block m_l), a new verification metadata ordered collection $\mathbb{E}'$ (containing the updated version of the verification metadata T_{m_l}) and a corresponding updating proof $\nu' = (U_1, \cdots, U_s, C_1, \cdots, C_s, d)$, such that w_l is randomly chosen from $\mathbb{Z}_p$, $d = T_{m_l}^{w_l}$, and for $j = 1, \cdots, s$, u_j is randomly chosen from $\mathbb{Z}_p$, $U_j = h_j^{r_j}$, $c_j = m_{l,j} \cdot w_l + u_j$ and $C_j = h_j^{c_j}$. $\mathcal{C}$ runs the algorithm **CheckOp** on the value ν' and sends the answer to $\mathcal{A}$. If the answer is "failure", then the challenger aborts; otherwise, it proceeds.

Setup. The adversary selects blocks m_i^* and the corresponding elements $info_i^*$, for $i \in \mathcal{I} \subseteq]0, n+1[\cap \mathbb{Q}$, and forwards them to the challenger who checks the data operations. In particular, the first $info_i^*$ indicates a full re-write.

Challenge. The challenger chooses a subset $I \subseteq \mathcal{I}$, randomly chooses $|I|$ elements $v_i \in_R \mathbb{Z}_p$ and sets $chal = \{(i, v_i)\}_{i \in I}$. It forwards $chal$ as a challenge to $\mathcal{A}$.

Forge. Upon receiving the challenge $chal$, the resulting proof of data possession on the correct stored file m should be $\nu = (R_1, \cdots, R_s, B_1, \cdots, B_s, c)$ and pass the Eq. 2. However, $\mathcal{A}$ generates a proof of data possession on an incorrect stored file $\tilde{m}$ as $\tilde{\nu} = (R_1, \cdots, R_s, \tilde{B}_1, \cdots, \tilde{B}_s, \tilde{c})$, such that r_j is randomly chosen from $\mathbb{Z}_p$, $R_j = h_j^{r_j}$, $\tilde{b}_j = \sum_{(i,v_i) \in chal} \tilde{m}_{i,j} \cdot v_i + r_j$ and $\tilde{B}_j = h_j^{\tilde{b}_j}$, for $j = 1, \cdots, s$. It also sets $\tilde{c} = \prod_{(i,v_i) \in chal} T_{\tilde{m}_i}^{v_i}$. Finally, it returns $\tilde{\nu} = (R_1, \cdots, R_s, \tilde{B}_1, \cdots, \tilde{B}_s, \tilde{c})$ to the challenger. If the proof of data possession still pass the verification, then $\mathcal{A}$ wins. Otherwise, it fails. We define $\Delta b_j = \tilde{b}_j - b_j$, for $j = 1, \cdots, s$. At least one element of $\{\Delta b_j\}_{j=1,\cdots,s}$ is non-zero.
Analysis: We provide the analysis of the above security proof in Appendix B.

Privacy against the TPA. For any probabilistic polynomial-time (PPT) adversary $\mathcal{A}$ who wins the game, there is a challenger $\mathcal{C}$ that interacts with the adversary $\mathcal{A}$ as follows.

KeyGen. $\mathcal{C}$ runs $\textbf{GroupGen}(\lambda) \rightarrow (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e)$ and selects two generators g_1 and g_2 of $\mathbb{G}_1$ and $\mathbb{G}_2$ respectively. Then, it randomly chooses s elements $h_1, \cdots, h_s \in_R \mathbb{G}_1$ and an element $a \in_R \mathbb{Z}_p$. It sets the public key $pk = (p, \mathbb{G}_1, \mathbb{G}_2, e, g_1, g_2, h_1, \cdots, h_s, g_2^a)$ and forwards it to $\mathcal{A}$. It sets the secret key $sk = a$ and keeps it.

Queries. $\mathcal{A}$ gives to the challenger two files $m_0 = m_{0,1} || \cdots || m_{0,n}$ and $m_1 = m_{1,1} || \cdots || m_{1,n}$ of equal length. $\mathcal{C}$ randomly selects a bit $b \in_R \{0, 1\}$ and for $i = 1, \cdots, n$, splits each block $m_{b,i}$ into s sectors $m_{b,i,j}$. Then, it computes

$T_{m_{b,i}} = (\prod_{j=1}^{s} h_j^{m_{b,i,j}})^{-sk} = (\prod_{j=1}^{s} h_j^{m_{b,i,j}})^{-a}$, for $i = 1, \cdots, n$, and gives them to $\mathcal{A}$.

Challenge. The adversary chooses a subset $I \subseteq \{1, \cdots, n\}$, randomly chooses $|I|$ elements $v_i \in_R \mathbb{Z}_p$ and sets $chal = \{(i, v_i)\}_{i \in I}$. It forwards $chal$ as a challenge to $\mathcal{C}$.

Generation of the Proof. Upon receiving the challenge $chal$, the challenger selects an ordered collection $F = \{m_i\}_{i \in I}$ of blocks and an ordered collection $\Sigma = \{T_{m_i}\}_{i \in I}$ which are the verification metadata corresponding to the blocks in F such that $T_{m_i} = (\prod_{j=1}^{s} h_j^{m_{i,j}})^{-sk} = (\prod_{j=1}^{s} h_j^{m_{i,j}})^{-a}$, for $i \in I$. It then randomly chooses $r_1, \cdots, r_s \in_R \mathbb{Z}_p$ and computes $R_1^* = h_1^{r_1}, \cdots, R_s^* = h_s^{r_s}$. It also randomly selects $b_1, \cdots, b_s \in \mathbb{Z}_p$ and computes $B_1^* = h_1^{b_1}, \cdots, B_s^* = h_s^{b_s}$. It sets $c^* = \prod_{(i,v_i) \in chal} T_{m_i}^{v_i}$ as well. Finally, the challenger returns $\nu^* = (R_1^*, \cdots, R_s^*, B_1^*, \cdots, B_s^*, c^*)$.

Guess. The adversary returns a bit b'.

Analysis: We provide the analysis of the above security proof in Appendix B.

5 Performance

5.1 Computational and Communication Costs

Sch.	Algorithm	Operation Total						$\mathsf{P}_{\_}$	PV	D
[8]	**KeyGen**	-	-	$3\mathsf{E}_{\mathbb{G}_1}$	$2\mathsf{E}_{\mathbb{G}_2}$	-	-	P_A	×	×
	TagGen	$1\mathsf{M}_{\mathbb{Z}_p}$	$1\mathsf{M}_{\mathbb{G}_1}$	$11\mathsf{E}_{\mathbb{G}_1}$	-	$3\mathsf{E}_{\mathbb{G}_T}$	$3\mathsf{P}$			
	GenPr.	$\|I\|\mathsf{M}_{\mathbb{Z}_p}$	$(\|I\|-1)\mathsf{M}_{\mathbb{G}_1}$	$\|I\|\mathsf{E}_{\mathbb{G}_1}$	-	-	-			
	CheckPr.	$(s+\|I\|)\mathsf{M}_{\mathbb{Z}_p}$	$(9\|I\|-6)\mathsf{M}_{\mathbb{G}_1}$	$(8+8\|I\|)\mathsf{E}_{\mathbb{G}_1}$	-	$\|I\|\mathsf{E}_{\mathbb{G}_T}$	$4\mathsf{P}$			
[13]	**KeyGen**	-	-	$2\mathsf{E}_{\mathbb{G}_1}$	/	-	-	P_S	✓	✓
	TagGen	-	$1\mathsf{M}_{\mathbb{G}_1}$	$(s+2)\mathsf{E}_{\mathbb{G}_1}$	/	-	-			
	GenPr.	$\|I\|\mathsf{M}_{\mathbb{Z}_p}$	$(s+\|I\|-2)\mathsf{M}_{\mathbb{G}_1}$	$(s+\|I\|+1)\mathsf{E}_{\mathbb{G}_1}$	/	-	$1\mathsf{P}$			
	CheckPr.	-	$(s+\|I\|-2)\mathsf{M}_{\mathbb{G}_1}$	$(s+\|I\|)\mathsf{E}_{\mathbb{G}_1}$	/	-	$3\mathsf{P}$			
Our PDP	**KeyGen**	-	-	-	$1\mathsf{E}_{\mathbb{G}_2}$	-	-	P_A	✓	✓
	TagGen	-	$(s-1)\mathsf{M}_{\mathbb{G}_1}$	$s\mathsf{E}_{\mathbb{G}_1}$	-	-	-			
	GenPr.	$\|I\|\mathsf{M}_{\mathbb{Z}_p}$	$(\|I\|-1)\mathsf{M}_{\mathbb{G}_1}$	$(2s+\|I\|)\mathsf{E}_{\mathbb{G}_1}$	-	-	-			
	CheckPr.	-	$(2s-2)\mathsf{M}_{\mathbb{G}_1}$	-	-	-	$3\mathsf{P}$			

Fig. 1. Group multiplication, group exponentiation and pairing benchmarks. $\mathsf{M}_{\mathbb{Z}_p}$ and $\mathsf{M}_{\mathbb{G}_1}$ denote multiplications in $\mathbb{Z}_p$ and $\mathbb{G}_1$ respectively. $\mathsf{E}_{\mathbb{G}_1}$, $\mathsf{E}_{\mathbb{G}_2}$ and $\mathsf{E}_{\mathbb{G}_T}$ denote exponentiations in $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_T$ respectively. P, P_A and P_S denote pairings, asymmetric pairings and symmetric pairings respectively. Let "PV" denote "public verifiability" and "D" denotes "dynamicity". Let s be the number of sector in one file block and $|I|$ be the cardinality of the set $I \subseteq]0, n+1[\cap \mathbb{Q}$ when setting the challenge $chal$. A blank means that there is no group operation performed. Let "/" point out symmetric pairings, i.e. $\mathbb{G}_1 = \mathbb{G}_2$ [13]. Let "-" denote no operation is performed.

In Figure 1, we compare the computational cost of our scheme with the ones in [8, 13]. We include the schemes from [8, 13] for comparison since they are recent and the offer similar features to our scheme. In all the schemes, during the execution of the algorithm **KeyGen**, the number of exponentiations in $\mathbb{G}_1$ and $\mathbb{G}_2$ is constant. The algorithm **TagGen** in [13] and ours requires $O(s)$ exponentiations in $\mathbb{G}_1$, where s is the number of sectors in each file block. In [8], **TagGen** needs only a constant number of exponentiations in $\mathbb{G}_1$, however there is an extra computation cost of generating the verification metadata, which is the cost of computing 3 exponentiations in $\mathbb{G}_T$ and 3 pairings. Moreover, the number of multiplications in $\mathbb{G}_1$ is constant in [8, 13], whereas it is linear in s in our case. In [13] and our scheme, the generation of proof of prossession in **GenProof** needs the computation of $O(s + |I|)$ exponentiations in $\mathbb{G}_1$; whereas in [8], the generation of proof of prossession only involves the computation of $O(|I|)$ exponentiations in $\mathbb{G}_1$. In addition, the number of multiplications in $\mathbb{G}_1$ is linear in $|I|$ in our scheme and in [8], whereas it it linear in both $|I|$ and s in [13]. The computation cost of checking the proof in **CheckProof** differs in the three schemes. In [8], the algorithm **CheckProof** requires $O(|I|)$ exponentiations in $\mathbb{G}_1$ and $\mathbb{G}_T$ and 4 pairings. In [13], the algorithm **CheckProof** needs $O(s + |I|)$ exponentiations in $\mathbb{G}_1$ and a constant number of pairings equal to 3. Moreover, the number of multiplications in both $\mathbb{Z}_p$ and $\mathbb{G}_1$ varies between the three systems. In [8], $O(s + |I|)$ and $O(|I|)$ multiplications are required in $\mathbb{Z}_p$ and $\mathbb{G}_1$ respectively. In [13], $O(s + |I|)$ multiplications in $\mathbb{G}_1$ are computed, while $O(s)$ multiplications in $\mathbb{G}_1$ are needed in our case. Finally, in our scheme, the computation cost is lighter; more precisely it is only the cost of computing 3 pairings (no exponentiation is required).

The communication cost of our protocol is mostly due to two factors: the challenge and the proof of data possession. The communication cost of a challenge $chal = \{(i, v_i)\}_{i \in I}$ is $|I|(|n| + |p|)$ bits, where $|I|$ is the number of selected file blocks, $|n|$ is the length of an index and $|p|$ is the length of an element in $\mathbb{Z}_p$. The communication cost of a proof of data possession $\nu = (R_1, \cdots, R_s, B_1, \cdots, B_s, c)$ is $(2 \cdot s + 1)|p|$ bits, where s is the number of sectors in each block. An additional cost can be the communication cost of an updating proof $\nu' = (U_1, \cdots, U_s, C_1, \cdots, C_s, d)$, which is $(2 \cdot s + 1)|p|$ bits, where s is the number of sectors in each block. However, this happens only when the client wants to update its data. We assume that the frequency of checking the integrity of the data is much higher than the frequency of performing data operations. Therefore, we leave out this additional cost.

5.2 Evaluation and Comparison of the Performance

We evaluate the practicality of our scheme and compare it to the one of the scheme in [13]. We use results of cryptographic operation implementations (exponentiations and pairings) using the MIRACL library, provided by Certivox for the MIRACL Authentication Server Project Wiki. All the following experiments are based on Borland C/C++ Compiler/Assembler and tested on a processor 2.4 GHz Intel i5 520M. For symmetric pairing-based systems (e.g. the scheme in [13]), AES with a 80-bit key and a Super Singular curve over $\mathbb{GF}_p$, for a 512-bit modulus p and an embedding degree equal to 2, are used. For asymmetric pairing-based systems (e.g. our scheme), AES with a 80-bit key and a Cocks-Pinch curve over $\mathbb{GF}_p$, for a 512-bit modulus p and an embedding degree equal to 2, are used.

		Expon. in $\mathbb{G}_1$	Expon. in $\mathbb{G}_2$	Expon. in $\mathbb{G}_T$	Pairings	Pairing Type
Time/computation		1.49	NA	0.36	3.34	Symmetric
		0.51	0.51	0.12	1.14	Asymmetric
[13] PDP	**KeyGen**	2.98	NA	-	-	Symmetric
	TagGen	151.98	NA	-	-	
	GenProof	835.89	NA	-	3.34	
	CheckProof	596	NA	-	10.02	
Our PDP	**KeyGen**	-	0.51	-	-	Asymmetric
	TagGen	51	-	-	-	
	GenProof	255	-	-	-	
	CheckProof	-	-	-	3.42	

Fig. 2. Timings for symmetric and asymmetric pairing types and pairing-based systems. Times are in milliseconds. Let "Expon." denote exponentations. Let "NA" denote "non available" to point out symmetric pairings, i.e. $\mathbb{G}_1 = \mathbb{G}_2$ [13]. Let "-" denote the results are equal to zero.

In Figure 2, we evaluate and compare the efficiency of our scheme. We assume that 2 GB data are stored. The file is split into one million blocks of size 2 KB, such that the size of the index is $|n| = 20$ bits. We assume that the number of sectors in each file block is $s = 100$ and the number of blocks determined in the challenge *chal* is $|I| = 460$. The main difference of time between the two protocols is not due to the exponentiation and pairing number difference but rather to the use of symmetric or asymmetric pairings. The total time in the algorithm **KeyGen** is 2.98 milliseconds in [13], whereas it is only 0.51 milliseconds in our construction. In the algorithm **TagGen**, it takes 151.98 milliseconds to generate the verification metadata in [13], whereas it takes one third of this time in ours. Then, it requires a total time of 839.23 milliseconds in the algorithm **GenProof** of [13], whereas, in our case, it requires only one third of this time. The reason may be that there are two stages to generate the proof in [13] (called "commitment" and "response"), while we need only one stage. Finally, in the algorithm **CheckProof**, it takes 844.42 milliseconds to check the proof of data possession in [13], whereas the time in our case is negligible. Indeed, the cost for verifying the proof is constant in our protocol.

Remarks. The probability of detecting corrupted block is $1-(1-\frac{|X|}{n})^{|I|}$, where $|X|$ is the number of corrupted blocks. Given $1,000,000$ of blocks, the challenge requires 460 blocks to allow the detection of 1% fraction of incorrect data with 99% probability of detecting misbehavior [3]. In several papers [3, 9, 7, 16], this observation is followed for the implementation and the experimentation of their protocol. Because the number of computations is constant in our verification process, we are able to check more than 460 blocks, nay all the one million of blocks. On the client's side, the computational cost remains the same during the algorithm **CheckProof** and slightly changes when creating the challenge: the client has to pick at random $|I|$ elements in $\mathbb{Z}_p$. If the number $|I|$ increases, then the client has to chooses more elements. However we consider that this cost is negligible. In addition, we assume that the server has huge storage space and computation stock: it is able to generate a proof of data possession on as many blocks as the client requests.

We note that the efficiency of the scheme in [13] and ours are dependent on the value of s. Thus, a bigger value s leads to a weaker efficiency for both of the systems. Nevertheless, our scheme still remains more practical than the one in [13] since the number of group exponentiations in the algorithms **TagGen** and **GenProof** is linear in s in both of the systems and the number of exponentiations in the algorithm **CheckProof** is also linear in s in [13] whereas there is no exponentation is required in our scheme.

6 Conclusion

We proposed an efficient Dynamic PDP with Public Verifiability and Data Privacy, which is more practical than the existing schemes in the literature. In particular, in order to check the proof of data possession generated by the server, the client is not required to compute any exponentiation but rather only a constant number of pairings.

References

1. A. Juels and Jr. B. S. Kaliski. Pors: Proofs of retrievability for large files. In *Proc. of CCS '07.*
2. H. Shacham and B. Waters. Compact proofs of retrievability. In *Proc. of ASIACRYPT '08.*
3. G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song. Provable data possession at untrusted stores. In *Proc. of CCS '07.*
4. G. Ateniese, R. Di Pietro, L. V. Mancini, and G.e Tsudik. Scalable and efficient provable data possession. In *Proc. of SecureComm '08.*
5. C. Wang, Q. Wang, K. Ren, and W. Lou. Privacy-preserving public auditing for data storage security in cloud computing. In *Proc. of INFOCOM '10.*
6. Z. Hao, S. Zhong, and N. Yu. A privacy-preserving remote data integrity checking protocol with data dynamics and public verifiability. *IEEE TKDE*, 23(9):1432–1437, September 2011.
7. B. Wang, B. Li, and H. Li. Oruta: privacy-preserving public auditing for shared data in the cloud. *IEEE TCC*, 2(1):43–56, 2012.
8. B. Wang, B. Li, and H. Li. Knox: Privacy-preserving auditing for shared data with large groups in the cloud. In *Proc. of ACNS '12.*
9. C. Erway, A. Küpçü, C. Papamanthou, and R. Tamassia. Dynamic provable data possession. In *Proc. of CCS '09.*

10. Q. Wang, C. Wang, J. Li, K. Ren, and W. Lou. Enabling public verifiability and data dynamics for storage security in cloud computing. In *Proc. of ESORICS '09*.
11. Y. Zhu, H. Wang, Z. Hu, G.-J. Ahn, H. Hu, and S. S. Yau. Dynamic audit services for integrity verification of outsourced storages in clouds. In *Proc. of SAC '11*.
12. Y. Yu, M. H. Au, Y. Mu, S. Tang, J. Ren, W. Susilo, and L. Dong. Enhanced privacy of a remote data integrity-checking protocol for secure cloud storage. *IJIS*, pages 1–12, 2014.
13. Y. Zhu, G.-J. Ahn, H. Hu, S. S. Yau, H. G. An, and C.-J. Hu. Dynamic audit services for outsourced storages in clouds. *IEEE TSC*, 6(2):227–238, 2013.
14. C. Wang, Q. Wang, K. Ren, and W. Lou. Ensuring data storage security in cloud computing. In *Proc. of IWQoS '09*, 2009.
15. A. Le and A. Markopoulou. Nc-audit: Auditing for network coding storage. *CoRR*, abs/1203.1730, 2012.
16. C. Wang, Q. Wang, K. Ren, N. Cao, and W. Lou. Toward secure and dependable storage services in cloud computing. *IEEE TSC*, 5(2):220–232, January 2012.
17. Y. Yu, L. Niu, G. Yang, Y. Mu, and W. Susilo. On the security of auditing mechanisms for secure cloud storage. *GCS*, 30(1):127–132, 2014.

A Definitions

Homomorphic Verifiable Tags (HVT). For each file block m, a HVT T_m is created. A HVT acts as a verification metadata for the file blocks and besides being unforgeable, it has the *blockless verification* property (the server constructs a proof of data possession allowing the client to verify if the server possesses certain file blocks even when the client does not have access to the actual file blocks along with the *homomorphic tags* property (given two verification metadata T_{m_i} and T_{m_j}, anyone can combine them into the verification metadata $T_{m_i+m_j}$ corresponding to the sum of the files $m_i + m_j$). The server stores the HVTs along with the file. The client should be able to check the HVTs on specific blocks, even though it does not possess any of these blocks, that is possible based on the blockless verification property of HVT.

Bilinear Maps. Let $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_T$ be three multiplicative cyclic groups of prime order $p = p(\lambda)$ (where λ is the security parameter). Let g_1 be a generator of $\mathbb{G}_1$, g_2 be a generator of $\mathbb{G}_2$ and $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$ be a bilinear map with the *bilinearity* property ($\forall u \in \mathbb{G}_1, \forall v \in \mathbb{G}_2, \forall a, b \in \mathbb{Z}_p, e(u^a, v^b) = e(u, v)^{ab}$) along with the *non-degeneracy* property ($e(g_1, g_2) \neq 1_{\mathbb{G}_T}$). $(\mathbb{G}_1, \mathbb{G}_2)$ is said to be a bilinear group if the group operation in $(\mathbb{G}_1, \mathbb{G}_2)$ and the bilinear map e are both efficiently computable.

B Analysis of the Security Proofs

Security against the Server. We prove that if the adversary can win the game, then a solution to the DL problem is found, which contradicts the assumption that the DL problem is hard in $\mathbb{G}_1$. Let assume that the server wins the game. Then, according to Eq. 2, we have $e(c, g_2^a) \cdot e(\prod_{j=1}^{s} R_j, g_2) = e(\prod_{j=1}^{s} \tilde{B}_j, g_2)$.

Since the proof $\nu = (R_1, \cdots, R_s, B_1, \cdots, B_s, c)$ is a correct one, we also have $e(c, g_2^a) \cdot e(\prod_{j=1}^{s} R_j, g_2) = e(\prod_{j=1}^{s} B_j, g_2)$. Therefore, we get that $\prod_{j=1}^{s} \tilde{B}_j = \prod_{j=1}^{s} B_j$. We can re-write as $\prod_{j=1}^{s} h_j^{\tilde{b}_j} = \prod_{j=1}^{s} h_j^{b_j}$ or even as $\prod_{j=1}^{s} h_j^{\Delta b_j} = 1$. For two elements $g, h \in \mathbb{G}_1$, there exists $x \in \mathbb{Z}_p$ such that $h = g^x$ since $\mathbb{G}_1$ is a cyclic group. Without loss of generality, given $g, h \in \mathbb{G}_1$, each h_j could randomly and correctly be generated by computing $h_j = g^{y_j} \cdot h^{z_j} \in \mathbb{G}_1$ such that y_j and z_j are random values of $\mathbb{Z}_p$. Then, we have $1 = \prod_{j=1}^{s} h_j^{\Delta b_j} = \prod_{j=1}^{s} (g^{y_j} \cdot h^{z_j})^{\Delta b_j} = g^{\sum_{j=1}^{s} y_j \cdot \Delta b_j} \cdot h^{\sum_{j=1}^{s} z_j \cdot \Delta b_j}$. Clearly, we can find a solution to the DL problem. More specifically, given $g, h = g^x \in \mathbb{G}_1$, we can compute $h = g^{\frac{\sum_{j=1}^{s} y_j \cdot \Delta b_j}{\sum_{j=1}^{s} z_j \cdot \Delta b_j}} = g^x$ unless the denominator is zero. However, as we defined in the game, at least one element of $\{\Delta b_j\}_{j=1,\cdots,s}$ is non-zero. Since z_j is a random element of $\mathbb{Z}_p$, the denominator is zero with probability equal to $1/p$, which is negligible. Thus, if the adversary wins the game, then a solution of the DL problem can be found with probability equal to $1 - \frac{1}{p}$, which contradicts the fact that the DL problem is assumed to be hard in $\mathbb{G}_1$. Therefore, for the adversary, it is computationally infeasible to win the game and generate an incorrect proof of data possession which can pass the verification.

Moreover, the simulation of the tag generation oracle $\mathcal{O}_{TG}$ is perfect. The simulation of the data operation performance oracle $\mathcal{O}_{DOP}$ is almost perfect except when the challenger aborts. This happens the data operation was not correclty performed. As previously, we can prove that if the adversary can pass the updating proof, then a solution to the DL problem is found. Following the above analysis and according to Eq. 1, if the adversary generates an incorrect updating proof which can pass the verification, then a solution of the DL problem can be found with probability equal to $1 - \frac{1}{p}$, which contradicts the fact that the DL problem is assumed to be hard in $\mathbb{G}_1$. Therefore, for the adversary, it is computationally infeasible to generate an incorrect updating proof which can pass the verification. The proof is completed.

Privacy against the TPA. The probability $Pr[b' = b]$ must be equal to $\frac{1}{2}$ since the verification metadata $T_{m_{b,i}}$, for $i = 1, \cdots, n$, and the proof ν^* are independent of the bit b. We now prove that the verification metadata and the proof of data possession given to the adversary are correctly distributed. The value $T_{m_{b,i}}$ is equal to $(\prod_{j=1}^{s} h_j^{m_{b,i,j}})^{-sk} = (\prod_{j=1}^{s} h_j^{m_{b,i,j}})^{-a}$. Since $sk = a$ is kept secret from $\mathcal{A}$, the above simulation is perfect. For a block file m_b, there exists $v_{b,i}$, for $(i, v_{b,i}) \in chal_b$, such that $b_{b,j} = \sum_{(i,v_{b,i}) \in chal_b} m_{b,i,j} \cdot v_{b,i} + r_{b,j}$. In addition, $R_{b,1}, \cdots, R_{b,s}, B_{b,1}, \cdots, B_{b,s}$ are statically indistinguishable with the actual outputs corresponding to m_0 or m_1. Thus, the answers given to the adversary are correctly distributed. The proof is completed.