# Publicly Verifiable Secure Cloud Storage for Dynamic Data Using Secure Network Coding

Binanda Sengupta
Indian Statistical Institute
Kolkata, India
binanda_r@isical.ac.in

Sushmita Ruj
Indian Statistical Institute
Kolkata, India
sush@isical.ac.in

## ABSTRACT

Cloud service providers offer storage outsourcing facility to their clients. In a secure cloud storage (SCS) protocol, the integrity of the client's data is maintained. In this work, we construct a publicly verifiable secure cloud storage protocol based on a secure network coding (SNC) protocol where the client can update the outsourced data as needed. To the best of our knowledge, our scheme is the first SNC-based SCS protocol for dynamic data that is secure in the standard model and provides privacy-preserving audits in a publicly verifiable setting. Furthermore, we discuss, in details, about the (im)possibility of providing a general construction of an efficient SCS protocol for dynamic data (DSCS protocol) from an arbitrary SNC protocol. In addition, we modify an existing DSCS scheme (DPDP I) in order to support privacy-preserving audits. We also compare our DSCS protocol with other SCS schemes (including the modified DPDP I scheme). Finally, we figure out some limitations of an SCS scheme constructed using an SNC protocol.

## Keywords

Cloud Storage, Provable Data Possession, Dynamic Data, Network Coding

## 1. INTRODUCTION

Cloud computing has emerged as a recent technology enabling a device with restricted resources to delegate heavy tasks that the device cannot perform by itself to a powerful cloud server. The services a cloud server offers include huge amount of computation, storage outsourcing and many more. A smart phone, for example, having a low-performance processor or a limited amount of storage capacity, cannot accomplish a heavy computation on its own or cannot store a large amount of data (say, in the order of terabytes) in its own storage. A client (cloud user) can delegate her computation or storage to the cloud server. Now, she can just download the result of the computation, or she can read (or update) only the required portion of the uploaded data.

For storage outsourcing, the cloud server stores a massive volume of data on behalf of its clients. However, a malicious cloud server can delete the client's data in order to save some space. Thus, the client (data owner) has to have a mechanism to check the integrity of her data outsourced to the server. Secure cloud storage (SCS) protocols (two-party protocols between the client and the server) provide a guarantee that the client's data are stored untampered in the server. Based on the nature of the data to be outsourced, secure cloud storage protocols are classified as: SCS protocols for *static* data (SSCS) and SCS protocols for *dynamic* data (DSCS). For static data, the client cannot change her data once they are uploaded to the server (suitable mostly for backup or archival data). Dynamic data are more generic in that the client can modify her data after the initial outsourcing. Some SSCS protocols include [3, 23, 36]; and some DSCS protocols include [18, 39, 10, 37]. In SCS protocols, the client can audit her data stored in the server without accessing the whole data file, and still, be able to detect an unwanted modification of the data done by a malicious server. The SCS protocols are *publicly verifiable* if the audits can be performed by any third party auditor (TPA) with the knowledge of public parameters only; they are *privately verifiable* if the secret information of the client is needed to perform audits. In *privacy-preserving* audits (for publicly verifiable SCS protocols only), the TPA cannot gain the knowledge of any portion of the data file.

Network coding technique [2, 25] serves as an alternative to the conventional store-and-forward routing technique used in a communication network. In a network coding (NC) protocol, every intermediate node (all nodes except the source and target nodes) in the network combines the incoming packets to output another packet. The network coding protocols enjoy much improved throughput, efficiency and scalability compared to simply relaying an incoming packet as it is. However, these protocols are susceptible to *pollution attacks* caused by a malicious intermediate node that injects invalid packets in the network. These invalid packets produce more such packets downstream. In the worst case, the target node cannot decode the original file sent to it via the network. Secure network coding (SNC) protocols provide countermeasures to resolve this issue using some cryptographic primitives. In an SNC protocol, the source node authenticates each of the packets to be transmitted through the network. For the authentication of the packets, a small tag is attached to each packet. These tags

*ASIA CCS '16, May 30-June 03, 2016, Xi'an, China*
© 2016 ACM. ISBN 978-1-4503-4233-9/16/05 . . . $15.00
DOI: http://dx.doi.org/10.1145/2897845.2897915

are generated using homomorphic message authentication codes (MACs) [1] or homomorphic signatures [13, 7, 19, 11]. Every intermediate node can combine the incoming packets to output another packet along with its authentication tag.

In a recent work, Chen et al. [14] explore the relation between a secure cloud storage (SSCS) protocol for static data and a secure network coding (SNC) protocol. They show that, given an SNC protocol, one can construct an SSCS protocol using the SNC protocol. However, for static data, the client (data owner) cannot perform any update (insertion, deletion or modification) efficiently on her data after she uploads them to the cloud server. This constraint makes an SSCS protocol insufficient in many cloud applications where a client needs to update her data frequently. Obviously, a naive way to update data in this scenario is to download the whole data file, perform the required updates and upload the file to the server again; but this procedure is highly inefficient as it requires huge amount of bandwidth for every update. Thus, further investigations are needed towards an efficient (and more generic) construction of a secure cloud storage (DSCS) protocol for dynamic data using an SNC protocol.

**Our Contribution** Following the work of Chen et al. [14], we provide a construction of a secure cloud storage (DSCS) protocol for dynamic data from a secure network coding (SNC) protocol. Unlike the construction of Chen et al., the client, in our scheme, can efficiently perform updates (insertion, deletion and modification) on her data outsourced to the cloud server. Our contributions in this paper are summarized as follows.

- We investigate whether we can provide a general construction of a DSCS protocol using any SNC protocol. We discuss about the challenges for such a general construction in details, and we identify some properties an SNC protocol must have such that an efficient DSCS protocol can be constructed from it. Based on these properties, we observe that, no *efficient* DSCS protocols can be constructed using *some* SNC protocols.

- We provide a construction of a DSCS protocol from an SNC protocol proposed by Catalano et al. [11]. Our construction is secure in the standard model and offers public verifiability. Moreover, since the audits are privacy-preserving in our scheme, a third party auditor cannot gain knowledge of the content of the data file.

- Erway et al. [18] propose an efficient dynamic provable data possession scheme (DPDP I). However, the audits in this scheme are not privacy-preserving. We modify this DPDP I scheme to make its audits privacy-preserving.

- We analyze the efficiency of our DSCS protocol and compare it with other existing secure cloud storage protocols. We discuss about some limitations of an SNC-based SCS protocol (for static or dynamic data).

The rest of the paper is organized as follows. In Section 2, we discuss about the notations used in this paper, and we describe the secure network coding and the secure cloud storage protocols briefly. Section 3 begins with a detailed discussion on the general construction of a DSCS protocol using an SNC protocol. Then, we describe our DSCS construction along with its security analysis and the probabilistic guarantees it offers. In Section 4, we modify the DPDP I scheme [18] to support privacy-preserving audits with proper security analysis of this modified DPDP I scheme. In Section 5, we analyze the efficiency of our DSCS scheme and compare its performance with the existing secure cloud storage schemes. In the concluding Section 6, we summarize the work done in this paper.

## 2. PRELIMINARIES AND BACKGROUND

### 2.1 Notation

We take $\lambda$ to be the security parameter. An algorithm $\mathcal{A}(1^\lambda)$ is a probabilistic polynomial-time algorithm when its running time is polynomial in $\lambda$ and its output $y$ is a random variable which depends on the internal coin tosses of $\mathcal{A}$. An element $a$ chosen uniformly at random from a set $S$ is denoted as $a \xleftarrow{R} S$. A function $f : \mathbb{N} \rightarrow \mathbb{R}$ is called negligible in $\lambda$ if for all positive integers $c$ and for all sufficiently large $\lambda$, we have $f(\lambda) < \frac{1}{\lambda^c}$. In general, $\mathbb{F}$ is used to denote a finite field. The multiplication of a vector v by a scalar $s$ is denoted by $s \cdot \mathrm{v}$. The terms *packet* and *vector* are used interchangeably in this work.

### 2.2 Secure Network Coding

Ahlswede et al. [2] introduce network coding as a replacement of the conventional store-and-forward routing for networks. In network coding, intermediate nodes (or routers) encode the received packets to output another packet which increases the throughput of the network (optimal in case of multicasting). Linear network coding was proposed by Li et al. [25]. Here, the file $F$ to be transmitted is divided into several (say, $m$) packets (or vectors) $\mathrm{v}_1, \mathrm{v}_2, \ldots, \mathrm{v}_m$ each consisting of $n$ components (or blocks), and each of these components is an element of a finite field $\mathbb{F}$. In other words, each $\mathrm{v}_i \in \mathbb{F}^n$ for $i \in [1, m]$. Then, the sender (or source) node augments each vector to form another vector $\mathrm{u}_i = [\mathrm{v}_i \ \mathrm{e}_i] \in \mathbb{F}^{n+m}$ for $i \in [1, m]$, where $\mathrm{e}_i$ is the $m$-dimensional unit vector containing 1 in $i$-th position and 0 in others. Finally, the sender transmits these augmented vectors to the network.

Let $V \subset \mathbb{F}^{n+m}$ be the linear subspace spanned by the augmented vectors $\mathrm{u}_1, \mathrm{u}_2, \ldots, \mathrm{u}_m$. A random file identifier `fid` is associated with the file $F$ (or $V$). In random (linear) network coding [22, 21], an intermediate node in the network, upon receiving $l$ packets $\mathrm{y}_1, \mathrm{y}_2, \ldots, \mathrm{y}_l \in \mathbb{F}^{n+m}$, chooses $l$ coefficients $\nu_1, \nu_2, \ldots, \nu_l \xleftarrow{R} \mathbb{F}$ and outputs another packet $\mathrm{w} \in \mathbb{F}^{n+m}$ such that $\mathrm{w} = \sum_{i=1}^{l} \nu_i \cdot \mathrm{y}_i$ (here, summation refers to vector additions). Thus, the output packet of each intermediate node is of the form

$$\mathrm{w} = [w_1, w_2, \ldots, w_n, c_1, c_2, \ldots, c_m] \in V$$

for some $c_1, c_2, \ldots, c_m \in \mathbb{F}$, where $w_j = \sum_{i=1}^{m} c_i v_{ij}$ for each $j \in [1, n]$. When the receiver (or target) node accumulates $m$ linearly independent vectors (or packets), it solves a system of linear equations to obtain the file destined to it.

In a secure network coding (SNC) protocol, an authentication information (or tag) is attached to each packet in order to prevent pollution attacks. The authentication tags are computed using homomorphic message authentication codes (MACs) [1] or homomorphic signatures [13, 7, 19, 5, 11]. In an SNC protocol based on homomorphic signatures, every node in the network can verify the authenticity of each of its incoming vectors. On the other hand, in case of SNC protocols based on homomorphic MACs, it requires the knowledge of the secret key to verify an incoming vector. We define a secure network coding (SNC) protocol below.

DEFINITION 2.1. *A secure network coding (SNC) protocol consists of the following algorithms.*

- SNC.KeyGen($1^\lambda, m, n$): *This algorithm generates a secret key-public key pair $K = (sk, pk)$ for the sender.*
- SNC.TagGen($V, sk, m, n, \texttt{fid}$): *On input a linear subspace $V \subset \mathbb{F}^{n+m}$, the secret key $sk$ and a random file identifier* fid *associated with $V$, the sender runs this algorithm to produce the authentication tag $t$ for $V$.*
- SNC.Combine($\{\mathrm{y}_i, t_i, \nu_i\}_{1\leqslant i\leqslant l}, pk, m, n, \texttt{fid}$): *Given $l$ incoming packets $\mathrm{y}_1, \mathrm{y}_2, \ldots, \mathrm{y}_l \in \mathbb{F}^{n+m}$ and their corresponding tags $t_1, t_2, \ldots, t_l$ for a file associated with* fid*, an intermediate node chooses $l$ random coefficients $\nu_1, \nu_2, \ldots, \nu_l \xleftarrow{R} \mathbb{F}$ and runs this algorithm. The algorithm outputs another packet $\mathrm{w} \in \mathbb{F}^{n+m}$ and its authentication tag $t$ such that $\mathrm{w} = \sum_{i=1}^{l} \nu_i \cdot \mathrm{y}_i$.*
- SNC.Verify($\mathrm{w}, t, K, m, n, \texttt{fid}$): *An intermediate node or the receiver node, on input a packet* w *and its tag $t$ for a file associated with* fid*, executes this algorithm which in turn returns 1 if $t$ is authentic for the packet* w*; returns 0, otherwise.*

In some schemes, the algorithm SNC.Verify requires only the public key $pk$ [7, 19, 11]. The knowledge of the secret key $sk$ is necessary to verify the incoming packets in other schemes [1].

### *Security of an SNC Protocol.*

The security of an SNC protocol based on a homomorphic signature scheme is defined by the security game between a challenger and a probabilistic polynomial-time adversary $\mathcal{A}$ as stated below [11].

- **Setup** The adversary $\mathcal{A}$ provides the values $m$ and $n$ of its choice to the challenger. The challenger runs SNC.KeyGen($1^\lambda, m, n$) to output $K = (sk, pk)$ and returns $pk$ to $\mathcal{A}$.
- **Queries** The adversary $\mathcal{A}$ specifies a sequence (adaptively chosen) of vector spaces $V_i \subset \mathbb{F}^{n+m}$ by respective augmented basis vectors $\{\mathrm{u}_{i1}, \mathrm{u}_{i2}, \ldots, \mathrm{u}_{im}\}$ and asks the challenger to authenticate the vector spaces. For each $i$, the challenger chooses a random file identifier $\texttt{fid}_i$ from a predefined space, generates an authentication tag $t_i$ by running SNC.TagGen($V_i, sk, m, n, \texttt{fid}_i$) and gives $t_i$ to $\mathcal{A}$.
- **Forgery** The adversary $\mathcal{A}$ outputs $(\texttt{fid}^*, \mathrm{w}^*, t^*)$.

Let $\mathrm{w}^* = [w_1^*, w_2^*, \ldots, w_{n+m}^*] \in \mathbb{F}^{n+m}$. The adversary $\mathcal{A}$ wins the security game if $[w_{n+1}^*, w_{n+2}^*, \ldots, w_{n+m}^*]$ is not the all-zero vector, SNC.Verify($\mathrm{w}^*, t^*, pk, m, n, \texttt{fid}^*$) outputs 1 and one of the following conditions is satisfied:
1. $\texttt{fid}^* \neq \texttt{fid}_i$ for all $i$ (*type-1 forgery*)
2. $\texttt{fid}^* = \texttt{fid}_i$ for some $i$, but $\mathrm{w}^* \notin V_i$ (*type-2 forgery*).

For a secure network coding (SNC) protocol, the probability that $\mathcal{A}$ wins the security game is negligible in the security parameter $\lambda$.

We note that the security game for an SNC protocol based on homomorphic MACs is exactly the same as the game described above, except that the algorithm SNC.KeyGen now produces a secret key only (unknown to the adversary $\mathcal{A}$) and the verification algorithm SNC.Verify requires the knowledge of this secret key.

## 2.3 Secure Cloud Storage

In the age of cloud computing, clients may want to outsource their huge amount of data to the cloud storage server. As the cloud service provider (possibly malicious) might discard old data to save some space, the clients need to be convinced that the outsourced data are stored untampered by the cloud server. A naive approach to ensure data integrity is that a client downloads the whole data from the server and verifies them individually segment by segment. However, this process is inefficient in terms of communication bandwidth required.

### *Building Blocks: PDP and POR.*

Researchers come up with *proofs of storage* in order to resolve the issue mentioned above. Ateniese et al. [3] introduce the concept of *provable data possession* (PDP) where the client computes an authenticator (for example, MAC) for each segment of her data (or file), and uploads the file along with the authenticators. During an audit protocol, the client samples a predefined number of random segment-indices (challenge) and sends them to the server. We denote the cardinality of the challenge set by $l$ which is typically taken to be $O(\lambda)$. The server does some computations (depending upon the challenge) over the stored data, and sends a proof (response) to the client who verifies the integrity of her data based on this proof. This scheme also introduces the notion of public verifiability[1] where the client (data owner) can delegate the auditing task to a third party auditor (TPA). The TPA with the knowledge of the public key performs an audit. For privately verifiable schemes, only the client having knowledge of the secret key can verify the proof sent by the server. This is illustrated in Figure 1. Other schemes achieving PDP include [4, 18, 39, 38].

The first paper introducing *proofs of retrievability* (POR) for static data is by Juels and Kaliski [23] (a similar idea is given for sublinear authenticators by Naor and Rothblum [30]). According to Shacham and Waters [36], the underlying idea of a POR scheme is to encode the original file with an erasure code [26, 33], authenticate the segments of the encoded file, and then upload them on the storage server. With this technique, the server has to delete or modify a considerable number of segments to actually delete or modify a data segment. This ensures that all segments of the file are retrievable from the responses of the server which passes an audit with some non-negligible probability. Following the work by Juels and Kaliski, several POR schemes have been proposed for static or dynamic data [9, 16, 10, 37, 12]. For a detailed list of POR schemes, we refer to [35].

As we deal with a single cloud server in this work, we

[1]The term "public verifiability" discussed in this paper denotes (only) whether a third party auditor having the knowledge of the public parameters can perform an audit on behalf of the client (data owner). We mention that this notion implicitly assumes that the client is honest. However, a malicious client can publish incorrect public parameters in order to get an honest server framed by a third party auditor [24].

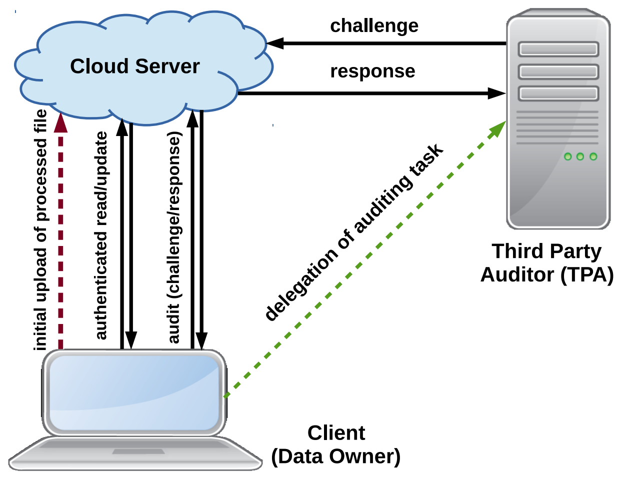
Cloud Server
challenge
response
Third Party
Auditor (TPA)
initial upload of processed file
authenticated read/update
audit (challenge/response)
delegation of auditing task
Client
(Data Owner)

**Figure 1: The architecture of a secure cloud storage protocol. The client processes the file and uploads it to the cloud server. For *static* data, she can read the outsourced data; for *dynamic* data, she can update her data as well. If the scheme is privately verifiable, the client having the secret key can perform audits on the data (through challenge and response). In a publicly verifiable protocol, she can delegate the auditing task to a TPA who performs audits on behalf of the client.**

only mention some secure cloud storage protocols in a distributed setting. Some of them include the works by Curtmola et al. [15] (using replication of data) and Bowers et al. [8] (using error-correcting codes and erasure codes). On the other hand, existing secure cloud storage schemes have found applications in other areas as well [28, 34].

We define an SCS protocol for static data (SSCS) below. We defer the definition of an SCS protocol for dynamic data (DSCS) to Section 3.1. In general, the term *verifier* is used to denote an auditor for a secure cloud storage. The client (for a privately verifiable protocol) or a third party auditor (for a publicly verifiable protocol) can act as the verifier.

Definition 2.2. *A secure cloud storage protocol for static data (SSCS) consists of the following algorithms.*

- SSCS.KeyGen($1^\lambda$): *This algorithm generates a secret key-public key pair $K = (sk, pk)$ for the client.*
- SSCS.Outsource($F, K, \texttt{fid}$): *Given a data file $F$ associated with a random file identifier* $\texttt{fid}$*, the client processes $F$ to form another file $F'$ (including authentication information computed using $sk$) and uploads $F'$ to the server.*
- SSCS.Challenge($pk, l, \texttt{fid}$): *During an audit, the verifier sends a random challenge set $Q$ of cardinality $l = O(\lambda)$ to the server.*
- SSCS.Prove($Q, pk, F', \texttt{fid}$): *Upon receiving the challenge set $Q$, the server computes a proof of storage $T$ corresponding to the challenge set $Q$ and sends $T$ to the verifier.*
- SSCS.Verify($Q, T, K, \texttt{fid}$): *The verifier checks whether $T$ is a valid proof of storage corresponding to the challenge set $Q$. The verifier outputs 1 if the proof passes the verification; she outputs 0, otherwise.*

An SSCS protocol is publicly verifiable if the algorithm SSCS.Verify described above involves only the public key $pk$. The algorithm SSCS.Verify in a privately verifiable SSCS protocol requires the knowledge of the secret key $sk$.

*Security of an SSCS Protocol.*

An SSCS protocol must satisfy the following properties [36].

1. **Authenticity** The authenticity of storage requires that the cloud server cannot forge a valid proof of storage $T$ (corresponding to the challenge set $Q$) without storing the challenged segments and their respective authentication information untampered, except with a probability negligible in $\lambda$.
2. **Extractability** The extractability (or retrievability) of data requires that, given a probabilistic polynomial-time adversary $\mathcal{A}$ that can respond correctly to a challenge $Q$ with some non-negligible probability, there exists a polynomial-time extractor algorithm $\mathcal{E}$ that can extract (at least) the challenged segments (except with negligible probability) by challenging $\mathcal{A}$ for a polynomial (in $\lambda$) number of times and verifying the responses sent by $\mathcal{A}$. The algorithm $\mathcal{E}$ has a non-black-box access to $\mathcal{A}$. Thus, $\mathcal{E}$ can rewind $\mathcal{A}$, if required.

The SSCS protocols based on PDP guarantee the extraction of *almost all* the segments of the file $F$. On the other hand, the SSCS protocols based on POR ensure the extraction of *all* the segments of $F$ with the help of erasure codes.

## 2.4 General Construction of an SSCS Protocol from an SNC Protocol

Chen et al. [14] propose a generic construction of a secure cloud storage protocol for static data from a secure network coding protocol. They consider the data file $F$ to be stored in the server to be a collection of $m$ vectors (or packets) each of which consists of $n$ blocks. The underlying idea is to store these vectors (without augmenting them with unit vectors) in the server. During an audit, the client sends an $l$-element subset of the set of indices $\{1, 2, \ldots, m\}$ to the server. The server augments those vectors with the corresponding unit vectors, combines them linearly in an authenticated fashion and sends the output vector along with its tag to the client. Finally, the client verifies the authenticity of the received tag against the received vector. Thus, the server acts as an intermediate node, and the client acts as both the sender and the receiver (or the next intermediate router). We briefly discuss the algorithms involved in the general construction below.

- SSCS.KeyGen($1^\lambda, m, n$): Initially, the client executes SNC.KeyGen($1^\lambda, m, n$) to generate a secret key-public key pair $K = (sk, pk)$.
- SSCS.Outsource($F, K, m, n, \texttt{fid}$): The file $F$ associated with a random file identifier $\texttt{fid}$ consists of $m$ vectors each of them having $n$ blocks. We assume that each of these blocks is an element of $\mathbb{F}$. Then, for each $1 \leqslant i \leqslant m$, the $i$-th vector $\mathrm{v}_i$ is of the form $[v_{i1}, \ldots, v_{in}] \in \mathbb{F}^n$. For each vector $\mathrm{v}_i$, the client forms

$u_i = [v_i \; e_i] \in \mathbb{F}^{n+m}$ by augmenting the vector $v_i$ with the unit coefficient vector $e_i$. Let $V \subset \mathbb{F}^{n+m}$ be the linear subspace spanned by $u_1, u_2, \ldots, u_m$. The client runs SNC.TagGen($V, sk, m, n, \texttt{fid}$) to produce an authentication tag $t_i$ for the $i$-th vector $u_i$ for each $1 \leqslant i \leqslant m$. Finally, the client uploads the file $F' = \{(v_i, t_i)\}_{1 \leqslant i \leqslant m}$ to the server.

- SSCS.Challenge($pk, l, m, n, \texttt{fid}$): During an audit, the verifier selects $I$, a random $l$-element subset of $[1, m]$. Then, she generates a challenge set $Q = \{(i, \nu_i)\}_{i \in I}$, where each $\nu_i \xleftarrow{R} \mathbb{F}$. The verifier sends the challenge set $Q$ to the server.

- SSCS.Prove($Q, pk, F', m, n, \texttt{fid}$): Upon receiving the challenge set $Q = \{(i, \nu_i)\}_{i \in I}$ for the file identifier fid, the cloud server, for each $i \in I$, forms $u_i = [v_i \; e_i] \in \mathbb{F}^{n+m}$ by augmenting the vector $v_i$ with the unit coefficient vector $e_i$. Then, the cloud server runs SNC.Combine($\{u_i, t_i, \nu_i\}_{i \in I}, pk, m, n, \texttt{fid}$) to produce another vector $w \in \mathbb{F}^{n+m}$ (along with its authentication tag $t$) such that $w = \sum_{i=1}^{l} \nu_i \cdot u_i$. Let $y \in \mathbb{F}^n$ be the first $n$ entries of w. The server sends $T = (y, t)$ to the verifier as a proof of storage corresponding to the challenge set $Q$.

- SSCS.Verify($Q, T, K, m, n, \texttt{fid}$): The verifier uses $Q = \{(i, \nu_i)\}_{i \in I}$ and $T = (y, t)$ to reconstruct the vector $w \in \mathbb{F}^{n+m}$, where the first $n$ entries of w are the same as those of y and the $(n+i)$-th entry is $\nu_i$ if $i \in I$ (0 if $i \notin I$). The verifier runs SNC.Verify($w, t, K, m, n, \texttt{fid}$) and returns the output of the algorithm SNC.Verify.

## 3. CONSTRUCTION OF AN SCS PROTOCOL FOR DYNAMIC DATA USING AN SNC PROTOCOL

In Section 2.4, we have discussed a general construction of an SSCS protocol from an SNC protocol proposed by Chen et al. [14]. In a secure network coding protocol, the number of packets (or vectors) in the file to be transmitted through the network is fixed. This is because the length of the coefficient vectors used to augment the original vectors has to be determined a priori. That is why, such a construction is suitable for static data in general. On the other hand, in a secure cloud storage protocol for dynamic data, clients can modify their data after they upload them to the cloud server initially. In this section, we discuss whether we can provide a general framework for constructing an efficient and secure cloud storage protocol for dynamic data (DSCS) from an SNC protocol.

### 3.1 On the General Construction of an Efficient DSCS Protocol from an SNC Protocol

In a secure network coding (SNC) protocol, a tag is associated with each packet such that the integrity of a packet can be verified using its tag. The SNC protocols found in the literature use homomorphic MACs [1] or homomorphic signatures [13, 7, 19, 5, 11]. Following are the challenges in constructing an *efficient* DSCS protocol from these existing SNC protocols. We exclude, in our discussion, the work of Attrapadung and Libert [5] as their scheme is not efficient due to its reliance on (inefficient) composite-order bilinear groups.

1. **The DSCS protocol must handle the varying values of $m$ appropriately**. In the network coding protocols mentioned above, the sender divides the file in $m$ packets and augments them with unit coefficient vectors before sending them into the network. The length of these coefficient vectors is $m$ which remains constant during transmission. In a secure cloud storage for dynamic data, the number of vectors may vary (for insertion and deletion). If we follow a similar general construction for a DSCS protocol as discussed in Section 2.4, we observe that the cloud server does not need to store the coefficient vectors. However, during an audit, the verifier selects a random $l$-element subset $I$ of $[1, m]$ and the server augments the vectors with unit coefficient vectors of dimension $m$ before generating the proof. Therefore, the verifier and the server need to keep an updated value of $m$.

   This issue can be resolved in a trivial way. The client includes the value of $m$ in her public key and updates its value for each authenticated insertion or deletion. Thus, its latest value is known to the verifier and the server. We assume that, for consistency, the client (data owner) does not update her data during an audit.

2. **The index of a vector should not be embedded in its authentication tag**. In an SNC protocol, the file to be transmitted is divided into $m$ packets $v_1, v_2, \ldots, v_m$, where each $v_i \in \mathbb{F}^n$ for $i \in [1, m]$ ([19] replaces $\mathbb{F}$ by $\mathbb{Z}$). The sender augments each vector to form another vector $u_i = [v_i \; e_i] \in \mathbb{F}^{n+m}$ for $i \in [1, m]$, where $e_i$ is the $m$-dimensional unit vector containing 1 in $i$-th position and 0 in others. Let $V \subset \mathbb{F}^{n+m}$ be the linear subspace spanned by these augmented basis vectors $u_1, u_2, \ldots, u_m$. The sender authenticates the subspace $V$ by authenticating these augmented vectors before transmitting them to the network [13, 1, 7, 19, 11]. In a scheme based on homomorphic MACs [1], the sender generates a MAC for the $i$-th basis vector $u_i$ and the index $i$ serves as an input to the MAC algorithm (for example, $i$ is an input to the pseudorandom function in [1]). On the other hand, for the schemes based on homomorphic signatures, the sender generates a signature $t_i$ on the $i$-th basis vector $u_i$. In some schemes based on homomorphic signatures, the index $i$ is embedded in the signature $t_i$ on the $i$-th augmented vector. For example, $H(\texttt{fid}, i)$ is embedded in $t_i$ [7, 19], where fid is the file identifier and $H$ is a hash function modeled as a random oracle [6].

   These schemes are not suitable for the construction of an *efficient* DSCS protocol due to the following reason. For dynamic data, the client can insert a vector in a specified position or delete an existing vector from a specified location. In both cases, the indices of the subsequent vectors are changed. Therefore, the client has to download all these subsequent vectors and compute fresh authentication tags for them before uploading the new vector-tag pairs to the cloud server. This makes the DSCS protocol inefficient. However, in a few schemes, instead of hashing vector indices as in [7, 19], there is a one-to-one mapping from the set of indices to some group [13, 11], and these group elements are made public. This increases the size of the public key of these schemes. However, an efficient DSCS protocol can be constructed from them. In fact, we construct a DSCS protocol (described in Section 3.3) based on the SNC protocol proposed by Catalano et al. [11]. We note that Chen et al. [14] construct an SCS protocol from the same SNC protocol, but for static data only.

3. **The freshness of data must be guaranteed**. For dynamic data, the client can modify an existing vector. However, a malicious cloud server may discard this change and keep an old copy of the vector. As the old copy of the vector and its corresponding tag are valid, the client has no way to detect if the cloud server is storing the latest copy.

   We ensure the freshness of the client's data, in our DSCS construction, using an *authenticated data structure (rank-based authenticated skip list) on the authentication tags of all the vectors*. In other words, the authenticity of the vectors is maintained by their tags, and the integrity of the tags is in turn maintained by the skip list. The advantage of building the skip list on the tags (over building it on the vectors) is that the tags are much shorter than a vector, and this decreases the size of the proof sent by the server. When a vector is inserted (or modified), its tag is also updated and sent to the server. The server updates the skip list accordingly. For deletion of a vector, the server simply removes the corresponding tag from the skip list. Finally, the server sends to the client a proof of performing the required update properly. We briefly discuss, in Section 3.2, about rank-based authenticated skip lists that we use in our construction described in Section 3.3.

In addition to the requirements mentioned above, it is often desired that a DSCS protocol (an SCS protocol, in general) satisfies the following properties.

4. **Public verifiability** For a publicly verifiable DSCS protocol, the auditing task can be delegated to a third party auditor (TPA). In a secure network coding protocol built on homomorphic MACs, the secret key (for example, the secret keys of the pseudorandom generator and the pseudorandom function in [1]) is needed to verify the authenticity of an incoming packet. This property restricts the secure cloud storage protocol built on such an SNC protocol to be privately verifiable only.

5. **Privacy-preserving audits** In privacy-preserving audits (for a publicly verifiable DSCS protocol), the third party auditor (TPA) cannot gain the knowledge of the challenged vectors.

Definition 3.1. *A secure cloud storage protocol for dynamic data (DSCS) consists of the following algorithms.*

- DSCS.KeyGen($1^\lambda$): *This algorithm generates a secret key-public key pair $K = (sk, pk)$ for the client.*
- DSCS.Outsource($F, K, m,$ `fid`): *The client divides the file $F$ associated with the file identifier* `fid` *into $m$ segments and computes authentication tags for these segments using her secret key $sk$. Then, she constructs an authenticated data structure $M$ on the authentication tags (for checking freshness of the data) and computes some metadata $d_M$ for $M$. Finally, the client uploads the file $F'$ (the file $F$ and the authentication tags) along with $M$ to the cloud storage server and stores $d_M$ (and $m$) at her end.*
- DSCS.InitUpdate($i,$ `updtype`$, d_M, m,$ `fid`): *The value of the variable* `updtype` *indicates whether the update is an insertion after or a modification of or the deletion of the i-th segment. Depending on the value of* `updtype`*, the client modifies $(m, d_M)$ at her end and asks the server to perform the required update on the file associated with* `fid` *(related information specified in* `info`*).*
- DSCS.PerformUpdate($i,$ `updtype`$, F', M,$ `info`$, m,$ `fid`): *The server performs the update on the file associated with* `fid` *and sends the client a proof* $\Pi$.
- DSCS.VerifyUpdate($i,$ `updtype`$, \Pi, m,$ `fid`): *On receiving the proof $\Pi$ for the file associated with* `fid` *from the server, the client checks whether $\Pi$ is a valid proof.*
- DSCS.Challenge($pk, l,$ `fid`): *During an audit, the verifier sends a challenge set $Q$ of cardinality $l = O(\lambda)$ to the server.*
- DSCS.Prove($Q, pk, F', m,$ `fid`): *The server, after receiving the challenge set $Q$, computes a proof of storage $T$ corresponding to $Q$ and a proof of freshness $\Pi$. Then, it sends $(T, \Pi)$ to the verifier.*
- DSCS.Verify($Q, T, K, m,$ `fid`): *The verifier checks if $T$ is a valid proof of storage corresponding to the challenge set $Q$ and $\Pi$ is a valid proof of freshness. The verifier outputs 1 if both the proofs pass the verification; she outputs 0, otherwise.*

A DSCS protocol can be privately verifiable if the algorithm DSCS.Verify described above involves the secret key $sk$ of the client; it is publicly verifiable, otherwise. Therefore, in publicly verifiable DSCS protocols, a third party auditor (TPA) can audit the client's data on behalf of the client who delegates her auditing task to the TPA.

*Security of a DSCS Protocol.*

In addition to the *authenticity* and *extractability* properties (as required by an SSCS protocol), a DSCS protocol must satisfy another property called *freshness* which guarantees that the server is storing an up-to-date version of the file $F$. The detailed discussion on the security of a DSCS protocol is deferred to Section 3.4.

## 3.2 Rank-Based Authenticated Skip Lists

For dynamic data, we need some tool to verify the freshness along with the authenticity of each of the vectors. Several data structures like Merkle hash trees [27], rank-based authenticated skip lists [18] and rank-based RSA trees [31, 18] are found in the literature which serve the purpose. Erway et al. [18] propose rank-based authenticated skip lists based on labeled skip lists [20, 32]. In our construction, we use this data structure since the number of levels in a skip list is logarithmic in $m$ with high probability [32]. We give a brief introduction to the procedures of rank-based authenticated skip lists stored remotely in a server as follows.

- ListInit$(t_1, \dots, t_m)$: Let $\{t_1, \dots, t_m\}$ be an ordered list of $m$ elements on which a rank-based authenticated skip list $M$ is to be built. These elements are kept in the bottom-level nodes of the skip list in an ordered fashion. For each node $z$ of the skip list: $right(z)$ and $down(z)$ are two pointers to the successors of $z$, $rank(z)$ is the number of bottom-level nodes reachable from $z$ (including $z$ if $z$ itself is a bottom-level node), $high(z)$ and $low(z)$ are the indices of the leftmost and rightmost bottom-level nodes reachable from $z$, $f(z)$ is the label associated with the node $z$, and $l(z)$ is the level of $z$ ($l(z) = 0$ for a bottom-level node $z$).

  Initially, all these information (except the label) are computed for each node in the skip list. In addition, the $i$-th bottom-level node $z$ contains $x(z) = t_i, \forall i \in [1, m]$. Finally, for each node $z$, the label $f(z)$ is computed using a *collision-resistant* hash function $h$ as

  $$f(z) = \begin{cases} 0, & \text{if } z \text{ is } \texttt{null} \\ f_1, & \text{if } l(z) = 0 \\ f_2, & \text{if } l(z) > 0 \end{cases}$$

  with $f_1 = h(l(z)||rank(z)||x(z)||f(right(z)))$ and $f_2 = h(l(z)||rank(z)||f(down(z))||f(right(z)))$.

  The skip list along with all the associated information are stored in the server. The client only stores the value of $m$ and the label of the root node $r$ (that is, $f(r)$) as the metadata $d_M$.

- ListAuthRead$(i, m)$: When the client wants to read the $i$-th element $t_i$, the server sends the requested element along with a proof $\Pi(i)$ to the client. Let the verification path of the $i$-th element be a sequence of nodes $z_1, \dots, z_k$, where $z_1$ is the bottom-level node storing the $i$-th element and $z_k = r$ is the root node of the skip list. Then, the proof $\Pi(i)$ is of the form

  $$\Pi(i) = (A(z_1), \dots, A(z_k)),$$

  where $A(z) = (l(z), q(z), d(z), g(z))$. Here, $l(z)$ is the level of the node $z$, $d(z)$ is 0 (or 1) if $down(z)$ (or $right(z)$) points to the previous node of $z$ in the sequence, and $q(z)$ and $g(z)$ are the rank and label (respectively) of the successor node of $z$ that is not present on the verification path.

- ListVerifyRead$(i, d_M, t_i, \Pi(i), m)$: Upon receiving the proof $(t_i, \Pi(i))$ from the server, the client checks if the proof corresponds to the latest metadata $d_M$ stored at her end. The client outputs 1 if the proof matches with the metadata; she outputs 0, otherwise.

  Due to the collision-resistance property of the hash function $h$ that is used to generate the labels of the nodes of the skip list, the server cannot pass the verification without storing the element $t_i$ properly, except with some probability negligible in the security parameter $\lambda$.

- ListInitUpdate$(i, \texttt{updtype}, d_M, t'_i, m)$: An update can be an insertion after or a modification of or the deletion of the $i$-th bottom-level node. The type of the update is stored in a variable `updtype`. The client defines $j = i$ (for an insertion or modification) or $j = i - 1$ (for a deletion). She calls ListAuthRead$(j, m)$ for the existing skip list $M$ and verifies the response sent by the server by calling ListVerifyRead$(j, d_M, t_j, \Pi(j), m)$. If the proof does not match with the metadata $d_M$ (the label of the root node of the existing skip list $M$), she aborts. Otherwise, she updates the value of $m$, computes the new metadata $d'_M$ using the proof and stores it at her end temporarily. Then, the client asks the server to perform the update specifying the location $i$, `updtype` (insertion, deletion or modification) and the new element $t'_i$ (`null` for deletion).

- ListPerformUpdate$(i, \texttt{updtype}, t'_i, M)$: Depending on the value of `updtype`, the server performs the update asked by the client, computes a proof $\Pi$ similar to the one generated during ListAuthRead and sends $\Pi$ to the client.

- ListVerifyUpdate$(i, \texttt{updtype}, t'_i, d'_M, \Pi, m)$: On receiving the proofs from the server, the client verifies the proof $\Pi$ and computes the new metadata $d_{new}$ based on $\Pi$. If $d'_M = d_{new}$ and $\Pi$ is a valid proof, the client sets $d_M = d'_M$, deletes the temporary value $d'_M$ and outputs 1. Otherwise, she changes $m$ to its previous value, deletes the temporary value $d'_M$ and outputs 0.

Due to the properties of a skip list [32], the number of levels in a skip list is logarithmic in $m$ with high probability. For this reason, the size of a proof, the computation time for the server and the verification time for the client are $O(\log m)$ with high probability.

## 3.3 Construction of a DSCS Protocol from an SNC Protocol

In this section, we construct a secure cloud storage protocol for dynamic data (DSCS) from the secure network coding (SNC) protocol proposed by Catalano et al. [11] which is secure in the standard model. This construction exploits a rank-based authenticated skip list (discussed in Section 3.2) to ensure the freshness of the dynamic data. This DSCS protocol consists of the following procedures. Let $h$ be the *collision-resistant* hash function used in the rank-based authenticated skip list we use in our construction. We assume that the file $F$ to be outsourced to the server is a collection of $m$ vectors where each of the vectors consists of $n$ blocks.

- KeyGen$(1^\lambda, m, n)$: The client selects two random safe primes $p, q$ of length $\lambda/2$ bits each and takes $N = pq$. The client chooses another random prime $e$ of length $\lambda+1$ (in bits) and sets the file identifier `fid` to be equal to $e$. She selects $g, g_1, \dots, g_n, h_1, \dots, h_m \xleftarrow{R} \mathbb{Z}_N^*$. The secret key $sk$ is $(p, q)$, and the public key $pk$ consists of $(N, e, g, g_1, \dots, g_n, h_1, \dots, h_m, d_M, m, n)$. Initially, $d_M$ is `null`. Let $K = (sk, pk)$.

- Outsource$(F, K, \texttt{fid})$: As mentioned above, the file $F$ (associated with the identifier `fid`) consists of $m$ vectors each of them having $n$ blocks. We assume that each of these blocks is an element of $\mathbb{F}_e$. Then, for each $1 \leqslant i \leqslant m$, the $i$-th vector $\mathrm{v}_i$ is of the form $[v_{i1}, \dots, v_{in}] \in \mathbb{F}_e^n$. For each vector $\mathrm{v}_i$, the client selects a random element $s_i \xleftarrow{R} \mathbb{F}_e$ and computes $x_i$ such that

  $$x_i^e = g^{s_i}(\prod_{j=1}^{n} g_j^{v_{ij}})h_i \bmod N. \qquad (1)$$

Now, $t_i = (s_i, x_i)$ acts as an authentication tag for the vector $\mathrm{v}_i$. The client constructs a rank-based authenticated skip list $M$ on the authentication tags $\{t_i\}_{1 \leqslant i \leqslant m}$ and computes the metadata $d_M$ (the label of the root node of $M$). Finally, the client updates $d_M$ in the public key and uploads the file $F' = \{(\mathrm{v}_i, t_i)\}_{1 \leqslant i \leqslant m}$ along with $M$ to the cloud server.

- InitUpdate($i, \texttt{updtype}, d_M, pk, \texttt{fid}$): The value of the variable `updtype` indicates whether the update is an insertion after or a modification of or the deletion of the $i$-th vector. The client performs one of the following operations depending on the value of `updtype`.

  1. If `updtype` is insertion, the client selects $h' \xleftarrow{R} \mathbb{Z}_N^*$ and generates the new vector-tag pair $(\mathrm{v}', t')$. She runs ListInitUpdate on $(i, \texttt{updtype}, d_M, t', m)$ and sends $(h', \mathrm{v}')$ to the server.
  2. If `updtype` is modification, the client generates the new vector-tag pair $(\mathrm{v}', t')$. Then, she runs ListInitUpdate$(i, \texttt{updtype}, d_M, t', m)$ and sends $\mathrm{v}'$ to the server.
  3. If `updtype` is deletion, the client runs the procedure ListInitUpdate on $(i, \texttt{updtype}, d_M, t', m)$, where $t'$ is `null`.

  The client stores the value of the new metadata $d'_M$ temporarily at her end.

- PerformUpdate($i, \texttt{updtype}, F', M, h', \mathrm{v}', t', pk, \texttt{fid}$): We assume that, for efficiency, the server keeps a local copy of the ordered list of $h_j$ values for $1 \leqslant j \leqslant m$. Based on the value of `updtype`, the server performs one of the following operations.

  1. If `updtype` is insertion, the server sets $m = m+1$, inserts $h'$ in the $(i+1)$-th position in the list of $h_j$ values (for $1 \leqslant j \leqslant m$) and inserts $\mathrm{v}'$ after the $i$-th vector. The server runs ListPerformUpdate on the input $(i, \texttt{updtype}, t', M)$.
  2. If `updtype` is modification ($h'$ is `null`), the server modifies the $i$-th vector to $\mathrm{v}'$ and runs the procedure ListPerformUpdate on $(i, \texttt{updtype}, t', M)$.
  3. If `updtype` is deletion ($h'$, $\mathrm{v}'$ and $t'$ are `null`), the server sets $m = m - 1$, deletes the particular $h_i$ value from the list of $h_j$ values ($j \in [1, m]$) and runs ListPerformUpdate$(i, \texttt{updtype}, \texttt{null}, M)$.

- VerifyUpdate($i, \texttt{updtype}, t', d'_M, \Pi, pk, \texttt{fid}$): After receiving the proof from the server, the client performs ListVerifyUpdate$(i, \texttt{updtype}, t', d'_M, \Pi, m)$. If the output of ListVerifyUpdate is 1, the client outputs 1 and updates her public key (the latest values of $m, d_M$ and $h_j$ for $j \in [1, m]$) accordingly. Otherwise, the client outputs 0.

- Challenge($pk, l, \texttt{fid}$): During an audit, the verifier selects $I$, a random $l$-element subset of $[1, m]$. Then, she generates a challenge set $Q = \{(i, \nu_i)\}_{i \in I}$, where each $\nu_i \xleftarrow{R} \mathbb{F}_e$. The verifier sends the challenge set $Q$ to the cloud server.

- Prove($Q, pk, F', M, \texttt{fid}$): The cloud server, after receiving the challenge set $Q = \{(i, \nu_i)\}_{i \in I}$, computes $s = \sum_{i \in I} \nu_i s_i \bmod e$ and $s' = (\sum_{i \in I} \nu_i s_i - s)/e$. The server, for each $i \in I$, forms $\mathrm{u}_i = [\mathrm{v}_i \ \mathrm{e}_i] \in \mathbb{F}_e^{n+m}$ by augmenting the vector $\mathrm{v}_i$ with the unit coefficient vector $\mathrm{e}_i$. Then, it computes $\mathrm{w} = \sum_{i \in I} \nu_i \cdot \mathrm{u}_i \bmod e \in \mathbb{F}_e^{n+m}$, $\mathrm{w}' = (\sum_{i \in I} \nu_i \cdot \mathrm{u}_i - \mathrm{w})/e \in \mathbb{F}_e^{n+m}$ and

  $$x = \frac{\prod_{i \in I} x_i^{\nu_i}}{g^{s'} \prod_{j=1}^{n} g_j^{w'_j} \prod_{j=1}^{m} h_j^{w'_{n+j}}} \bmod N. \tag{2}$$

  Let $\mathrm{y} \in \mathbb{F}_e^n$ be the first $n$ entries of w and $t = (s, x)$. The server sends $T = (T_1, T_2)$ as a proof of storage corresponding to the challenge set $Q$, where $T_1 = (\mathrm{y}, t)$ and $T_2 = \{(t_i, \Pi(i))\}_{i \in I}$.

- Verify($Q, T, pk, \texttt{fid}$): Using $Q = \{(i, \nu_i)\}_{i \in I}$ and $T = (\mathrm{y}, t)$ sent by the server, the verifier constructs a vector $\mathrm{w} = [w_1, \ldots, w_n, w_{n+1}, \ldots, w_{n+m}] \in \mathbb{F}_e^{n+m}$, where the first $n$ entries of w are the same as those of y and the $(n+i)$-th entry is $\nu_i$ if $i \in I$ (0 if $i \notin I$). Then, the verifier checks whether

  $$x^e \overset{?}{=} g^s \prod_{j=1}^{n} g_j^{w_j} \prod_{j=1}^{m} h_j^{w_{n+j}} \bmod N. \tag{3}$$

  She also verifies if, for each $i \in I$, $\Pi(i)$ is a valid proof (with respect to $d_M$) for $t_i$. The verifier outputs 1 if the proof passes all the verifications; she outputs 0, otherwise.

The DSCS protocol described above is publicly verifiable, that is, a third party auditor (TPA) having the knowledge of the public key of the client (data owner) can perform an audit. Chen et al. [14] construct a secure cloud storage protocol for *static* data using the same SNC protocol [11]. They show that, in order to make an audit privacy-preserving, the server adds a random linear combination of some random vectors to the computed value of y to form the final response. Each of these random vectors is augmented with the all-zero vector of dimension $m$ ($0^m$), and the client outsources these augmented vectors (along with their tags) to the server initially. Due to the addition of this random component to the resulting vector y, the third party auditor (TPA) cannot gain knowledge of the challenged vectors. It is easy to see that a similar change in the algorithm Prove in our DSCS protocol mentioned above makes this scheme privacy-preserving as well.

## 3.4 Security Analysis

The DSCS protocol described in Section 3.3 offers the guarantee of dynamic provable data possession (DPDP) [18]. We describe the data possession game of DPDP between the challenger (acting as the client) and the adversary (acting as the cloud server) as follows.

- The challenger generates a key pair $(sk, pk)$ and gives $pk$ to the adversary.

- The adversary selects a file $F$ associated with the identifier `fid` to store. The challenger processes the file to form another file $F'$ with the help of $sk$ and returns $F'$ to the adversary. The challenger stores only some metadata to verify the updates to be performed by the adversary later. The adversary chooses a sequence of updates (of its choice) defined by $(\texttt{updtype}_i, \texttt{info}_i)$ for

$1 \leqslant i \leqslant q_1$ ($q_1$ is polynomial in the security parameter $\lambda$) and asks the challenger to initiate the update. For each update, the challenger runs InitUpdate and stores the latest metadata at her end. The adversary sends a proof after executing PerformUpdate. The challenger verifies this proof by running VerifyUpdate and updates her metadata if and only if the proof passes the verification. The adversary is notified about the output of VerifyUpdate for each update.

- Let $F^*$ be the final state of the file after $q_1$ updates. The challenger has the latest metadata for the file $F^*$. Now, she challenges the adversary with a random challenge set $Q$, and the adversary returns a proof to the challenger. The adversary wins the game if the proof passes the verification. The challenger can challenge the adversary $q_2$ (polynomial in $\lambda$) number of times in an attempt to extract (at least) the challenged vectors of $F^*$.

Our scheme satisfies the following three properties required for security.

1. **Authenticity** The authenticity of storage demands that the cloud server cannot produce (or forge) a valid response $T_1 = (\mathrm{y}, t)$ (corresponding to the challenge set $Q = \{(i, \nu_i)\}_{i \in I}$) without storing the challenged vectors and their respective authentication tags appropriately. Since the SNC protocol proposed by Catalano et al. [11] is secure in the standard model and the random challenge set $Q$ (precisely, the coefficients $\nu_i$, for $i \in I$, used in the algorithm Prove for computing w as a linear combination of the augmented vectors) is chosen by the verifier, the DSCS protocol we have constructed provides a guarantee of authenticity (in the standard model) except with a probability negligible in $\lambda$.

2. **Freshness** The freshness of storage requires that the cloud server must store an up-to-date version of the data file outsourced by the client. In our scheme, for each update, the freshness of data is guaranteed using the algorithm VerifyUpdate (by computing $d_{new}$ from $\Pi$ and checking if $d'_M \overset{?}{=} d_{new}$). Moreover, for each challenge $Q$, the freshness of data is guaranteed by checking the validity of the proof $T_2 = \{(t_i, \Pi(i))\}_{i \in I}$ (in the algorithm Verify) for the rank-based authenticated skip list $M$. Thus, given the hash function $h$ (see Section 3.2) used to compute the labels of the nodes in the skip list $M$ is collision-resistant, the DSCS protocol described above ensures the freshness of data.

3. **Extractability** The extractability (or retrievability) of data requires that, given a probabilistic polynomial-time adversary $\mathcal{A}$ that wins the data possession game mentioned above with some non-negligible probability, there must be a polynomial-time extractor algorithm $\mathcal{E}$ that can extract (at least) the challenged vectors (except with negligible probability) by challenging $\mathcal{A}$ for a polynomially (in $\lambda$) many times and verifying the responses sent by $\mathcal{A}$. The algorithm $\mathcal{E}$ has a non-black-box access to $\mathcal{A}$. Thus, $\mathcal{E}$ can rewind $\mathcal{A}$, if required. Given the DSCS protocol satisfies the *authenticity* and *freshness* properties mentioned above, it is not hard to see that a polynomial-time extractor algorithm for such an adversary $\mathcal{A}$ can extract (at least) the challenged vectors (for known linear combinations of these vectors) for the DSCS scheme described above with the help of Gaussian elimination [36].

*Probabilistic Guarantees.*

If the server has corrupted a fraction (say, $\beta$) of vectors in a file, then the server passes an audit with probability $p_{cheat} = (1-\beta)^l$, where $l$ is the cardinality of the challenge set $Q$. The probability $p_{cheat}$ is very small for large values of $l$. Typically, $l$ is taken to be $O(\lambda)$ in order to make the probability $p_{cheat}$ negligible in $\lambda$. Thus, the verifier detects a malicious server corrupting $\beta$-fraction of the file with probability $p_{detect} = 1 - p_{cheat} = 1 - (1-\beta)^l$, and it guarantees the integrity of *almost all* vectors of the file.

## 4. DPDP I: A DYNAMIC PROVABLE DATA POSSESSION SCHEME

Erway et al. [18, 17] propose two efficient and fully dynamic provable data possession schemes: DPDP I (based on rank-based authenticated skip lists) and DPDP II (based on rank-based RSA trees). We consider only the DPDP I scheme here.

### 4.1 Blockless Verification in DPDP I

Let there be a key generation algorithm KeyGen that produces a public key $pk = (N, g)$, where $N = pq$ is a product of two large primes and $g$ is an element of $\mathbb{Z}_N^*$ with large order. Suppose the initial data file consists of $\tilde{m}$ blocks $b_1, b_2, \ldots, b_{\tilde{m}}$. For each block $b$, the client computes a tag $\mathcal{T}(b) = g^b \bmod N$. Now, the client builds a rank-based authenticated skip list $\tilde{M}$ on the tags of the blocks and uploads the data, tags and the skip list to the cloud server. The insertion, deletion and modification operations are performed in a similar fashion as discussed in Section 3.3. There is no secret key involved in the DPDP I scheme. Although Erway et al. do not claim explicitly the public verifiability of the DPDP I scheme, we observe that the scheme can be made publicly verifiable by simply making the metadata $d_{\tilde{M}}$ of the up-to-date skip list and the value $\tilde{m}$ public (see the footnote in Section 2.3).

During an audit, the verifier selects $I$, a random $l$-element subset of $\{1, 2, \ldots, \tilde{m}\}$, and generates a challenge set $Q = \{(i, \nu_i)\}_{i \in I}$, where each $\nu_i$ is a random value. The verifier sends the challenge set $Q$ to the server. The server computes an aggregated block $B = \sum_{i \in I} \nu_i b_i$ and sends $\{\mathcal{T}(b_i)\}_{i \in I}$, $B$ and proofs $\{\Pi(i)\}_{i \in I}$ (see Section 3.2) to the verifier. The verifier computes $\mathcal{T} = \prod_{i \in I} \mathcal{T}(b_i)^{\nu_i}$. Finally, the verifier accepts the proof if and only if the following two conditions hold: $\Pi(i)$ is a valid proof for each $i \in I$ and $\mathcal{T} = g^B \bmod N$.

### 4.2 Modified DPDP I to Make Audits Privacy-Preserving

The secure cloud storage scheme for dynamic data discussed in Section 3.3 offers privacy-preserving audits where a third party auditor (TPA) cannot learn about the actual data while auditing. Let us investigate whether the scheme DPDP I provides this facility.

As in the original scheme [18] (see Section 4.1), the server sends the aggregated block $B = \sum_{i \in I} \nu_i b_i$ to the verifier (or TPA) where $|I| = l$. Now, a TPA can obtain the $b_i$ values by solving a system of linear equations. Therefore, the audits in the original scheme are not privacy-preserving. However, it is not hard to make these audits privacy-preserving. We modify the procedures involved in an audit as follows. As before, the verifier sends the challenge set $Q = \{(i, \nu_i)\}_{i \in I}$ to the server. The server computes an aggregated block $B = \sum_{i \in I} \nu_i b_i$. Now, the server chooses a random value $r$, and it computes $B' = B + r$ and $R = g^r \bmod N$. The server sends $\{\mathcal{T}(b_i)\}_{i \in I}$, $B'$, $R$ and proofs $\{\Pi(i)\}_{i \in I}$ to the verifier. The verifier computes $\mathcal{T} = R \prod_{i \in I} \mathcal{T}(b_i)^{\nu_i}$. Finally, the verifier accepts the proof if and only if the following two conditions hold: $\Pi(i)$ is a valid proof for each $i \in I$ and $\mathcal{T} = g^{B'} \bmod N$.

As discussed in Section 4.1, in order to make the scheme publicly verifiable, the client includes the pair $(d_{\tilde{M}}, \tilde{m})$ in her public key and updates it after every authenticated update on the outsourced data.

### *Security Analysis.*

The modified DPDP I scheme satisfies the authenticity and freshness properties as described in Section 3.4 (this directly follows from the same guarantees provided by the original DPDP I). Given a probabilistic polynomial-time adversary $\mathcal{A}$ that wins the data possession game (see Section 3.4) with some non-negligible probability, there exists a polynomial-time extractor algorithm $\mathcal{E}$ for the original DPDP I which can extract the challenged vectors (except with a negligible probability) by interacting with $\mathcal{A}$. Now, the extractor algorithm $\mathcal{E}'$ for the modified DPDP I challenges the adversary with two different challenge sets $Q = \{(i, \nu_i)\}_{i \in I}$ and $Q' = \{(i, \nu'_i)\}_{i \in I}$ on the same commitment $r$, where each $\nu_i$ (or $\nu'_i$) is a random value. Then, $\mathcal{E}'$ gets two responses of the form $B' = \sum_{i \in I} \nu_i b_i + r$ and $B'' = \sum_{i \in I} \nu'_i b_i + r$, and the extractor now forms another $B''' = \sum_{i \in I} \nu''_i b_i$ where $\nu''_i = \nu_i - \nu'_i$ for each $i \in I$. We note that $B''' = \sum_{i \in I} \nu''_i b_i$ is similar to a response from the adversary in the original DPDP I scheme described in Section 4.1. Thus, $\mathcal{E}'$ can extract (at least) the challenged vectors in a similar fashion as done by $\mathcal{E}$.

### *Privacy-Preserving Audits.*

We observe that the TPA does not have an access to the value of $B$. To get the value of $B$, the TPA has to solve either $B$ from $g^B \bmod N$, or $r$ from $R = g^r \bmod N$, both of which are infeasible for any probabilistic polynomial-time adversary $\mathcal{A}$, except with some negligible probability. Thus, the audits are privacy-preserving in this modified scheme.

## 5. PERFORMANCE ANALYSIS

In this section, we discuss about the efficiency of our DSCS protocol (described in Section 3.3) and compare this scheme with other existing SCS protocols achieving provable data possession guarantees. We also identify some limitations of an SNC-based SCS scheme (for static or dynamic data) compared to the DPDP I scheme (described in Section 4).

## 5.1 Efficiency

The computational cost of the algorithms in our DSCS protocol is dominated by the cost of exponentiations (modulo $N$) required. To generate the value $x$ in an authentication tag for each vector (in the algorithm Outsource), the client has to perform a multi-exponentiation [29] and calculate the $e$-th root of the result (see Eqn. 1 in Section 3.3). The server requires two multi-exponentiations to calculate the value of $x$ (see Eqn. 2 in the algorithm Prove). To verify a proof using the algorithm Verify, the verifier has to perform a multi-exponentiation and a single exponentiation (see Eqn. 3).

As mentioned in Section 3.2, due to the properties of a skip list [32], the size of each proof $\Pi$ (related to the rank-based authenticated skip list), the time required to generate $\Pi$ and the time required to verify $\Pi$ are $O(\log m)$ with high probability.

## 5.2 Comparison among PDP Schemes

As our DSCS protocol provides provable data possession (PDP) guarantees, we compare our scheme with some other PDP schemes found in the literature. The comparison shown in Table 1 is done based on different parameters related to an audit.

Now, we discuss about a few limitations of our DSCS protocol compared to DPDP I (specifically), since both of them are secure in the *standard model*, handle *dynamic* data and offer *public verifiability.* In the DSCS protocol, the audits are privacy-preserving, that is, a third party auditor (TPA) cannot gain knowledge of the data actually stored in the cloud server. Although the original DPDP I scheme does not offer privacy-preserving audits, this scheme can be modified to support the same (see Section 4.2). The issues of our scheme compared to the modified DPDP I scheme are mentioned below.

1. The size of the public key is $O(m + n)$ in our scheme. On the other hand, the size of the public key in the modified DPDP I scheme is constant.

2. The authentication tags in the DSCS protocol are of the form $(s, x)$, where $s \in \mathbb{F}_e$ and $x \in \mathbb{Z}_N^*$. An authentication tag in the modified DPDP I scheme is an element of $\mathbb{Z}_N^*$. Thus, the size of a tag in the DSCS protocol is larger than that in the modified DPDP I scheme by $\lambda + 1$ bits (as $e$ is a $(\lambda + 1)$-bit prime).

3. In our DSCS scheme, the value of $(d_M, m)$ and the $h_i$ values in the public key must be changed for each insertion or deletion (only change in $d_M$ is required for modification), whereas only the value of $(d_{\tilde{M}}, \tilde{m})$ needs to be changed in the modified DPDP I scheme. However, if the server keeps a local copy of the public key (an ordered list containing $h_i$ values for $i \in [1, m]$), then small changes are required at the server side. The server inserts the new $h$ value (sent by the client) in $(i + 1)$-th position in the list (for insertion) or discards the $i$-th $h$ value (for deletion).

Thus, the proposed DSCS scheme suffers from the limitations mentioned above. We note that the existing SSCS protocol [14] based on the same SNC protocol [11] also suffers from the first two of these limitations. However, in this work, we explore if an efficient secure cloud storage protocol can be constructed from a secure network coding protocol. A more efficient (in terms of the size of the public key or the size of an authentication tag) SNC protocol having the properties mentioned in Section 3.1 can lead us to construct a more efficient DSCS protocol in future.

| Secure cloud storage protocols | Type of data | Computation for verifier | Computation for server | Communication complexity | Publicly verifiable | Privacy-preserving audits | Security model |
|---|---|---|---|---|---|---|---|
| PDP [3] | Static | $O(1)$ | $O(1)$ | $O(1)$ | Yes | No | RO$^\dagger$ |
| Scalable PDP [4] | Dynamic$^\ddagger$ | $O(1)$ | $O(1)$ | $O(1)$ | No | No | RO |
| DPDP I [18] | Dynamic | $O(\log \tilde{m})$ | $O(\log \tilde{m})$ | $O(\log \tilde{m})$ | Yes$^\S$ | No | Standard |
| Modified DPDP I (in this work) | Dynamic | $O(\log \tilde{m})$ | $O(\log \tilde{m})$ | $O(\log \tilde{m})$ | Yes | Yes | Standard |
| DPDP II [18] | Dynamic | $O(\log \tilde{m})$ | $O(\tilde{m}^\epsilon \log \tilde{m})^\star$ | $O(\log \tilde{m})$ | Yes$^\S$ | No | Standard |
| Wang et al. [39] | Dynamic | $O(\log \tilde{m})$ | $O(\log \tilde{m})$ | $O(\log \tilde{m})$ | Yes | No | RO |
| Wang et al. [38] | Dynamic | $O(\log \tilde{m})$ | $O(\log \tilde{m})$ | $O(\log \tilde{m})$ | Yes | Yes | RO |
| Chen et al. [14] | Static | $O(1)$ | $O(1)$ | $O(1)$ | Yes | Yes | Standard |
| Our DSCS scheme (in this work) | Dynamic | $O(\log m)$ | $O(\log m)$ | $O(\log m)$ | Yes | Yes | Standard |

**Table 1: Comparison of the secure cloud storage schemes achieving PDP guarantees. For simplicity, we exclude the security parameter $\lambda$ from complexity parameters (for an audit). The value $m$ denotes the number of vectors in our DSCS scheme, and $\tilde{m}$ denotes the number of segments the data file is divided in (such that an authentication tag is associated with each segment). The term $O(\tilde{n})$ is added implicitly to each complexity parameter, where $\tilde{n}$ is the size of each segment. For example, $\tilde{n} = n$ in the last two schemes, where a vector having $n$ blocks is considered to be a segment. For all the schemes, the storage at the verifier side is $O(1)$, and the storage at the server side is $O(|F'|)$ where $F'$ is the outsourced file. If $l$ is the cardinality of the challenge set and the server corrupts $\beta$-fraction of the file, the detection probability $p_{detect} = 1 - (1-\beta)^l$ for all the schemes (except, in DPDP II, $p_{detect} = 1 - (1-\beta)^{\Omega(\log \tilde{m})}$).**

$\dagger$ **RO denotes the random oracle model [6].**
$\ddagger$ **Scalable PDP scheme supports deletion, modification and append only for a predefined number of times, and insertion is not supported in this scheme.**
$\S$ **A small change (making the latest values of $d_{\tilde{M}}$ and $\tilde{m}$ public) is required in the original scheme (see Section 4.1).**
$\star$ **$\epsilon$ is a constant such that $0 < \epsilon < 1$.**

## 6. CONCLUSION

In this work, we have proposed a DSCS protocol based on an SNC protocol. To the best of our knowledge, this is the first SNC-based DSCS protocol that is secure in the standard model, enjoys public verifiability and offers privacy-preserving audits. We have also discussed about some properties an SNC protocol must have such that an efficient DSCS protocol can be constructed using this SNC protocol. We have modified an existing DSCS scheme (DPDP I [18]) to make its audits privacy-preserving. We have analyzed the efficiency of our DSCS construction and compare it with other existing secure cloud storage protocols achieving the guarantees of provable data possession. Finally, we have identified some limitations of an SNC-based secure cloud storage protocol. However, some of these limitations follow from the underlying SNC protocols used. A more efficient SNC protocol can give us a DSCS protocol with a better efficiency.

## 7. REFERENCES

[1] S. Agrawal and D. Boneh. Homomorphic MACs: MAC-based integrity for network coding. In *Applied Cryptography and Network Security - ACNS 2009*, pages 292–305, 2009.

[2] R. Ahlswede, N. Cai, S. R. Li, and R. W. Yeung. Network information flow. *IEEE Transactions on Information Theory*, 46(4):1204–1216, 2000.

[3] G. Ateniese, R. C. Burns, R. Curtmola, J. Herring, L. Kissner, Z. N. J. Peterson, and D. X. Song. Provable data possession at untrusted stores. In *ACM Conference on Computer and Communications Security, CCS 2007*, pages 598–609, 2007.

[4] G. Ateniese, R. D. Pietro, L. V. Mancini, and G. Tsudik. Scalable and efficient provable data possession. In *International Conference on Security and Privacy in Communication Networks, SECURECOMM 2008*, pages 9:1–9:10, 2008.

[5] N. Attrapadung and B. Libert. Homomorphic network coding signatures in the standard model. In *Public Key Cryptography - PKC 2011*, pages 17–34, 2011.

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

[7] D. Boneh, D. M. Freeman, J. Katz, and B. Waters. Signing a linear subspace: Signature schemes for network coding. In *Public Key Cryptography - PKC 2009*, pages 68–87, 2009.

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

[9] K. D. Bowers, A. Juels, and A. Oprea. Proofs of retrievability: Theory and implementation. In *ACM Cloud Computing Security Workshop, CCSW 2009*, pages 43–54, 2009.

[10] D. Cash, A. Küpçü, and D. Wichs. Dynamic proofs of retrievability via oblivious RAM. In *Advances in Cryptology - EUROCRYPT 2013*, pages 279–295,

2013.

[11] D. Catalano, D. Fiore, and B. Warinschi. Efficient network coding signatures in the standard model. In *Public Key Cryptography - PKC 2012*, pages 680–696, 2012.

[12] N. Chandran, B. Kanukurthi, and R. Ostrovsky. Locally updatable and locally decodable codes. In *Theory of Cryptography Conference, TCC 2014*, pages 489–514, 2014.

[13] D. X. Charles, K. Jain, and K. E. Lauter. Signatures for network coding. *International Journal of Information and Coding Theory*, 1(1):3–14, 2009.

[14] F. Chen, T. Xiang, Y. Yang, and S. S. M. Chow. Secure cloud storage meets with secure network coding. In *IEEE Conference on Computer Communications, INFOCOM 2014*, pages 673–681, 2014.

[15] R. Curtmola, O. Khan, R. C. Burns, and G. Ateniese. MR-PDP: multiple-replica provable data possession. In *IEEE International Conference on Distributed Computing Systems - ICDCS 2008*, pages 411–420, 2008.

[16] Y. Dodis, S. P. Vadhan, and D. Wichs. Proofs of retrievability via hardness amplification. In *Theory of Cryptography Conference, TCC 2009*, pages 109–127, 2009.

[17] C. C. Erway, A. Küpçü, C. Papamanthou, and R. Tamassia. Dynamic provable data possession. In *ACM Conference on Computer and Communications Security, CCS 2009*, pages 213–222, 2009.

[18] C. C. Erway, A. Küpçü, C. Papamanthou, and R. Tamassia. Dynamic provable data possession. *ACM Transactions on Information and System Security*, 17(4):15, 2015.

[19] R. Gennaro, J. Katz, H. Krawczyk, and T. Rabin. Secure network coding over the integers. In *Public Key Cryptography - PKC 2010*, pages 142–160, 2010.

[20] M. T. Goodrich, R. Tamassia, and A. Schwerin. Implementation of an authenticated dictionary with skip lists and commutative hashing. In *DARPA Information Survivability Conference and Exposition (DISCEX) II*, pages 68–82, 2001.

[21] T. Ho, R. Koetter, M. Médard, D. R. Karger, and M. Effros. The benefits of coding over routing in a randomized setting. In *IEEE International Symposium on Information Theory - ISIT 2003*, page 442, 2003.

[22] T. Ho, M. Médard, R. Koetter, D. R. Karger, M. Effros, J. Shi, and B. Leong. A random linear network coding approach to multicast. *IEEE Transactions on Information Theory*, 52(10):4413–4430, 2006.

[23] A. Juels and B. S. Kaliski, Jr. PORs: Proofs of retrievability for large files. In *ACM Conference on Computer and Communications Security, CCS 2007*, pages 584–597, 2007.

[24] A. Küpçü. Official arbitration with secure cloud storage application. *The Computer Journal*, 58(4):831–852, 2015.

[25] S. R. Li, R. W. Yeung, and N. Cai. Linear network coding. *IEEE Transactions on Information Theory*, 49(2):371–381, 2003.

[26] F. J. MacWilliams and N. J. A. Sloane. *The Theory of Error-Correcting Codes*. North-Holland Publishing Company, 1977.

[27] R. C. Merkle. A digital signature based on a conventional encryption function. In *Advances in Cryptology - CRYPTO 1987*, pages 369–378, 1987.

[28] A. Miller, A. Juels, E. Shi, B. Parno, and J. Katz. Permacoin: Repurposing Bitcoin work for data preservation. In *IEEE Symposium on Security and Privacy - S&P 2014*, pages 475–490, 2014.

[29] B. Möller. Algorithms for multi-exponentiation. In *Selected Areas in Cryptography - SAC 2001*, pages 165–180, 2001.

[30] M. Naor and G. N. Rothblum. The complexity of online memory checking. *Journal of the ACM*, 56(1):2:1–2:46, February 2009.

[31] C. Papamanthou, R. Tamassia, and N. Triandopoulos. Authenticated hash tables. In *ACM Conference on Computer and Communications Security, CCS 2008*, pages 437–448, 2008.

[32] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

[33] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.

[34] B. Sengupta, S. Bag, S. Ruj, and K. Sakurai. Retricoin: Bitcoin based on compact proofs of retrievability. In *International Conference on Distributed Computing and Networking, ICDCN 2016*, pages 14:1–14:10, 2016.

[35] B. Sengupta and S. Ruj. *Guide to Security Assurance for Cloud Computing*, chapter Cloud Data Auditing Using Proofs of Retrievability, pages 193–210. Springer International Publishing, 2015.

[36] H. Shacham and B. Waters. Compact proofs of retrievability. *Journal of Cryptology*, 26(3):442–483, 2013.

[37] E. Shi, E. Stefanov, and C. Papamanthou. Practical dynamic proofs of retrievability. In *ACM Conference on Computer and Communications Security, CCS 2013*, pages 325–336, 2013.

[38] C. Wang, S. S. M. Chow, Q. Wang, K. Ren, and W. Lou. Privacy-preserving public auditing for secure cloud storage. *IEEE Transactions on Computers*, 62(2):362–375, 2013.

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