Robust Dynamic Provable Data Possession*

Bo Chen Reza Curtmola
Department of Computer Science
New Jersey Institute of Technology
Newark, USA
Email: bc47@njit.edu, crix@njit.edu

***Abstract*—Remote Data Checking (RDC) allows clients to efficiently check the integrity of data stored at untrusted servers. This allows data owners to assess the risk of outsourcing data in the cloud, making RDC a valuable tool for data auditing. A *robust* RDC scheme incorporates mechanisms to mitigate arbitrary amounts of data corruption. In particular, protection against small corruptions (*i.e.*, bytes or even bits) ensures that attacks that modify a few bits do not destroy an encrypted file or invalidate authentication information. Early RDC schemes have focused on static data, whereas later schemes such as DPDP support the full range of *dynamic* operations on the outsourced data, including insertions, modifications, and deletions. Robustness is required for both static and dynamic RDC schemes that rely on spot checking for efficiency.**

However, under an adversarial setting there is a fundamental tension between efficient dynamic updates and the encoding required to achieve robustness, because updating even a small portion of the file may require retrieving the entire file. We identify the challenges that need to be overcome when trying to add robustness to a DPDP scheme. We propose the first RDC schemes that provide robustness and, at the same time, support dynamic updates, while requiring small, constant, client storage. Our first construction is efficient in encoding, but has a high communication cost for updates. Our second construction overcomes this drawback through a combination of techniques that includes RS codes based on Cauchy matrices, decoupling the encoding for robustness from the position of symbols in the file, and reducing insert/delete operations to append/modify operations when updating the RS-encoded parity data.

I. Introduction

Remote Data Checking (RDC) is a technique that allows to check the integrity of data stored at a third party, such as a Cloud Storage Provider (CSP). Especially when the CSP is not fully trusted, RDC can be used for data auditing, allowing data owners to assess the risk of outsourcing data in the cloud.

In an RDC protocol, the data owner (client) initially stores data and metadata with the cloud storage provider (server); at a later time, an auditor (the data owner or another client) can challenge the server to prove that it can produce the data that was originally stored by the client; the server then generates a proof of data possession based on the data and the metadata. Several RDC schemes have been proposed, including Provable Data Possession (PDP) [1], [2] and Proofs of Retrievability (PoR) [3], [4], both for the single server [1], [3], [4] and for the multiple server setting [5]–[8].

Early RDC schemes have focused on *static* data, in which the client cannot modify the original data [1], [3], [4] or can only perform a limited set of updates [9]. Erway et al. [10] have proposed DPDP, a scheme that supports the full range of *dynamic updates* on the outsourced data, while providing the same strong guarantees about data integrity. The ability to perform updates such as insertions, modifications, or deletions, extends the applicability of RDC to practical systems for file storage [11], [12], database services [13], peer-to-peer storage [14], [15], and more complex cloud storage systems [16], [17].

A scheme for auditing remote data should be both *lightweight* and *robust* [2]. *Lightweight* means that it does not unduly burden the server; this includes both overhead (*i.e.*, computation and I/O) at the server and communication between the server and the client. This goal can be achieved by relying on *spot checking*, in which the client randomly samples small portions of the data and checks their integrity, thus minimizing the I/O at the server. Spot checking allows the client to detect if a fraction of the data stored at the server has been corrupted, but it cannot detect corruption of small parts of the data (*e.g.*, 1 byte).

Robust means that the auditing scheme incorporates mechanisms for mitigating arbitrary amounts of data corruption. Protecting against **large corruptions** ensures the CSP has committed the contracted storage resources: Little space can be reclaimed undetectably, making it unattractive to delete data to save on storage costs or sell the same storage multiple times. Protecting against **small corruptions** protects the data itself, not just the storage resource. Many data have value well beyond their storage costs, making attacks that corrupt small amounts of data practical. For example, modifying a single bit may destroy an encrypted file or invalidate authentication information. Thus, *robustness is a necessary property* for all RDC schemes that rely on spot checking, which includes the majority of static and dynamic RDC schemes.

Robustness is usually achieved by integrating forward error-correcting codes (FECs) with remote data checking [2], [18], [19]. Attacks that corrupt small amounts of data do no damage, because the corrupted data may be recovered by the FEC. Attacks that do unrecoverable amounts of damage are easily detected using spot checking, because they must corrupt many blocks of data to overcome the FEC redundancy. Unfortunately, under an adversarial setting, there is a fundamental tension between the dynamic nature of the updates supported in the DPDP scheme and FEC codes (which are mostly designed for static data) because securely updating even a small portion of the file may require retrieving the entire file.

*This is the full version of the paper that appears in the proceedings of SPCC '12 [35].

In this paper, we make the following contributions:

- We identify Reed-Solomon (RS) codes based on Cauchy matrices which provide communication-efficient code updates and propose methods to efficiently update the code parity under a benign setting (*i.e.*, when the server is trustworthy). We observe that append/modify updates have a much lower bandwidth overhead than insert/delete updates (Section III-B).
- We identify the challenges that need to be overcome when trying to add robustness to a DPDP scheme in an adversarial setting (Section IV). Reed-Solomon codes provide efficient error correction capabilities in the static case, but their linear nature imposes a high communication cost when even a small portion of the original data needs to be updated (insertions/deletions). Moreover, it is difficult to hide the relationship among file symbols (required for robustness) while achieving a low communication overhead for updates.
- We give the definition of a Robust DPDP (R-DPDP) scheme, which is a remote data checking scheme that supports dynamic updates and at the same time provides robustness. We propose two R-DPDP constructions that realize this definition. The first one, πR-D, achieves robustness by extending techniques from the static to the dynamic setting. The resulting R-DPDP scheme is efficient in encoding, but requires a high communication cost for updates (insertions/deletions). Our second construction, VLCG (Variable Length Constraint Group), overcomes this drawback by: (a) decoupling the encoding for robustness from the position of symbols in the file and instead relying on the value of symbols, and (b) reducing expensive insert/delete operations to append/modify operations when updating the RS-coded parity data, which ensures efficient updates even under an adversarial setting. The improvement provided by VLCG over πR-D is beneficial, as our source code analysis of a few popular software projects shows that insert/delete operations represent a majority of all updates (Section IV-C).

Although DPDP schemes and robustness for the static RDC setting have been individually considered previously, we are the first to propose R-DPDP schemes that simultaneously provide robustness and support dynamic updates, while requiring small, constant, client storage.

On the Adversarial Model. We work under the assumption that the cloud storage server is not trustworthy. Protection against corruption of a large portion of the data is necessary in order to handle servers that discard a significant fraction of the data. This applies to servers that are financially motivated to sell the same storage resource to multiple clients.

Protection against corruption of a small portion of the data is necessary in order to handle servers that try to hide data loss incidents. This applies to servers that wish to preserve their reputation. Data loss incidents may be accidental (*e.g.*, management errors or hardware failures) or malicious (*e.g.*, insider or outsider attacks).

Moreover, the storage server may try to provide a stale, older version of the data.

II. Background and Related Work

A. Remote Data Checking (RDC)

Remote Data Checking (RDC) allows a client to check the integrity of data outsourced at an untrusted server, and thus to audit whether the server fulfills its contractual obligations. For simplicity, we assume the client's data consists of a file $\mathtt{F}$.

A RDC protocol consists of three phases: Setup, Challenge and Retrieve. During Setup, the data owner preprocesses the file $\mathtt{F}$ generating metadata Σ, and then stores both $\mathtt{F}$ and Σ at the server. The data owner deletes $\mathtt{F}$ and Σ from its local storage and only keeps a small amount of secret key material K (**constant client storage**). During Challenge, an auditor (the data owner or another client) challenges the server to prove that it can produce the data that was originally stored. The server produces a proof of data possession based on the data and the metadata. The client uses the secret key material K to check the validity of the proof provided by the server. During the Retrieve phase, the data owner recovers the original data. Depending on how robustness is added to RDC, recovering the data can be as simple as retrieving the original file or using the redundancy provided by error correction to compensate for small data corruption.

B. Remote Data Checking for Dynamic Settings

Early RDC schemes have focused on *static* data, in which the client cannot modify the original data [1], [3], [4] or can only perform a limited set of updates [9].

Dynamic Provable Data Possession (DPDP) [10] proposes a model that provides strong guarantees about data integrity while supporting the full range of dynamic operations on the outsourced data, including modifications, insertions, deletions, and appends. A DPDP protocol contains the three phases as in an RDC protocol for static data (Setup, Challenge, and Retrieve), but also allows another phase, Update. During Update, the original file may be updated. During Challenge, the auditor obtains an integrity guarantee about the latest version of the file (due to updates, this may be different from the original file). In Retrieve, the client recovers the latest version of the file. As opposed to handling static data, the main challenge in DPDP is ensuring that the client obtains guarantees about the latest version of the file (*i.e.*, prevent the server from passing the client's challenges by using old file versions) while meeting the low overhead requirements for RDC.

A DPDP scheme is a collection of seven polynomial-time algorithms (KeyGen_DPDP, PrepareUpdate_DPDP, PerformUpdate_DPDP, VerifyUpdate_DPDP, GenChallenge_DPDP, Prove_DPDP, Verify_DPDP) that can be used to construct a DPDP protocol as follows. During the Setup phase, the client uses KeyGen_DPDP to setup the scheme and PrepareUpdate_DPDP to preprocess the file and generate metadata. The server stores the client's data using PerformUpdate_DPDP and the client uses VerifyUpdate_DPDP to check the success of the initial file submission (note that the initial file submission can be seen as an update in which the client re-

writes the entire file). In the Update phase, the client and server use PrepareUpdate_DPDP, PerformUpdate_DPDP and VerifyUpdate_DPDP to prepare the update, apply the update on the file, and verify if the update was applied correctly, respectively. During the Challenge phase, the client uses GenChallenge_DPDP to generate a challenge, the server generates a proof of data possession using Prove_DPDP, and the client verifies the proof using Verify_DPDP.

For the complete definition of a DPDP scheme, we refer the reader to [10]. We note that DPDP does not include provisions for robustness.

Dynamic Proofs of Retrievability. Concurrently with our work, Stefanov *et al.* [20] proposed Iris, a system that supports dynamic proofs of retrievability (D-PoR), including protection against small data corruption. For practical reasons, Iris achieves robustness by storing the parity data on the client. As this may place an additional burden on lightweight clients, our work focuses on a more challenging setting which has stood as an open problem: All data, including parity, is stored at the server, in order to minimize client storage.

Another proposal for D-PoR [21] does not offer protection against small data corruption when clients rely on spot checking data stored at untrusted servers.

Authenticated Data Structures. In all the DPDP and D-PoR constructions, the client uses an *authenticated data structure* to ensure the freshness of the retrieved file and to prevent the server from using an old file version when answering challenges. This data structure is usually a tree-like structure computed over the verification tags, and the client keeps a copy of the root of this structure (*e.g.*, skip lists [10], RSA trees [10], Merkle hash trees [20], [22], or 2-3 trees [21]). Our work can rely on any of these data structures to ensure file data freshness and prevent the server from conducting replay attacks.

C. Error-correcting codes

We say that a code $\mathcal{C}$ is a $(n, k, d+1)$ error-correcting code if it encodes a message of k symbols into a codeword of n symbols and has minimum distance $d+1$. The minimum distance $d+1$ is the minimum Hamming distance between any two distinct codewords of $\mathcal{C}$ and reflects the code's ability to handle errors. We will use Reed-Solomon [23] (RS) codes which are Maximum Distance Separable (MDS) codes (*i.e.*, can tolerate as many erasures as their overhead, *e.g.*, $n-k$). We use the notation (n, k) RS code to denote a RS code that can correct up to $d = n - k$ erasures.

For data checking protocols, we are concerned with erasure correction not error correction. The integrity of data is independently verifiable, *e.g.*, through verification tags in PDP [1]. Any data error will be detected and the erroneous data will be omitted, converting an error into an erasure.

We consider *systematic* forward error-correcting (FEC) codes, which are codes that embed the unmodified input in the encoded output (*e.g.*, the first k symbols of a codeword are the same k original symbols). We refer to the redundant symbols in the codeword as *parity* symbols.

D. Robust Auditing of Outsourced Data

A *robust* auditing scheme incorporates mechanisms for mitigating arbitrary amounts of data corruption. We consider a notion of mitigation that includes the ability to both efficiently detect data corruption and be impervious to data corruption. When data corruption is detected, the owner can act in a timely fashion (*e.g.*, data can be restored from other replicas). Even when data corruption is not detected, a robust auditing scheme ensures that no data will be lost. More formally, we define a robust auditing scheme as follows [2]:

***Definition** 2.1:* A *robust auditing scheme* $\mathcal{RA}$ is a tuple $(\mathcal{C}, \mathcal{T})$, where $\mathcal{C}$ is a remote data checking scheme for a file $\mathtt{F}$ and $\mathcal{T}$ is a transformation that yields $\tilde{\mathtt{F}}$ when applied on $\mathtt{F}$. We say $\mathcal{RA}$ provides δ-*robustness* when:

– the auditor will *detect* with high probability if the server corrupts more than a δ-fraction of $\tilde{\mathtt{F}}$ (**protection against corruption of a *large* portion of $\tilde{\mathtt{F}}$**)

– the auditor will *recover* the data in $\mathtt{F}$ with high probability if the server corrupts at most a δ-fraction of $\tilde{\mathtt{F}}$ (**protection against corruption of a *small* portion of $\tilde{\mathtt{F}}$**)

Several methods can be employed to add robustness to a remote data checking scheme [2], [18]. The most straightforward method is to use an FEC code over the entire file. For a file of f symbols, this can be achieved with an (n, f) Reed-Solomon code and would give an even stronger guarantee than δ-robustness, because this code can deterministically correct up to $n - f$ erasures and not just with high probability. However, such an FEC code would be impractical because RS codes become quite inefficient to compute even for moderate-size files if the code were to be applied over the entire file.

For efficiency reasons, it is desirable to apply a RS code over a smaller number of symbols: the file $\mathtt{F}$ is divided into k-symbol chunks and a (n, k) RS code is applied to each chunk, expanding it into a n-symbol codeword. The first k symbols of the codeword are the original k data symbols, followed by $d = n-k$ parity symbols. We define a *constraint group* as the group of symbols from the same codeword, *i.e.*, the original k data symbols and their corresponding $n-k$ parity symbols. The number of constraint groups in the encoded file is the same as the number of chunks in the original file, namely, $\frac{f}{k}$.

To ensure the δ-robustness guarantee, **it is necessary that the association between symbols and constraint groups remain hidden** (*i.e.*, the server should not know which symbols belong to the same constraint group). This can be achieved through a combination of permuting and then encrypting the symbols of the file. Different encoding schemes to add robustness can lead to remote data checking schemes with different properties and performance characteristics [2], [3], [18], [19]. We review two of them next.

Let $(\mathsf{G}, \mathsf{E}, \mathsf{D})$ be a symmetric-key encryption scheme and π, ψ, ω be pseudo-random permutations (PRPs) defined as:

– $\pi : \{0,1\}^{\kappa} \times \{0,1\}^{\log_2(fn/k)} \rightarrow \{0,1\}^{\log_2(fn/k)}$

– $\psi : \{0,1\}^{\kappa} \times \{0,1\}^{\log_2(f)} \rightarrow \{0,1\}^{\log_2(f)}$

– $\omega : \{0,1\}^{\kappa} \times \{0,1\}^{\log_2(fd/k)} \rightarrow \{0,1\}^{\log_2(fd/k)}$

We use the keys w, z, v, u for the encryption scheme, PRP π, PRP ψ and PRP ω, respectively.

Permute-All (πA). The constraints among symbols can be concealed by randomly permuting and then encrypting *all* the symbols of the encoded file. Starting from the file $\mathtt{F} = \mathtt{b}_1, \ldots, \mathtt{b}_f$, we use a (n, k) RS code (with $d = n - k$) to generate the encoded file $\hat{\mathtt{F}} = \mathtt{b}_1, \ldots, \mathtt{b}_f, \mathtt{c}_1, \ldots, \mathtt{c}_{\frac{f}{k}d}$, in which symbols $\mathtt{b}_{ik+1}, \ldots, \mathtt{b}_{(i+1)k}$ are constrained by parity symbols $\mathtt{c}_{id+1}, \ldots, \mathtt{c}_{(i+1)d}$, for $0 \leq i \leq \frac{f}{k} - 1$. We then use π and E to randomly permute and then encrypt all the symbols of $\hat{\mathtt{F}}$, obtaining the encoded file $\tilde{\mathtt{F}}$, where $\tilde{\mathtt{F}}[i] = \mathsf{E}_w(\hat{\mathtt{F}}[\pi_z(i)])$, for $1 \leq i \leq fn/k$.

This strategy leads to a δ-robustness guarantee [2], [3], [18], but has two major drawbacks: permuting the entire encoded file can be inefficient and the systematic nature of the RS code is sacrificed.

Permute-Redundancy (πR). The drawbacks of the πA scheme can be overcome by observing that it is sufficient to *permute and then encrypt only the parity symbols*. The input file $\mathtt{F} = \mathtt{b}_1, \ldots, \mathtt{b}_f$ is encoded as follows:

1) Use ψ to randomly permute the symbols of $\mathtt{F}$ to obtain the file $\mathtt{P} = \mathtt{p}_1, \ldots, \mathtt{p}_f$, where $\mathtt{p}_i = \mathtt{b}_{\psi_v(i)}$, $1 \leq i \leq f$.
2) Compute parity symbols $\mathtt{C} = \mathtt{c}_1, \ldots, \mathtt{c}_{\frac{f}{k}d}$ so that symbos $\mathtt{p}_{ik+1}, \ldots, \mathtt{p}_{(i+1)k}$ are constrained by $\mathtt{c}_{id+1}, \ldots, \mathtt{c}_{(i+1)d}$, for $0 \leq i \leq \frac{f}{k} - 1$.
3) Permute and then encrypt the parity symbols to obtain $\mathtt{R} = \mathtt{r}_1, \ldots, \mathtt{r}_{\frac{f}{k}d}$, where $\mathtt{r}_i = \mathsf{E}_w(\mathtt{c}_{\omega_u(i)})$, $1 \leq i \leq \frac{f}{k}d$.
4) Output redundancy encoded file $\tilde{\mathtt{F}} = \mathtt{F}||\mathtt{R}$.

By computing RS codes over the permuted input file, rather than the original input file, an attacker does not know the relationship among symbols of the input file. By permuting the parity symbols, the attacker does not know the relationship among the symbols in the redundant portion $\mathtt{R}$ of the output file. By encrypting the parity symbols, an attacker cannot find the combinations of input symbols that correspond to output symbols.

When compared to πA, πR is more efficient (as it requires to permute and encrypt only the parity symbols) and preserves the systematic nature of the RS code. πR was shown to achieve the δ-robustness guarantee [18], [19] (this construction is also known as a "server code" in the literature [6], [19]).

III. Preliminaries

Towards achieving robustness, in this section we first review Reed-Solomon encoding and decoding based on Cauchy matrices. We then study how to update a Reed-Solomon code when an update is applied to the original data. Note that in this section we study these operations under a benign setting (*i.e.*, when the server is trustworthy).

We consider a (n, k) Reed-Solomon (RS) code that can correct up to $d = n - k$ known erasures or $\lfloor \frac{d}{2} \rfloor$ unknown errors, or any combination of E errors and S erasures with $2E + S \leq d$. The minimum Hamming distance of the RS code is $d + 1$, where $d = n - k$. If two RS codes have the same value d, we say they provide *the same fault tolerance level*.

To encode a k-symbol message into a n-symbol codeword, we need a $n * k$ encoding matrix, known as the *distribution matrix*. Typically, Vandermonde or Cauchy matrices are used to construct the distribution matrix. We use Cauchy RS codes, which are Reed-Solomon codes based on Cauchy matrices [24], for two reasons: they are more suitable to handle dynamic operations on the original data (as we show in Sec. III-B) and they were shown to be approximately twice as fast as the classical Reed-Solomon encoding based on Vandermonde matrices [25]–[28].

A. Cauchy RS Encoding and Decoding

For ease of presentation, we present the Cauchy RS encoding and decoding using a (6,4) RS code as an example (*i.e.*, $n = 6, k = 4, d = 2$). All the arithmetic operations are in $GF(2^w)$, assuming the condition $2^w > n$ always holds (the $+$, $-$ operations can be regarded as $\oplus$, logical XOR). We use $\mathbf{L}^{\mathrm{T}}$ to denote the transpose of a vector $\mathbf{L}$.

Encode. The message $\mathbf{L}$ contains 4 data symbols, all of which are in the Galois Field $GF(2^w)$: $\mathbf{L} = (\ b_1 \quad b_2 \quad b_3 \quad b_4\ )$.

We use the method introduced in [29] to construct the Cauchy matrix, which has the useful property that it can be re-generated on the fly based on a constant amount of information. The distribution matrix M_1, which is composed of the identity matrix in the first 4 rows and Cauchy matrix in the remaining 2 rows, is as follows:

$$M_1 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \end{pmatrix}, \text{where } a_{ij} = \frac{1}{i \oplus (d + j)}$$

The codeword C is computed as

$$C = M_1 \times \mathbf{L}^{\mathrm{T}} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \end{pmatrix} \times \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ p_1 \\ p_2 \end{pmatrix}$$

where the parity symbols p_1 and p_2 are
$p_1 = a_{11} * b_1 + a_{12} * b_2 + a_{13} * b_3 + a_{14} * b_4$
$p_2 = a_{21} * b_1 + a_{22} * b_2 + a_{23} * b_3 + a_{24} * b_4$

Decode. When using a $(6, 4)$ RS code, any 4 out of 6 symbols are enough to recover $\mathbf{L}$. Assume that b_3 and b_4 are corrupted. To recover $\mathbf{L}$, the decoding process is:

$$\mathbf{L}^{\mathrm{T}} = M^{-1} \times \begin{pmatrix} b_1 \\ b_2 \\ p_1 \\ p_2 \end{pmatrix}, \text{where } M = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \end{pmatrix}$$

The decoding matrix, M, is invertible based on the fact that all of the $4*4$ submatrices of M_1 are invertible [30]. Moreover, M has the useful property that it can be re-generated by knowing the indices of the non-corrupted symbols in the codeword. By putting the non-corrupted symbols into their right locations in the codeword for decoding, M can be re-generated based on a constant amount of information.

B. Cauchy RS Updating

Consider a (n, k) Cauchy RS code computed over a message. If the symbols in the original message are updated (*e.g.*,

modified, appended, inserted, deleted), we are interested to update the RS parity data so that it reflects the updated message. We seek to answer the question: how can we minimize the cost of updating the RS code? More precisely, how can we update the parity symbols efficiently by minimizing the number of symbols that need to be read from the orginal RS code?[1]. We answer this question using the same example from Sec. III-A. The conclusion is that *modify/append operations have a lower bandwidth overhead than insert/delete operations*.

Modify a data symbol. For example, if symbol b_1 is modified to b'_1, we should update the parity data correspondingly: p_1 is updated to p'_1, and p_2 is updated to p'_2. To compute p'_1 and p'_2, only *the old parity symbols* (p_1, p_2) and *the old data symbol* (b_1) are required to be retrieved (*i.e.*, there is no need to retrieve any other data symbol except the one to be modified):

$$\begin{aligned} p'_1 &= a_{11} * b'_1 + a_{12} * b_2 + a_{13} * b_3 + a_{14} * b_4 \\ &= a_{11} * b_1 + a_{12} * b_2 + a_{13} * b_3 + a_{14} * b_4 + a_{11} * b'_1 - a_{11} * b_1 \\ &= p_1 + a_{11} * b'_1 - a_{11} * b_1 \end{aligned}$$

$$\begin{aligned} p'_2 &= a_{21} * b'_1 + a_{22} * b_2 + a_{23} * b_3 + a_{24} * b_4 \\ &= a_{21} * b_1 + a_{22} * b_2 + a_{23} * b_3 + a_{24} * b_4 + a_{21} * b'_1 - a_{21} * b_1 \\ &= p_2 + a_{21} * b'_1 - a_{21} * b_1 \end{aligned}$$

Append a data symbol. For example, if b_5 is appended to $\mathbf{L}$, to maintain *the same fault tolerance level*, the $(6, 4)$ RS code should become a $(7, 5)$ RS code and the new distribution matrix M_2 is (assuming the condition $2^w > n$ still holds):

$$M_2 = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \end{pmatrix}, \text{ where } a_{ij} = \frac{1}{i \oplus (d+j)}$$

Compared to M_1, most of the elements in M_2 are the same although n has changed (this is a useful property of Cauchy RS codes). To update the parity data correspondingly, only *the old parity symbols* (p_1, p_2) are required to be retrieved (*i.e.*, there is no need to retrieve any of the data symbols):

$$\begin{aligned} p'_1 &= a_{11} * b_1 + a_{12} * b_2 + a_{13} * b_3 + a_{14} * b_4 + a_{15} * b_5 \\ &= p_1 + a_{15} * b_5 \end{aligned}$$

$$\begin{aligned} p'_2 &= a_{21} * b_1 + a_{22} * b_2 + a_{23} * b_3 + a_{24} * b_4 + a_{25} * b_5 \\ &= p_2 + a_{25} * b_5 \end{aligned}$$

Insert a data symbol. For example, if b'_2 is inserted into $\mathbf{L}$ after b_2, to maintain *the same fault tolerance level*, the $(6, 4)$ RS code should become a $(7, 5)$ RS code. The new distribution matrix will be M_2, thus, we can update the parity data (p_1 to p'_1, p_2 to p'_2) by retrieving whichever one is smaller between:

[1]For the purpose of RDC, the client needs to update the RS code on the server after each update operation, so we are interested in minimizing the data communication required to update the RS code.

- $\mathtt{D}$ is the encoded file, $\mathtt{F}$ is the original file, $\mathtt{P}$ is the redundancy added after applying a RS code over $\mathtt{F}$. We have $\mathtt{D} = \mathtt{F}||\mathtt{P}$.
- n is the number of symbols in a constraint group: $n = k + d$, where k is the number of data symbols and d is the number of parity symbols. A (n, k) RS code is applied over each constraint group.
- M is the server metadata computed over D (stored at the server, includes the verification tags).
- $\mathtt{F}_i$ is the i-th version of $\mathtt{F}$. We have $\mathtt{D}_i = \mathtt{F}_i||\mathtt{P}_i$ and M_i is the server metadata for $\mathtt{D}_i$.
- M_c is the client metadata (*e.g.*, the root of the skip list/RSA tree [10]).
- $info$ is the information about the update operation (*e.g.*, full rewrite, delete block i, modify block i, insert a block after block i, etc.).

Fig. 1: Reference sheet for various notations.

(i) *all the data symbols* (b_1, b_2, b_3, b_4), or (ii) *all the parity symbols* (p_1, p_2) and *the data symbols after* b_2 (b_3, b_4):

$$\begin{aligned} p'_1 &= a_{11} * b_1 + a_{12} * b_2 + a_{13} * b'_2 + a_{14} * b_3 + a_{15} * b_4 \\ &= p_1 - a_{13} * b_3 - a_{14} * b_4 + a_{13} * b'_2 + a_{14} * b_3 + a_{15} * b_4 \end{aligned}$$

$$\begin{aligned} p'_2 &= a_{21} * b_1 + a_{22} * b_2 + a_{23} * b'_2 + a_{24} * b_3 + a_{25} * b_4 \\ &= p_2 - a_{23} * b_3 - a_{24} * b_4 + a_{23} * b'_2 + a_{24} * b_3 + a_{25} * b_4 \end{aligned}$$

Delete a data symbol. Similar to inserting a data symbol, to update the parity when deleting the i-th symbol from $\mathbf{L}$, we need to retrieve *the smaller between either all the data symbols, or all the parity symbols and the data symbols after position* i.

IV. Robust Dynamic Provable Data Possession schemes

In this section, we present R-DPDP, a new framework to add robustness to DPDP. R-DPDP allows to audit remote data that is dynamically changing and, at the same time, offers protection against both large and small data corruption. To the best of our knowledge, robustness has not been previously considered for dynamic remote data checking while maintaining small, constant, client storage.

We start by summarizing the challenges that need to be overcome when adding robustness to DPDP. We then present the definition of a R-DPDP scheme and propose two R-DPDP constructions: πR-D (an extension of the πR scheme presented in Sec. II-D) and VLCG (Variable Length Constraint Group, a new scheme that improves the communication efficiency of πR-D in the Update phase). To facilitate the exposition, we include a reference sheet with various notations in Fig. 1.

Challenges. It is challenging to add robustness to DPDP in an adversarial setting and also maintain low bandwidth overhead for updates, because:

- Adding robustness to DPDP requires encoding the data using Reed-Solomon codes. RS codes, as a type of linear codes, provide error correction in a static setting, as they compute redundancy over every portion of the original data. However, they are not immediately suitable when the data

can be dynamically updated. As shown in Sec. III-B, for certain update operations (insert/delete), updating even a small portion of the original data imposes a high communication cost (this holds even under a benign setting, in which the server is trustworthy).

- Robustness applies RS encoding over groups of symbols (*constraint groups*) and it requires to **hide the association between symbols and constraint groups** (*i.e.*, the server should not know which symbols belong to the same constraint group). When dynamic updates are performed over file data, the parity of the affected constraint groups should also be updated, which requires knowledge of the data and the parity symbols in those constraint groups (Sec. III-B). However, the client cannot simply retrieve only the symbols of the affected constraint groups, as that would reveal the contents of the corresponding constraint groups and break robustness. Moreover, the client cannot simply update only the parity symbols in the affected constraint groups, as that may allow the server to infer which parity symbols are in the same constraint group by comparing the new parity with the old parity.

File representation. We use two independent logical representation of the file for different purposes:
– For the purpose of file updating (during the Update phase), the file is seen as an ordered collection of blocks. Basically, update operations occur at the block level. This is also the representation used for checking data possession (during the Challenge phase), as each block has one corresponding verification tag.
– For the purpose of encoding for robustness, the file is seen as a collection of symbols, which are grouped into *constraint groups* and each constraint group is encoded independently.

For each file block, there is a corresponding verification tag which needs to be stored at the server. Thus, larger file blocks result in smaller additional server storage overhead due to verification tags. On the other hand, efficient encoding and decoding requires the symbols to be from a small size field. As a result, one file block will usually contain multiple symbols. *Each file update operation which is performed at the block level results into several operations applied to the symbols in that block* (for example, when modifying a data block, all the symbols in that block and the corresponding parity symbols should be modified).

Metric. To measure the communication overhead for data updates, we use as a metric the *update bandwidth factor* α defined as

$$\alpha = \frac{the\ amount\ of\ data\ downloaded\ for\ updating\ one\ file\ block}{the\ total\ amount\ of\ data\ at\ the\ server}$$

A. R-DPDP Definition

We introduce the definition of a robust dynamic data possession (R-DPDP) scheme. Compared to a DPDP scheme, R-DPDP adds robustness, which is reflected in slightly different definitions for the data-updating algorithms. We have also added an explicit algorithm to decode the data, since data decoding is more challenging when robustness is needed. We advise readers familiar with the definition of DPDP to skip this section and continue with Section IV-B.

Definition 4.1: (R-DPDP SCHEME) A Robust Dynamic Provable Data Possession (R-DPDP) scheme is a collection of eight polynomial-time algorithms:

- $\mathsf{KeyGen}(1^\kappa) \rightarrow \{sk, pk\}$: a probabilistic key generation algorithm run by the **client** to setup the scheme. Input: the security parameter κ. Output: the secret key sk and public key pk.
- $\mathsf{PrepareUpdate}(sk, pk, \Delta F, \Delta D_{i-1}, info, M_c) \rightarrow \{e(\Delta D), e(info^{'}), e(\Delta M)\}$: an algorithm run by the **client** to prepare (a part of) the file for untrusted storage. *Input:* the secret key sk, the public key pk, (a part of) the file ΔF, (a part of) the previous version of the encoded file ΔD_{i-1}, information about the update operation $info$, and the client metadata M_c. *Output:* the "encoded" version of the update data $e(\Delta D)$ (add to ΔD randomness, sentinels, or simply let $e(\Delta(D)) = \Delta D$. ΔD is the data to be updated), the "encoded" version of the update information $e(info')$ ($info$ will be changed to $info'$, since updating ΔF may lead to updating of the redundancy. $info'$ should be changed to fit the encoded version of ΔD), and the new server metadata $e(\Delta M)$. The client will send $e(\Delta D), e(info')$, and $e(\Delta M)$ to the server.
- $\mathsf{PerformUpdate}(pk, D_{i-1}, M_{i-1}, e(\Delta D), e(info), e(\Delta M)) \rightarrow \{D_i, M_i, M_c^{'}, P_{M_c^{'}}\}$: an algorithm run by the **server** in response to an update request from the client. *Input:* public key pk, the old version of the encoded file D_{i-1}, the metadata M_{i-1}, and the values $e(\Delta D), e(info)$, $e(\Delta M)$ provided by the client. *Output:* the new version of the encoded file D_i and metadata M_i, the metadata to be sent to client $M_c^{'}$ and its proof of correctness $P_{M_c'}$. The server will send $M_c^{'}$ and $P_{M_c'}$ back to the client.
- $\mathsf{VerifyUpdate}(sk, pk, \Delta F, \Delta D_{i-1}, info, M_c, M_c^{'}, P_{M_c^{'}}) \rightarrow \{accept, reject\}$: an algorithm run by the **client** to verify the server's behavior during the update. *Input:* all the inputs from PrepareUpdate, $M_c^{'}$ and the proof $P_{M_c^{'}}$ which are sent back by the server. *Output: accept* if the check succeeds, *reject* otherwise.
- $\mathsf{GenChallenge}(sk, pk, M_c) \rightarrow \{c\}$: a probabilistic algorithm run by the **client** to issue a challenge for the server. Input: the secret key sk, public key pk, and the latest client metadata M_c. Output: the challenge c that will be sent to the server.
- $\mathsf{Prove}(pk, D_i, M_i, c) \rightarrow \{\Pi\}$: an algorithm run by the **server** to generate the proof of possession upon receiving the challenge from the client. Input: the public key pk, the latest version of the encoded file D_i, the metadata M_i, and the challenge c. Output: a proof of possession Π that will be sent back to the client.
- $\mathsf{Verify}(sk, pk, M_c, c, \Pi) \rightarrow \{accept, reject\}$: an algorithm run by the **client** to validate a proof of possession upon receiving the proof Π from the server. Input: the secret key sk, the public key pk, the client metadata M_c, the challenge c, and the proof Π. Output: *accept* if Π is a valid proof of possession, *reject* otherwise.

- $\mathsf{Decode}(sk, pk, D_i, M_i, M_c) \rightarrow \{F_i, failure\}$: an algorithm run by the **client** to decode the latest version of the encoded file D_i (repair it if small corruption exists). *Input:* the secret key sk, the public key pk, the latest version of the encoded file D_i (where $D_i = F_i||P_i$), metadata M_i, and client metadata M_c. *Output:* the latest version of the file F_i if the decode process is successful, *failure* otherwise.

A R-DPDP protocol can be constructed in four phases, Setup, Challenge, Update, and Retrieve.

Setup: The client C who is in possession of file F runs $(pk, sk) \leftarrow \mathsf{KeyGen}(1^\kappa)$, followed by $\{e(\Delta D), e(info'), e(\Delta M)\} \leftarrow \mathsf{PrepareUpdate}(sk, pk, F, NULL,$ *"full re-write"*, $NULL)$. C sends $e(\Delta D), e(info'), e(\Delta M)$ to the server S. S runs $\{D_1, M_1, M_c^{'}, P_{M_c^{'}}\} \leftarrow \mathsf{PerformUpdate}(pk, NULL, NULL, e(\Delta D), e(info'), e(\Delta M))$ and sends $M_c^{'}, P_{M_c^{'}}$ back to C. C then runs $\mathsf{VerifyUpdate}(sk, pk, F, NULL,$ *"full re-write"*, $NULL, M_c^{'}, P_{M_c^{'}})$ to check whether the initial data outsourcing is successful or not. If successful, C sets $M_c = M_c^{'}$, and deletes F.

Challenge: C generates challenge c by running $\mathsf{GenChallenge}(sk, pk, M_c)$, and sends c to S. S runs $\{\Pi\} \leftarrow \mathsf{Prove}(pk, D_i, M_i, c)$ and sends to C the proof of possession Π. C can check the validity of the proof Π by running $\mathsf{Verify}(sk, pk, M_c, c, \Pi)$.

Update: C downloads ΔD_{i-1} from S, and runs $\{e(\Delta D), e(info'), e(\Delta M)\} \leftarrow \mathsf{PrepareUpdate}(sk, pk, \Delta F, \Delta D_{i-1}, info, M_c)$. C sends $e(\Delta D), e(info'), e(\Delta M)$ to S. S runs $\{D_i, M_i, M_c^{'}, P_{M_c^{'}}\} \leftarrow \mathsf{PerformUpdate}(pk, D_{i-1}, M_{i-1}, e(\Delta D), e(info'), e(\Delta M))$ and sends $M_c^{'}, P_{M_c^{'}}$ back to C. C then runs $\mathsf{VerifyUpdate}(sk, pk, \Delta F, \Delta D_{i-1}, info, M_c, M_c^{'}, P_{M_c^{'}})$ to check whether the update is successful or not. If successful, C sets $M_c = M_c^{'}$, then deletes ΔF and ΔD_{i-1}.

Retrieve: C downloads the current version of the encoded file D_i and the server metadata M_i, then runs $\{F_i, failure\} \leftarrow \mathsf{Decode}(sk, pk, D_i, M_i, M_c)$.

B. Enhancing πR: πR-D

We first describe πR-D, a new scheme obtained by adapting the πR scheme (Sec. II-D) to add robustness on top of a DPDP scheme. In the Setup phase, the file F is first processed according to πR, and the parity data P is generated. The encoded file D = F||P is then pre-processed further using the DPDP algorithms PrepareUpdate_DPDP, PerformUpdate_DPDP, and VerifyUpdate_DPDP. During the Challenge phase, we can use directly the DPDP algorithms GenChallenge_DPDP, Prove_DPDP, and Verify_DPDP.

In the Update phase, the main operations are:

- *Insert/Delete a data block.* A data block may contain multiple symbols, which may belong to more than one constraint groups. Inserting/deleting a data block is equivalent to inserting/deleting all the symbols in that block. Inserting/deleting a data symbol will affect the indices of the following data symbols in the whole file, as well as the parameter f of the PRP ψ in πR. Since in πR the contents of each constraint group are decided based on the indices provided by the PRP ψ, the changing of the parameter f of PRP ψ will require the client to download the entire file F and re-compute the parity P based on a new set of constraint groups. The update bandwidth factor is $\alpha = \frac{|\mathtt{F}|}{|\mathtt{D}|}$. The updated file F is pre-processed using the technique described in πR, but the new parity P will be permuted and encrypted by a new key (the client will only keep the new key and discard the previous key). For an insert operation, the newly inserted block is sent back to the server using the corresponding DPDP update algorithms, whereas for a delete operation the corresponding block should be deleted. Also, P should replace the old parity at the server.
- *Modify a data block.* The client downloads the data block to be modified (*i.e.*, the old data block) and the latest version of the parity P, decrypts P and restores the original order, updates the parity symbols in the affected constraint groups according to P, the data symbols in the old and the new data block (exact procedure described in Sec. III-B). The update bandwidth factor is $\alpha = \frac{|\mathtt{P}|}{|\mathtt{D}|}$. To prevent the server from learning the contents of the constraint groups by comparing the new parity with the old parity, the client should use a new key to permute and encrypt the parity symbols (the client will keep the new key and discard the previous key). The new data block and the new parity P are sent back to the server using the DPDP update algorithms, replacing the corresponding old block and old P.

	OpenSSL	Eclipse
dates of activity	1998-2011	2001-2011
# of files	4,283	180,662
# of commits	67,846	883,045
# of insertions (lines)	707,978	8,579,577
# of deletions (lines)	678,936	7,009,582
# of modifications (lines)	371,159	6,714,823
Avg. # commits/file	15.8	4.9
Avg. # insertions/commit	10.4	9.7
Avg. # deletions/commit	10	7.9
Avg. # modifs./commit	5.5	7.6

TABLE I: Statistics for update operations, based on the CVS repositories for OpenSSL and Eclipse.

In the Retrieve phase, the client simply retrieves the file F and may use the parity P to correct data corruption.

Performance analysis. πR-D is efficient during Setup and Retrieve, but has high communication overhead during Update, since for every insertion/deletion the update bandwidth factor is $\alpha = \frac{|\mathtt{F}|}{|\mathtt{D}|}$, which approaches 1 in practice. We have analyzed the pattern of updates for the source files of two popular projects, OpenSSL [31] and Eclipse [32]. Table I shows that the number of insert and delete operations, compared to modifications, represent a majority of the total number of updates. Thus, it is likely that one small update may require to download the entire file F. We need a scheme with lower communication overhead for data updates.

Security analysis. The δ-robustness of the πR-D scheme can

be established similarly as for the πR scheme [2]: Once we fix a target for the probability of a successful attack (*e.g.*, 10^{-10}), we can determine the RS encoding parameters that minimize the number of blocks being spot-checked during a challenge. The resulting RS encoding provides the value of δ. If the adversary corrupts more than a δ-fraction of the encoded file, the spot-checking based strategy and the authentication structure of the underlying DPDP scheme guarantee that the auditor can detect such corruptions with high probability. If the adversary corrupts at most a δ-fraction, the data can be retrieved based on the recovery capability of the RS code.

C. Variable Length Constraint Group

Though efficient in encoding, πR-D has a high communication overhead for updates. In πR-D, the PRP ψ is applied to the index of data symbols, thus making it sensitive to insert/delete operations (*e.g.*, one simple insertion/deletion may require the client to download the entire file $\mathtt{F}$).

To mitigate the drawbacks of πR-D, we propose a second construction called Variable Length Constraint Group (VLCG). Like in πR-D, we still use the notion of *constraint groups*, which are groups of symbols over which a RS code is computed. However, we rely on two additional main insights.

Firstly, unlike in πR-D, in which symbols are assigned to constrained groups based on the position (*i.e.*, index) of the symbols in the file, VLCG assigns symbols to constraint groups based on the value of the symbols. More precisely, for a data symbol b, we use $h_K(b)$ to decide the index of the constraint group to which b belongs. This has the advantage that, in order to insert/delete a data symbol into/from f, we can insert/delete the data symbol into/from the corresponding constraint group, *without affecting other constraint groups*.

Secondly, we employ several techniques to preserve robustness and minimize the bandwidth overhead. For example, we reduce insert operations to append operations and delete operations to modify operations when updating the RS-coded parity data.

We seek to maintain *the same fault tolerance level* (see definition in Sec. III-A) for all constraint groups (that is, for every (n, k) constraint group, $d = n - k$ will be kept the same after each update operation). All the parity symbols $\mathtt{P}$ should be permuted like in πR, but there is no need to encrypt them (this is explained later in more detail).

Next, we give an overview of the VLCG construction, focusing on the Update and Retrieve phases.

Update operations. For all update operations, we first execute the actual block update on the file data, but the challenging step is how to efficiently update the RS-coded parity data.

Inserting a symbol into the file requires updating the parity symbols of the constraint group to which the symbol is assigned. According to the analysis in Sec. III-B, inserting a symbol into a RS code (equivalent to inserting it into a constraint group) requires to retrieve either all the data symbols in that constraint group, or all the parity symbols and some of the data symbols in that constraint group. However, as we argued for the πR scheme (Sec. II-D), to ensure the δ-robustness guarantee, it is necessary that the association between symbols and constraint groups remains hidden. Thus, it is insufficient to only retrieve symbols from the corresponding constraint group. Moreover, it is not possible to efficiently determine which other symbols belong to that constraint group. For these reasons, the client would have to retrieve the entire file $\mathtt{F}$. We overcome this limitation by updating the parity symbols of the symbol's constraint group *as if the symbol was appended to the end of the data symbols in that constraint group* (of course, the symbol is still inserted in the file at the desired location). The advantage of this method is that appending a data symbol to a Cauchy RS code does not require to download any data symbols and we can update the corresponding parity symbols based only on the old parity (cf. Sec. III-B). We note that ensuring δ-robustness prevents us from retrieving only the parity symbols from this constraint group. Instead, we retrieve all the parity data $\mathtt{P}$. Thus the update bandwidth factor is $\alpha = \frac{|\mathtt{P}|}{|\mathtt{D}|}$.

Deleting a data symbol is more complex. Although under a benign setting this operation could be achieved by only retrieving symbols from the same constraint group, ensuring δ-robustness prevents us from using this strategy. Instead, we use a different strategy: to delete a data symbol, we ask the server to physically delete the symbol from $\mathtt{F}$, but we update the parity symbols from the corresponding constraint group *as if that symbol was modified to have the value 0*. As a result, the delete operation is converted into a modify operation when updating the RS-encoded parity data for the corresponding constraint group (a similar strategy was previously used in [7], [9]). A code modify operation, according to Sec. III-B, only requires to download the parity data and the old symbol from the corresponding constraint group. This means that the update bandwidth factor can be kept as $\alpha = \frac{|\mathtt{P}|}{|\mathtt{D}|}$ for deletion.

Modifying a data symbol is the most complex operation because if a symbol is modified to a new different value, it may be re-assigned to a different constraint group. The old symbol must first be deleted from its current constraint group and the new symbol must be inserted into a (possibly) new constraint group. The update bandwidth factor remains $\alpha = \frac{|\mathtt{P}|}{|\mathtt{D}|}$.

In Sec. IV-D, we propose an alternative method to optimize the communication overhead for updates ($\alpha = \frac{\log^2|\mathtt{P}|}{|\mathtt{D}|}$) by using Oblivious RAMs to only retrieve the parity symbols from the corresponding constraint groups.

Retrieve data. The method we use for deciding to which constraint group does a symbol belong in VLCG introduces an additional challenge in the Retrieve phase. By using a PRF over the value of the symbol to decide its constraint group, the encoded file contains no information about the relative position of the symbols inside a constraint group. Note that the initial position of the symbols inside a constraint group may change because of update operations (*i.e.*, modification of symbols). In case of data corruption, the RS code computed over a constraint group will be used to recover the original symbols; however, successfully decoding the RS code requires knowledge of the correct position of symbols inside the constraint group. During file recovery, the correct position of symbols inside a constraint group may be uncertain because of two reasons: (a) if a symbol is corrupted, the client does

not know to which constraint group did that symbol belong, and thus the symbol will be missing from that constraint group during RS decoding; (b) if a symbol is deleted (*i.e.*, a valid delete operation), it does not exist anymore at the server (*i.e.*, we cannot find this symbol in the latest file version), but for RS decoding purposes the client should still use a symbol with value 0 at the corresponding position in the constraint group to which the symbol belongs.

We illustrate the uncertainty in decoding with an example. Assume that a constraint group is a $(6, 4)$ RS code $(b_1\ b_2\ b_3\ b_4\ p_1\ p_2)$. If b_3 was corrupted, the RS decoding should take as input $(b_1\ b_2\ ?\ b_4\ p_1\ p_2)$, whereas if b_3 was deleted, the input for RS decoding should be $(b_1\ b_2\ 0\ b_4\ p_1\ p_2)$. But how does the client know that in position 3 there should be a corrupted symbol or a 0 symbol?

To deal with the uncertainty about symbol position in a constraint group during decoding, we propose a strategy similar with the one used in HAIL [6]: to identify the correct locations of data symbols (healthy and 0 symbols) in their corresponding constraint groups, we convert the parity symbols into cryptographically secure Message Authentication Codes (MACs). Based on these MACs, we use a brute force approach to determine the correct position of symbols for RS decoding (full details in Fig. 2). The parity symbols are converted to secure MACs by composing them with a pseudorandom function (PRF) (we call this operation *masking*). The PRF is computed over the file identifier, the index of the corresponding constraint group, and the index of the parity symbol in the constraint group. For example, using the (6,4) RS code described in Sec. III-A, the new parity symbols p'_1 and p'_2, which are also secure MACs over b_1, b_2, b_3, b_4, are computed as:
$p'_1 = p_1 + g_{K'}(file\_id||constraint\_group\_index||5)$
$p'_2 = p_2 + g_{K'}(file\_id||constraint\_group\_index||6)$,
where g is a PRF and all operations are over $GF(2^w)$, in which "+" and "−" can be regarded as bitwise XORs. To strip off g from p'_1 and p'_2, we compute:
$p_1 = p'_1 - g_{K'}(file\_id||constraint\_group\_index||5)$
$p_2 = p'_2 - g_{K'}(file\_id||constraint\_group\_index||6)$

Since different constraint groups may end up having different sizes, the client needs to keep track of the size of each constraint group [2]. This can be done by recording either n or k for each (n, k) RS code (because $d = n - k$ is fixed for all constraint groups). Let Φ be the set of (n, k) parameters for all constraint groups. For ease of presentation, we assume that Φ is stored (and updated) at the client. In Sec. IV-D we discuss how to store Φ more efficiently.

Finally, we note that since the parity symbols are masked with a PRF, there is no need to further encrypt them as in step 3 of the πR construction (described in Sec. II-D).

*1) **The VLCG construction**:* We are now ready to present the details of our main R-DPDP construction, Variable Length Constraint Group (VLCG). We fix the parameters n and k as the initial size of the RS code computed over each constraint group of k data symbols and let $d = n - k$ be the fault tolerance level of the code. For a file with f symbols $\mathtt{F} = \{\mathtt{b}_1, \mathtt{b}_2, \ldots, \mathtt{b}_f\}$, there will be $m = f/k$ constraint groups and each constraint group will initially have approximately n symbols (k data and d parity symbols). As file updates are performed, the (n, k) parameters for different constraint groups will change, but the fault tolerance level $d = n - k$ will be preserved. Each file symbol is an element in the Galois Field $GF(2^w)$.

Let κ be a security parameter. In addition, we make use of a pseudo-random permutation (PRP) φ and two pseudo-random functions (PRF) h and g with the following parameters:
$-\ \varphi : \{0,1\}^\kappa \times \{0,1\}^{md} \rightarrow \{0,1\}^{md}$
$-\ h : \{0,1\}^\kappa \times \{0,1\}^w \rightarrow \{0,1\}^{\log m}$
$-\ g : \{0,1\}^\kappa \times \{0,1\}^* \rightarrow \{0,1\}^w$

Figures 2 and 3 describe our VLCG construction, which can be built on top of any DPDP scheme DPDP = (KeyGen_DPDP, PrepareUpdate_DPDP, PerformUpdate_DPDP, VerifyUpdate_DPDP, GenChallenge_DPDP, Prove_DPDP, Verify_DPDP) (refer to [10] for the definition of a DPDP scheme). We construct a R-DPDP protocol from the VLCG scheme in four phases Setup, Challenge, Update, and Retrieve as follows.

Setup. The client C runs $\{sk, pk\} \leftarrow \mathsf{KeyGen}(1^\kappa)$, and then runs PrepareUpdate on the file F. PrepareUpdate applies the PRF h over every symbol in F, determining the group where each symbol is assigned to (there are m groups). For every group of k symbols (k may be different for different groups, depending on how many symbols are assigned to the groups), PrepareUpdate applies a $(n = k + d, k)$ RS code and every group becomes a (n, k) constraint group. The md parity symbols from all the constraint groups will form the parity data P. All the symbols in P are masked using PRF g and permuted using PRP φ, both keyed with key K' (similar as in πR). PrepareUpdate then calls the PrepareUpdate_DPDP algorithm of DPDP to further process $\mathtt{D} = \mathtt{F}||\mathtt{P}$.

The output of PrepareUpdate is sent to the server S. S runs PerformUpdate to fully re-write the data (D and the corresponding verification tags in M) and sends back the proof. C verifies the proof by running VerifyUpdate. If the verification is successful, C discards D and M, and keeps all the (n, k) parameters; otherwise, C quits the protocol and retries with a different server.

Challenge. As described in Section IV-A, the client challenges the server to prove data possession using the GenChallenge, Prove, and Verify algorithms which simply call their counterpart algorithms of DPDP.

Update. Three operations are available in the Update phase: insert, delete, and modify a data block.
$-$ *Insert a data block B.* After having downloaded the whole file parity $\mathtt{P}_{i-1}$ (*i.e.*, $\Delta D_{i-1} = \mathtt{P}_{i-1}$), C runs PrepareUpdate. C first restores the original order of the parity symbols in $\mathtt{P}_{i-1}$ and then strips off PRF g from them. For each symbol b in B, C updates the constraint group with index $h_K(\mathtt{b})$ by appending b to the data symbols of the constraint group (cf. Sec. III-B). C obtains $\mathtt{P}_i$ by using PRF g to mask all the symbols in the file parity P and by using PRP φ to permute

[2] Knowledge of the (n, k) parameters is required for the operations in the Update and Retrieve phases.

KeyGen(1^κ):

1. $\{sk_{DPDP}, pk\} \leftarrow$ KeyGen_DPDP(1^κ), and $K, K' \xleftarrow{R} \{0,1\}^\kappa$
2. Return $\{sk = \{sk_{DPDP}, K, K'\}, pk\}$

PrepareUpdate($sk, pk, \Delta F, \Delta D_{i-1}, info, M_c$):

1. If $info$ ="$full\ re\text{-}write$ " /*occurs in the Setup phase, to prepare the original file for outsourcing. ΔF is the original version of the file, $i = 1$*/
   - P = ComputeParityData($sk, pk, \Delta F, NULL, 0$)
   - $\Delta D = \Delta F || P$
   - $info'$ = "$full\ re\text{-}write$"
   - Return PrepareUpdate_DPDP($sk_{DPDP}, pk, \Delta D, info', M_c$)

   /*For ease of presentation, we only consider block-level updates (this can be easily extended to arbitrary file portions), thus, in the following, the block $B = \Delta F$. If $info$ is "delete" or "modify" a block, then ΔD_{i-1} is $P_{i-1}||B'$, and B' denotes the block to be deleted or modified. If $info$ is "insert" a block, then ΔD_{i-1} is P_{i-1}.*/
2. Restore the original order of symbols in P_{i-1} and strip off PRF g from them
3. If $info$ = "$insert\ B$"
   - P_i = ComputeParityData($sk, pk, B, P_{i-1}, 1$)
   - $\Delta D = P_i || B$
   - $info'$ = "$replace\ P_{i-1}\ with\ P_i,\ insert\ block\ B$"
4. Else if $info$ = "$delete\ B'$ "
   - P_i = ComputeParityData($sk, pk, B', P_{i-1}, 2$)
   - $\Delta D = P_i$
   - $info'$ = "$replace\ P_{i-1}\ with\ P_i,\ delete\ block\ B'$"
5. Else if $info$ = "$modify\ B' to\ B$"
   - P_i = ComputeParityData($sk, pk, B'||B, P_{i-1}, 3$)
   - $\Delta D = P_i || B$
   - $info'$ = "$replace\ P_{i-1}\ with\ P_i,\ modify\ B'\ to\ B$ "
6. Return PrepareUpdate_DPDP($sk_{DPDP}, pk, \Delta D, info', M_c$)

PerformUpdate($pk, D_{i-1}, M_{i-1}, e(\Delta D), e(info), e(\Delta M)$):
Return PerformUpdate_DPDP($pk, D_{i-1}, M_{i-1}, e(\Delta D), e(info), e(\Delta M)$)

VerifyUpdate($sk, pk, \Delta F, \Delta D_{i-1}, info, M_c, M_c', P_{M_c'}$):

1. Re-compute ΔD and $info'$ according to the procedure in PrepareUpdate /*In fact, ΔD and $info'$ can directly be stored at the client after PrepareUpdate, and there is no need to re-compute them*/
2. Return VerifyUpdate_DPDP($sk_{DPDP}, pk, \Delta D, info', M_c, M_c', P_{M_c'}$)

GenChallenge(sk, pk, M_c): Return GenChallenge_DPDP(sk_{DPDP}, pk, M_c)

Prove(pk, D_i, M_i, c): Return Prove_DPDP(pk, D_i, M_i, c)

Verify(sk, pk, M_c, c, Π): Return Verify_DPDP($sk_{DPDP}, pk, M_c, c, \Pi$)

Decode($sk, pk, D_i = F_i||P_i, M_i, M_c$):

1. Check the freshness of the retrieved file using the dynamic verification structure (*e.g.*, for DPDP that uses a skip list, re-compute the root of the skip list using the verification tags as leaves and compare to the root stored at the client, which is part of M_c [10]). If the check fails, then return $failure$.
2. Check the data blocks in F_i using the corresponding verification tags in M_i and discard the corrupted blocks. If there are no corrupted data blocks, return F_i. Otherwise, assign the symbols in healthy data blocks to their constraint groups (*i.e.*, for a symbol b, the index of its constraint group is $h_K(b)$).
3. Re-order all the parity symbols in P_i, strip off PRF g from them, and put them back to their right locations in the corresponding constraint groups based on the (n, k) parameters of each constraint group.
4. For each (n, k) constraint group, apply brute force decoding as follows. Let k' be the number of healthy symbols that have been assigned to this constraint group. Consider all permutations of k' symbols in k' locations (out of k locations for data symbols), together with $k-k'$ parity symbols (out of d parity symbols): (a) apply RS decoding on the k symbols to recover the original data symbols; (b) re-compute the parity symbols; (c) if at least one of the newly computed parity symbols match the parity symbols retrieved from the server, then the decoding for this constraint group is considered successful. If there are no successful decodings, then further consider ℓ of the $k-k'$ locations as zero symbols (with $1 \le \ell \le k-k'$), together with $k-k'-\ell$ parity symbols (out of d parity symbols), and further decode the k symbols as previously described. If there are still no successful decodings, then return $failure$.
5. Return the successfully decoded file F_i.

Fig. 2: VLCG: an R-DPDP scheme

ComputeParityData($sk, pk, \mathcal{B}, P_{i-1}, flag$):
(run by the client to compute the parity in Setup phase or update the parity in Update phase)

1. If $flag$ = 0 /*compute the parity for the original file in Setup phase. $\mathcal{B}$ represents the original file*/
   - For each symbol b in file $\mathcal{B}$, compute $h_K(b)$ to determine to which constraint group will b be assigned
   - For each constraint group with k symbols (k may be different for different groups), apply a $(k+d, k)$ RS code
   - Let P_i be the collection of all the md parity symbols
   - All the parity symbols in P_i are masked using PRF g and permuted using PRP φ, both keyed with K'
   - Return P_i
2. Else if $flag$ = 1 /*insertion, $\mathcal{B}$ represents the block to be inserted*/
   - For each symbol b in block $\mathcal{B}$: update in P_{i-1} the constraint group with index $h_K(b)$ by appending b to the data symbols of this constraint group
   - $P_i = P_{i-1}$
3. Else if $flag$ = 2 /*deletion, $\mathcal{B}$ represents the block to be deleted*/
   - For each symbol b in block $\mathcal{B}$: update in P_{i-1} the constraint group with index $h_K(b)$ by modifying b to the zero symbol
   - $P_i = P_{i-1}$
4. Else if $flag$ = 3 /*modification of B' to B, $\mathcal{B} = B'||B$*/
   - For each symbol b in block B': update in P_{i-1} the constraint group with index $h_K(b)$ by setting b to be the zero symbol
   - For each symbol b in block B: update in P_{i-1} the constraint group with index $h_K(b)$ by appending b to the data symbols of this constraint group
   - $P_i = P_{i-1}$
5. All the parity symbols in P_i are masked using PRF g and permuted using PRP φ, both with a new key K' (the previous K' will be discarded after having verified the update successfully)
6. Return P_i

Fig. 3: Computing the parity symbols in VLCG

them (g and φ are keyed with a new key K'). PrepareUpdate then calls the PrepareUpdate_DPDP algorithm of DPDP to further process the data $\mathtt{P}_i||B$. The output of PrepareUpdate is sent to S. S runs PerformUpdate and sends back the proof for updating the corresponding data correctly. C then runs VerifyUpdate to check the proof. If successful, C discards $\mathtt{P}_{i-1}$, B, and the old key K', and updates the (n, k) parameters for the corresponding constraint groups to $(n+1, k+1)$; otherwise, C aborts the protocol and raises an alarm.

– *Delete a data block B'.* After having downloaded the whole parity data $\mathtt{P}_{i-1}$ and the block B' that is to be deleted (*i.e.*, $\Delta D_{i-1} = \mathtt{P}_{i-1}||B'$), C runs PrepareUpdate. It first restores the original order of the parity symbols in $\mathtt{P}_{i-1}$ and strips off PRF g from them. For each symbol $\mathtt{b}$ in B', it updates the constraint group with index $h_K(\mathtt{b})$ by modifying the value of $\mathtt{b}$ to be zero (cf. Sec. III-B). C obtains $\mathtt{P}_i$ by using PRF g to mask all the symbols in the file parity $\mathtt{P}$ and by using PRP φ to permute them (g and φ are keyed with a new key K'). PrepareUpdate then calls the PrepareUpdate_DPDP algorithm of DPDP to further process $\mathtt{P}_i$. The output of PrepareUpdate is sent to S. PerformUpdate and VerifyUpdate are run just like in the insert a block operation above, except that there is no need to update the (n, k) parameters after C verifies the update successfully.

– *Modify a data block B' to B.* Modifying a data block B' to B is equivalent to first deleting the old block B' and then inserting the new block B.

For each of these operations (insert, delete, modify a block), the client discards the previous key K' after running VerifyUpdate and replaces it with the new K'. Thus, C only stores a constant amount of key material.

Retrieve. C downloads D_i (the latest version of the encoded file $\mathtt{D}$) and metadata M_i. C then applies the Decode algorithm. If Decode returns $failure$, C should raise an alarm.

The Decode algorithm relies on brute force decoding to recover the file (VLCG can tolerate up to $d-1$ erasures in each constraint group). We argue this is not a major concern because, first of all, file recovery is usually a rare event;

secondly, in Sec. IV-D we present an optimization that trades-off client computation during decoding for additional server storage. We also note that during Decoding, verification tags are used to determine the healthy data blocks and only symbols in healthy data blocks are assigned to their constraint groups (thus, if the server swaps two symbols in the data file, the corresponding blocks will be considered corrupted).

D. Discussion

Unlike πR-D in which all the constraint groups have the same fixed size, in VLCG the length of each constraint group is variable. The client can determine which symbols belong to the same constraint group by keeping track of the set Φ of (n, k) parameters for all the constraint groups. A basic approach is to store Φ at the client, which will result in $O(m)$ additional client storage. An alternative approach is to store Φ at the server and securely manage it using a Merkle hash tree; the client only stores the root of the tree, thus preserving the $O(1)$ client storage overhead; the communication overhead is increased by $O(m)$ during challenges of data possession, and by $O(\log m)$ during updates.

The number k of data symbols in a constraint group should be kept relatively small, so that the (n, k) RS code can be computed efficiently over the constraint group. Once the n and k parameters are fixed, the number of constraint groups m is determined as $m = f/k$. As updates are performed, m remains the same, but the number of data symbols k in each constraint groups may change (though the level of fault tolerance $d = n - k$ is preserved). Since delete operations do not reduce the size of the (n, k) RS code, k will increase over time due to insert/modify operations. To avoid a prohibitive increase of k and keep the (n, k) RS codes efficient to compute, as well as keep the zero symbols as few as possible (to keep the brute force computation in the Decode algorithm at a minimum), the client C should periodically perform a *dynamic adjustment*: after a certain number of updates C retrieves the file and runs Setup to pre-process the file again; in this process, C may pick a different m, depending on the size of the updated file. The amortized bandwidth factor for updates will remain $\alpha = \frac{|\mathtt{P}|}{|\mathtt{D}|}$.

We now describe one step that was previously omitted to simplify the description of the VLCG scheme. The assignment of data symbols to constraint groups is done based on the value of each symbol. To ensure robustness, the server should not know which symbols belong to the same constraint group. The client applies a layer of encryption before storing the file at the server in order to hide that two equal symbols are mapped to the same constraint group. Specifically, in the Setup phase, the succession of steps for the client is: (1) encode the file and obtain the parity, (2) encrypt the file blocks (using randomized encryption), (3) mask and permute the parity, (4) generate verification metadata over the encrypted file and parity, (5) store the encrypted file, parity, and metadata at the server. As a result, the client needs to remove this encryption layer whenever it gets file data from the server, and add the encryption whenever it stores blocks at the server.

Finally, we remark that our VLCG construction is better suited for files with random data in order to ensure that PRF h provides a balanced distribution of symbols into constraint groups. For instance, files could be encrypted before applying the PRF, and then be stored encrypted at the server (this may be desirable anyway if the data is of sensitive nature).

Security analysis for VLCG. The main difference between VLCG and πR-D in the Setup phase is that VLCG determines the constraint groups by applying a PRF over the content of data symbols, whereas πR-D determines them by applying a PRF over the indices of the data symbols. The security properties of the PRF used to decide the constraint groups and of the randomized encryption applied on the file data before being stored at the server, ensure that the server can infer which symbols are in the same constraint group with negligible probability. Moreover, VLCG masks and permutes the parity in a similar fashion with the permuting and encrypting of the parity in πR-D, ensuring a similar δ-robustness guarantee as for πR-D (see security analysis for πR-D).

The security of the Challenge and Update phases in VLCG relies on the security of the underlying DPDP scheme (which ensures that large corruptions will be detected based on spot-checking and that the server possesses the latest version of the file based on the authenticated structure).

In the Update phase, for insert/delete operations, VLCG only downloads the parity, updates the corresponding parity symbols, then re-masks and re-permutes the parity with a new key, which is comparative to πR-D, in which although the whole file data is downloaded (for the purpose of re-computing the parity), only the newly generated parity is re-permuted and re-encrypted with a new key; for modify operations, both VLCG and πR-D only download the parity (rather than the whole file data) and update the parity correspondingly. In the Retrieve phase, both schemes rely on the same mechanisms (metadata and authenticated data structure) to detect corruption. This implies that VLCG and πR-D provide a similar security level. We leave a rigorous security analysis of VLCG as future work.

Performance analysis for VLCG. For every update operation (insertion, deletion, and modification), the update bandwidth factor α for VLCG is approximately $\frac{|\mathtt{P}|}{|\mathtt{D}|}$, whereas for πR-D α is approximately $\frac{|\mathtt{F}|}{|\mathtt{D}|}$ for insertion/deletion. Since d is usually small compared to k (*i.e.*, $|\mathtt{P}|$ is a lot smaller than $|\mathtt{F}|$), the communication overhead for updates in VLCG is significantly smaller than in πR-D.

The encoding computation for VLCG is close to that of πR-D, as expressed in the following theorem (proof provided in App. A):

Theorem 4.2: Let f be the number of symbols in the file $\mathtt{F}$. If $C_1(f)$ is the computation for encoding in πR-D, and $C_2(f)$ is the computation for encoding in VLCG, then we have: $C_2(f) = \Theta(C_1(f))$.

Optimizing the worst-case computation for decoding in VLCG. The Decode algorithm in VLCG has a high worst-case computation, since we do not know the correct positions of healthy data symbols in their corresponding constraint groups. We propose to record the position of each data symbol in its constraint group. For a file $\mathtt{F}$ with f symbols, we

maintain $\mathbf{V}$ a vector of positions with f elements. The vector $\mathbf{V}$ is stored encrypted at the server (each element in $\mathbf{V}$ is encrypted independently), which prevents the server from learning information about constraint groups. $\mathbf{V}$ is regarded as part of the file data during the Setup and Challenge phases (thus, there will be verification tags computed over blocks made of elements of $\mathbf{V}$).

In the Update phase, changes on the file symbols should be mirrored by changes in $\mathbf{V}$: whenever symbols are inserted, deleted, or modified in the file, a corresponding element in $\mathbf{V}$ should be inserted, deleted, or modified, respectively (reflecting the symbol's new position in its constraint group).

In the Retrieve phase, $\mathbf{V}$ is retrieved together with all the data D. After having checked the verification tags on $D||\mathbf{V}$, the elements in $\mathbf{V}$ which have been found corrupted are discarded. The healthy elements in $\mathbf{V}$ will indicate the position of each symbol inside its constraint group. A lightweight brute force computation may still be required because, first of all, for the corrupted elements in $\mathbf{V}$, the client will lose the position information of the corresponding (healthy) data symbols, thus, the positions of these symbols in their constraint groups would have to be determined by brute force; secondly, symbols with 0 value introduced by "delete" and "modify" operations may exist in some constraint groups, and the right positions for such symbols (if needed) will be determined by brute force search. The parameters of the RS code should be selected to ensure that the brute force search during Decode remains reasonable (*e.g.*, for a code with $n = 140$, $k = 128$, $d = 12$, which requires to spot-check 1188 blocks in the Challenge phase to guarantee high probability of corruption detection [2], the worse case running time for brute force is $\binom{140}{12}$, which is approximately 2^{38}). This computation is reasonable considering that data retrieval is a rare event.

Further optimizing the update communication. Compared to πR-D, which requires to download the entire file for every update, VLCG only needs to download the parity symbols. Although the parity is usually very small compared to the whole data, the asymptotic communication per update is $O(f)$, where f is the number of file symbols. To further optimize the update communication to $O(\log^2 f)$, we can use Oblivious RAM (ORAM) techniques [33]. To update one symbol, instead of downloading the parity $\mathtt{P}$ over the entire file, we use ORAM over $\mathtt{P}$ to only retrieve the d parity symbols of the constraint group corresponding to the updated symbol. These symbols are updated (cf. Sec. IV-C1) and then written back to the server using ORAM. For example, when using the ORAM scheme in [34], the amortized communication for every update will be reduced to $O(\log^2 f)$, at the cost of a slight increase in server storage (asymptotically, the server storage remains $O(f)$) and server computation (by $O(\log^2 f)$).

Verification tags in VLCG. Remote data checking schemes designed for the static case [1], [4] embed the index of a block (*i.e.* its position in the file) inside the corresponding verification tag. In order to support efficient dynamic updates, verification tags in DPDP [10] are fundamentally different, as they do not embed the block indices inside the verification tags. In DPDP, the tag for a block B is computed as $g^B \mod N$, where $N = pq$ is a product of two large primes p, q, and g is an element of high order in $\mathbb{Z}_N^*$. Since N should be at least 1024 bits, when the block size is smaller than 1024 bits the size of a tag will be larger than the size of a data block, which may result in an undesirably high additional storage overhead. [22] provides another tag construction which can avoid this issue, but the tags will be of the same size as the data blocks and the verification process is based on expensive bilinear maps. In Appendix B, we provide an alternative to compute the verification tags in a prime-order field and the size of the verification tags will be smaller than the data even when choosing the block size as small as a few hundred bits.

V. Conclusion

Adding protection against small corruptions to remote data checking schemes that support dynamic updates extends the range of applications that rely on outsourcing data at untrusted servers. In this paper, we have proposed the first Robust Dynamic Provable Data Possession (R-DPDP) schemes that provide robustness and, at the same time, support dynamic data updates, while requiring small, constant, client storage. The main challenge that had to be overcome was to reduce the client-server communication overhead during updates under an adversarial setting. This work initiates the study of R-DPDP schemes, but more investigation is required in order to improve the efficiency of our VLCG construction (such as further reducing the update bandwidth factor and the computation required by the brute force search during data retrieval).

Acknowledgment

A preliminary version of this paper appeared in [35]. This research was sponsored by the US National Science Foundation CAREER grant 1054754-CNS.

References

[1] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song, "Provable data possession at untrusted stores," in *Proc. of ACM CCS*, 2007.

[2] G. Ateniese, R. Burns, R. Curtmola, J. Herring, O. Khan, L. Kissner, Z. Peterson, and D. Song, "Remote data checking using provable data possession," *ACM Trans. Inf. Syst. Secur.*, vol. 14, June 2011.

[3] A. Juels and B. S. Kaliski, "PORs: Proofs of retrievability for large files," in *Proc. of ACM CCS*, 2007.

[4] H. Shacham and B. Waters, "Compact proofs of retrievability," in *Proc. of Asiacrypt 2008*, 2008.

[5] R. Curtmola, O. Khan, R. Burns, and G. Ateniese, "MR-PDP: Multiple-replica provable data possession," in *Proc. of ICDCS*, 2008.

[6] K. Bowers, A. Oprea, and A. Juels, "HAIL: A high-availability and integrity layer for cloud storage," in *Proc. of ACM CCS*, 2009.

[7] C. Wang, Q. Wang, K. Ren, and W. Lou, "Ensuring data storage security in cloud computing," in *Proc. of IWQoS*, 2009.

[8] B. Chen, R. Curtmola, G. Ateniese, and R. Burns, "Remote data checking for network coding-based distributed storage systems," in *Proc. of ACM CCSW*, 2010.

[9] G. Ateniese, R. D. Pietro, L. V. Mancini, and G. Tsudik, "Scalable and efficient provable data possession," in *Proc. of SecureComm*, 2008.

[10] C. Erway, A. Kupcu, C. Papamanthou, and R. Tamassia, "Dynamic provable data possession," in *Proc. of ACM CCS*, 2009.

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

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

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

[14] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao, "Oceanstore: an architecture for global-scale persistent storage," *SIGPLAN Not.*, vol. 35, pp. 190–201, November 2000.

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

[16] S. Kamara and K. Lauter, "Cryptographic cloud storage," in *Workshop on Real-Life Cryptographic Protocols and Standardization*, 2010.

[17] S. Kamara, C. Papamanthou, and T. Roeder, "CS2: A semantic cryptographic cloud storage system," Microsoft Research, Tech. Rep. MSR-TR-2011-58, 2011.

[18] R. Curtmola, O. Khan, and R. Burns, "Robust remote data checking," in *Proc. of ACM StorageSS*, 2008.

[19] K. D. Bowers, A. Juels, and A. Oprea, "Proofs of retrievability: Theory and implementation," in *Proc. of the 2009 ACM workshop on Cloud computing security (CCSW '09)*, 2009.

[20] E. Stefanov, M. van Dijk, A. Oprea, and A. Juels, "Iris: A scalable cloud file system with efficient integrity checks," IACR ePrint Cryptography Archive, Tech. Rep. 2011/585, 2011.

[21] Q. Zheng and S. Xu, "Fair and dynamic proofs of retrievability," in *Proc. of the ACM CODASPY '11*, 2011.

[22] Q. Wang, C. Wang, K. Ren, W. Lou, and J. Li, "Enabling public auditability and data dynamics for storage security in cloud computing," *IEEE Trans. on Parallel and Distributed Syst.*, vol. 22, no. 5, May 2011.

[23] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *Journal of the Society for Industrial and Applied Mathematics*, vol. 8, no. 2, pp. 300–304, 1960.

[24] J. S. Plank and L. Xu, "Optimizing cauchy reed-solomon codes for fault-tolerant network storage applications," *Proc. of IEEE NCA '06*, 2006.

[25] M. O. Rabin, "Efficient dispersal of information for security, load balancing, and fault tolerance," *J. of the ACM*, vol. 36, no. 2, 1989.

[26] J. Blomer, M. Kalfane, R. Karp, M. Karpinski, M. Luby, and D. Zuckerman, "An xor-based erasure-resilient coding scheme," International Computer Science Institute, Tech. Rep. TR-95-048, August 1995.

[27] J. S. Plank, "Erasure codes for storage applications," Tutorial Slides, presented at *FAST-2005: 4th Usenix Conference on File and Storage Technologies*, http://www.cs.utk.edu/~plank/plank/papers/FAST-2005.html, San Francisco, CA, 2005.

[28] J. S. Plank and Y. Ding, "Note: Correction to the 1997 tutorial on reed-solomon coding," 2003.

[29] J. S. Plank, S. Simmerman, and C. D. Schuman, "Jerasure: A library in C/C++ facilitating erasure coding for storage applications - Version 1.2," University of Tennessee, Tech. Rep. CS-08-627, August 2008.

[30] J. S. Plank, "Optimizing Cauchy Reed-Solomon codes for fault-tolerant storage applications," University of Tennessee, Tech. Rep. CS-05-569, December 2005.

[31] "OpenSSL," http://www.openssl.org/.

[32] "Eclipse," http://archive.eclipse.org/arch/.

[33] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious rams," *Journal of the ACM*, vol. 43, no. 3, May 1996.

[34] B. Pinkas and T. Reinman, "Oblivious ram revisited," in *CRYPTO 2010*.

[35] B. Chen and R. Curtmola, "Robust dynamic provable data possession," in *Proc. of the Third IEEE International Workshop on Security and Privacy in Cloud Computing (SPCC '12)*, 2012.

[36] D. Boneh, D. Freeman, J. Katz, and B. Waters, "Signing a linear subspace: Signature schemes for network coding," in *Proc. of PKC '09*.

APPENDIX A
PROOF OF THEOREM 4.2

In πR-D, the encoding computation of one constraint group is approximately $c(\frac{f}{m})^2$ (the computation for Cauchy RS encoding is approximately quadratic. d is small compared to $\frac{f}{m}$), while c is constant, thus, $C_1(f) \approx m * c(\frac{f}{m})^2 = c\frac{f^2}{m}$.

In VLCG, the f symbols are distributed to m groups by using PRF h. Assume the number of data symbols in the m constraint groups are k_1, k_2, ..., k_m. Then, $C_2(f) \approx c(k_1^2 + k_2^2 + ... + k_m^2)$.

Let X be a random variable that denotes the number of data symbols in one constraint group. We have:

The expected value of X: $E(X) = \frac{f}{m}$

The variance of X: $Var(X) = E(X^2) - (E(X))^2$

$$\begin{aligned} C_2(f) &\approx c(k_1^2 + k_2^2 + ... + k_m^2) \approx c(mE(X^2)) \\ &= cm(E(X)^2 + Var(X)) = cm((\frac{f}{m})^2 + Var(X)) \\ &= c(\frac{f^2}{m} + mVar(X)) \end{aligned}$$

$\lim_{f\to\infty}(\frac{C_2(f)}{C_1(f)}) = \lim_{f\to\infty}(1 + Var(X)\frac{m^2}{f^2}) \approx 1$ (if h is a good PRF and the input for h is random enough, $Var(X)$ will be approximately constant). Thus, $C_2(f) = \Theta C_1(f)$.

APPENDIX B
AN ALTERNATIVE FOR COMPUTING THE VERIFICATION TAGS FOR DPDP

We adopt the tag construction method from [36]. Suppose every file block contains s symbols (*i.e.*, block m_i consists of symbols $m_{i1}, m_{i2}, ..., m_{is}$). Choose a group $\mathbb{G}$ of prime order p, with $p > max(2^\lambda, 2^w)$ (λ is a security paramter, and w is the length of the symbol). Choose generators $g_i \xleftarrow{R} \mathbb{G}$ for $i = 1, .., s$. The public key $pk = (p)$, and secret key $sk = (g_1, ..., g_s)$.

In the Setup phase, the tag $\mathcal{T}_{m_i}$ for the block m_i is computed as $\mathcal{T}_{m_i} = \Pi_{k=1}^s g_k^{m_{ik}}$.

In the Challenge phase, the client sends to the server pairs (i_j, a_j), for $1 \le j \le C$, where C is the number of blocks to be challenged. The server runs Prove_DPDP and returns a proof (T, μ_k), where

$$T = \Pi_{j=1}^C \mathcal{T}_{m_{i_j}}^{a_j}, \;\; \mu_k = \sum_{j=1}^C a_j m_{i_j k}$$

for $1 \le k \le s$ ($m_{i_j k}$ is the k-th symbol in block m_{i_j}).

The client runs Verify_DPDP and accepts if $T = \Pi_{k=1}^s g_k^{u_k}$ and if M_c verifies.

Proof of correctness:

$$\begin{aligned} T &= \Pi_{j=1}^C \mathcal{T}_{m_{i_j}}^{a_j} = \Pi_{j=1}^C (\Pi_{k=1}^s g_k^{m_{i_j k}})^{a_j} = \Pi_{k=1}^s g_k^{\sum_{j=1}^C a_j m_{i_j k}} \\ &= \Pi_{k=1}^s g_k^{u_k} \end{aligned}$$