POSTER: Robust Dynamic Remote Data Checking for Public Clouds

Bo Chen, Reza Curtmola
Department of Computer Science
New Jersey Institute of Technology
{bc47,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 public cloud, making RDC a valuable tool for data auditing. 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. In this paper, we propose the first RDC schemes that provide robustness and, at the same time, support dynamic updates, while requiring small, constant, client storage.

Categories and Subject Descriptors

H.3.2 [**Information Storage and Retrieval**]: Information Storage; E.4 [**Coding and Information Theory**]: Error Control Codes

General Terms

Security, Design, Performance

Keywords

cloud storage, remote data integrity checking, dynamic provable data possession, robustness, small corruption

1. 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 for static data, including Provable Data Possession (PDP) [1,2] and Proofs of Retrievability (PoR) [9, 12], both for the single server [2, 9, 12] and for the multiple server setting [3, 5, 7, 13]. RDC schemes have also been proposed for the *dynamic* setting (DPDP) [8], which supports updates on the outsourced data (insertions, modifications, appends, and deletions).

A scheme for auditing remote data should be both *lightweight* and *robust* [1]. *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.

Robustness is usually achieved by integrating forward error-correcting codes (FECs) with remote data checking [1,6]. 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 dynamic remote data checking schemes 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.

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.
- We identify the challenges that need to be overcome when trying to add robustness to a DPDP scheme in an adversarial setting. 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, $RDC1$, 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, $RDC2$, overcomes this drawback by: (a) decoupling the encoding for

Copyright is held by the author/owner(s).
CCS'12, October 16–18, 2012, Raleigh, North Carolina, USA.
ACM 978-1-4503-1651-4/12/10.

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 $RDC2$ over $RDC1$ is beneficial, as our source code analysis of a few popular software projects shows that insert/delete operations represent a majority of all updates [4].

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.

2. BACKGROUND

2.1 Remote Data Checking Schemes

A remote data checking scheme consists of four phases, Setup, Challenge, Update, 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 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 auditor uses the secret key material K to check the validity of the proof. During Update, the data owner securely updates the outsourced data. During Retrieve, the data owner recovers the original data.

2.2 Cauchy Reed-Solomon Code

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 $2E + S \leq d$. 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 [11], for two reasons: they are more suitable to handle dynamic operations on the original data, and they were shown to be approximately twice as fast as the classical Reed-Solomon encoding based on Vandermonde matrices.

Encode. We take a (6, 4) Cauchy RS code as example. 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 [10] 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, which is composed of the identity matrix in the first 4 rows and Cauchy matrix in the remaining 2 rows, is as follows:

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

The codeword C is computed as

$$C = M \times \mathbf{L}^{\mathrm{T}} = (\ b_1 \quad b_2 \quad b_3 \quad b_4 \quad p_1 \quad p_2\ )^{\mathrm{T}}$$

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

Cauchy RS codes have some useful properties: When appending/modifying a symbol, we only need the parity symbols and the new symbol to update the codeword. By retrieving only the parity symbols, the update communication can be significantly reduced. Our $RDC2$ solution will utilize these properties.

3. ROBUST DYNAMIC REMOTE DATA CHECKING

Preliminaries. The outsourced file is seen as a collection of symbols, which are grouped into chunks. Each chunk is encoded independently and we call the resulting codewords *constraint groups*. To achieve robustness, the association between symbols and constraint groups must be hidden from the possibly malicious cloud servers.

R-DPDP Definition. The definition of an R-DPDP scheme is presented in the full version of this paper [4].

The $RDC1$ scheme: Our first robust dynamic remote data checking scheme, $RDC1$, extends our previous work [6] to a dynamic setting.

Setup phase: The file is divided into k-symbol chunks, and a (n, k) Cauchy RS code is computed over the symbols from every chunk (*i.e.*, one chunk will result in one constraint group). The encoded file is $\tilde{\mathtt{F}} = \mathtt{F}||\mathtt{P}$, where $\mathtt{F}$ is the original file and $\mathtt{P}$ is the parity. To achieve robustness for $\tilde{\mathtt{F}}$, we: (1) determine the chunk to which a symbol is assigned by applying a Pseudo Random Permutation (PRP) ψ over the index of the symbol, and (2) permute and encrypt all the parity symbols. We then compute a verification tag for every block in $\tilde{\mathtt{F}}$, and a Merkle hash tree T over all the verification tags. The encoded file as well as the tags are outsourced to the server. The client keeps the root of T and a small amount of secret key material.

Challenge phase: The client randomly samples a small number of blocks (*e.g.*, 400) from the server. By checking the relationship between the sampled blocks and their verification tags, as well as checking the root of T, the client can detect data corruption with high probability [1].

Update phase: To insert/delete a symbol into/from $\mathtt{F}$ (outsourced to the server), $\mathtt{P}$ should be updated. Since $RDC1$ relies on a PRP over the indices of symbols in $\mathtt{F}$ to determine their constraint groups, inserting/deleting a file symbol requires to re-compute the whole $\mathtt{P}$. Thus, for insert/delete operations, the whole $\mathtt{F}$ needs to be retrieved. To modify a symbol in $\mathtt{F}$ does not affect the indices, and only the parity in the corresponding constraint group must be re-computed. Thus retrieving $\mathtt{P}$ is sufficient for modify operations.

Retrieve phase: The client simply retrieves all the data and may use $\mathtt{P}$ to correct data corruption.

The $RDC2$ scheme. Though efficient in encoding, $RDC1$ has a high communication overhead for updates because 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 to re-encode the whole file, thus requiring to retrieve the entire previous version of the file).

To mitigate the drawbacks of $RDC1$, we propose a second scheme, $RDC2$, in which we still use the notion of *constraint groups*. However, we rely on two additional main insights.

Firstly, unlike in $RDC1$, in which symbols are assigned to constraint groups based on the position (*i.e.*, index) of the symbols in the file, $RDC2$ assigns symbols to constraint groups based on the

value of the symbols. More precisely, for a data symbol b, the client uses $h_K(b)$ to decide the index of the constraint group to which b belongs, where h is a cryptographic hash function and K is a secret key. This has the advantage that, when inserting/deleting a data symbol into/from the file, we only need to update the parity symbols from the corresponding constraint group, without affecting other constraint groups.

Secondly, we employ several techniques to preserve robustness and minimize the bandwidth overhead. To utilize the useful properties of Cauchy RS codes, we reduce insert operations to append operations, and delete operations to modify operations when updating the RS-coded parity data. Inserting a symbol to F is equivalent to inserting this symbol into the corresponding constraint group. Our strategy is to update the parity symbols of the corresponding constraint group *as if the symbol was appended to the end of the data symbols in that constraint group*. Deleting a symbol from F is equivalent to deleting this symbol from the corresponding constraint group. We use another strategy: To delete a data symbol, we ask the server to physically delete the symbol from F, but we update the parity symbols from the corresponding constraint group *as if that symbol was modified to have the value 0*. This results in a more expensive Retrieve procedure (a more complex decoding algorithm is required to recover the file from corruptions), but we argue this is not a major concern because file recovery is usually a rare event.

4. 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 Remote Data Checking schemes that provide robustness and, at the same time, support dynamic data updates, while requiring small, constant, client storage. The main challenge was to reduce the client-server communication overhead during updates under an adversarial setting. This work initiates the study of robust dynamic remote data checking, but more investigation is required in order to improve the efficiency of our $RDC2$ solution (such as further reducing the update bandwidth factor and the computation required by the brute force search during data retrieval).

5. ACKNOWLEDGEMENTS

The full version of this paper [4] contains a detailed description. This research was sponsored by the US National Science Foundation CAREER grant 1054754-CNS.

6. REFERENCES

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

[2] 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.

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

[4] B. Chen and R. Curtmola. Robust dynamic provable data possession. In *Proc. of ICDCS-SPCC*, 2012.

[5] 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.

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

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

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

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

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

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

[12] H. Shacham and B. Waters. Compact proofs of retrievability. In *Proc. of Asiacrypt*, 2008.

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