International Journal on Applications in Basic and Applied Sciences
Volume 5 : Issue 1: December 2019, pp 15 – 19 www.aetsjournal.com ISSN (Online) : 2455 - 1007

ENERGETIC PROOFS OF RETRIEVABILITY PATTERN IN CLOUD COMPUTING

Dr.D.SATISHKUMAR, Dr. SIVAKUMAR.K , Mr. D. KALEESWARAN

***Abstract* —** In the recent years cloud storage brought many interesting security issues to user attention. Two efficient storage integrity protocols Proofs of Retrievability (PoR) and Provable Data Possession (PDP) are mainly used. The first proposals worked for static or limited dynamic data, whereas later proposals enabled fully dynamic data integrity and retrievability. The DPoR framework encapsulates all known DPoR schemes as its special cases that enable even better performance than the existing solutions. The audit bandwidth for DPoR that is independent of the data size, and the client can greatly speed up updates with O (lpn) local storage (where n is the number of blocks, and l is the security parameter), which corresponds to less than 3 MB for 10 GB outsourced data, and can easily be obtained in today's smart phones, let alone computers.

Keywords- Retrievability, DPoR, Protocols, Dynamic Data Integrity.

I. INTRODUCTION

The outsourcing of storage data through the cloud (e.g., Google Drive, Amazon S3, Microsoft One Drive), brings advantages such as cost saving, global access to data, and reduced management overhead. The important disadvantage is that the data owner (client), by outsourcing his data to a cloud storage provider (server), loses the direct control and maintenance over his data. So the client expects having an authenticated data storage and guaranteed retrievability.

Dr.D.SatishKumar PhD , Associate Professor , Department of Computer Science and Engineering,, Nehru Institute of Engineering and Technology (Email: satishcoimbatore@gmail.com)
Dr. Sivakumar K . Ph.D., MIEEE.Principal; Professor & Dean Computer Science and Engineering,Rathinam Technical Campus - Coimbatore (Email: rksivakumar@gmail.com)
Mr. D. Kaleeswaran M.E., Assistant Professor - Computer Science and Engineering , Arjun College of Tehnology - Coimbatore.
(Email: kaleeswaranme@gmail.com)

The former means that the client wants each storage data access to return the correct value; i.e., a value that is the most recent version of data that has been written by the client.

The latter means that the client wants to make sure that their data is retrievable. These authenticity and retrievability checks should be much more efficient than downloading the whole data.

A simple mechanism to provide an authenticated storage is to compute a digest (e.g., hash, MAC, or signature) of data and keep it locally after transferring the data to the server (or in case of a MAC or signature, the key is kept locally, while the tags can be stored at the server). But, the client needs to download the whole data and check it against the locally-stored digest to investigate the authenticity of their data, which is prohibitively given current trends of outsourcing tens of gigabytes of data.

II. RELATED WORK

PoR was first proposed by Juels and Kaliski [19] for static data. The data is erasure-coded and encrypted. Then, a set of sentinel blocks are appended, the result is permuted randomly, and outsourced. The sentinel blocks are used to check authenticity with high probability, and in case of any unauthorized manipulation, the erasure-correcting code will help recover the original data. Juels and Kaliski's PoR supports only a limited number of challenges.

The first really dynamic PoR scheme with full security definition and proof was proposed by Cash et al. [9]. The scheme has constant client storage and polylogarithmic communication and computation. As a building block,they use an ORAM satisfying a special property called next-read-pattern-hiding. Although it achieves asymptotic efficiency, ORAM is a complicated and heavy cryptographic primitive that is (currently) not practically efficient.

Chandran et al. [10] proposed a locally updatable and locally decodable code, and used it to construct a dynamic PoR scheme. They erasure-code the data, and store it remotely inside a hierarchical authenticated data structure similar to ORAM in nature. Later updates are also erasure-coded and stored in the same data structure. Reading through that structure requires O(n) cost, hence, they store the plain data and subsequent updates in another similar structure to support read operations efficiently.

Shi et al. proposed a dynamic PoR scheme similar to [10], using the fast incrementally-constructable codes to achieve efficiency. Later, they improved their scheme by outsourcing some part of computation to the server, reducing the communication and client computation. Using the introduced homomorphic checksum, the client only performs the computation on these checksums, leaving computation on data itself to the server.

III. PROPOSED SYSTEM

Erasure-Coded Authenticated Log is a log store plays an important role in the scheme. It is a special authenticated data structure (ADS) that inspects integrity and guarantees retrievability of the logs, which in turn, guarantee retrievability of the outsourced data. The ECAL first erasure-codes the logs (to guarantee retrievability), and garbles the result (e.g., by encrypting the blocks and permuting them randomly) to make locating any part of the original data difficult for the server. Finally, it provides authenticity using a homomorphic tag. Any scheme supplying retrievability and authenticity can be used to store the update logs and also PoR scheme provides efficient update and audit, without caring about the read efficiency (since it is used rarely, only for lost data recovery and infrequent rebuilds).

IV. IMPLEMENTATION

1) Dynamic Proof of Retrievability

The ECAL stores client data and guarantees its retrievability, it seems that the ECAL itself is a dynamic PoR scheme. However, due to the nature and application of the ECAL, reading a data block requires reading, decoding, and reconstructing all logs, necessitating O(n) cost. Therefore, read is not an efficient operation in ECAL, and it should not normally be fulfilled through the ECAL. This means that retrieving through the ECAL should be the last resort when other options fail. This is the important reason why ECAL is not an efficient DPoR scheme in its own. If read efficiency is not necessary, the ECAL can be used as a DPoR scheme. Access privacy is not a requirement in PoR definition [19], the client data can store in a dynamic memorychecking scheme, e.g., DPDP [15], preserving its authenticity. Read operations will be handled through this memorychecking part. But, update operations will affect both the memory-checking part and the ECAL part. This will solve the read inefficiency problem of the ECAL. This means that given a dynamic memory-checking scheme and a static PoR scheme, e.g., compact PoR [25]. Moreover, given an erasure-correcting code scheme and a static memory-checking scheme, e.g., PDP [1], (or any –homomorphic for efficiency– MAC or signature scheme), a static PoR [25] can be constructed. Therefore, a dynamic PoR scheme can be constructed given black box access to a dynamic memory-checking scheme, an erasure-correcting code scheme, and a static memory-checking scheme.

2) Dynamic PoR using ECAL

According to our observations on making PoR dynamic, storing updates inside the data is neither efficient nor secure. Therefore, we store the updates inside an ECAL scheme (which aims to support retrievability), separately from the data. Moreover, we use the update logs differently from [10, 25], as they only support modification on the original data, but we support insertion and deletion, too. They only require the last version of data blocks to reconstruct the data (the number of blocks is fixed), but we need the whole logs. The data itself is stored inside a dynamic memory-checking scheme (e.g., DPDP [15]) in plaintext form, since we apply later updates on and read through it. We refer to the DPDP part as ‘D’ and the ECAL part as ‘E’. An informal description of dynamic PoR operations is given below.

• Read is used to retrieve the most up-to-date version of the data at a specific location. Since D maintains the last version of the data, read can be done through D. Moreover, D provides an authenticity proof (of size O(llogn)) for all data blocks read, that works as the proof of retrievability for those blocks.

• Update is performed on the outsourced data, and brings both D and L in an up-to-date state. Updating D requires O(l) communication and O(logn) server computation, and updating L depends on the underlying ECAL structure. The equibuffers structure with O(p n) buffers of size (p n) each, and O(p n) client storage, performs updates with O(l) communication and O(1) server computation (amortized).

• Audit challenges and checks authenticity of the outsourced logs to see if the server keeps storing them intact. It challenges l random blocks from each non-empty buffer of L and verifies them. Since the challenge vector can bu generated by the server given the required keys, the client-to-server communication is O(l) [1]. Using the equibuffers setting, the server finds and aggregates the challenged blocks and their tags in O(lp n) time. The proof includes two values, and is of size O(l). The client verification time is also O(lp n).

• Periodic rebuild. Our scheme in the equibuffers setting eliminates the reshuffling operation, which is executed more frequently, and needs only the rare rebuilds. Moreover, we run the rebuild operation using the fresh data from D instead of combining and using the update logs, resulting in a much more efficient rebuild. The server sends the whole data from D to the client, which is an O(n) operation. The client first verifies the whole data with DPDP. If accepted, this guarantees that it is the correct last version of the data, and the logs can be discarded completely. She then runs LInit and uploads the result. Hence, the communication and the client computation are also O(n) at the worst case. Since this operation is executed once in every n updates, the amortized complexities will all be O(1).

3) Definition (Dynamic PoR)

A dynamic PoR scheme includes the following protocols (mostly from [9]) run between a stateful client and a stateful server. The client, using these interactive protocols, can outsource and later update her data at an untrusted server, while retrievability of her data is guaranteed (with overwhelming probability):

• PInit(1^{λ};1^{w};n;M): given the alphabet $\Sigma = \{0;1\}^{w}$ and the security parameter $^{\lambda}$, the client uses this protocol to initialize an empty memory of size n on the server, outsourcing there the initial data M.

• PUpdate(ί;OP; υ): the client performs the operation OP ϵ {I;D;M} on the ith location of the memory (on the server) with input value υ (if required).

• (υ;π) PRead(ί): is used to read the value stored at the ith location of the memory managed by the server. Theclient specifies the location ί as input, and outputs some value υ, and a proof p proving authenticity of υ.

• {*accept*,*reject*} ← *PAudit*(): The client starts this protocol to check if the server keeps
storing her data correctly. She emits an acceptance or a rejection signal.

4) Dynamic PoR Security Definitions

Since ECAL is a (inefficient) DPoR scheme, all its security definitions with proper protocol names are applicable here. In both games, the server e Ŝ asks the challenger to start a protocol execution (PRead, PUpdate or PAudit) by providing the required information.

5) DPoR Construction

Let *n*;*k* ϵ z^{+} (*k*<*n*), and $\Sigma ί=\{0,1\}^{w}$ and $Sm=\{0;1\}^{w}$ be two finite alphabets. The client is going to outsource a data $M=(m_1,\ldots,m_k) \epsilon\ \Sigma_m{}^{k}$. She stores M inside a DPDP construction D=(KeyGen,PrepareUpdate,Perform-Update,VerifyUpdate,Challenge,Prove,Verify). She also initializes an ECAL instantiation E=(LInit,LAppend,LAudit) to store the encoded logs. On each update, she updates both D and E, which support read and audit, respectively. Our dynamic PoR construction is as follows:

PInit(1^{λ};1^{w},*n*,M):

• The client runs (*pk*,*sk*) D:KeyGen(1^{λ}) and shares pk with the server.

• The client runs (*e*(*M*);*e*('full rewrite'), e(st'_c)) ← D:PrepareUpdate (*sk*,*pk*;M,'full rewrite',st_c).

• The server runs (M^1,st_s,st'_c ,$P_{st'c}$) ← *D*.PerformUpdate(pk, *e*(*M*);e('full rewrite');*e*(st'_c)), where M^1 is the first version of the client data hosted by the server, and st'_c and P $_{st'c}$ c are the client's metadata and its proof, respectively, computed by the server, to be sent to the client.The client executes D.VerifyUpdate(*sk*,*pk*,M;'full rewrite',st_c,st'_c,$P_{st'c}$), and outputs the corresponding acceptance or rejection notification.

• The client also stores the initial data ECAL:E:LInit(1^{λ},1^w,*n*,M).PUpdate(*i*;*OP*;*v*)

• The client runs (*e(v)*,*e(OP*, *i)*,*e*(st'_c) D.PrepareUpdate*(sk; pk,v,(OP,i),st_c)*.

•The server runs *(m^j,st_s,st'_c,Pst'_c)* D. Perform Update*(pk,e(m^{j-1}),e(OP,i),e(st'_c))*, where m^{j-1} is the current version of the data on the server (to be updated into m^j). The server sends st'_c and P st'_c to the client.

•The client executes D. Verify Update*(sk,pk,v,(OP,i),st_c,st'_c,P st'_c)*, and outputs the corresponding acceptance or rejection signal.

• The client, in parallel, prepares the corresponding log, *l = 'iOPv',* and runs E:LAppend(*l*). PRead(*i*):

• The client creates a DPDP challenge ch containing the block index *i* only, and sends it to the server. (Challenging only one block is a 'read' operation.)

•The server executes P← D:Prove*(pk,m_j,st_c;ch)* to generate and send the proof *P* (for the i^{th} block only).

• The client runs D.Verify*(sk,pk,st_c,ch,P)* to verify the proof, and emits an acceptance or a rejection notification based on the result.

• If there was a problem reading from D, then she tries to read through the log structure E. In such a case, she needs to read the whole logs.

• If reading from E is not possible too, server misbehavior is detected. She goes to the arbitrator, e.g., [21].

PAudit:

• The client starts E:LAudit(). If it results in an acceptance, she outputs accept, otherwise, she outputs reject and contacts the arbitrator.

6) Dynamic PoR Security Proof

Theorem1:
IfD=(KeyGen;PrepareUpdate;PerformUpdate;VerifyUpdate;Challenge;Prove;Verify) is a secure DPDP scheme, and E = (LInit;LAppend;LAudit) is a secure ECAL scheme, then DPoR = (PInit;PRead, PUpdate;PAudit) is a secure DPoR scheme.

Proof: Correctness of DPoR follows from the correctness of DPDP and ECAL. Since DPoR has nothing to do apart from DPDP and ECAL, if both of them operate correctly, any PRead(i) will return the most recent version stored at the ith location through DPDP, and all PAudit operations will lead to acceptance. Authenticity of the plain data is provided by the underlying DPDP and ECAL schemes. Whenver a data is read, the underlying DPDP scheme sends a proof of integrity, assuring authenticity. When that fails, the logs will be read through ECAL, which also provides authenticity. In particular, if a PPT adversary A wins the DPoR authenticity game with non-negligible probability, we can use it in a straightforward reduction to construct a PPT algorithm B who breaks security of the underlying ECAL or DPDP schemes with non-negligible probability. Since both the ECAL and DPDP schemes are secure, the adversary has negligible probability of winning either of their respective games. Therefore, our DPOR is authentic supposed that the underlying ECAL and DPDP schemes are authentic. Retrievability immediately follows from retrievability of the ECAL. Since ECAL is secure, it guarantees retrievability of the logs that can be used to reconstruct and retrieve the plaintext data. We bypass the proof details as it is straightforward to reduce the retrievability of our dynamic PoR scheme to that of the underlying ECAL.

V. CONCLUSION

This paper describes a general framework for constructing DPoR schemes. This framework encapsulates all known DPoR schemes as its special cases and further show practical and interesting optimizations that enable even better performance than the existing frameworks.

References

[1] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song. Provable data possession at untrusted stores. In CCS'07. ACM, 2007.

[2] G. Ateniese, R. Di Pietro, L. V. Mancini, and G. Tsudik. Scalable and efficient provable data possession. in SecureComm, page 9. ACM, 2008.

[3] G. Ateniese, S. Kamara, and J. Katz. Proofs of storage from homomorphic identification protocols. In ASIACRYPT, 2009.Barsoum and A. Hasan. Enabling dynamic data and indirect mutual trust for cloud computing storage systems. Parallel and Distributed Systems, IEEE Transactions on, 24(12):2375–2385, 2013.

[4] F. Barsoum and M. A. Hasan. Provable possession and replication of data over cloud servers. Centre For Applied Cryptographic Research (CACR),University of Waterloo, Report, 32:2010, 2010.

[5] D. Boneh, B. Lynn, and H. Shacham. Short signatures from the weil pairing. In ASIACRYPT. Springer, 2001.

[6] K. Bowers, A. Juels, and A. Oprea. Hail: A high-availability and integrity layer for cloud storage. In CCS'09, pages 187–198. ACM, 2009.

[7] K. D. Bowers, A. Juels, and A. Oprea. Proofs of retrievability: Theory and implementation. In CCSW, pages 43–54. ACM, 2009.

[8] D. Cash, A. K¨upc¸ ¨u, and D. Wichs. Dynamic proofs of retrievability via oblivious ram. In EUROCRYPT'13, pages 279–295. Springer, 2013.

[9] N. Chandran, B. Kanukurthi, and R. Ostrovsky. Locally updatable and locally decodable codes. In TCC, 2014.

[10] Curtmola, O. Khan, and R. Burns. Robust remote data checking. In Proceedings of the 4th ACM international workshop on Storage security and survivability, pages 63–68. ACM, 2008.

[11] R. Curtmola, O. Khan, R. Burns, and G. Ateniese. Mr-pdp: Multiple-replica provable data possession. In ICDCS'08, pages 411–420. IEEE, 2008.

[12] Y. Dodis, S. Vadhan, and D. Wichs. Proofs of retrievability via hardness amplification. In TCC. Springer, 2009.

[13] Dwork, M. Naor, G. N. Rothblum, and V. Vaikuntanathan. How efficient can memory checking be? In TCC, pages 503–520. Springer, 2009

[14] Erway, A. K¨upc¸ ¨u, C. Papamanthou, and R. Tamassia. Dynamic provable data possession. In CCS'09, pages 213–222. ACM, 2009.

[15] Esiner, A. Kachkeev, S. Braunfeld, A. K¨upc¸ ¨u, and O. Ozkasap. Flexdpdp: Flexlist-based optimized dynamic provable data possession. Technical report, Cryptology ePrint Archive, Report 2013/645, 2013.

[16] M. Etemad and A. K¨upc¸ ¨u. Transparent, distributed, and replicated dynamic provable data possession. In ACNS, 2013.

[17] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious rams. Journal of the ACM (JACM), 43(3):431–473, 1996.

[18] A. Juels and B. S. Kaliski, Jr. Pors: proofs of retrievability for large files. In CCS'07, pages 584–597, New York NY, USA, 2007. ACM.

[19] K¨upc¸ ¨u. Efficient cryptography for the next generation secure cloud: protocols, Proofs and Implementation. Lambert Academic Publishing, 2010.

[20] K¨upc¸ ¨u. Official arbitration with secure cloud storage application. The Computer Journal, 2013.

[21] Kushilevitz, S. Lu, and R. Ostrovsky. On the (in) security of hash-based oblivious ram and a new balancing scheme. In ACM SODA. SIAM, 2012.

[22] Z. Mo, Y. Zhou, and S. Chen. A dynamic proof of retrievability (por) scheme with o(logn) complexity. In IEEE ICC, pages 912–916. IEEE, 2012.

[23] H. Shacham and B. Waters. Compact proofs of retrievability. In Advances in Cryptology-ASIACRYPT 2008, pages 90–107. Springer, 2008.

[24] H. Shacham and B. Waters. Compact proofs of retrievability. Journal of cryptology, 26(3), 2013.

[25] M. Etemad and A. K¨upc¸ ¨u. Transparent, distributed, and replicated dynamic provable data possession. In ACNS, 2013.