Energy Efficiency in Secure and Dynamic Cloud Storage

Adilet Kachkeev Ertem Esiner Alptekin Küpçü Öznur Özkasap

Koç University Department of Computer Science and Engineering, İstanbul, Turkey
{akachkeev,eesiner,akupcu,oozkasap}@ku.edu.tr

Abstract. The popularity of the cloud storage systems has brought a number of challenges. Two of them are data integrity and energy efficiency. There are many proposed static solutions to prove the integrity of a file. For the dynamic case, Rank-Based Authenticated Skip list (RBASL) has been presented. It provides the update operations with logarithmic complexity. However, an RBASL expects the block size to be fixed. In a realistic scenario, where the updates can be of variable size, an RBASL performs $O(n)$ update operations on the data structure when a change in the file occurs, where n is the number of blocks. To overcome this problem, we propose Flexible-length based authenticated skip list (FlexList) to make $O(u)$ update operations where u is the number of the update operations. Moreover, we developed an algorithm to carry out multiple challenges at once. We have tested our algorithm, and the results show time and energy efficiencies of 60%, 45%, 35% and 20% for file sizes 4MB, 40MB, 400MB, and 4GB respectively.

Keywords: Energy efficiency, cloud storage, skip list, provable data possession.

1 Introduction

Cloud data storage systems have become popular in recent years both in academia [1, 3, 6, 7, 10, 13] and industry (e.g., Sky Drive, Google Drive, Amazon S3), where energy efficiency and proving data integrity have become important challenges [4]. These systems have two entities, a client and a server. A client sends her data to the server - data storage provider (third party), which promises to store the data intact and provide its availability. However, the server can be malicious, and even if the server is trustworthy, there may be hardware or software related failures that may cause data corruption or loss. Therefore, the client should be able to efficiently (in terms of energy and time) and securely check the integrity of her data without downloading the whole file [1].

One of the first proposed models with provable data integrity is *Provable Data Possession* (PDP) [1]. The client, in this model, has the ability to challenge the server on randomly chosen blocks, the server sends a proof, and she can verify the data integrity through that proof. PDP and other related schemes are applicable to the static cases, if the blockwise update operations (insert, remove, modify) are possible, they demonstrate poor performance [1, 6, 10, 13]. The static scenario can be used in some systems (e.g., archival storage at the libraries), but many other applications may necessitate a dynamic scenario, where the client can interact with her data in a read/write manner, while preserving the data possession guarantees. Ateniese et al. [3] proposed Scalable PDP, where the client has a pre-determined number of a limited set of operations. Erway

et al. [7] proposed a model called *Dynamic Provable Data Possession*, which not only extends the PDP model, but also provides a dynamic solution. However, an underlying authenticated data structure based on a skip list [12] is needed for the implementation of the DPDP scheme.

For the dynamic scenario, Erway et al. [7] introduced the new data structure *rank-based authenticated skip list (RBASL)*, which is a special type of authenticated skip list [9] to be used in DPDP. In this model, the client preprocesses the file and stores meta data to verify the later proofs from the server. Then she outsources the file to the server. At any time, she can challenge some blocks to check the integrity of her file. Upon such request, the server prepares a proof for the challenged blocks. In opposition to an authenticated skip list, where a search is done using a key, in an RBASL one can search with indices of the blocks. This feature gives the opportunity to efficiently check the integrity of the data using block indices. Authenticated ranks are used as a search key in an RBASL. Each node has a *rank*, indicating the count of the nodes at leaf-level that are reachable from that particular node. Leaf-level nodes, having no $after$ links, have a rank of 1.

In a realistic scenario, the client may want to alter some part of a particular block, not the whole. It can be problematic to perform in an RBASL. Modification of a particular block in an RBASL may cause the modifications in all consequent blocks as well. Therefore, it is subject to $O(n)$ modifications for DPDP and PDP, which is inefficient in terms of time and energy. We propose a new data structure called FlexList, which is based on an authenticated skip list. It performs dynamic operations (modify, insert, remove) for cloud data storage, while having $O(1)$ variable block sized updates. Moreover, the client has a capability to challenge multiple blocks at once and the server using the multi-proof algorithm prepares the proof for the client.

Our main contribution is as follows:

- The client can challenge the server for multiple blocks using authenticated skip lists, rank-based authenticated skip lists and FlexLists. Our algorithm provides an *optimal* proof, without any repeated items. The current experimental results show time and energy efficiencies of 60%, 45%, 35% and 20% for file sizes 4MB, 40MB, 400MB, and 4GB respectively.

2 Secure and Dynamic Cloud Storage using FlexList

2.1 FlexList

A fixed block size is suitable for an RBASL, since a search (and other methods) by byte index of the data is not possible with rank information. A FlexList, unlike an RBASL, supports the variable size of a block. Due to the problem of providing variable block sized operations with an RBASL, we present a FlexList, which overcomes the problem and serves as an underlying data structure in our cloud storage system. A FlexList stores, at every node, the total number of *bytes* that can be reached from that node, instead of the number of blocks reachable from it. The *rank* of each leaf-level node is computed as the sum of the $length$ of its data and the *rank* of its $after$ node (0 if *null*). The *rank* for every non-leaf node is computed as the summation of the *ranks* of its $below$ and $after$ links [8].

2.2 FlexDPDP

FlexList is employed as an authenticated data structure in our secure and dynamic cloud storage systems, which supports energy efficient data integrity checking. We define a FlexDPDP as a DPDP scheme with FlexList. In our system we have two main parties: a client and a server. The server provides storage space for the client's file. An RBASL can be constructed on the top of the file as shown by Erway et al. [7]. A FlexList, in contrast to an RBASL, can search and reach the data bytes easily, even though the data blocks are of variable sizes. Therefore, a FlexList can perform a variable sized update of length $O(u)$ in $O(u)$ operations. However, an RBASL will need $O(n)$ operations, where n is the number of the blocks in the file. A FlexList represents file blocks as the leaves. So the search path for particular block is the proof membership (i.e., integrity) from this leaf node to the root. We developed an optimized proof generation algorithm, which handles the multiple block challenge at once [8].

The FlexDPDP scheme employs *homomorphic verifiable tags* (as DPDP), so that a number of tags can be combined to obtain a single tag that corresponds to the combined blocks [2]. Small size of tags compared to data blocks enables storage in memory. The authenticity of the skip list gives guarantees for the integrity of tags, and tags protect the integrity of the data blocks.

2.3 Multi-Proof Generation

The client server system starts with the client preprocessing her data, where a FlexList on the file is created and tags are calculated for each block. Then, the client send the random seed to the server and the file itself. The server using the seed can create the identical FlexList using the data blocks and tags (sent by the client). The server sends the hash value of the root of the FlexList for the client to verify the correct construction of the FlexList. After the successful verification, the client can safely delete the file and keep the hash value of the root as the meta data. At any time later, the client can send a random seed to the server to challenge a number of blocks. The server using the seed creates the challenges, runs the *genMultiProof* algorithm and returns proof generated by this algorithm to the client. She can verify the proof using the meta data and the verification algorithm.

genMultiProof : The proof generation algorithm is run by the server, upon the receipt of the random seed from the client. The server first generates a predetermined number of challenges, and random values accordingly. Then, the server runs the *genMultiProof* algorithm to obtain the proof, file blocks and tags for the challenged indices. The algorithm traverses to the leaf-level nodes holding the challenged blocks. Along the traverse path, it stores visited nodes. We have observed that the regular search for each challenged block is inefficient. Since the regular search always starts from the root, there are a lot of repeated nodes in the proof. To handle this problem, we save the states at each intersection point. A node is an intersection point of proof paths of two indices when the first index can be found following the *below* link and the second index is detected by following the $after$ link. Note that all the challenges (indices) are in ascending order. In our *optimal* proof, we visit and take information stored for each node

on the proof path only once [8]. Therefore, the algorithm achieves significant gains in terms of time and energy.

3 Evaluation

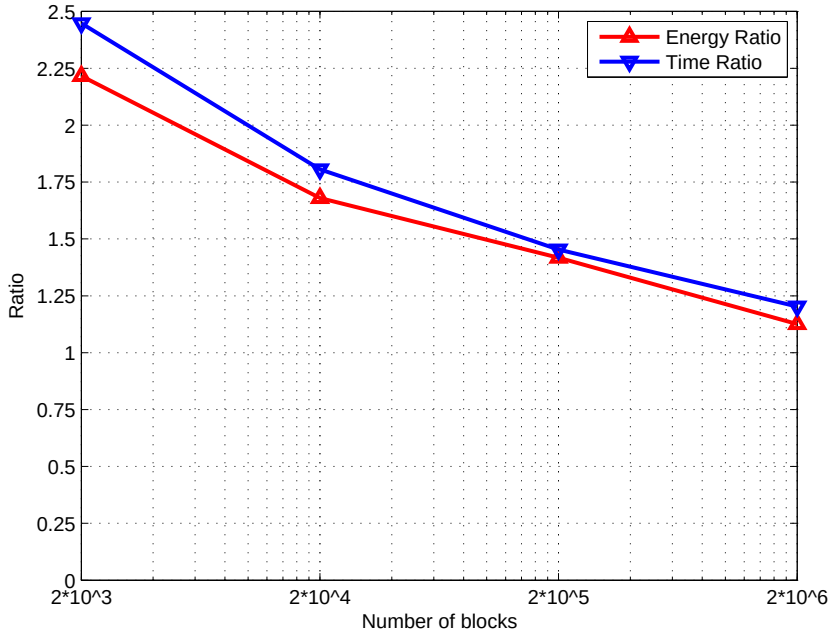

Fig. 1. Ratio graph on *genMultiProof* algorithm.

We developed a prototype implementaion of an optimized FlexList and used it in our FlexDPDP scheme. C++ is used as the programming language and some of the methods from the *Cashlib* library [11, 5] are employed. The experiment was conducted on a 64-bit machine with a 2.4GHz Intel 4 core CPU (only one core is active), 4GB main memory and 8MB L2 cache, running Ubuntu 12.10. As security parameters, we used 1024-bit RSA modulus, 80-bit random numbers, and SHA-1 hash function, overall resulting in an expected security of 80-bits. Watts up Pro meter was used for the energy efficiency tests. It measures the total energy consumption of the connected device and displays this information. We measured the average energy consumption while the experiment was running. Then, we measured the average energy cost for the idle time, while no tests were taking place. The difference between these two measurements were used in the calculation of the results. Energy consumption and time (CPU) ratio results are close in our experiment. Since there was no I/O delay in the test due to disk access,

we argue that the energy efficiency of our *genMultiProof* algorithm is directly impacted by the CPU time. The test is the average of 10 runs.

We have tested our *genMultiProof* algorithm, which is used to accumulate the proof along with the FlexList for the received challenge request from the client. The ratio graph for the *genMultiProof* algorithm is shown in Figure 1. We had different file size scenarios : starting a file size from 4MB to 4GB (all with a block size of 2KB). In every scenario we had the same challenge size of 460, which is sufficient for high probability (constitutes to 99% with the assumption that 1% of the file altered) of catching the cheating server. The time/energy ratio is a division of the time/energy spent for the operation of 460 challenges (one by one) to the time/energy spent for the *genMultiProof* algorithm. Even though the results show a decline in the time and energy ratio while the file size grows, the efficiency of the algorithm is still satisfactory. The algorithm gets its advantage through the minimization of the proof size, therefore no repeated proof nodes for the same node are created. Once created, the proof node is used as a common proof node for other nodes as well. As the file size grows, the ratio on the number of common proof nodes to the total proof size decreases, since we have the constant number of challenges as 460. Note that the maximum efficiency of the *genMultiProof* algorithm is reached when the challenged blocks are near each other. Nevertheless, the graph clearly shows the efficiency gains in terms of energy and time for sufficiently large file sizes. So for the file of size 4MB, 40MB, 400MB and 4GB, we have time and energy gains of 60%, 45%, 35% and 20% respectively.

4 Conclusion and Future Work

With the emergence of the distributed and cloud storage services, energy efficiency has become one of the important challenges [4]. Early works have shown that the static solutions with optimal complexity [1, 13], and the dynamic solutions with logarithmic complexity [7] are within reach. However, the DPDP [7] solution is not applicable to the realistic scenarios since it supports only fixed block size and therefore lacks flexibility on the data updates, while updates in the realistic scenario are more likely to be of a variable size. We have extended their work in several ways and provided a new data structure (FlexList) and its usage in the cloud data storage. A FlexList efficiently supports the variable block sized multiple updates, and we showed how handling multiple challenges at once greatly improves scalability and energy efficiency. As a part of future work, we plan to further develop FlexDPDP algorithms. Currently, we are working on energy efficient algorithms to create a FlexList from a scratch, perform and verify multiple updates. Subsequently, a P2P model for the FlexDPDP will be investigated and designed. We plan to deploy such a system on PlanetLab and run tests for energy efficiency at both the client and the server.

Acknowledgements

Our work was partially supported by the COST (European Cooperation in Science and Technology) framework, under Action IC0804: “Energy Efficiency in Large Scale Distributed Systems” and Action IC1206, by TÜBİTAK (the Scientific and Technological

Research Council of Turkey) under grants 109M761 and 112E115, by Türk Telekom, Inc. under Grant 11315-06, and by Koç Sistem, Inc.

References

1. G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song. Provable data possession at untrusted stores. In *ACM CCS*, 2007.
2. G. Ateniese, S. Kamara, and J. Katz. Proofs of storage from homomorphic identification protocols. In *ASIACRYPT*, 2009.
3. G. Ateniese, R. D. Pietro, L. V. Mancini, and G. Tsudik. Scalable and efficient provable data possession. In *SecureComm*, 2008.
4. A. Berl, E. Gelenbe, M. Di Girolamo, G. Giuliani, H. De Meer, M. Q. Dang, and K. Pentikousis. Energy-efficient cloud computing. *Comput. J.*, 53(7):1045–1051, Sept. 2010.
5. Brownie cashlib cryptographic library. http://github.com/brownie/cashlib.
6. Y. Dodis, S. Vadhan, and D. Wichs. Proofs of retrievability via hardness amplification. In *TCC*, 2009.
7. C. Erway, A. Küpçü, C. Papamanthou, and R. Tamassia. Dynamic provable data possession. In *ACM CCS*, 2009.
8. E. Esiner, A. Kachkeev, A. Küpçü, and Ö. Özkasap. Flexlist: Optimized skip list for secure cloud storage. Technical Report, Koç University, 2013. http://crypto.ku.edu.tr/sites/crypto.ku.edu.tr/files/papers/techreport-flexlist.pdf.
9. M. T. Goodrich, R. Tamassia, and A. Schwerin. Implementation of an authenticated dictionary with skip lists and commutative hashing. In *DARPA*, 2001.
10. A. Juels and B. S. Kaliski. PORs: Proofs of retrievability for large files. In *ACM CCS*, 2007.
11. S. Meiklejohn, C. Erway, A. Küpçü, T. Hinkle, and A. Lysyanskaya. Zkpdl: Enabling efficient implementation of zero-knowledge proofs and electronic cash. In *USENIX Security*, 2010.
12. W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 1990.
13. H. Shacham and B. Waters. Compact proofs of retrievability. In *ASIACRYPT*, 2008.