Asian Journal of Information Technology 15 (19): 3801-3809, 2016
ISSN: 1682-3915

A Secured Proof of Ownership and Enhanced Deduplication Scheme for Cloud Storage

[1]S. Umamaheswari, [2]T. Purusothaman, [3]N P. Supriya and [4]Arun Thundyill Saseendran
[1]Anna University, Chennai, Tamil Nadu, India
[2]Department of CSE/IT, Government College of Technology, Coimbatore, India
[3]HP Software R and D, Bangalore, Karnataka, India
[4]TATA Consultancy Services, Bangalore, Karnataka, India

Abstract: Cloud storage services use cross-user deduplication techniques to maintain unique copies of duplicate data. Client-side deduplication detects and prevents duplicates at the client before the data is sent across the network bringing in huge benefits in terms of network bandwidth. However, these techniques are prone to security breaches. Proof of Ownership (PoW) is a cryptographic concept that requires a client to prove to the server that it possesses the data before the server accepts its upload request and marks the client as the owner of the data. In this paper, a secure PoW technique for cloud storage systems is presented by overcoming the drawbacks of existing research works such as overhead due to pre-computed challenges, third party entities and I/O computational overhead.The proposed solution is implemented in OpenStack Cloud environment for the proof of practicability. A trusted entity called Pseudo Open Stack Manager (POM) is introduced for enhancing security which acts as an interface isolating the cloud storage from malicious users. POM is assumed to be a part of the cloud storage system. The security and performance of the proposed solution is evaluated and the results demonstrate significant improvement in the upload bandwidth usage and efficient use of storage space without compromising security.

Key words: Cloud storage, deduplication, proof of ownership, data security, Pseudo OpenStack manager

INTRODUCTION

Cloud computing provides an economical method for leveraging computing resources to manage large amounts of digital data with efficient storage capabilities. Rapid growth in the amount of data has led industry-wide initiatives to provide efficient online storage services such as Dropbox, Wuala, MozyHome and many public and private OpenStack based cloud storage systems. This development has increased the amount of redundant data stored in the cloud and this redundancy is typically addressed using deduplication techniques. Data deduplication stores a single copy of redundant data and creates links to the single copy instead of storing multiple copies of the data (Harnik *et al.,* 2010). Server side deduplication has the overhead that the data has to be sent to the server over the network before applying the deduplication technique. Hence, client-side deduplication is preferred in the industry (Keelveedhi *et al.*, 2013). Client-side deduplication focuses on reducing redundant data by identifying it before it is sent to the server, thereby reducing the network bandwidth usage and upload time to a significant amount.

Security issues: Though client-side deduplication brings in advantages such as reduction of network bandwidth and data upload time, several security threats are associated with it. These threats have to be addressed in order to reap the full potential of client-side deduplication. Halevi *et al.* (2011) identified various threats that affect a remote storage system that implements client-side deduplication. First, in traditional storage systems, privacy and confidentiality are compromised when an attacker learns the hash value of a file and gain access to the entire file from the server. For example, Dropbox uses Secure Hash Algorithm 256 (SHA256) in a straightforward manner; thus, it is possible for an attacker to obtain hashes for confidential files of others. Using the hash, a hacker can easily download an unauthorized file from the server. A practical way of performing this hack is demonstrated by the use of the Dropship Application Programming Interface (API) utilities for Dropbox which allows a hacker to download confidential files of others from Dropbox servers by obtaining the file hashes which is comparatively easy to obtain and exploiting the weak client-side deduplication technique that is applied (Dropship API). Second, if an attacker can access the

Corresponding Author: S. Umamaheswari, Anna University, Chennai, Tamil Nadu, India

server cache, then all confidential hashes will be disclosed. Hence the attacker can make the cloud storage service behave as a Content Distribution Network (CDN) by publishing the hacked hash values. In a case where the file hashes are published globally, the only way to protect the files from hackers is to turn off client-side deduplication for the files for which the hashes have been compromised, for lifetime in most cases (Halevi *et al*., 2011). The above said two security threats are mitigated by the use of Remote Data Auditing (RDA) (Sookhak *et al*., 2014) procedures. RDA refers to the set of protocols or procedures that can be used to verify the correctness of the data over cloud environment. Proof of Ownership, first explained by Haveli et al is a RDA procedure that can be used to mitigate the security threats faced during client-side deduplication.

PoW can be used in industry for its efficiency in terms of computation, I/O, memory usage and bandwidth. Also, it should be optimized in such a way that it should not require a large amount of file data to be loaded in memory by the server and client to execute PoW. Though several research works have addressed various necessary characteristics of PoW, a robust solution which addresses all the characteristics has not been published.The issues that are addressed in this work are an efficient Proof of Ownership scheme without the use of pre-computed challenges and third party entities with efficiency in terms of computation, I/O and memory utilization.

OpenStack: The need for secure, scalable and reliable cloud storage services inside organizations has raised concerns regarding open-source cloud solutions such as OpenStack. OpenStack is an open-source cloud operating system that enables the creation and management of large virtual machine clusters of private and public clouds. Open Stack software is capable of controlling large clusters of compute, storage and networking resources spread across datacenters. These resources can be controlled by making use of the web-based graphical user interface or by using the REST based APIs provided by OpenStack (http://www.openstack.org/software/).

The object storage component of OpenStack is called Swift. It is an object store that enables storing and retrieving data from the cloud using APIs. It provides durability, availability and concurrency for the entire data set. Swift is suitable for storing unstructured data sets of arbitrary size. Though OpenStack Swift provides efficient object storage, it does not offer deduplication out of the box. However, it provides a rich set of REST based APIs through which deduplication can be enabled. Hence, in our research, we use OpenStack Swift to demonstrate the applicability, practicality and efficiency of our solution.

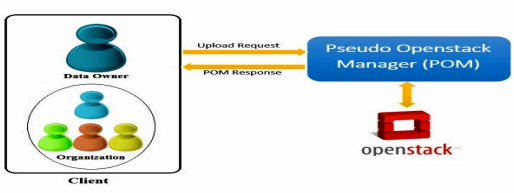

Fig. 1: Solution overview diagram

The software layer that is developed to interface between OpenStack Swift and the client requests is called as Pseudo OpenStack Manager (POM). It is explained in the following study.

PSEUDO Openstack Manager (POM): POM (Fig. 1) is a trusted entity for OpenStack that acts as a secure abstract layer between OpenStack and a client (the data owner or a group of users). POM is always synchronized with OpenStack Swift metadata. The metadata includes information about the files present in the OpenStack Swift Meta data store such as object name, its owner(s), file size and its digest value.

It is not a third party entity or does not require a separate host. During the implementation of the scheme proposed in this work, POM is implemented as a part of OpenStack and is responsible for executing the RDA procedures. Hence there is no network delay in the transaction between POM and OpenStack Swift. The implementation of POM gives the Cloud Service Providers an additional layer of security as it isolates the cloud environment from malicious users. POM also acts as the metrics store which provides the essential metrics for performance and security tuning. The various performance metrics such as upload time comparison and network bandwidth usage presented in this study are collected from POM.

Pseudo Openstack Manager (PoM): POM (Fig 1) is a trusted entity for OpenStack that acts as a secure abstract layer between OpenStack and a client (the data owner or a group of users). POM is always synchronized with OpenStack Swift metadata. The metadata includes information about the files present in the OpenStack Swift Meta data store such as object name, its owner(s), file size and its digest value. It is not a third party entity or does not require a separate host. During the implementation of the scheme proposed in this work, POM is implemented as a part of OpenStack and is responsible for executing the RDA procedures. Hence there is no network delay in the transaction between POM and OpenStack Swift. The implementation of POM gives the Cloud Service Providers an additional layer of security as it isolates the cloud

environment from malicious users. POM also acts as the metrics store which provides the essential metrics for performance and security tuning. The various performance metrics such as upload time comparison and network bandwidth usage presented in this paper are collected from POM.

Literature review: The objective of deduplication is to identify duplication of data in the server and avoid it so as to save storage space and bandwidth. The related solutions that have been proposed to counter the threats of client-side cross-user data deduplication.

Proof of Ownership (PoW) The PoW concept was introduced by Halevi *et al.* (2011). PoW is a two-part protocol that is used to verify that the client owns a particular file. It functions between two players that operate on a joint input file, F. First, the verifier generates a shorter verification information v by summarizing to itself F. Later, the prover having F and verifier having only v, engage in an interactive protocol; at the end of this protocol, the verifier either accepts or rejects the proof (Halevi *et al*., 2011). The ressearch describe that PoW as the combination of a summary function and an interactive two party protocol. The authors present three schemes that differ in terms of security and performance. In all the schemes, the client is challenged by server to give the paths of the sibling for a leaves subset in Merkle tree (Merkle, 1989). Both the client and the server build the Merkle tree; the server only keeps the root and challenges clients that claim to own the file. This scheme is efficient in terms of CPU, bandwidth and I/O for the server and should not require the server to load the file from its back-end storage at each execution of PoW. But it has high I/O-bound operation at the client side (reading files from disk) and calculating cryptographic hashes is a CPU-intensive task

s-PoW: Next we shall describe the s-PoW scheme presented by DiPietro *et al.* (2012). If the server receives the file for the first time, it pre computes the number of challenges for a file for both the cases of file store request and precomputed challenges has been used. The server sends the unused challenge to the client. The client computes and sends the response to a challenge of the server. The server checks the response for equality with the precomputed challenge and outputs success or failure. In the modified PoW scheme (DiPietro *et al.*, 2012) considered, the cost of computing the hash value for small files is negligible; for the distribution of files with the same size and for sizes of more than 1MiB, the file size is a good indexing function. s-PoW trades information-theoretical security for improved space efficiency, by deriving challenge seeds from a master secret. It is achieved by using Client-side algorithm to generate a fresh random seed s for each new challenge.

bf-PoW: The next scheme that is explained in this paper is a PoW scheme proposed by Blasco. The authors use a space-efficient randomized data structure called Bloom Filters. Bloom filters is a probabilistic data structure in which false positives are allowed but false negatives are not allowed. It is a data structure that consumes very less memory (Bloom and Burton, 1970). The research claim that Bloom Filters have been used successfully in other domains of computer security (Geravand and Ahmadi, 2013), however it was never used in data deduplication. In reality, the scheme proposed by Blasco works in two phases. The solution provided in this scheme is more efficient in the client side than the solution proposed by Halevi *et al.* (2011) and is efficient in the server side than the solution provided by Roberto. With the use of bloom filters, the authors have reduced the space and computation complexity incurred in the server side. Yet, this solution also depends on pre-computed challenges. This may prove ineffective in terms of practicability. Also, the research by Zheng and Xu (2012) addresses PoW efficiently and also presents a formal security analysis. However, a third party entity named "AUDITOR" is introduced which makes it less practicable.

MATERIALS AND METHODS

This study describes the proposed protocol called o-PoW, Proof of Ownership for OpenStack (Fig. 2). The main idea of o-PoW is to implement secure deduplication in Open Stack through POM. The challenge to the client is calculated by POM following the method described in Algorithm 1. The client sends the hash d of the file which is to be uploaded to the server. On receiving an upload request from the client, the POM first creates a session identifier s_{id}. The study identifier is used for authenticating the client session. Once a study is created, POM subsequently computes a seed s from the current date and time in POM. The seed s is then input into a PRNG (Pseudo Random Number Generator) which initializes a random pointer r to generate a sequence of random numbers. A constant integer N is also chosen by POM. N corresponds to the number of bits that should be

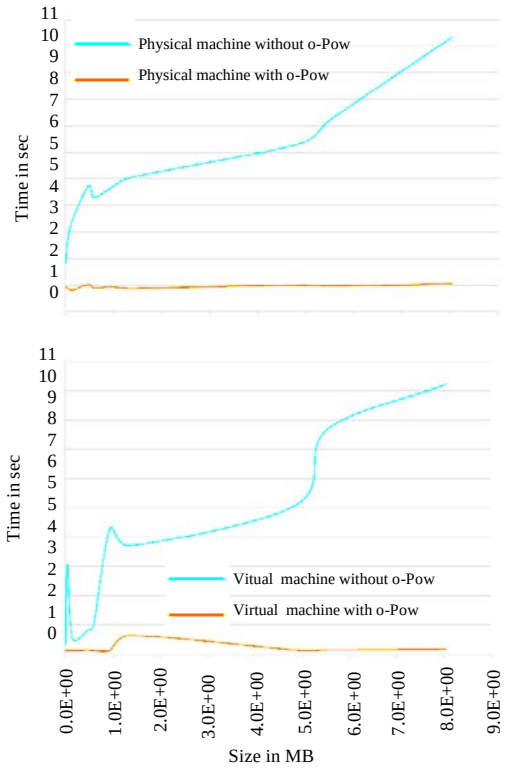

Fig. 2: Comparison of data upload rate of small size files with and without o-PoW

concatenated to form the challenge from the server and the response from the client. Let f_d be the file in OpenStack Swift that corresponds to the hash d received from the client. get_bit() is a simple method to retrieve a bit from a file at a specified position. Challenge $res_d$is calculated by concatenating N bits generated using the get_bit() function. The pseudo code for POMChallenge is given in Algorithm 1.

Algorithm 1: POM challenge:

Input: file f_d in OpenStack swift with hash d, Seed s, constant integer N
Output: Seed s, Session Identifier s_{id}, constant value N
begin
r <- PRNG(s)
Compute session identifier s_{id}
for i in range(0, N)do
res_d<- res_d + get_bit(fd,r.next)
done
store res_d with s_{id}
return (Seed, Sid, N)
End

Once the challenge is sent from POM to the client, the Algorithm 2 is executed on the client side to calculate the response res_c. The client receives the session identifier s_{id}, the seed s and the constant integer N from POM. The client should also possess the file fc for which the upload request was posted to POM. As in the POM side, the client uses the seed s to input to a PRNG which initializes a random pointer r, capable of producing random integers. Since the same seed is used in both the POM and client side, the random integers produced by the PRNG will be equivalent. The client then concatenated N bits from the file fc for each integer generated by r. The client sends the computed response $res_c$and the input session identifier s_{id} to POM. The pesudo code for oPoWClient is given in Algorithm 2.

Algorithm 2: oPoWClient (POMResponse):

Input: Seed s, constant integer N, file to upload f_c, Session Identifier s_{id}
Output: Response to the DupChallenge res_c
begin
r <- PRNG(s)
for i in range(0, N)do
res_c<- res_c + get_bit(f_c,r.next)
done
return (res_c, s_{id})
End

As soon as the response is received from the client, POM executes Algorithm 3 to verify the response. The input to the Algorithm 3 (PoMValidate) is the session identifier s_{id} and the response from the client res_c.The PoMValidate Algorithm first uses the session identifier received from the client to check if the session is valid. If the session is timed-out or invalidated for any reason, the client is notified to retry the upload. If the session is active, the challenge res_d for the session identifier s_{id} is fetched from the respective POM session cache and is compared with the response res_creceived from the client. If the challenge $res_d$and the response $res_c$are equal, then the client is marked as the owner of the file. The PoMValidate is given as pseudo code in Algorithm 3.

Algorithm 3: POMValidate()

Input: Response from the Client res_c, Stored Challenge res_d(Fetched from the cache using session identifier s_{id})
Output: Dup Response to the Client
begin
if res_c == res_d **then**
accept UploadRequest
//Store client as a new owner of the file
else
reject UploadRequest
endif
End

The overall interaction between the OpenStack POM and the client is described in Algorithm 4. Algorithm 4 consists of two scenarios. First, the file that is intended to be uploaded by the client is already present with the service provider. Second, the file that is intended to be uploaded is not present with the service provider and is uploaded for the first time. The interaction between the client and POM is given as a pseudo code in Algorithm 4.

Algorithm 4: Interaction between a client and POM

Client
To Upload File f_c to Openstack Swift
begin
d <- H(f_c)

```
        send d to POM
end
POM
Upon Receipt of d
begin
if d exists in Openstack Swift then
        POMResponse <- POMChallenge()
        oPoWClient(POMResponse)
        POMValidate()
else
        POMResponse <- AllowFileUpload
        f_temp <- AllowUpload
        if d equals H(f_temp)
        f_perm <- f_temp
        Mark the user as owner of file
        delete(f_temp)
        Send positive response to user
else
        delete(f_temp)
        Send negative response to user
endif
End
```

The check for proof of ownership is mandatory during the upload of the file to prevent a malicious user from downloading a file for which he/she is not the owner. Consider the scenario where a malicious user can try to gain ownership of a file with only the metadata (hash of file, size, cached parts of the file etc.) of the file. The algorithm should be able to detect and stop such an attempt. Also, a malicious user can try to perform a poison attack where the cloud server is made to associate the key (in this case the hash of the file) to a poisoned (not the file that corresponds to the hash of the file) file. The algorithm should be able to detect such an attack and stop the malicious user. The four possible scenarios that a malicious user can attempt and the preventions mechanism employed by o-PoW with the help of Algorithms 1-4 is as follows.

Legends

- **Enuine User:** A normal user will no harmful intentions
- **Alicious User:** A user who intends to steal the data from the cloud or create problems for others
- **Loud Server / Server:** OpenStack Swift Store

Scenario 1: A genuine user tries to upload a file that is not present in the cloud server: Consider the scenario in which a genuine user is trying to upload a file to the cloud server. In this scenario let's consider that the file is not present in the cloud server. As per Algorithm 4, the hash of the file (d) is computed by the client and sent to the cloud server. The cloud server prompts the user to upload the file since the file is not present in the cloud server. Since the genuine user possess the file from which the hash (d) was computed, the user will be able to upload this file to the cloud server. The cloud server stores the file in a temporary location f_{temp}. Once the file is present in the cloud server, the server computes the hash $H(f_{temp})$ of the uploaded file and compares it with the hash (d) that was initially sent by the client. Since both d and $H(f_{temp})$ will be equal, the file will be stored permanently in the cloud server and the genuine user will be marked as the owner of the file.

Scenario 2: A malicious user tries to upload a poisoned file that is not present in the cloud server: In this scenario a malicious user who wants to poison the server by associating a file with a wrong hash is considered in this scenario. In this scenario too, it is assumed that the file is not present in the server. As the first step, the client uploads the hash to the file (d) and since the file is not present in the server, the server prompts the client to upload the file. In this scenario, the intent of the malicious user is to poison the cloud server so that when a genuine user uploads the file with the same hash, the server will be misguided that the file is already present in the server. Hence the genuine user will be challenged as per the PoM Challenge algorithm with challenges generated from the poisoned file. Hence the genuine user will not be able to upload the file to the cloud server. However as defined in Algorithm 4, the malicious user will not be able to upload a poisoned file since the file is temporarily stored in the cloud server (f_{temp}) once it is uploaded the freshly computed hash ($H(f_{temp})$) is compared with the hash uploaded by the client (d) before permanently storing the file are marking the user as the owner of the file. In this scenario, since the $H(f_{temp})$ and d will be different the file will not be stored permanently and the malicious user's request to upload the file will be rejected.

Scenario 3: A genuine user tries to upload a file that is already present in the cloud server: In this scenario, a genuine user tries to upload a file that is already present in the server. As per the algorithm, the user first sends the hash of the file to the server. The server identifies that the file is already present in the server. The chance of a poisoned file is eradicated as explained in scenario 2. The cloud server challenges the client using the POMChallenge Algorithm and then validates it using the POMValidate algorithm. Since the genuine user possesses the original file, the user will be able to succeed the challenge. Hence the user will be marked as the owner of the file. From now onwards the user can download the file anytime from the server

Scenario 4: A malicious user tries to upload a file that is already present in the cloud server: In this scenario, a malicious user having hacked metadata about the file tries to gain ownership of the actual file in the cloud server. As per the algorithm 4, initially only the hash of the file is uploaded to the server. The server identifies that the file is already present in the server. The cloud server challenges the user as per the POMChallenge Algorithm and gets the response. Since, the POMChallenge

challenges the user with a sequence of bits distributed throughout the length of the file, the user cannot overcome the POMChallenge with a part of the file. Hence the malicious user will not be able to gain access to a file that he/she does not possess.

RESULTS AND DISCUSSION

The implementation presented in this research was run on a client-server duo with the following configuration.

Client configuration: The client system that was used is a laptop computer with a 2.5-GHz Intel (R) Core (TM) i5-4300U CPU, 8 GB of Random Access Memory (RAM) and a 7200-RPM hard disk. The operating system used was 64-bit version of Windows 7 Enterprise and the file system format were EXT4 with a 4-kB page size.

Cloud instance configuration: The cloud instance was set-up using OpenStack (Juno) on both a virtual and a physical machine. The implementation of the algorithms discussed in this research was done in Python. The python-swiftclient library for the OpenStack Swift API was used to implement POM. The communication to the OpenStack Swift instance was done using RESTful APIs exposed by Swift. Swift has the benefits of ease of use, open source and industry adoption. Performance evaluation was done by uploading a set of 30 real-world files ranging from the size 0.005 MB to 975 MB which were already present in the OpenStack Swift in three ranges as follows:

- Small size files-from 0.005MiB up to 10 MiB
- Medium size files-from 10 MiB upto 100 MiB
- Large size files-from 100 MiB upto 1000 MiB

The total time taken to upload the file to OpenStack Swift is measured as the time from the upload request is initiated by the client to the time the final response is received from the server. The unit of time is set as seconds. The time measured is inclusive of the time taken for the execution of the algorithms, if any. The o-PoW algorithm is implemented on an OpenStack Swift instance set up on a physical machine and a virtual machine. The time measured on different scenarios for smaller size files is tabulated in Table 1. As described, the files are uploaded to both the setups and the upload time is measured and tabulated. The graphical representation of the scenario in which the small files are uploaded to an OpenStack Swift instance on a physical machine and a virtual machine is given in Fig. 2. It can be seen that in case of a physical instance, for small files the difference in upload time is comparatively narrow. However, in virtual instances, the implementation of o-PoW reduces the upload time to a considerable extent.The next set of the experiment was done using a set of 10 medium-sized files and the results are tabulated in Table 2 and the readings

Table 1: The O-PoW data upload time metrics for small size files

	Upload time (sec)			
	Open stack swift on physical machine		Open stack swift on virtual machine	
File size (MiB)	Without O-PoW	With O-PoW	Without O-PoW	With O-PoW
0.005	1.858	0.925	3.622	1.497
0.150	3.502	0.811	5.574	1.450
0.475	4.739	1.036	8.367	1.534
0.592	4.310	0.889	9.314	1.487
0.949	4.669	0.936	43.266	1.528
1.353	5.055	0.873	37.079	6.576
4.875	6.332	1.003	51.068	1.541
5.523	7.254	0.967	76.724	1.586
8.032	10.335	1.061	92.160	1.825

Table 2: O-PoW data upload time metrics for medium size files

	Upload time (sec)			
	OpenStack Swift on Physical Machine		OpenStack Swift on Virtual Machine	
File size (MiB)	Without o-PoW	With o-PoW	Without o-PoW	With o-PoW
10.174	7.521	1.076	87.795	6.791
11.048	28.619	1.092	96.621	6.671
16.571	7.516	1.217	138.070	1.732
19.086	7.909	1.233	158.603	6.807
22.095	8.423	1.311	187.587	1.786
27.618	9.060	1.326	229.169	1.913
33.141	9.541	1.438	262.651	1.937
38.172	10.151	2.133	308.741	7.117
60.786	17.083	1.883	496.693	7.576
91.180	180.996	2.355	765.633	2.798

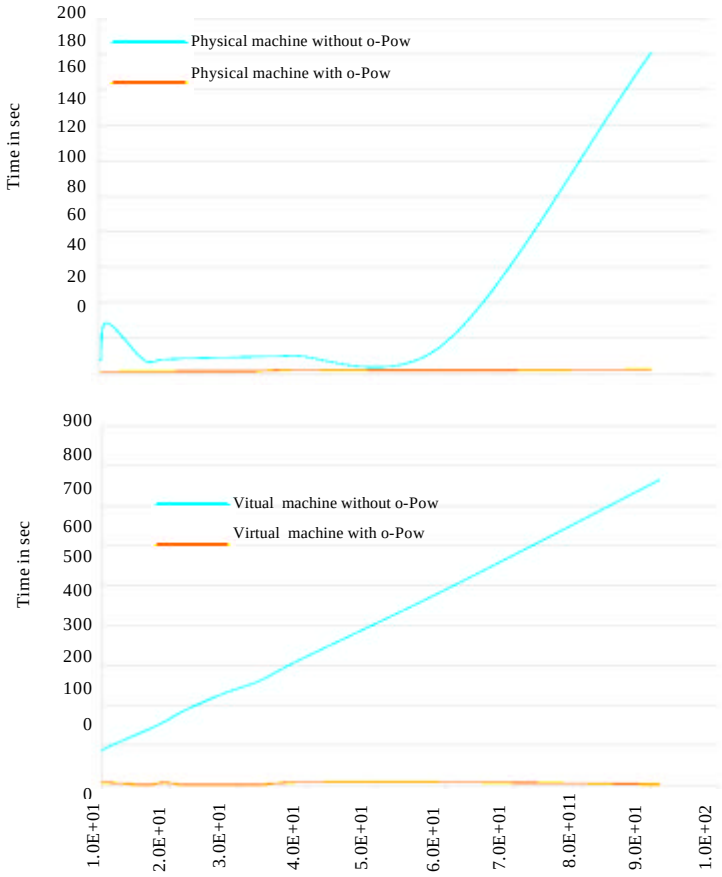

Fig. 3: Comparison of data upload rate of medium size files with and without o-PoW

Table 3: o-PoW data upload time metrics for large size files

	Upload Time (seconds)			
	Open stack swift on physical machine		Open stack swift on virtual machine	
File size (MiB)	Without O-PoW	With O-PoW	Without O-PoW	With O-PoW
112.995	69.751	2.296	907.692	8.055
188.664	40.107	3.724	1516.941	6.234
208.392	44.301	4.113	1675.559	6.886
212.752	45.228	3.256	1718.245	6.800
275.911	68.111	3.567	2228.336	8.819
483.049	119.245	2.134	3456.877	5.200
571.440	141.065	2.524	4089.440	6.152
691.558	149.123	3.055	4387.432	5.900
893.017	192.564	2.912	5393.345	6.100
974.558	210.147	3.178	5885.809	6.657

are plotted in a graph for comparison in Fig. 3. From the Fig. 3 we can see that, unlike the case of small size files, the time for upload increases by a sharp margin even in physical instance when the size of the file increases above 50 MiB. In case of virtual instance, the upload time increases linearly when o-PoW is not implemented, whereas with the implementation of o-PoW algorithm, the upload time almost remains a constant. The third set of upload time measurements was taken from large files. In case of larger files, it is clearly observed that the upload time remains a constant. We have compared the upload time in the case of both physical and virtual instances without o-PoW implementation. The upload time with o-PoW implementation is very less. The upload time measurement for large files is given in Table 3 and the graph with upload time comparison for large files with and without o-PoW implementation is given in Fig. 4.

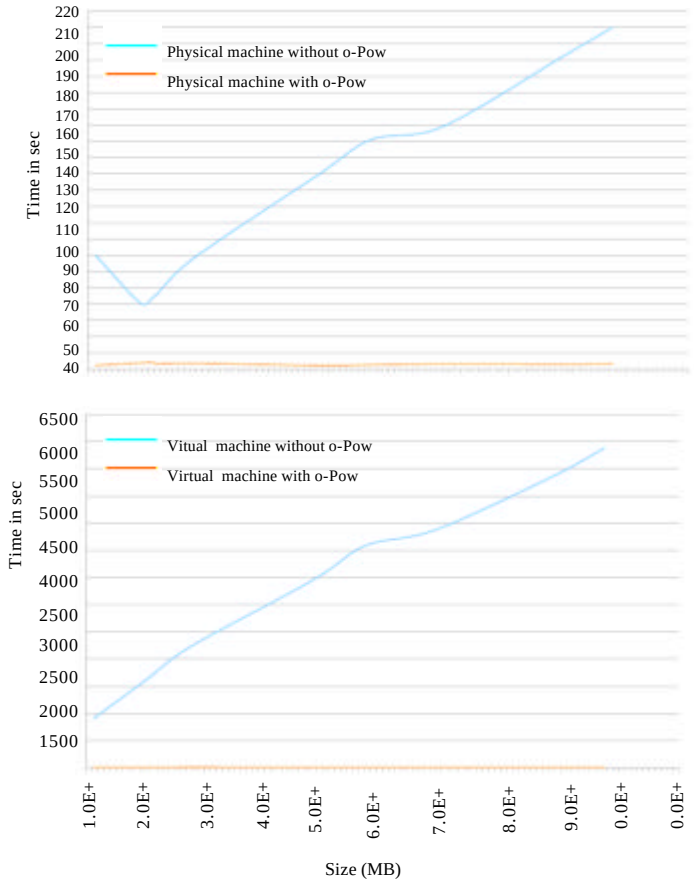

Fig. 4: Comparison of Data Upload Rate of Large Size Files with and Without o-PoW

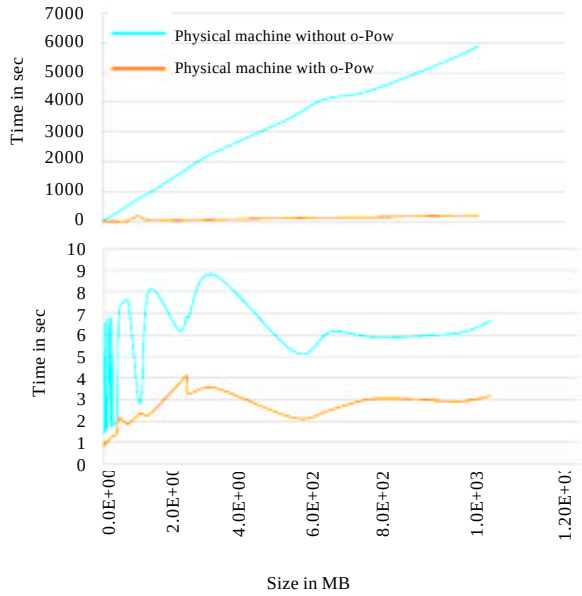

Fig. 5: Data upload rate comparison overview on physical and virtual instance with and without o-PoW

From the measurements, it is calculated that the average time for upload of 1 MB onto OpenStack Swift on a physical instance is 0.003 seconds and 0.216 sec with and without o-PoW implementation respectively. Similarly, it is 0.027 seconds and 6.919 sec with and without o-PoW implementation respectively on a virtual instance. It can be observed that the time taken for upload increases by 32x on a virtual instance compared to a physical instance without o-PoW implementation. However, the upload time increases only 8x on a virtual instance compared to a physical instance with the implementation of o-PoW. It stands as the proof to the efficiency of o-PoW algorithm implementation. The comparison is shown as a in Fig. 5.

CONCLUSION

Researchers have presented an improved mechanism for PoW named o-PoW algorithm which offers an effective Proof of Ownership scheme without the use of third party entities. The o-PoW algorithm can be used in

cases where secure deduplication is the primary goal. The advantage of the proposed scheme is a minimal-security-state server with high performance on the client side.Since all the algorithms proposed in work are implemented and experimented in real time on OpenStack Swift Cloud Storage instance, it ensures that the proposed solution is practicable and can be implemented on any Cloud Storage system.

REFERENCES

Bloom, B.H., 1970. Space-time trade-offs in hash coding with allowable errors. Commun. ACM., 13: 422-426.

DiPietro, R. and A. Sorniotti, 2012. Boosting efficiency and security in proof of ownership for deduplication. Proceedings of the 7th ACM Symposium on Information Computer and Communications Security, May 2-4, 2012, ACM, Korea, ISBN: 978-1-4503-1648-4, pp: 81-82.

Geravand, S. and M. Ahmadi, 2013. Bloom filter applications in network security: A state-of-the-art survey. Comput. Networks, 57: 4047-4064.

Halevi, S., D. Harnik, B. Pinkas and A. Shulman-Peleg, 2011. Proofs of ownership in remote storage systems. Proceedings of the 18th ACM Conference on Computer and Communications Security, October 17-21, 2011, Chicago, IL., USA., pp: 491-500.

Harnik, D., B. Pinkas and A. Shulman-Peleg, 2010. Side channels in cloud services: Deduplication in cloud storage. IEEE Secur. Privacy, 8: 40-47.

Keelveedhi, S., M. Bellare and T. Ristenpart, 2013. Dupless: Server-aided encryption for deduplicated storage. Proceedings of the 22nd USENIX Symposium on Presented as Part of the Security (USENIX Security 13), August 14-16, 2013, Usenix Publications, Washington, USA., ISB N: 978 -1-931971-03-4, pp: 179-194.

Merkle, R.C., 1989. A Certified Digital Signature. In: Advances In Cryptology-CRYPTO 89, Brassard, G., (Ed.), LNCS 435, Springer-Verlag, Berlin, pp: 218-238.

Sookhak, M., H. Talebian, E. Ahmed, A. Gani and M.K. Khan, 2014. A review on remote data auditing in single cloud server: Taxonomy and open issues. J. Network Comput. Appl., 43: 121-141.

Zheng, Q. and S. Xu, 2012. Secure and efficient proof of storage with deduplication. Proceedings of the 2nd ACM Conference on Data and Application Security and Privacy, February 7-9, 2012, San Antonio, TX., USA., pp: 1-12.