H. G. Reyes-Anastacio, J. L. Gonzalez-Compean, M. Morales-Sandoval and J. Carretero, "A data integrity verification service for cloud storage based on building blocks," *2018 8th International Conference on Computer Science and Information Technology (CSIT)*, 2018, pp. 201-206

DOI: 10.1109/CSIT.2018.8486274

A data integrity verification service for cloud storage based on building blocks

Hugo G. Reyes-Anastacio
Cinvestav Tamaulipas
Cd. Victoria, Mexico
hreyes@tamps.cinvestav.mx

J.L. Gonzalez-Compean
Cinvestav Tamaulipas
Cd. Victoria, Mexico
jgonzalez@tamps.cinvestav.mx

Miguel Morales-Sandoval
Cinvestav Tamaulipas
Cd. Victoria, Mexico
mmorales@tamps.cinvestav.mx

Jesus Carretero
Arcos-UC3M
Madrid, Spain
jcarrete@inf.uc3m.es

Abstract—Cloud storage is a popular solution for organizations and users to store data in ubiquitous and cost-effective manner. However, violations of confidentiality and integrity are still issues associated to this technology. In this context, there is a need for tools that enable organizations/users to verify the integrity of their information stored in cloud services. In this paper, we present the design and implementation of an efficient service based on provable data possession cryptographic model, which enables organizations to verify, on-demand, the data integrity without retrieving files from the cloud. The storage and cryptographic components have been developed in the form of building blocks, which are deployed on the user-side using the Manager/Worker pattern that favors exploiting parallelism when executing data possession challenges. An experimental evaluation in a private cloud revealed the efficacy of launching integrity verification challenges to cloud storage services and the feasibility of applying containerized task parallel scheme that significantly improves the performance of the data possession proof service in real-world scenarios in comparison with the implementation of the original possession data proof scheme.

Index Terms—Virtual containers, Data Integrity, Task parallelism, Cloud storage, proof of possession.

I. Introduction

The production of new data has dramatically grown over few last years [1], [2]. Studies predict that, between 2005 and 2020, the digitized data will grow by a factor of 300, from 130 exabytes (EB) to 40,000 EB [3]. This growth produces a data accumulation effect that represents a critical issue for organizations in terms of management and costs. In this context, the cloud has become an outsourcing popular solution for organization to delegate the manage/store of their data to a public provider [4]. The pay-as-you-go model associated to this technology enables organization to manage large data volumes in a cost-effective manner.

Nevertheless, delegating the management and storage of data to a cloud provider also means delegating control over the data [5], which results in security risky situations such as events of violations of confidentiality and integrity of data [6], [7] as well as incidents of temporal [5] and permanent [5] outages affecting the cloud service reliability.

The violations of data integrity are issues particularly critical for organizations when managing sensitive contents.
For instance, in health domain, bit errors in images are not admissible by both specialists and applications performing automatic processing of images. In remote sensing and earth observation domains, the integrity of satellite images is critical for producing earth observation products (thematic maps, climate maps, etc.), which are considered as a heritage for the scientific community. Moreover, governments establish regulation to be applied to this type of content including the conservation for large periods of time (five years for the case of medical images) through infrastructures meeting reliability and integrity requirements.

The integrity verification of data stored in the cloud using the Provable Data Possession (PDP) scheme has been evaluated in literature [8], [9]. This type of scheme prevents the owner from having to store a local copy D_L of its content D stored in the cloud, download D and verify if it has been modified in the cloud using D_L as reference. PDP is a mechanism that is based on zero knowledge proof and implemented using arithmetic in finite fields and groups, particularly in the additive group of an elliptic curve defined on the prime field F_p [10].

However, in this type of scenarios, the end-users oversee performing the challenges and to adapt their storage clients to include a PDP service. In real scenarios, for instance, the management of any of clinical files, organizational documents, government data or satellite images could come with high and possibly unfeasible performance costs. In this context, there is a need for tools that enables organizations/users not only to verify the integrity of their information but also to perform it in an efficient and flexible manner in real-world scenarios.

In this paper, we present an efficient a data integrity verification service based on building blocks for organizations that use cloud storage services. This service enables organizations to verify, on-demand, the data integrity without retrieving files from the cloud. To improve the deployment of

this service on real-world scenarios, the design of this service is based on building blocks, which enables organizations to implement Manager/Worker patterns to explore parallel realizations thus improving the service performance. A prototype of the proposed PDP service was developed, tested, and evaluated using a federated cloud storage, created by a space agency [11], for the verification of satellite images.

The rest of the document is organized as follows: Section II describes the PDP service design. Section III describes the experimental PDP prototype. Section IV presents the results of the prototype implementation. Finally, the conclusions and final comments derived from this work are presented in Section V.

II. A DATA INTEGRITY VERIFICATION SERVICE FOR CLOUD STORAGE BASED ON BUILDING BLOCKS

This service has been built by using building block (BB) structure [12]. In this service, a BB represents an application and I/O interfaces encapsulated into a virtual container. Three types of BBs were created in this service, the first one (*PDPClient*) includes a PDP Client, the second one (*PDPServer*) includes a PDP Server application and the last one (*PDPDispatcher*) includes a sub-service that creates Manager/Worker patterns launching BBs (either *PDPServer* BB or *PDPClient*). The *PDPDispatcher* BB includes a load balancing module to distribute tasks to the *Workers* (*PDPClient* BBs); as a results, the verification operations in this service are performed in parallel.

A. Components of the PDP scheme

The concepts of PDP were introduced by [9]. A PDP scheme includes a *client* and a *server* where the client is the user of the storage and performs the verification service, whereas the server is in charge of performing the challenges to the cloud services. This scheme is composed of a collection of the following polynomial time complexity algorithms:

- *KeyGen*: *KeyGen*(λ) → (pk,sk) is a probabilistic key generator algorithm executed by the client. It takes a security parameter λ as input, and return an elliptic curve cryptography key pair {pk,sk}, where *pk* is public and *sk* private. The key pair is generated using Algorithm 1. The security parameter λ defines the security strength of the scheme, generally expressed as the size of the keys.
- *TagBlock*: *TagBlock*(pk,sk,m) → T_m is an algorithm (Possibly probabilistic) executed by the client to generate the metadata that is used for verification. It takes as input the public key *pk*, the secret key *sk* and a block *m* of a file F. It returns the metadata of verification T_m corresponding to the block m.
- *GenProof*: *GenProof*(pk,F,*chal*,T) → V is executed by the server in order to generate the possession test. It takes as input parameters the public key pk, an ordered collection of data blocks F, a challenge *chal* of the client and an ordered collection T which are the verification metadata corresponding to the blocks in F.
- *CheckProof*: *CheckProof*(pk,sk,*chal*,v) → {"*success*","*failure*"} is executed by the client in order to validate the possession test. It takes as input the public key *pk*, the secret key *sk*, a challenge *chal*, and its corresponding proof of possession V. It returns "success" if V is a correct proof of possession associated with *chal*, otherwise it returns "failure".

Algorithm 1: KeyGen algorithm

Input: System security parameter λ
Output: Key pair {pk,sk}
Construct an elliptic curve E over the prime finite field F_p, with prime order n.
Select a point P in E;
Use a safe deterministic pseudo-random number generator to generate $d \in [1,n]$; $sk \leftarrow d$;
$pk \leftarrow d \times P$ (elliptic curve scalar multiplication); return {pk,sk}

A PDP system can be built in the following phases (see [8] for details):

- *Setup*: The client (C) has the data F in n blocks (m_1,m_2,...m_n) and executes (pk,sk) ← *KeyGen*(λ), followed by T_{Mi} ← *TagBlock*(pk,sk,m_i) for all $1 \le i \le n$. C then sends pk, F and T = {T_{M1},...,T_{Mn}} to the server S for storage and removes F and T from your local storage.
  Given a data file $D \in \{0,1\}$ and the public key pk, the *setup* algorithm generates the identifier of D (*IDF*) and its corresponding public elements (σ_1,σ_2).
- *Challenge*: C generates a challenge *chal* that, among other things, indicates the specific blocks for which C requires a proof of possession. C then sends *chal* to the server (S), S executes ← *GenProof*(pk,F,*chal*,T) and sends the returned value V to C for test possession.
- *GenChal* algorithm generates a random challenge *chal*. Given a challenge *chal*, and the D version of the data stored on the server, the *ChalProof* algorithm produces a possession test PT = (y_1,y_2) of D.
- *Verify*: Given a test PT, the public elements of D and *sk* of the data owner, this algorithm executed by C performs the verification test and returns as result "*accept*" or "*reject*".

Figure 1 depicts the utilization of the components of this scheme in the context of Cloud storage; PDPClient deployed

on the end-user side, whereas PDPServer deployed on the cloud side.

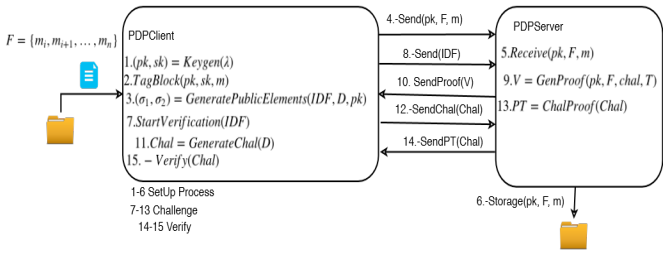

Fig. 1. Uploading and verification process using the PDP scheme

Notice that, in the *setup* phase, C calculates labels for each block file and stores them together with the file in S. In the *challenge* phase, C requests possession tests for a subset of blocks in F. This phase can be executed an unlimited number of times in order to verify if S still contains the selected blocks.

B. Proposed task-parallel scheme for PDP service

A task-parallel scheme for integrity verification was designed to create Manager/Worker pattern. As previously commented, the applications of the PDP scheme were encapsulated into three BBs: a manager called *PDPDispatcher* BB, a worker called *PDPClient* BB and a Service called *PDPServer* BB.

PDPDispatcher BB includes an initialization module that prepares Manager/Worker pattern by launching as many *PDPClient* BBs and *PDPServer* BBs as configured in the service by the end-user. Configurations such as *1-PDPClient to 1PDPServer, 1 to N, N to 1 and N to N* can be chosen by endusers to configure this BB. When the BBs have been launched in Cloud/Cluster/Server/PC, the initialization modules invoke a load balancing module that oversees reading the list of files arriving in the source folder and to determine the worker best suitable to perform a given task. This is key for the performance of this solution as a heterogeneous and unknown load is expected to reach at the folder source. The decisions taken by the load balancing module are sent to an assignation module, which is in charge of preparing the task distribution to the workers (*PDPClient* BB).

Figure 2 shows an example of lists produced by the dispatcher and the files in the list are balanced using a two choices algorithm [13], which produces two random numbers between 0 and the number of available workers minus one ($w-1$) and chooses the one with the smaller load included the previously load assigned to these workers. The algorithm updates the load counter of the chosen worker and processes the next request as this task is performed by each file in the list.

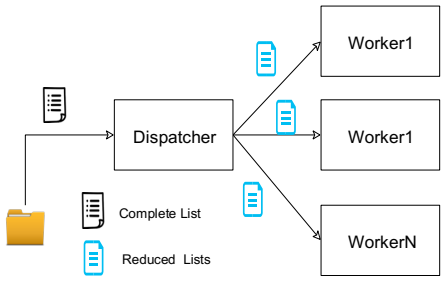

Fig. 2. Task distribution performed by the Dispatcher BB.

The *PDPClient* BB includes an application that listens for *PDPDispatcher* messages through TCP sockets. The messages include the paths for this BB to read the files to be processed. It also includes the client of PDP previously defined for sending integrity verification operations to *PDPServer* BBs. This type of BB completes each operation sent by the dispatcher when a *PDPServer* BB responses to its requests.

The *PDPServer* BB is a virtual container running on a virtual machine in the cloud/cluster, which receives the files sent by *PDPClient* BB and stores them in a cloud storage service by using a file system tool. It includes a module including the previously described *PDPServer* BB, which performs the possession tests requested by *PDPClient* to verify the data integrity.

C. Integrity verification operations

A private data verification is considered in this service as only the data owner can request the integrity test. This verification is invoked on the client side and the generation of integrity tests are performed on the server side, which avoids retrieving files to the end-user's device to perform the integrity verification.

The *PDPDispatcher* BB can deploy Manager/Worker pattern on the end-user side to perform operations such as Upload (Including data transfer and setup securing integrity), Verify/Download (Including an integrity verification before downloading files) and Verify (only sending verification to the servers running in the cloud).

For *Upload* operations, the *PDPDispatcher* BB defines an elliptic curve (E) and a generator point P to be used in Algorithm 1 for all workers (*PDPClient* BBs) that execute *KeyGen*. The *PDPDispatcher* BB invokes multiple *PDPClient* BBs to process the load, each *PDPClient* BBs first store a list of t data files D_t in the cloud by decomposing each file D_i $(0 \leq i < t)$ into two blocks s and n, which represents the quotient and the remainder respectively by applying the Euclidean Division (ED) to the D_i. It is the integer value of the hash obtained from D_i with a divisor b. Note that the value b is kept secret by the *PDPClient* BB and is used to process different files. This b represents the only secret information that the *PDPClient* BB must store to carry out the data possession tests. As such, the definition of several data divisors can extend our proposal. This

means the data owner can rely on different secrets regarding the sensitivity of the data shared in the cloud. In Upload operations, the *KeyGen* is executed by each *PDPClient* BB for the generation of its public and private keys, whereas *Setup* is executed by each worker when storing a data set D in the cloud, its corresponding public elements are generated.

For *Verify* operations, each *PDPClient* BB executes the protocol for the generation of the challenge-response with its corresponding *PDPServer* BB. In this operation, each *PDPClient* BB executes *GenChal* by generating a challenge for a *PDPServer* BB, which executes *ChalProof* to generate a valid proof of the possession of the data D_i and sends the results to the *PDPClient* BB (Worker invoking the verify operation). The *PDPClient* BB executes *Verify* by using data sent by *PDPServer* BB to perform the verification of the possession test.

For *Download* operations, the *PDPClient* BB can be configured to perform secure acquisition of files by invoking a *verify* operation before to the execution of the download of the files included in the list sent by the *PDPDispatcher* BB. *PDPClient* BB also can be configured to only download files without previous verification.

D. PDP Service prototype: Implementation details

The applications of the PDP modules were performed in Java by using jPBC library [14] for managing elliptic curves and bilinear pairings.

TABLE I
VIRTUAL CONTAINERS IMAGES (CI) AND BUILDING BLOCKS (BBS) OF THE VERIFICATION SERVICE

CI/BB	Containers	Interface(In/Out:ports)
PDPDispatcher	dispatcher	workers:4 Output:4500 Output:4501 Output:4502 Output:4503
PDPClient	Worker1	services:1 input:4500 Output:4600
	Worker2	services:1 Input:4501 Output:4601
	Worker3	services:1 Input:4502 Output:4602
	Worker4	services:1 Input:4503 Output:4603
PDPDispatcher	Server1	storage:1 Input:4600
	Server2	storage:1 Input:4601
	Server3	storage:1 Input:4602
	Server4	storage:1 Input:4603

To create Manager/Worker patterns, the three BB types were implemented by using container images (CI) and Docker CE platform [15] was used for the deployment of container images. The generated images can be executed multiple times in different infrastructures using a different configuration files. In the case of *PDPServer* BB only the port of the container must be enabled before executing its applications. The *PDPDispatcher* BB configuration file includes the configuration of the Manager/Worker pattern (any of 1-1, 1-N N-1 or N-N). This configuration file also includes the number of *PDPClient* BBs (workers) and *PDPServer* BBs (storage services) and the list of IPs addresses and ports connecting *PDPClient* BBs with *PDPServer* BBs considered in the configuration of the Manager/Worker pattern.

Table 1 shows the CI used to launch the BBs, the name of the containers that were launched using that BB, and the ports used by the input/output interfaces of each BB. As it can be seen, the interconnection of the *PDPDispatcher* BB and *PDPClient* BBs can be established by following input and output ports of these BBs, whereas the *PDPServer* BB always listening by the same port.

Figure 3 shows the BBs used to build the prototype of the PDP service as well as the operation performed by these BBs (Upload, verification and download). As it can be seen, the BBs of the verification service were deployed on five machines of a private cloud. Please notice that the *PDPDispatcher* BB (1) and the *PDPClient* BBs (4) are running in the same physical machine (labeled as Disys0), whereas he *PDPServer* BBs were deployed on different physical machines (Disys1-4).
All the physical machines have 6 cores, 12 GB RAM and 500 GB HD.

III. EVALUATION METHODOLOGY

The performance of the PDP service prototype was evaluated in two phases. In the first one, controlled experiments were conducted to identify the performance of each component of this service, whereas in the last one a study case based on satellite imagery was conducted to evaluate the functionality of the whole service.

A. Metrics, data source and experiments

The metrics of interest are the *service time*, which represents the time spent by a given BB or component of a given BB and the *response time*, which is measured from the file is read from the data source to this file is either uploaded or downloaded.

The controlled experimentation was performed by varying the size of the files in 10 MB form 1MB to 100MB.

For the evaluation of the study case, a sample of 70 satellite images from a repository of satellite imagery produced by the Soil Moisture Ocean Salinity [16] mission of the European Space Astronomy Center (ESAC). (Figure 4 shows a histogram with the different sizes of these images). These images were processed and stored in the cloud by using the PDP prototype

by using multiple workers (1, 2, 4) to evaluate the impact of the task parallelism on the service performance.

For each experiment in the study case, three different operations were evaluated: the first one is the *upload of the images* where the *PDPDispatcher* BB generates the lists of satellite images that the *PDPClient* BBs must process, each *PDPClient* BB (worker) performs the generation of its keys and transfers the data to its corresponding *PDPServer* BB in the cloud. The second one is the *Integrity Verification* where each *PDPClient* BB with its keys and the identifiers of the images that it uploaded, executes verification operations of

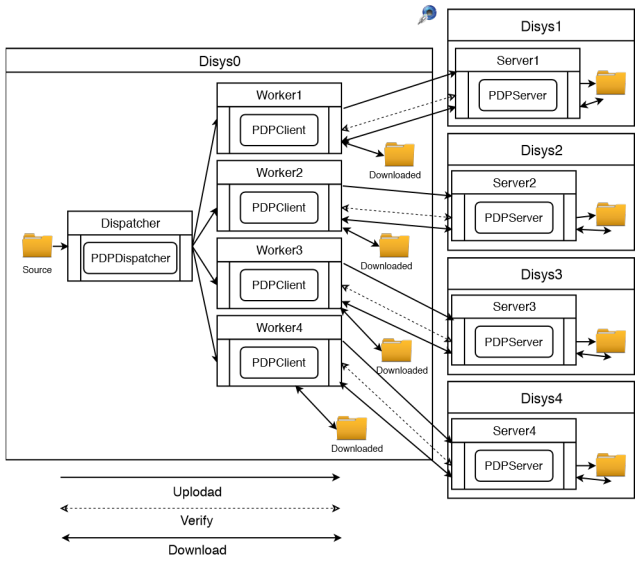

Fig. 3. The BBs of the PDP service prototype and the workflows of the upload, verification and download operations

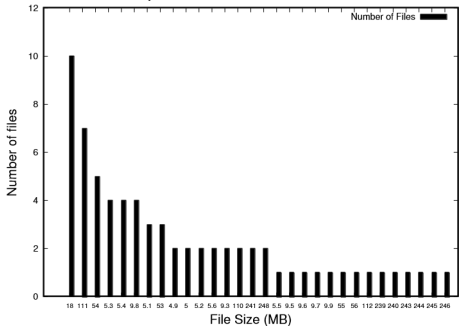

Fig. 4. Histogram of the 70 satellite image size

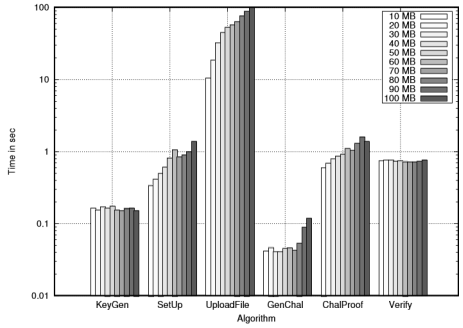

Fig. 5. Service time of PDP algorithms

these files. The last one is the download of the images, where each *PDPClient* BB downloads the files from its list that have not been modified (previously verified by a PDPServer).

IV. Results

In this section, the results are described in the two phases described in the methodology section: a controlled experimentation and a study case.

A. Controlled experimentation

The service and response times of the service with one single container (no parallel pattern is created in this experiment) are presented in this section.

Figure 5 shows the service times (vertical axis) produced by all the algorithms executed in *PDPClient* BB (horizontal axis). As it can be seen, the performance of $V\ erify$ and $KeyGen$ algorithms does not depend on the file size, which is the case for the rest of algorithms. Moreover, as expected, the major portion of the time of this service is spent in the uploading of the files.

Figure 6 shows the response times (vertical axis) produced by the challenges performed by *PDPServer* BB (horizontal axis). As expected, the file size impacts on the response time of this service: the more file size, the more response time.

B. Controlled experimentation

In this section, the prototype was evaluated in a scenario where a bot automatically sends operations of upload, verification and download by using the Manager/Worker pattern

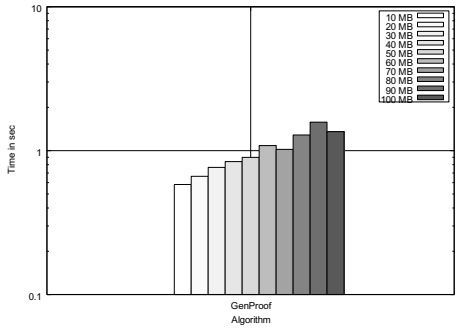

Fig. 6. Response time of verification operations

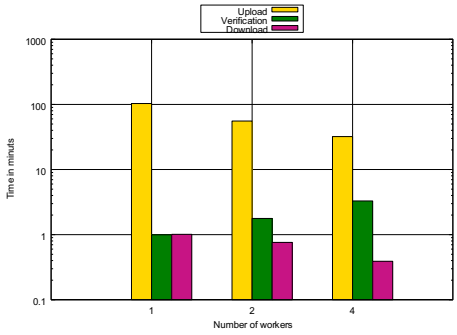

Fig. 7. Times obtained when loading, checking and downloading the data.

of 1,2 and 4 PDPClient-PDPServer containers (We recall the Figure 3 where the deployment of service is depicted).

Figure 7 shows the running times obtained when performing the different operations in logarithmic scale (Y axis) when using a different number of workers (X axis). As it can be observed, the data loading is the most expensive process, for example, the sending of 70 satellite images consumes 103.23 min (median). However, by increasing the number of workers the files can be loaded in a faster way. For example, when two workers are used, the time to load the files is 1.8X faster and 3.2X times faster when using 4 workers than the original algorithm (when using only one worker).

V. Conclusions

This paper presented the design and implementation of an efficient service based on provable data possession cryptographic model, which enables organizations to verify, ondemand, the data integrity without retrieving files from the cloud. The storage and cryptographic components have been developed in the form of building blocks, which are deployed on the user-side using the Manager/Worker pattern that favors exploiting parallelism when executing data possession challenges. An experimental evaluation in a private cloud revealed the efficacy of launching integrity verification challenges to cloud storage services and the feasibility of applying containerized task parallel scheme that significantly improves the performance of the data possession proof service in real-world scenarios in comparison with the implementation of the original algorithm. An important contribution of this paper, as shown in the study case, is the feasibility to apply building block model to adapt the patterns to the resources available in a given infrastructure. This prototype is used in on-going project for the conservation of satellite images captured by an antenna managed by a space agency.

Acknowledgment

This work has been partially funded by https://doi.org/10.13039/501100003141 GRANT Fondo Sectorial Mexican Space Agency-CONACYT Num. 262891 and by EU under the COST programme Action IC1305, Network for Sustainable Ultrascale Computing (NESUS).

References

[1] John Gantz and David Reinsel. Extracting value from chaos. *IDC iview*, 1142(2011):1–12, 2011.

[2] TJ Bittman and L Leong. Worldwide archival storage solutions 20112015 forecast: Archiving needs thrive in an information-thirsty world. *IDC. Market Analysis*, pages 1–21, 2011.

[3] John Gantz and David Reinsel. The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east. *IDC iView: IDC Analyze the future*, 2007(2012):1–16, 2012.

[4] Peter Mell, Tim Grance, et al. The nist definition of cloud computing. 2011.

[5] Richard Chow, Philippe Golle, Markus Jakobsson, Elaine Shi, Jessica Staddon, Ryusuke Masuoka, and Jesus Molina. Controlling data in the cloud: Outsourcing computation without outsourcing control. In *Proceedings of the 2009 ACM Workshop on Cloud Computing Security*, CCSW '09, pages 85–90. ACM, 2009.

[6] Amit Sangroya, Saurabh Kumar, Jaideep Dhok, and Vasudeva Varma. Towards analyzing data security risks in cloud computing environments. In *International Conference on Information Systems, Technology and Management*, pages 255–265. Springer, 2010.

[7] Zhifeng Xiao and Yang Xiao. Security and privacy in cloud computing. *IEEE Communications Surveys & Tutorials*, 15(2):843–859, 2013.

[8] Nesrine Kaaniche, Ethmane El Moustaine, and Maryline Laurent. A novel zero-knowledge scheme for proof of data possession in cloud storage applications. In *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*, pages 522–531. IEEE, 2014.

[9] Giuseppe Ateniese, Randal Burns, Reza Curtmola, Joseph Herring, Lea Kissner, Zachary Peterson, and Dawn Song. Provable data possession at untrusted stores. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 598–609. Acm, 2007.

[10] Darrel Hankerson, Alfred J Menezes, and Scott Vanstone. *Guide to elliptic curve cryptography*. Springer Science & Business Media, 2006.

[11] JL Gonzalez-Compean, Victor J Sosa-Sosa, Arturo Diaz-Perez, Jesus Carretero, and Ricardo Marcelin-Jimenez. Fedids: a federated cloud storage architecture and satellite image delivery service for building dependable geospatial platforms. *International Journal of Digital Earth*, pages 1–22, 2017.

[12] Jose Luis Gonzalez, Arturo Diaz-Perez, Victor Sosa-Sosa, Jesus Carretero Perez, and Jedidiah Yanez-Sierra. Sacbe: A modular software architecture for secure, reliable and flexible end-to-end cloud storage (sin publicar). 2016.

[13] Michael Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1094–1104, 2001.

[14] Angelo De Caro and Vincenzo Iovino. jpbc: Java pairing based cryptography. In *Proceedings of the 16th IEEE Symposium on Computers and Communications, ISCC 2011*, pages 850–855, Kerkyra, Corfu, Greece, June 28 - July 1, 2011.

[15] Docker inc. *Docker Community Edition*., 2016. [Online; accessed May 01, 2018].

[16] Pierluiggi Silvestrin, Michael Berger, Yann Kerr, and Jordi Font. Esas second earth explorer opportunity mission: The soil moisture and ocean salinity mission-smos. *IEEE Geoscience and Remote Sensing Newsletter*, 118:11–14, 2001.