

Freely Provable Ensure Circulated Responsibility for Data Distribution in the Cloud

Gattu Muni Kodhanda Ramakrishna ¹, S.Swetha ²

Research Scholar (M.Tech), Department of Computer Science and Engineering, Sree Rama Engineering College,
Tirupati, A.P, India¹

Assistant Professor, Department of Computer Science and Engineering, Sree Rama Engineering College, Tirupati,
A.P, India²

ABSTRACT: We focus on the more challenging multi-key scenario where data streams are uploaded by multiple data sources with distinct keys. We first present a novel homomorphism verifiable tag technique to publicly verify the outsourced inner product computation on the dynamic data streams, and then extend it to support the verification of matrix product computation. We prove the security of our scheme in the random oracle model. Moreover, the experimental result also shows the practicability of our design. we propose a novel highly decentralized information accountability framework to keep track of the actual usage of the users' data in the cloud. In particular, we propose an object-centered approach that enables enclosing our logging mechanism together with users' data and policies. We leverage the JAR programmable capabilities to both create a dynamic and traveling object, and to ensure that any access to users' data will trigger authentication and automated logging local to the JARs. To strengthen user's control, we also provide distributed auditing mechanisms. We provide extensive experimental studies that demonstrate the efficiency and effectiveness of the proposed approaches.

KEYWORDS: Efficiency, Cloud, JARs, Authentication.

I. INTRODUCTION

Cloud computing presents a new way to supplement the current consumption and delivery model for IT services based on the Internet, by providing for dynamically scalable and often virtualized resources as a service over the Internet. To date, there are a number of notable commercial and individual cloud computing services, including Amazon, Google, Microsoft, Yahoo, and Salesforce [19]. Details of the services provided are abstracted from the users who no longer need to be experts of technology infrastructure. Moreover, users may not know the machines which actually process and host their data. While enjoying the convenience brought by this new technology, users also start worrying about losing control of their own data. The data processed on clouds are often outsourced, leading to a number of issues related to accountability, including the handling of personally identifiable information. Such fears are becoming a significant barrier to the wide adoption of cloud services [30]. To allay users' concerns, it is essential to provide an effective mechanism for users to monitor the usage of their data in the cloud. For example, users need to be able to ensure that their data are handled according to the service level agreements made at the time they sign on for services in the cloud. Conventional access control approaches developed for closed domains such as databases and operating systems, or approaches using a centralized server in distributed environments, are not suitable, due to the following features characterizing cloud environments. First, data handling can be outsourced by the direct cloud service provider (CSP) to other entities in the cloud and these entities can also delegate the tasks to others, and so on. Second, entities are allowed to join and leave the cloud in a flexible manner. As a result, data handling in the cloud goes through a complex and dynamic hierarchical service chain which does not exist in conventional environments. To overcome the above problems, we propose a novel approach, namely Cloud Information Accountability (CIA) framework, based on the notion of information accountability [44]. Unlike privacy protection technologies which are built on the hide-it-or-

lose-it perspective, information accountability focuses on keeping the data usage transparent and trackable. Our proposed CIA framework provides end-to-end accountability in a highly distributed fashion. One of the main innovative features of the CIA framework lies in its ability of maintaining lightweight and powerful accountability that combines aspects of access control, usage control and authentication. By means of the CIA, data owners can track not only whether or not the service-level agreements are being honored, but also enforce access and usage control rules as needed. Associated with the accountability feature, we also develop two distinct modes for auditing: push mode and pull mode. The push mode refers to logs being periodically sent to the data owner or stakeholder while the pull mode refers to an alternative approach whereby the user (or another authorized party) can retrieve the logs as needed.

II. CLOUD INFORMATION ACCOUNTABILITY

we present an overview of the Cloud Information Accountability framework and discuss how the CIA framework meets the design requirements discussed in the previous section. The Cloud Information Accountability framework proposed in this work conducts automated logging and distributed auditing of relevant access performed by any entity, carried out at any point of time at any cloud service provider. It has two major components: logger and log harmonizer.

A. MAJOR COMPONENTS

There are two major components of the CIA, the first being the logger, and the second being the log harmonizer. The logger is the component which is strongly coupled with the user's data, so that it is downloaded when the data are accessed, and is copied whenever the data are copied. It handles a particular instance or copy of the user's data and is responsible for logging access to that instance or copy. The log harmonizer forms the central component which allows the user access to the log files. The logger is strongly coupled with user's data (either single or multiple data items). Its main tasks include automatically logging access to data items that it contains, encrypting the log record using the public key of the content owner, and periodically sending them to the log harmonizer. The logger requires only minimal support from the server (e.g., a valid Java virtual machine installed) in order to be deployed. The tight coupling between data and logger, results in a highly distributed logging system, therefore meeting our first design requirement. Furthermore, since the logger does not need to be installed on any system or require any special support from the server, it is not very intrusive in its actions, thus satisfying our fifth requirement. Finally, the logger is also responsible for generating the error correction information for each log record and send the same to the log harmonizer.

B. DATA FLOW

The overall CIA framework, combining data, users, logger and harmonizer is sketched in Fig. 1. At the beginning, each user creates a pair of public and private keys based on Identity-Based Encryption [4] (step 1 in Fig. 1). This IBE scheme is a Weil-pairing-based IBE scheme, which protects us against one of the most prevalent attacks to our architecture as described in Section 7. Using the generated key, the user will create a logger component which is a JAR file, to store its data items. The JAR file includes a set of simple access control rules specifying whether and how the cloud servers, and possibly other data stakeholders (users, companies) are authorized to access the content itself. Then, he sends the JAR file to the cloud service provider that he subscribes to. To authenticate the CSP to the JAR (steps 3-5 in Fig. 1), we use OpenSSLbased certificates, wherein a trusted certificate authority certifies the CSP. In the event that the access is requested by a user, we employ SAML-based authentication [8], wherein a trusted identity provider issues certificates verifying the user's identity based on his username.

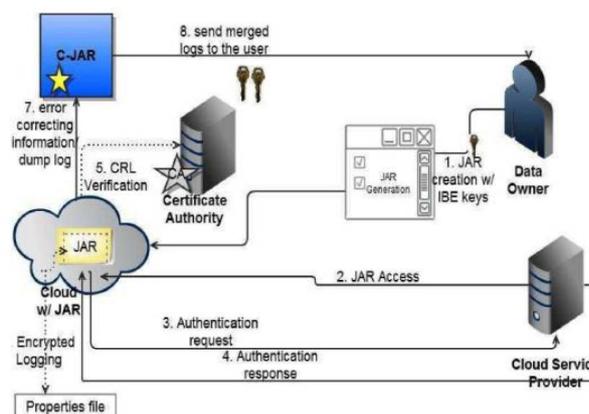


Fig. 1. Overview of the cloud information accountability framework.

Once the authentication succeeds, the service provider (or the user) will be allowed to access the data enclosed in the JAR. Depending on the configuration settings defined at the time of creation, the JAR will provide usage control associated with logging, or will provide only logging functionality. As for the logging, each time there is an access to the data, the JAR will automatically generate a log record, encrypt it using the public key distributed by the data owner, and store it along with the data (step 6 in Fig. 1). The encryption of the log file prevents unauthorized changes to the file by attackers.

III. AUTOMATED LOGGING MECHANISM

In this section, we first elaborate on the automated logging mechanism and then present techniques to guarantee dependability.

A. THE LOGGER STRUCTURE

We leverage the programmable capability of JARs to conduct automated logging. A logger component is a nested Java JAR file which stores a user's data items and corresponding log files. As shown in Fig. 2, our proposed JAR file consists of one outer JAR enclosing one or more inner JARs.

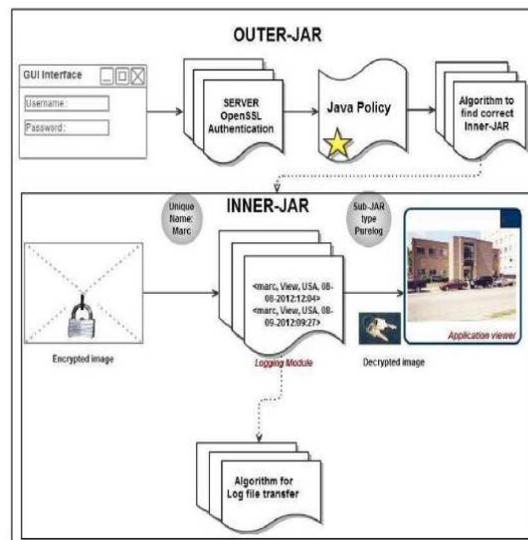


Fig. 2. The structure of the JAR file.

The main responsibility of the outer JAR is to handle authentication of entities which want to access the data stored in the JAR file. In our context, the data owners may not know the exact CSPs that are going to handle the data. Hence, authentication is specified according to the servers' functionality (which we assume to be known through a lookup service), rather than the server's URL or identity.

B. LOG RECORD GENERATION

Log records are generated by the logger component. Logging occurs at any access to the data in the JAR, and new log entries are appended sequentially, in order of creation $LR = \langle r_1, \dots, r_i \rangle$. Each record r_i is encrypted individually and appended to the log file. In particular, a log record takes the following form:

$$r_i = \langle ID, Act, T, Loc, h((ID, Act, T, Loc) \parallel r_{i-1} \parallel \dots \parallel r_1), sig \rangle.$$

Here, r_i indicates that an entity identified by ID has performed an action Act on the user's data at time T at location Loc . The component $h((ID, Act, T, Loc) \parallel r_{i-1} \parallel \dots \parallel r_1)$ corresponds to the checksum of the records preceding the newly inserted one, concatenated with the main content of the record itself (we use \parallel to denote concatenation). The checksum is computed using a collision-free hash function [37]. The component sig denotes the signature of the record created by the server. If more than one file is handled by the same logger, an additional $ObjID$ field is added to each record. An example of log record for a single file is shown below.

The entity (e.g., the cloud service provider) can only read the data but is not allowed to save a raw copy of it anywhere permanently. For this type of action, the PureLog will simply write a log record about the access, while the Access Logs will enforce the action through the enclosed access control module. Recall that the data are encrypted and 560 stored in the inner JAR.

Download.

The entity is allowed to save a raw copy of the data and the entity will have no control over this copy neither log records regarding access to the copy.

Timed access.

This action is combined with the view-only access, and it indicates that the data are made available only for a certain period of time.

Location-based access

In this case, the Pure Log will record the location of the entities. The Access Log will verify the location for each of such access. The access is granted and the data are made available only to entities located at locations specified by the data owner.

D. DEPENDABILITY OF LOGS

In this section, we discuss how we ensure the dependability of logs. In particular, we aim to prevent the following two types of attacks. First, an attacker may try to evade the auditing mechanism by storing the JARs remotely, corrupting the JAR, or trying to prevent them from communicating with the user. Second, the attacker may try to compromise the JRE used to run the JAR files.

JARs Availability

To protect against attacks perpetrated on offline JARs, the CIA includes a log harmonizer which has two main responsibilities: to deal with copies of JARs and to recover corrupted logs. Each log harmonizer is in charge of copies of logger components containing the same set of data items. The harmonizer is implemented as a JAR file. It does not contain the user's data items being audited, but consists of class files for both a server and a client processes to allow it to communicate with its logger components. The harmonizer stores error correction information sent from its logger components, as well as the user's IBE decryption key, to decrypt the log records and handle any duplicate records. Duplicate records result from copies of the user's data JARs.

Log Correctness

For the logs to be correctly recorded, it is essential that the JRE of the system on which the logger components are running remain unmodified. To verify the integrity of the logger component, we rely on a two-step process:

- 1) We repair the JRE before the logger is launched and any kind of access is given, so as to provide guarantees of integrity of the JRE.
- 2) We insert hash codes, which calculate the hash values of the program traces of the modules being executed by the logger component. This helps us detect modifications of the JRE once the logger component has been launched, and are useful to verify if the original code flow of execution is altered.

III. END-TO-END AUDITING MECHANISM

In this section, we describe our distributed auditing mechanism including the algorithms for data owners to query the logs regarding their data.

Push and Pull Mode

To allow users to be timely and accurately informed about their data usage, our distributed logging mechanism is complemented by an innovative auditing mechanism. We support two complementary auditing modes: 1) push mode; 2) pull mode.

Push mode

In this mode, the logs are periodically pushed to the data owner (or auditor) by the harmonizer. The push action will be triggered by either type of the following two events: one is that the time elapses for a certain period according to the temporal timer inserted as part of the JAR file; the other is that the JAR file exceeds the size stipulated by the content owner at the time of creation. After the logs are sent to the data owner, the log files will be dumped, so as to free the space for future access logs. Along with the log files, the error correcting information for those logs is also dumped.

Pull mode

This mode allows auditors to retrieve the logs anytime when they want to check the recent access to their own data. The pull message consists simply of an FTP pull command, which can be issues from the command line. For naive users, a wizard comprising a batch file can be easily built. The request will be sent to the harmonizer, and the user will be informed of the data's locations and obtain an integrated copy of the authentic and sealed log file.

IV. SECURITY DISCUSSION

We now analyze possible attacks to our framework. Our analysis is based on a semihonest adversary model by assuming that a user does not release his master keys to unauthorized parties, while the attacker may try to learn extra information from the log files. We assume that attackers may have sufficient Java programming skills to disassemble a JAR file and prior knowledge of our CIA architecture. We first assume that the JVM is not corrupted, followed by a discussion on how to ensure that this assumption holds true.

Copying Attack

The most intuitive attack is that the attacker copies entire JAR files. The attacker may assume that doing so allows accessing the data in the JAR file without being noticed by the data owner. However, such attack will be detected by our auditing mechanism. Recall that every JAR file is required to send log records to the harmonizer. In particular, with the push mode, the harmonizer will send the logs to data owners periodically. That is, even if the data owner is not aware of the existence of the additional copies of its JAR files, he will still be able to receive log files from all existing copies.

Disassembling Attack

Another possible attack is to disassemble the JAR file of the logger and then attempt to extract useful information out of it or spoil the log records in it. Given the ease of disassembling JAR files, this attack poses one of the most serious threats to our architecture. Since we cannot prevent an attacker to gain possession of the JARs, we rely on the strength of the cryptographic schemes applied to preserve the integrity and confidentiality of the logs. Once the JAR files are disassembled, the attacker is in possession of the public IBE key used for encrypting the log files, the encrypted log file itself, and the *.class files. Therefore, the attacker has to rely on learning the private key or subverting the encryption to read the log records.

Man-in-the-Middle Attack

An attacker may intercept messages during the authentication of a service provider with the certificate authority, and reply the messages in order to masquerade as a legitimate service provider. There are two points in time that the attacker can replay the messages. One is after the actual service provider has completely disconnected and ended a session with the certificate authority. The other is when the actual service provider is disconnected but the session is not over, so the attacker may try to renegotiate the connection. The first type of attack will not succeed since the certificate typically has a time stamp which will become obsolete at the time point of reuse. The second type of attack will also fail since renegotiation is banned in the latest version of Open SSL and cryptographic checks have been added.

Compromised JVM Attack

An attacker may try to compromise the JVM. To quickly detect and correct these issues, we discussed in Section 5.3 how to integrate oblivious hashing to guarantee the correctness of the JRE [11] and how to correct the JRE prior to execution, in case any error is detected. OH adds hash code to capture the computation results of each instruction and computes the oblivious-hash value as the computation proceeds. These two techniques allow for a first quick detection of errors due to malicious JVM, therefore mitigating the risk of running subverted JARs. To further strengthen our solution, one can extend OH usage to guarantee the correctness of the class files loaded by the JVM.

V. PERFORMANCE STUDY

In this section, we first introduce the settings of the test environment and then present the performance study of our system.

Experimental Settings

We tested our CIA framework by setting up a small cloud, using the Emulab testbed [42]. In particular, the test environment consists of several OpenSSL-enabled servers: one head node which is the certificate authority, and several computing nodes.

Experimental Results

In the experiments, we first examine the time taken to create a log file and then measure the overhead in the system. With respect to time, the overhead can occur at three points: during the authentication, during encryption of a log record, and during the merging of the logs. Also, with respect to storage overhead, we notice that our architecture is very lightweight, in that the only data to be stored are given by the actual files and the associated logs.

Log Creation Time

In the first round of experiments, we are interested in finding out the time taken to create a log file when there are entities continuously accessing the data, causing continuous logging. Results are shown in Fig. 5. It is not surprising to see that the time to create a log file increases linearly with the size of the log file.

Authentication Time

The next point that the overhead can occur is during the authentication of a CSP. If the time taken for this authentication is too long, it may become a bottleneck for accessing the enclosed data.

Time Taken to Perform Logging

This set of experiments studies the effect of log file size on the logging performance. We measure the average time taken to grant an access plus the time to write the corresponding log record. The time for granting any access to the data items in a JAR file includes the time to evaluate and enforce the applicable policies and to locate the requested data items.

Log Merging Time

To check if the log harmonizer can be a bottleneck, we measure the amount of time required to merge log files. In this experiment, we ensured that each of the log files had 10 to 25 percent of the records in common with one other. The exact number of records in common was random for each repetition of the experiment. We proposed innovative approaches for automatically logging any access to the data in the cloud together with an auditing mechanism. Our approach allows the data owner to not only audit his content but also enforce strong back-end protection if needed. Moreover, one of the main features of our work is that it enables the data owner to audit even those.

REFERENCES

- [1] P. Ammann and S. Jajodia, "Distributed Timestamp Generation in Planar Lattice Networks," *ACM Trans. Computer Systems*, vol. 11, pp. 205-225, Aug. 1993.
- [2] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song, "Provable Data Possession at Untrusted Stores," *Proc. ACM Conf. Computer and Comm. Security*, pp. 598-609, 2007.
- [3] E. Barka and A. Lakas, "Integrating Usage Control with SIP-Based Communications," *J. Computer Systems, Networks, and Comm.*, vol. 2008, pp. 1-8, 2008.
- [4] D. Boneh and M.K. Franklin, "Identity-Based Encryption from the Weil Pairing," *Proc. Int'l Cryptology Conf. Advances in Cryptology*, pp. 213-229, 2001.
- [5] R. Bose and J. Frew, "Lineage Retrieval for Scientific Data Processing: A Survey," *ACM Computing Surveys*, vol. 37, pp. 1-28, Mar. 2005.
- [6] P. Buneman, A. Chapman, and J. Cheney, "Provenance Management in Curated Databases," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '06)*, pp. 539-550, 2006.
- [7] B. Chun and A.C. Bavier, "Decentralized Trust Management and Accountability in Federated Systems," *Proc. Ann. Hawaii Int'l Conf. System Sciences (HICSS)*, 2004.
- [8] OASIS Security Services Technical Committee, "Security Assertion Markup Language (saml) 2.0," http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=security, 2012.
- [9] R. Corin, S. Etalle, J.I. den Hartog, G. Lenzini, and I. Staicu, "A Logic for Auditing Accountability in Decentralized Systems," *Proc. IFIP TC1 WG1.7 Workshop Formal Aspects in Security and Trust*, pp. 187-201, 2005.
- [10] B. Crispo and G. Ruffo, "Reasoning about Accountability within Delegation," *Proc. Third Int'l Conf. Information and Comm. Security (ICICS)*, pp. 251-260, 2001.
- [11] Y. Chen et al., "Oblivious Hashing: A Stealthy Software Integrity Verification Primitive," *Proc. Int'l Workshop Information Hiding*, F. Petitcolas, ed., pp. 400-414, 2003.
- [12] S. Etalle and W.H. Winsborough, "A Posteriori Compliance Control," *SACMAT '07: Proc. 12th ACM Symp. Access Control Models and Technologies*, pp. 11-20, 2007.
- [13] X. Feng, Z. Ni, Z. Shao, and Y. Guo, "An Open Framework for Foundational Proof-Carrying Code," *Proc. ACM SIGPLAN Int'l Workshop Types in Languages Design and Implementation*, pp. 67-78, 2007.
- [14] Flickr, <http://www.flickr.com/>, 2012.
- [15] R. Hasan, R. Sion, and M. Winslett, "The Case of the Fake Picasso: Preventing History Forgery with Secure Provenance," *Proc. Seventh Conf. File and Storage Technologies*, pp. 1-14, 2009.
- [16] J. Hightower and G. Borriello, "Location Systems for Ubiquitous Computing," *Computer*, vol. 34, no. 8, pp. 57-66, Aug. 2001.
- [17] J.W. Holford, W.J. Caelli, and A.W. Rhodes, "Using Self-Defending Objects to Develop Security Aware Applications in Java," *Proc. 27th Australasian Conf. Computer Science*, vol. 26, pp. 341-349, 2004.
- [18] Trusted Java Virtual Machine IBM, <http://www.almaden.ibm.com/cs/projects/jvm/>, 2012.
- [19] P.T. Jaeger, J. Lin, and J.M. Grimes, "Cloud Computing and Information Policy: Computing in a Policy Cloud?," *J. Information Technology and Politics*, vol. 5, no. 3, pp. 269-283, 2009.
- [20] R. Jagadeesan, A. Jeffrey, C. Pitcher, and J. Riely, "Towards a Theory of Accountability and Audit," *Proc. 14th European Conf. Research in Computer Security (ESORICS)*, pp. 152-167, 2009.