

Promoting Distributed Accountability in the Cloud

Smitha Sundareswaran
 Pennsylvania State University
 sus263@psu.edu

Anna Squicciarini
 Pennsylvania State University
 asquicciarini@ist.psu.edu

Dan Lin
 Missouri S&T
 lindan@mst.edu

Shuo Huang
 Pennsylvania State University
 shuohuang@cse.psu.edu

Abstract—Cloud computing enables highly scalable services to be easily consumed over the Internet on an as-needed basis. A major feature of the cloud services is that users’ data is usually processed remotely in unknown machines that users do not own or operate. While enjoying the convenience brought by this new emerging technology, users’ fears of losing control of their own data (particularly financial and health data) can become a significant barrier to the wide adoption of cloud services. To address this problem, in this paper, 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 leverage the programmable capability of Java JAR files to enclose our logging mechanism together with users’ data and policies. Our approach ensures 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. Our experimental study demonstrates the efficiency and effectiveness of the proposed approaches.

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 [11]. 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 is 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 [19].

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 is 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* [28]. 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 light-weight 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.

The design of the CIA framework presents substantial challenges, including uniquely identifying CSPs, ensuring the reliability of the log, adapting to a highly decentralized infrastructure, etc. Our basic approach toward addressing these issues is to leverage and extend the programmable capability of JAR (Java ARchives) files to automatically log the usage of the users’ data by any entity in the cloud. Users will send their data along with any policies such as access control policies and logging policies that they want to enforce, enclosed in JAR files, to cloud service providers. Any access to the data will trigger an automated and authenticated logging mechanism local to the JARs. We refer to this type of enforcement as “strong binding” since the policies and the logging mechanism travel with the data. This strong binding exists even when copies of the JARs are created, thus the user will have control over his data at any location. Such decentralized logging mechanism meets the dynamic nature of the cloud but also imposes challenges on ensuring the integrity of the logging. To cope with this issue, we provide the JARs with a central point of contact which forms a link between them and the user. It records the error correction information sent by the JARs,

which allows it to monitor the loss of any logs from any of the JARs. Moreover, if a JAR is not able to contact its central point, any access to its enclosed data will be denied.

Currently, we focus on image files since images represent a very common content type for end-users and organizations, and are increasingly hosted in the cloud as part of the storage services offered by the utility computing paradigm featured by cloud computing. Further, images often reveal social and personal habits of users, or are used for archiving important files from organizations.

In summary, our main contributions are as follows:

- We propose a novel automatic and enforceable logging mechanism in the cloud. To our knowledge, this is the first time a systematic approach to data accountability through the novel usage of JAR files is proposed.
- Our proposed architecture is platform-independent and highly decentralized, in that it does not require any dedicated authentication or storage system in place.
- We go beyond traditional access control in that we provide a certain degree of usage control for the protected data after it is delivered to the receiver.
- We conducted experiments on a real cloud testbed. The results demonstrate the efficiency, scalability and granularity of our approach. We also provide a detailed security analysis and discuss the reliability and strength of our architecture in the face of various non-trivial attacks.

The rest of the paper is organized as follows. Section II lays out our problem statement. Section III presents our proposed Cloud Information Accountability framework, and Section IV and V describe the detailed algorithms for automated logging mechanism and auditing approaches respectively. Section VI presents a security analysis of our framework, followed by an experimental study in Section VII. Section VIII discusses related work. We conclude in Section X.

II. PROBLEM STATEMENT

We begin this work by identifying the common requirements and develop several guidelines to achieve data accountability in the cloud. A user who subscribed to a certain cloud service, usually needs to send his/her data as well as associated access control policies (if any) to the service provider. After the data is received by the cloud service provider, the service provider will have granted access rights, such as read, write and copy, on the data. Using conventional access control mechanisms, once the access rights are granted, the data will be fully available at the service provider. In order to track the actual usage of the data, we aim to develop novel logging and auditing techniques which satisfy the following requirements:

- 1) The logging should be decentralized in order to adapt to the dynamic nature of the cloud. More specifically, log files should be tightly bound with the corresponding data being controlled, and require minimal infrastructural support from any server.
- 2) Every access to the user's data should be correctly and automatically logged. This requires integrated techniques to authenticate the entity who accesses the data,

verify and record the actual operations on the data as well as the time that the data has been accessed.

- 3) Log files should be reliable and tamper proof to avoid illegal insertion, deletion and modification by malicious parties. Recovery mechanisms are also desirable to restore damaged log files caused by technical problems.
- 4) Log files should be sent back to their data owners periodically to inform them of the current usage of their data. More importantly, log files should be retrievable anytime by their data owners when needed regardless the location where the files are stored.
- 5) The proposed technique should not intrusively monitor data recipients' systems, nor it should introduce heavy communication and computation overhead, which otherwise will hinder its feasibility and adoption in practice.

III. CLOUD INFORMATION ACCOUNTABILITY

The Cloud Information Accountability (CIA) 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*.

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 and periodically sending them to the log harmonizer. It may also be configured to ensure that access and usage control policies associated with the data are honored. For example, a data owner can specify that user X is only allowed to view but not to modify the data. The logger will control the data access even after it is downloaded by user X .

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. The error correction information combined with the encryption and authentication mechanism, provides a robust and reliable recovery mechanism, therefore meeting the third requirement.

The log harmonizer is responsible for auditing. It supports two auditing strategies: *push* and *pull*. Under the push strategy, the log file is pushed back to the data owner periodically in an automated fashion. The pull mode is an on-demand approach, whereby the log file is obtained by the data owner as often as requested. These two modes allow us to satisfy the aforementioned fourth design requirement. In case there exist multiple loggers for the same set of data items, the log harmonizer will merge log records from them before sending back to the data owner. The log harmonizer is also responsible for handling log file corruption. In addition, the log harmonizer can itself carry out logging in addition to auditing. Separating the logging

and auditing functions improves the performance. The logger and the log harmonizer are both implemented as lightweight and portable JAR files. The JAR file implementation provides automatic logging functions, which meets the second design requirement.

The data flow in the CIA framework is as follows. At the beginning, each user creates a pair of public and private keys based on Identity-Based Encryption (IBE) [2]. Using the generated key, the user will create a logger (i.e., a JAR file) to store its data items, and sign and seal it. The JAR file includes a set of access control rules specifying whether and how the cloud servers, and possibly other data stakeholders are authorized to access the data. Then, he/she sends the JAR file to the cloud service provider (CSP) that he/she subscribes to. To authenticate the CSP to the JAR, we use OpenSSL based 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 [18], wherein a trusted identity provider issues certificates verifying the user's identity based on his username. Using SAML-based authentication for the users allows us to increase or decrease the number of users to whom the access is granted by simply adding more user identities and corresponding SAML certificates. Using OpenSSL for the CSP allows us to ensure that we grant access based on the role of the CSP. Thus, this design ensures that our architecture scales well both for users and CSPs.

Once the authentication succeeds, the CSP (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. The encryption of the log file prevents unauthorized changes to the file by attackers. The data owner could opt to reuse the same key pair for all JARs or create different key pairs for separate JARs. Using separate keys can enhance the security without introducing any overhead except in the initialization phase. In addition, some error correction information will be sent to the log harmonizer to handle possible log file corruption.

The encrypted log files can later be decrypted and accessed by the data owner or other authorized stakeholders at any time for auditing purposes with the aid of the log harmonizer.

As discussed in Section VI, our proposed framework prevents various attacks such as detecting illegal copies of users' data. Note that our work is different from traditional logging methods which use encryption to protect log files. With only encryption, their logging mechanisms are neither automatic nor distributed. They require the data to stay within the boundaries of the centralized system for the logging to be possible, which is however not suitable in the cloud.

IV. AUTOMATED LOGGING MECHANISM IN THE CLOUD

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

A. Overview

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 Figure 1, our proposed JAR file consists of one outer JAR enclosing one or more inner JARs.

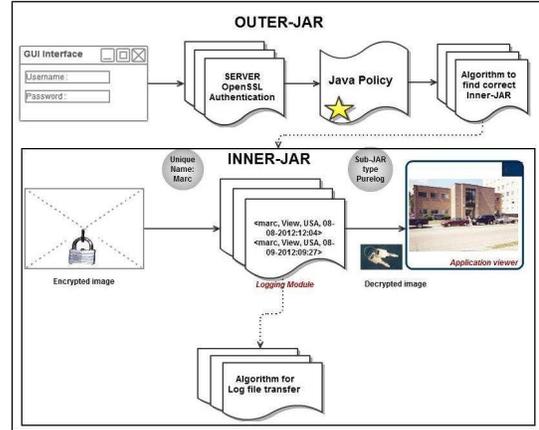


Fig. 1. The Structure of the JAR File

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. For example a policy may state that Server X is allowed to download the data if it is a storage server. As discussed below, the outer JAR may also have the access control functionality to enforce the data owner's requirements, specified as Java policies, on the usage of the data. A Java policy specifies which permissions are available for a particular piece of code in a Java application environment¹. Moreover, the outer JAR is also in charge of selecting correct inner-JAR according to the identity of the entity who requests the data.

Each inner JAR contains the encrypted data, class files to facilitate retrieval of log files and display enclosed data in a suitable format, and a log file for each encrypted item. We support options.

- *PureLog*: Its main task is to record every access to the data. The log files are used for pure auditing purpose.
- *AccessLog*: It has two functions: logging actions and enforcing access control. In case an access request is

¹The permissions expressed in the Java policy are in terms of File System Permissions. However, the data owner can specify the permissions in user-centric terms as opposed to the usual code-centric security offered by Java.

denied, the JAR will record the time when the request is made. If the access request is granted, the JAR will additionally record the access information along with the duration for which the access is allowed.

The two kinds of logging modules allow the data owner to enforce certain access conditions either pro-actively (in case of AccessLogs) or reactively (in case of PureLogs). For example, services like billing may just need to use PureLogs. AccessLogs will be necessary for services which need to enforce service level agreements such as limiting the visibility to some sensitive content at a given location.

B. Log Record Generation

Log records are generated by the logger which is triggered by any access to the data in the JAR. Each record is encrypted individually and added to the log file one by one. In particular, a log record is in the form of $\langle ID, Act, Time, Loc \rangle$, indicating that an entity identified by ID has performed an action Act on the user's data at time $Time$ at location Loc . 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.

To ensure the correctness of the log records, we verify the access time, locations as well as actions. In particular, the time of access is determined using the Network Time Protocol [21] to avoid suppression of the correct time by a malicious entity. The location of the cloud service provider (CSP) can be determined using IP address. The JAR can perform an IP lookup and use the range of the IP address to find the most probable location of the CSP. More advanced techniques for determining locations can also be used [10]. The most critical part is to log the actions on the users' data. In the current system, we support four types of actions, i.e., Act has one of the following four values: *view*, *download*, *timed_access*, and *Location-based_access*. For each action, we propose a specific method to correctly record or enforce it depending on the type of the logging module, which are elaborated as follows.

View: 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 AccessLogs will enforce the action through the enclosed access control module. Recall that the data is encrypted and stored in the inner JAR. When there is a view-only access request, the inner JAR will decrypt the data on the fly and create a temporary decrypted file. The decrypted file will then be displayed to the entity using the Java application viewer in case the file is displayed to a human user. Presenting the data in the Java application viewer disables the copying functions using right click or other hot keys such as PrintScreen. Further, to prevent the use of some screen capture software, the data will be hidden whenever the application viewer screen is out of focus. The content is displayed using the headless mode in Java on the command line when it is presented to a CSP.

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.

If PureLog is adopted, the user's data will be directly downloadable in a pure form using a link. When an entity clicks this download link, the JAR file associated with the data will decrypt the data and give it to the entity in raw form. In case of AccessLogs, the entire JAR file will be given to the entity. If the entity is a human user, he/she just needs to double click the JAR file to obtain the data. If the entity is a CSP, it can run a simple script to execute the JAR.

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

The PureLog will just record the access starting time and its duration, while the AccessLog will enforce that the access is allowed only within the specified period of time. The duration for which the access is allowed is calculated using the Network Time Protocol (NTP). To enforce the limit on the duration, the AccessLog records the start time using the NTP, and then uses a timer to stop the access. Naturally, this type of access can be enforced only when it is combined with the *View* access right and not when it is combined with the *Download*.

Location-based_access: In this case, the PureLog will record the location of the entities. The AccessLog will verify the location for each of such access. The access is granted and the data is made available only to entities located at locations specified by the data owner.

C. Dependability of Logs

The distributed nature of our approach poses some non-trivial security challenges. For example, an attacker can try to evade the auditing mechanism by storing the JARs remotely, corrupting the JAR or trying to prevent them from communicating with the user. To address this problem, 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. Since user's data is strongly coupled with the logger component in a data JAR file, the logger will be copied together with the user's data. Consequently, the new copy of the logger contains the old log records with respect to the usage of data in the original data JAR file. Such old log records are redundant and irrelevant to the new copy of the data. To present the data owner an integrated view, the harmonizer will merge log records from all copies of the data JARs by eliminating redundancy.

For recovering purpose, logger components are required to send error correction information to the harmonizer after

writing each log record. Therefore, logger components always ping the harmonizer before they grant any access right. If the harmonizer is not reachable, the logger components will deny all access. In this way, the harmonizer helps prevent attacks which attempt to keep the data JARs offline for unnoticed usage. If the attacker took the data JAR offline after the harmonizer was pinged, the harmonizer still has the error correction information about this access and will quickly notice the missing record.

In case of corruption of JAR files, the harmonizer will recover the logs with the aid of Reed-Solomon error correction code [29]. Specifically, each individual logging JAR, when created, contains a Reed-Solomon based encoder. For every n symbols in the log file, n redundancy symbols are added to the log harmonizer in the form of bits. This creates an error correcting code of size $2n$ and allows the error-correction to detect and correct n errors. We choose the Reed-Solomon code as it achieves the equality in the Singleton Bound [22], making it a maximum distance separable code and hence leads to an optimal error correction.

The log harmonizer is located at a known IP address. Typically, the harmonizer resides at the user's end as part of his local machine, or alternatively, it can either be stored in a user's desktop or in a proxy server.

V. END-TO-END AUDITING MECHANISM

To allow users to be timely and accurately informed about their data usage, our distributed logging mechanism is complemented by an innovative auditing mechanism which has two complementary auditing modes: (i) push mode; (ii) pull mode.

Push Mode: In this mode, the logs are periodically pushed to the data owner 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.

This push mode is the basic mode which can be adopted by both the PureLog and the AccessLog, regardless of whether there is a request from the data owner for the log files. This mode serves two essential functions in the logging architecture: 1) It ensures that the size of the log files does not explode and 2) It enables timely detection and correction of any loss or damage to the log files.

Pull Mode: This mode allows data owners 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 issued 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.

Pushing or pulling strategies have interesting trade-offs. The pushing strategy is beneficial when there are a large number of accesses to the data within a short period of time. In this case, if the data is not pushed out frequently enough, the log file may become very large, which may increase cost of operations like copying data (see Section VII). The pushing mode may be preferred by data owners who are organizations and need to keep track of the data usage consistently over time. For such data owners, receiving the logs automatically can lighten the load of the data analyzers. The maximum size at which logs are pushed out is a parameter which can be easily configured while creating the logger component. The pull strategy is most needed when the data owner suspects some misuse of his data; the pull mode allows him to monitor the usage of his content immediately. A hybrid strategy can actually be implemented to benefit of the consistent information offered by pushing mode and the convenience of the pull mode. Further, as discussed in Section VI, supporting both pushing and pulling modes helps protecting from some non-trivial attacks.

VI. SECURITY DISCUSSION

We now analyze possible attacks to our framework. Our analysis is based on a semi-honest 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. If attackers move copies of JARs to places where the harmonizer cannot connect, the copies of JARs will soon become inaccessible. This is because each JAR is required to write redundancy information to the harmonizer periodically. If the JAR cannot contact the harmonizer, the access to the content in the JAR will be disabled. Thus the logger component provides more transparency than conventional log files encryption; it allows the data owner to detect when an attacker has created copies of a JAR, and it makes offline files inaccessible.

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.

As for the first case, the attacker may try to identify which encrypted log records correspond to his actions by mounting a chosen plaintext attack to obtain some pairs of encrypted

log records and plain texts. However, the adoption of the Weil Pairing algorithm ensures that the CIA framework has both chosen ciphertext security and chosen plaintext security in the random oracle model [2]. Therefore, the attacker will not be able to decrypt any data or log files in the disassembled JAR file. Even if the attacker is an authorized user, he can only access the actual content file but he is not able to decrypt any other data including the log files which are viewable only to the data owner. From the disassembled JAR files, the attackers are not able to directly view the access control policies either, since the original source code is not included in the JAR files. If the attacker wants to infer access control policies, the only possible way is through analyzing the log file. This is, however, very hard to accomplish since, as mentioned earlier, log records are encrypted and breaking the encryption is computationally hard.

For the same reason, the attacker cannot modify the log files extracted from a disassembled JAR. Attackers will also not be able to write fake records to log files since JAR files are signed and sealed by the data owner. Sealing ensures that all packages within the JAR file come from the same source code [17]. Sealing is one of the Java properties, which allows creating a signature that does not allow the code inside the JAR file to be changed. That is to say, an attacker cannot write to the JAR even if he can read from it by disassembling it - he cannot “reassemble” it with modified packages. In case the attacker guesses or learns the data owner’s key from somewhere, all the JAR files using the same key will be compromised. Thus, using different IBE key pairs for different JAR files will be more secure and prevent such attack.

Finally, the attacker may try to modify the Java classloader in the JARs in order to subvert the class files when they are being loaded. This attack is prevented as follows. The JARs check the classloader each time before granting any access right. If the classloader is found as a custom classloader, the JARs will throw an exception and halt.

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 timestamp 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 OpenSSL and cryptographic checks have been added.

Compromised JVM Attack. Our solution relies on the correctness of the JVM. If the JVM is compromised, all the code executed on it can be compromised. However, these attacks can be prevented in several ways. One approach is to verify the integrity of a virtual machine prior to running the JARs,

for which the data owner needs to check whether the image of the virtual machine being loaded into the memory matches that of the provider. To verify the virtual machine’s image, we can employ an additional shell script which downloads the checksum for the JRE from Sun. Additionally, the script tests the version and author of the JRE. The shell script is bundled with the JAR file and will be executed before JAR. Therefore, simply copying the JAR does not allow an attacker to use his own compromised JRE as the bundled JVM is also copied with it. While the JAR itself can be disassembled, the content will not be accessible unless the entire JAR is executed, thus guaranteeing that the JVM packaged with the JAR is always used when the content is accessed. This approach is very efficient (order of 12 ms) and can be used for files which are not very sensitive.

Alternatively, prior to opening the JAR, a command can be included in the script to repair the JVM using the version provided by Sun. If we assume the cloud sitting on top of an untrusted network, we can also package a JRE to be used along with the JARs. In this way, we eliminate the possibility of a corrupted JRE and therefore the corrupted JVM at the server. The JRE itself can be easily packaged using any of the readymade open source installers such as VMKit or the GCJ toolkit. These approaches, while effective can still be bypassed by a skilled adversary. A more secure approach is to integrate Oblivious Hashing (OH) to guarantee the correctness of the JRE [8]. OH works by adding additional hash codes into the software executing a particular program. The hash code captures the computation results of each instruction and computes the oblivious-hash value as the computation proceeds. To leverage this technique, the JRE installer needs to be augmented so that it can capture the results of the instructions being carried out while installing the JRE. Therefore, we still need to package the JRE and its shell-script based installer with the JAR file. OH verifies the integrity of the JVM during loading and during run time. To further strengthen our solution, one can extend OH usage to guarantee the correctness of the class files loaded by the JVM.

VII. PERFORMANCE STUDY

We tested our CIA framework by setting up a small cloud, using the Emulab testbed [26]. In particular, the test environment consists of several OpenSSL-enabled servers: one head node which is the certificate authority, and several computing nodes. Each of the servers is installed with Eucalyptus [25]. Eucalyptus is an open source cloud implementation for Linux based systems. It is loosely based on Amazon EC2, therefore bringing the powerful functionalities of Amazon EC2 into the open source domain. We used Linux-based servers running Fedora 10 OS for our experiments. Each server has a 64-bit Intel Quad Core Xeon E5530 processor, 4 GB RAM, and a 500 GB Hard Drive. Each server is equipped to run the OpenJDK runtime environment with a version of IcedTea6 1.8.2. In the experiments, we first examine the time taken to create a log file and then measure the storage overhead in the system.

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. It is not surprising that the time to create a log file increases linearly with the size of the log file. Specifically, the time to create a 100Kb file is about 114.5ms while the time to create a 1 MB file averages at 731ms. With this experiment as the baseline, one can decide the amount of time to be specified between dumps, keeping other variables like space constraints or network traffic in mind.

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. To evaluate this, the head node issued OpenSSL certificates for the computing nodes and we measured the total time for the OpenSSL authentication to be completed and the certificate revocation to be checked. Considering one access at the time, we find that the authentication time averages around 920 ms which proves that not too much overhead is added during this phase. As of present, the authentication takes place each time the CSP needs to access the data. The performance can be further improved by caching the certificates.

The time for authenticating an end user is about the same when we consider only the actions required by the JAR, viz. obtaining a SAML certificate and then evaluating it. This is because both the OpenSSL and the SAML certificates are handled in a similar fashion by the JAR. When we consider the user actions, such as submitting his username to the JAR, it averages at 1.2 minutes.

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.

In the experiment, we let multiple servers continuously access the same data JAR file for a minute and recorded the number of log records generated. Each access is just a view request and hence the time for executing the action is negligible. As a result, the average time to log an action is about 10 seconds, which includes the time taken by a user to double click the JAR or by a server to run the script to open the JAR. We also measured the log encryption time which is about 300ms (per record) and is relatively constant when the log file size increases.

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% of the records in common with one other. The exact number of records in common was random for each repetition of the experiment. The time was averaged over 10 repetitions. We tested the time to merge up to 70 log files of 100KB, 300KB, 500KB, 700KB, 900KB, and 1 MB each. The results are shown in Figure 2. We can observe that

the time increases almost linearly to the number of files and size of files, with the least time being taken for merging two 100KB log files at 59 ms, while the time to merge 70 1 MB files was 2.35 minutes.

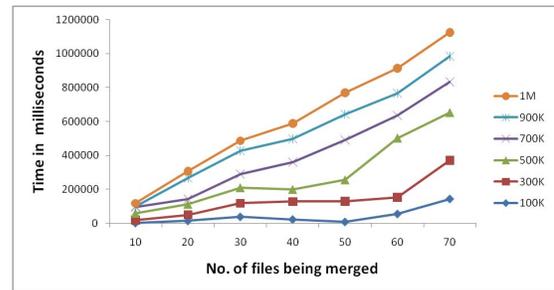


Fig. 2. Time to Merge Log Files

Size of the Data JAR Files Finally, we investigate whether a single logger, used to handle more than one file, results in storage overhead. We measure the size of the loggers (JARs) by varying the number and size of data items held by them. We tested the increase in size of the logger containing 10 content files (i.e. images) of the same size as the file size increases. Intuitively, in case of larger size of data items held by a logger, the overall logger also increases in size. The size of logger grows from 3500KB to 4035KB when the size of content items changes from 200KB to 1MB. Overall, due to the compression provided by JAR files, the size of the logger is dictated by the size of the largest files it contains. Notice that we purposely did not include large log files (less than 5KB), so as to focus on the overhead added by having multiple content files in a single JAR.

VIII. RELATED WORK

Cloud computing has raised a range of important *privacy* and *security* issues [11], [15], [19]. Such issues are due to the fact that, in the cloud, users' data and applications reside – at least for a certain amount of time – on the cloud cluster which is owned and maintained by a third party. Concerns arise since in the cloud it is not always clear to individuals why their personal information is requested or how it will be used or passed on to other parties. To date, little work has been done in this space, in particular with respect to accountability. Pearson et al. have proposed accountability mechanisms to address privacy concerns of end users [19] and then develop a privacy manager [20]. Their basic idea is that the user's private data is sent to the cloud in an encrypted form, and the processing is done on the encrypted data. The output of the processing is de-obfuscated by the privacy manager to reveal the correct result. However, the privacy manager provides only limited features in that it does not guarantee protection once the data is being disclosed. In [5], the authors present a layered architecture for addressing the end-to-end trust management and accountability problem in federated systems. The authors' focus is very different from ours, in that they mainly leverage trust relationships for accountability, along with authentication

and anomaly detection. Further, their solution requires third party services to complete the monitoring and focuses on lower level monitoring of system resources.

Researchers have investigated accountability mostly as a provable property through cryptographic mechanisms, particularly in the context of electronic commerce [13], [7]. A representative work in this area is given by [6]. The authors propose the usage of policies attached to the data and present a logic for accountability data in distributed settings. Similarly, Jagadesaan et al. recently proposed a logic for designing accountability-based distributed systems [12]. However, these works stay at a theoretical level and do not include any algorithm for tasks like mandatory logging.

To the best of our knowledge, the only work proposing a distributed approach to accountability is from Lee and colleagues [14]. The authors have proposed an agent-based system specific to grid computing. Distributed jobs, along with the resource consumption at local machines are tracked by static software agents. The notion of accountability policies in [14] is related to ours, but it is mainly focused on resource consumption and on tracking of sub-jobs processed at multiple computing nodes, rather than access control.

In previous work, we provided a Java-based approach to prevent privacy leakage from indexing [24], which could be integrated with the CIA framework proposed in this work since they build on related architectures. Further, Mont et al. provided an approach for strongly coupling content with access control, using IBE [16]. We also leverage IBE techniques, but in a very different way; we do not rely on IBE to bind the content with the rules. Instead, we use it to provide strong guarantees for the encrypted content and the log files, such as protection against chosen plaintext and ciphertext attacks.

In addition, general outsourcing techniques have been investigated over the past few years [1], [23]. Although only [27] is specific to the cloud, some of the outsourcing protocols may also be applied in this realm. In this work, we do not cover issues of data storage security which are a complementary aspect of the privacy issues.

Finally, our work may look similar to works on secure data provenance [9], [3], [4], but in fact greatly differs from them in terms of goals, techniques and application domains. Works on data provenance aim to guarantee data integrity by securing the data provenance. They ensure that no one can add or remove entries in the middle of a provenance chain without detection, so that data is correctly delivered to the receiver. Differently, our work is to provide data accountability, to monitor the usage of the data and ensure that any access to the data is tracked.

IX. CONCLUSION AND FUTURE RESEARCH

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 copies of its data that were made without his knowledge.

REFERENCES

- [1] G. Ateniese and *et. al.* Provable data possession at untrusted stores. In *ACM conference on Computer and communications security*, pages 598–609, 2007.
- [2] D. Boneh and M. K. Franklin. Identity-based encryption from the weil pairing. In *International Cryptology Conference on Advances in Cryptology*, pages 213–229, 2001.
- [3] R. Bose and J. Frew. Lineage retrieval for scientific data processing: a survey. *ACM Comput. Surv.*, 37:1–28, March 2005.
- [4] P. Buneman, A. Chapman, and J. Cheney. Provenance management in curated databases. In *2006 ACM SIGMOD international conference on Management of data*, pages 539–550, 2006.
- [5] B. Chun and A. C. Bavier. Decentralized trust management and accountability in federated systems. In *HICSS*, 2004.
- [6] R. Corin, S. Etalle, J. I. den Hartog, G. Lenzi, and I. Staicu. A logic for auditing accountability in decentralized systems. In *IFIP TC1 WG1.7 Workshop on Formal Aspects in Security and Trust*, pages 187–201, 2005.
- [7] B. Crispo and G. Ruffo. Reasoning about accountability within delegation. In *ICICS*, pages 251–260, 2001.
- [8] Y. Chen *et al.* Oblivious hashing: A stealthy software integrity verification primitive. In Fabien Petitcolas, editor, *Information Hiding*, volume 2578 of *Lecture Notes in Computer Science*, pages 400–414. 2003.
- [9] R. Hasan, R. Sion, and M. Winslett. The case of the fake picasso: preventing history forgery with secure provenance. In *7th conference on File and storage technologies*, pages 1–14, Berkeley, CA, USA, 2009. USENIX Association.
- [10] J. Hightower and G. Borriello. Location systems for ubiquitous computing. *Computer*, 34(8):57–66, August 2001.
- [11] P. Jaeger, J. Lin, and J. M. Grimes. Cloud computing and information policy: Computing in a policy cloud? *Journal of Information Technology and politics*, 5(3), 2009.
- [12] R. Jagadeesan, A. Jeffrey, C. Pitcher, and J. Riely. Towards a theory of accountability and audit. In *ESORICS*, pages 152–167, 2009.
- [13] R. Kailar. Accountability in electronic commerce protocols. *IEEE Trans. Software Eng.*, 22(5):313–328, 1996.
- [14] W. Lee, A. Squicciarini, and E. Bertino. The design and evaluation of accountable grid computing system. In *29th IEEE International Conference on Distributed Computing Systems (ICDCS 2009)*, pages 145–154, 2009.
- [15] T. Mather, S. Kumaraswamy, and S. Latif. *Cloud Security and Privacy: An Enterprise Perspective on Risks and Compliance (Theory in Practice)*. O’Reilly, first edition, 2009.
- [16] M. Casassa Mont, S. Pearson, and P. Bramhall. Towards accountable management of identity and privacy: Sticky policies and enforceable tracing services. In *DEXA Workshops*, pages 377–382, 2003.
- [17] Scott Oaks. *Java security*. O’Really, 2001.
- [18] OASIS. Security assertion markup language (SAML) 2.0.
- [19] S. Pearson and A. Charlesworth. Accountability as a way forward for privacy protection in the cloud. *Hewlett-Packard Development Company (HPL-2009-178)*, 2009.
- [20] S. Pearson, Y. Shen, and M. Mowbray. A privacy manager for cloud computing. In *CloudCom*, pages 90–106, 2009.
- [21] NTP: The Network Time Protocol. <http://www.ntp.org/>.
- [22] S. Roman. *Coding and information theory*. Springer-Verlag New York, Inc., 1992.
- [23] T. J. E. Schwarz and E. L. Miller. Store, forget, and check: Using algebraic signatures to check remotely administered storage. In *IEEE International Conference on Distributed Systems*, page 12, 2006.
- [24] A. Squicciarini, S. Sundareswaran, and D. Lin. Preventing information leakage from indexing in the cloud. In *IEEE International Conference on Cloud Computing*, 2010.
- [25] Eucalyptus Systems. <http://www.eucalyptus.com/>.
- [26] Emulab Network Emulation Testbed. www.emulab.net.
- [27] Q. Wang, C. Wang, J. Li, K. Ren, and W. Lou. Enabling public verifiability and data dynamics for storage security in cloud computing. In *ESORICS*, pages 355–370, 2009.
- [28] D. Weitzner, H. Abelson, T. Berners-Lee, J. Feigenbaum, J. Hendler, and G. Sussman. Information accountability. *Commun. ACM*, 51(6):82–87, 2008.
- [29] Stephen B. Wicker and Vijay K. Bhargava, editors. *Reed-Solomon Codes and Their Applications*. John Wiley & Sons, Inc., USA, 1999.