Ensuring Distributed Accountability for Data Sharing in the Cloud

Smitha Sundareswaran, Anna C. Squicciarini, *Member*, *IEEE*, and Dan Lin

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 are 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 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.

Index Terms—Cloud computing, accountability, data sharing.

1 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 theses 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.

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

- *S. Sundareswaran and A.C. Squicciarini are with the College of Information Sciences and Technology, The Pennsylvania State University, University Park, PA 16802. E-mail: {sus263, asquicciarini}@ist.psu.edu.*
- *D. Lin is with the Department of Computer Science, Missouri University of Science & Technology, Rolla, MO 65409. E-mail: lindan@mst.edu.*

Manuscript received 30 June 2011; revised 31 Jan. 2012; accepted 17 Feb. 2012; published online 2 Mar. 2012.
For information on obtaining reprints of this article, please send e-mail to: tdsc@computer.org, and reference IEEECS Log Number TDSC-2011-06-0169.
Digital Object Identifier no. 10.1109/TDSC.2012.26.

Published by the IEEE Computer Society

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 (as is proven by the popularity of Flickr [14]) 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 addition, our approach can handle personal identifiable information provided they are stored as image files (they contain an image of any textual content, for example, the SSN stored as a .jpg file).

We tested our CIA framework in a cloud testbed, the Emulab testbed [42], with Eucalyptus as middleware [41]. Our experiments demonstrate the efficiency, scalability and granularity of our approach. In addition, we also provide a detailed security analysis and discuss the reliability and strength of our architecture in the face of various nontrivial attacks, launched by malicious users or due to compromised Java Running Environment (JRE).

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 these are delivered to the receiver.
- We conduct 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.

This paper is an extension of our previous conference paper [40]. We have made the following new contributions. First, we integrated integrity checks and oblivious hashing (OH) technique to our system in order to strengthen the dependability of our system in case of compromised JRE. We also updated the log records structure to provide additional guarantees of integrity and authenticity. Second, we extended the security analysis to cover more possible attack scenarios. Third, we report the results of new experiments and provide a thorough evaluation of the system performance. Fourth, we have added a detailed discussion on related works to prepare readers with a better understanding of background knowledge. Finally, we have improved the presentation by adding more examples and illustration graphs.

The rest of the paper is organized as follows: Section 2 discusses related work. Section 3 lays out our problem statement. Section 4 presents our proposed Cloud Information Accountability framework, and Sections 5 and 6 describe the detailed algorithms for automated logging mechanism and auditing approaches, respectively. Section 7 presents a security analysis of our framework, followed by an experimental study in Section 8. Finally, Section 9 concludes the paper and outlines future research directions.

2 Related Work

In this section, we first review related works addressing the privacy and security issues in the cloud. Then, we briefly discuss works which adopt similar techniques as our approach but serve for different purposes.

2.1 Cloud Privacy and Security

Cloud computing has raised a range of important *privacy* and *security* issues [19], [25], [30]. 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 [30] and then develop a privacy manager [31]. Their basic idea is that the user's private data are sent to the cloud in an encrypted form, and the processing is done on the encrypted data. The output of the processing is deobfuscated 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 are being disclosed. In [7], 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 [10], [21]. A representative work in this area is given by [9]. The authors propose the usage of policies attached to the data and present a logic for accountability data in distributed settings. Similarly, Jagadeesan et al. recently proposed a logic for designing accountability-based distributed systems [20]. In [10], Crispo and Ruffo proposed an interesting approach related to accountability in case of delegation. Delegation is complementary to our work, in that we do not aim at controlling the information workflow in the clouds. In a summary, all 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 [22]. 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 [22] is related to ours, but it is mainly focused on resource consumption and on tracking of subjobs processed at multiple computing nodes, rather than access control.

2.2 Other Related Techniques

With respect to Java-based techniques for security, our methods are related to self-defending objects (SDO) [17]. Self-defending objects are an extension of the object-oriented programming paradigm, where software objects that offer sensitive functions or hold sensitive data are responsible for protecting those functions/data. Similarly, we also extend the concepts of object-oriented programming. The key difference in our implementations is that the authors still rely on a centralized database to maintain the access records, while the items being protected are held as separate files. In previous work, we provided a Java-based approach to prevent privacy leakage from indexing [39], which could be integrated with the CIA framework proposed in this work since they build on related architectures.

In terms of authentication techniques, Appel and Felten [13] proposed the Proof-Carrying authentication (PCA) framework. The PCA includes a high order logic language that allows quantification over predicates, and focuses on access control for web services. While related to ours to the extent that it helps maintaining safe, high-performance, mobile code, the PCA's goal is highly different from our research, as it focuses on validating code, rather than monitoring content. Another work is by Mont et al. who proposed an approach for strongly coupling content with access control, using Identity-Based Encryption (IBE) [26]. 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, our work may look similar to works on secure data provenance [5], [6], [15], 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 are 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. Since it is in a distributed environment, we also log where the data go. However, this is not for verifying data integrity, but rather for auditing whether data receivers use the data following specified policies.

Along the lines of extended content protection, usage control [33] is being investigated as an extension of current access control mechanisms. Current efforts on usage control are primarily focused on conceptual analysis of usage control requirements and on languages to express constraints at various level of granularity [32], [34]. While some notable results have been achieved in this respect [12], [34], thus far, there is no concrete contribution addressing the problem of usage constraints enforcement, especially in distributed settings [32]. The few existing solutions are partial [12], [28], [29], restricted to a single domain, and often specialized [3], [24], [46]. Finally, general outsourcing techniques have been investigated over the past few years [2], [38]. Although only [43] 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.

3 Problem Statement

We begin this section by considering an illustrative example which serves as the basis of our problem statement and will be used throughout the paper to demonstrate the main features of our system.

Example 1. Alice, a professional photographer, plans to sell her photographs by using the SkyHigh Cloud Services. For her business in the cloud, she has the following requirements:

- Her photographs are downloaded only by users who have paid for her services.
- Potential buyers are allowed to view her pictures first before they make the payment to obtain the download right.
- Due to the nature of some of her works, only users from certain countries can view or download some sets of photographs.
- For some of her works, users are allowed to only view them for a limited time, so that the users cannot reproduce her work easily.
- In case any dispute arises with a client, she wants to have all the access information of that client.
- She wants to ensure that the cloud service providers of SkyHigh do not share her data with other service providers, so that the accountability provided for individual users can also be expected from the cloud service providers.

With the above scenario in mind, we identify 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 are 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 bounded 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 have 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.

4 Cloud Information Accountability

In this section, 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*.

4.1 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. 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.

Being the trusted component, the log harmonizer generates the master key. It holds on to the decryption key for the IBE key pair, as it is responsible for decrypting the logs. Alternatively, the decryption can be carried out on the client end if the path between the log harmonizer and the client is not trusted. In this case, the harmonizer sends the key to the client in a secure key exchange.

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.

4.2 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 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 [8], wherein a trusted identity provider issues certificates verifying the user's identity based on his username.

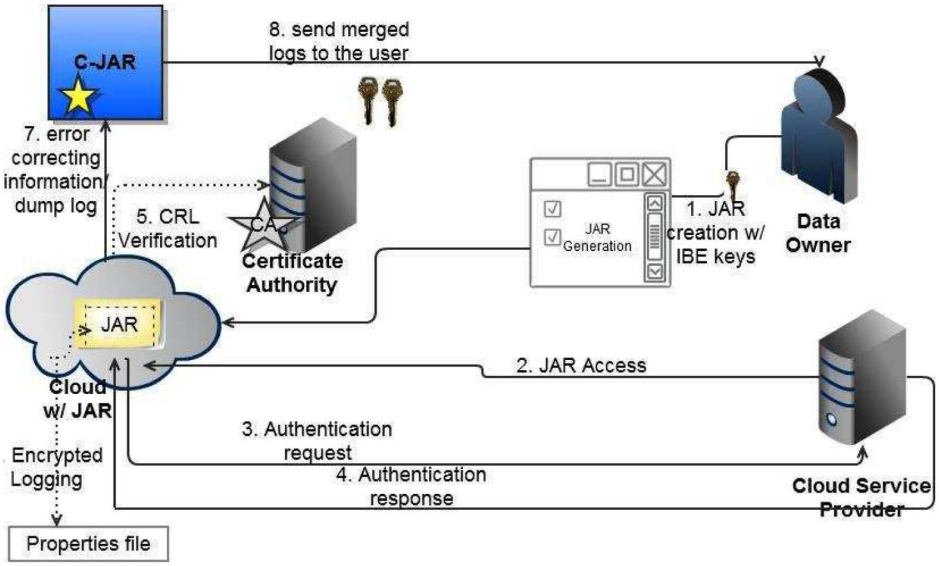


Fig. 1. Overview of the cloud information accountability framework.

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. 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 (detailed discussion is in Section 7) 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 (step 7 in Fig. 1). To ensure trustworthiness of the logs, each record is signed by the entity accessing the content. Further, individual records are hashed together to create a chain structure, able to quickly detect possible errors or missing records. The encrypted log files can later be decrypted and their integrity verified. They can be accessed by the data owner or other authorized stakeholders at any time for auditing purposes with the aid of the log harmonizer (step 8 in Fig. 1).

As discussed in Section 7, 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.

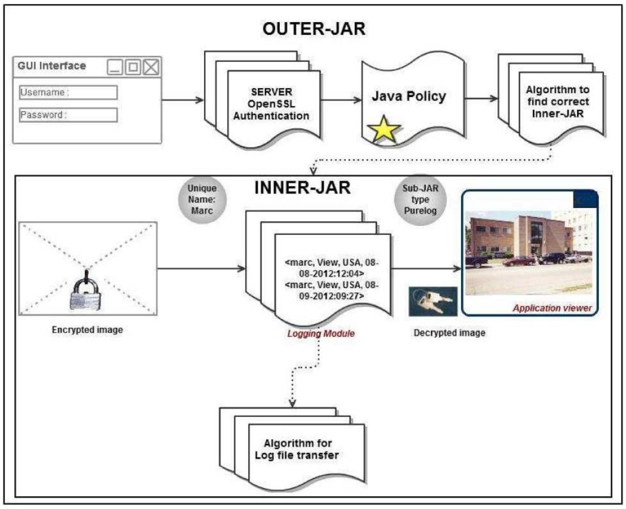
Example 2. Considering Example 1, Alice can enclose her photographs and access control policies in a JAR file and send the JAR file to the cloud service provider. With the aid of control associated logging (called AccessLog in Section 5.2), Alice will be able to enforce the first four requirements and record the actual data access. On a regular basis, the push-mode auditing mechanism will inform Alice about the activity on each of her photographs as this allows her to keep track of her clients' demographics and the usage of her data by the cloud service provider. In the event of some dispute with her clients, Alice can rely on the pull-mode auditing mechanism to obtain log records.

5 Automated Logging Mechanism

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

5.1 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.



```
<marc, View, USA, 08-08-2012:12:04>
<marc, View, USA, 08-09-2012:09:27>
```

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. 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 environment. The 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, using Java Authentication and Authorization Services. Moreover, the outer JAR is also in charge of selecting the correct inner JAR according to the identity of the entity who requests the data.

Example 3. Consider Example 1. Suppose that Alice's photographs are classified into three categories according to the locations where the photos were taken. The three groups of photos are stored in three inner JAR J1 , J2, and J3 , respectively, associated with different access control policies. If some entities are allowed to access only one group of the photos, say J1 , the outer JAR will just render the corresponding inner JAR to the entity based on the policy evaluation result.

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 two 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 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 proactively (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.

To carry out these functions, the inner JAR contains a class file for writing the log records, another class file which corresponds with the log harmonizer, the encrypted data, a third class file for displaying or downloading the data (based on whether we have a PureLog, or an AccessLog), and the public key of the IBE key pair that is necessary for encrypting the log records. No secret keys are ever stored in the system. The outer JAR may contain one or more inner JARs, in addition to a class file for authenticating the servers or the users, another class file finding the correct inner JAR, a third class file which checks the JVM's validity using oblivious hashing. Further, a class file is used for managing the GUI for user authentication and the Java Policy.

5.2 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 $\mathrm{LR} = \langle r_1, \dots, r_k \rangle$. Each record r_i is encrypted individually and appended to the log file. In particular, a log record takes the following form:

$$r_i = \langle ID, Act, T, Loc, h((ID, Act, T, Loc)|r_i - 1|\dots|r_1), sig\rangle.$$

Here, r_i indicates that an entity identified by I D has performed an action Act on the user's data at time T at location Loc. The component $h((ID, Act, T, Loc)|r_i - 1|\dots|r_1)$ corresponds to the checksum of the records preceding the newly inserted one, concatenated with the main content of the record itself (we use I 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 ObjI D field is added to each record. An example of log record for a single file is shown below.

Example 4. Suppose that a cloud service provider with ID Kronos, located in USA, read the image in a JAR file (but did not download it) at 4:52 pm on May 20, 2011. The corresponding log record is
⟨*Kronos, View, 2011-05-29 16:52:30,USA, 45rftT024g, r94gm30130ff*⟩.

The location is converted from the IP address for improved readability.

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 (NTP) [35] to avoid suppression of the correct time by a malicious entity. The location of the cloud service provider 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 location can also be used [16]. Similarly, if a trusted time stamp management infrastructure can be set up or leveraged, it can be used to record the time stamp in the accountability log [1]. 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 are 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 are 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. 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 are made available only to entities located at locations specified by the data owner.

5.3 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.

5.3.1 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. Since user's data are 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 purposes, 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.

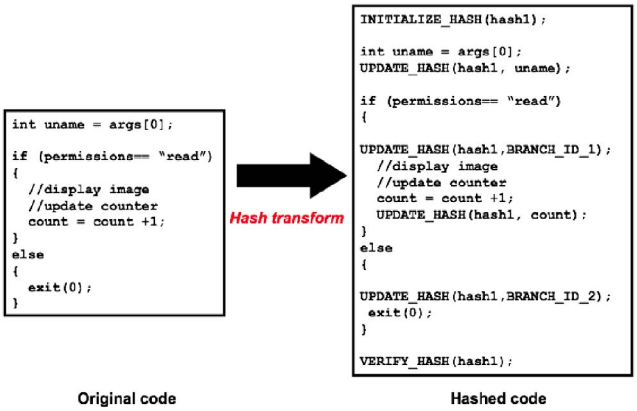
In case of corruption of JAR files, the harmonizer will recover the logs with the aid of Reed-Solomon error correction code [45]. Specifically, each individual logging JAR, when created, contains a Reed-Solomon-based encoder. For every n symbols in the log file, n redundancy symbols are added to the 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 [36], 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.

5.3.2 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.

These tasks are carried out by the log harmonizer and the logger components in tandem with each other. The log



```
int uname = args[0];

if (permissions== "read")
{
  //display image
  //update counter
  count = count +1;
}
else
{
  exit(0);
}
```

```
INITIALIZE_HASH(hash1);

int uname = args[0];
UPDATE_HASH(hash1, uname);

if (permissions== "read")
{

UPDATE_HASH(hash1,BRANCH_ID_1);
  //display image
  //update counter
  count = count +1;
  UPDATE_HASH(hash1, count);
}
else
{

UPDATE_HASH(hash1,BRANCH_ID_2);
 exit(0);
}

VERIFY_HASH(hash1);
```

Fig. 3. Oblivious hashing applied to the logger.

harmonizer is solely responsible for checking the integrity of the JRE on the systems on which the logger components exist before the execution of the logger components is started. Trusting this task to the log harmonizer allows us to remotely validate the system on which our infrastructure is working. The repair step is itself a two-step process where the harmonizer first recognizes the Operating System being used by the cloud machine and then tries to reinstall the JRE. The OS is identified using nmap commands. The JRE is reinstalled using commands such as `sudo apt install` for Linux-based systems or `$ <jre>.exe [lang=] [s] [IEXPLORER=1] [MOZILLA=1] [INSTALLDIR=:] [STATIC=1]` for Windows-based systems.

The logger and the log harmonizer work in tandem to carry out the integrity checks during runtime. These integrity checks are carried out using oblivious hashing [11]. OH works by adding additional hash codes into the programs being executed. The hash function is initialized at the beginning of the program, the hash value of the result variable is cleared and the hash value is updated every time there is a variable assignment, branching, or looping. An example of how the hashing transforms the code is shown in Fig. 3.

As shown, the hash code captures the computation results of each instruction and computes the oblivious-hash value as the computation proceeds. These hash codes are added to the logger components when they are created. They are present in both the inner and outer JARs. The log harmonizer stores the values for the hash computations. The values computed during execution are sent to it by the logger components. The log harmonizer proceeds to match these values against each other to verify if the JRE has been tampered with. If the JRE is tampered, the execution values will not match. Adding OH to the logger components also adds an additional layer of security to them in that any tampering of the logger components will also result in the OH values being corrupted.

6 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.

6.1 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.

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.

Concerning the latter function, we notice that the auditor, upon receiving the log file, will verify its cryptographic guarantees, by checking the records' integrity and authenticity. By construction of the records, the auditor, will be able to quickly detect forgery of entries, using the checksum added to each and every record.

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.

6.2 Algorithms

Pushing or pulling strategies have interesting tradeoffs. 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 are not pushed out frequently enough, the log file may become very large, which may increase cost of operations like copying data (see Section 8). 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 7, supporting both pushing and pulling modes helps protecting from some nontrivial attacks.

```
Require: $size$: maximum size of the log file specified by the data owner, $time$: maximum time allowed to elapse before the log file is dumped, $tbeg$: timestamp at which the last dump occurred, $log$: current log file, $pull$: indicates whether a command from the data owner is received.
1: Let TS(NTP) be the network time protocol timestamp
2: $pull = 0$
3: $rec := \langle UID, OID, AccessType, Result, Time, Loc \rangle$
4: $curtime := TS(NTP)$
5: $lsize := sizeof(log)$ //current size of the log
6: if $((cutime\ \text{-}tbeg) < time) \&\& (lsize < size) \&\& (pull == 0)$ then
7:    $log := log + ENCRYPT(rec)$ // ENCRYPT is the encryption function used to encrypt the record
8:    $PING$ to CJAR //send a PING to the harmonizer to check if it is alive
9:    if $PING$-CJAR then
10:      PUSH $RS(rec)$ // write the error correcting bits
11:   else
12:      EXIT(1) // error if no PING is received
13:   end if
14: end if
15: if $((cutime - tbeg) > time) || (lsize >= size) || (pull \neq 0)$ then
16:   // Check if PING is received
17:   if $PING$-CJAR then
18:      PUSH $log$ //write the log file to the harmonizer
19:      $RS(log) := NULL$ // reset the error correction records
20:      $tbeg := TS(NTP)$ // reset the $tbeg$ variable
21:      $pull := 0$
22:   else
23:      EXIT(1) // error if no PING is received
24:   end if
25: end if
```

Fig. 4. Push and pull PureLog mode.

The log retrieval algorithm for the Push and Pull modes is outlined in Fig. 4.

The algorithm presents logging and synchronization steps with the harmonizer in case of PureLog. First, the algorithm checks whether the size of the JAR has exceeded a stipulated size or the normal time between two consecutive dumps has elapsed. The size and time threshold for a dump are specified by the data owner at the time of creation of the JAR. The algorithm also checks whether the data owner has requested a dump of the log files. If none of these events has occurred, it proceeds to encrypt the record and write the error-correction information to the harmonizer.

The communication with the harmonizer begins with a simple handshake. If no response is received, the log file records an error. The data owner is then alerted through e-mails, if the JAR is configured to send error notifications. Once the handshake is completed, the communication with the harmonizer proceeds, using a TCP/IP protocol. If any of the aforementioned events (i.e., there is request of the log file, or the size or time exceeds the threshold) has occurred, the JAR simply dumps the log files and resets all the variables, to make space for new records.

In case of AccessLog, the above algorithm is modified by adding an additional check after step 6. Precisely, the AccessLog checks whether the CSP accessing the log satisfies all the conditions specified in the policies pertaining to it. If the conditions are satisfied, access is granted; otherwise, access is denied. Irrespective of the access control outcome, the attempted access to the data in the JAR file will be logged.

Our auditing mechanism has two main advantages. First, it guarantees a high level of availability of the logs. Second, the use of the harmonizer minimizes the amount of workload for human users in going through long log files sent by different copies of JAR files. For a better understanding of the auditing mechanism, we present the following example.

Example 5. With reference to Example 1, Alice can specify that she wants to receive the log files once every week, as it will allow her to monitor the accesses to her photographs. Under this setting, once every week the JAR files will communicate with the harmonizer by pinging it. Once the ping is successful, the file transfer begins. On receiving the files, the harmonizer merges the logs and sends them to Alice. Besides receiving log information once every week, Alice can also request the log file anytime when needed. In this case, she just need to send her pull request to the harmonizer which will then ping all the other JARs with the "pull" variable to 1. Once the message from the harmonizer is received, the JARs start transferring the log files back to the harmonizer.

7 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.

7.1 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 unaccessible.

7.2 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.

To compromise the confidentiality of the log files, 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 [4]. 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.[1] 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.

Also, the attacker cannot modify the log files extracted from a disassembled JAR. Would the attacker erase or tamper a record, the integrity checks added to each record of the log will not match at the time of verification (see Section 5.2 for the record structure and hash chain), revealing the error. Similarly, attackers will not be able to write fake records to log files without going undetected, since they will need to sign with a valid key and the chain of hashes will not match. The Reed-Solomon encoding used to create the redundancy for the log files, the log harmonizer can easily detect a corrupted record or log file.

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 by the sealing techniques offered by Java. Sealing ensures that all packages within the JAR file come from the same source code [27]. 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. More importantly, this attack is stopped as the JARs check the classloader each time before granting any access right. If the classloader is found to be a custom classloader, the JARs will throw an exception and halt. Further, JAR files are signed for integrity at the time of creation, to avoid that an attacker writes to the JAR. Even if an attacker 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.

7.3 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 OpenSSL and cryptographic checks have been added.

7.4 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.

8 Performance Study

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

8.1 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:

[1] Notice that we do not consider the attack on the log harmonizer component, since it is stored separately in either a secure proxy or at the user end and the attacker typically cannot access it. As a result, we assume that the attacker cannot extract the decryption keys from the log harmonizer.

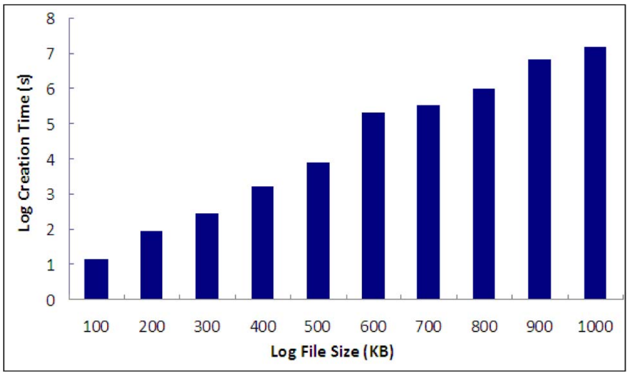


Fig. 5. Time to create log files of different sizes.

one head node which is the certificate authority, and several computing nodes. Each of the servers is installed with Eucalyptus [41]. 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. Each server has a 64-bit Intel Quad Core Xeon E5530 processor, 4 GB RAM, and a 500 GB Hard Drive. Each of the servers is equipped to run the OpenJDK runtime environment with IcedTea6 1.8.2.

8.2 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. Further, JAR act as a compressor of the files that it handles. In particular, as introduced in Section 3, multiple files can be handled by the same logger component. To this extent, we investigate whether a single logger component, used to handle more than one file, results in storage overhead.

8.2.1 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. Specifically, the time to create a 100 Kb file is about 114.5 ms while the time to create a 1 MB file averages at 731 ms. 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.

8.2.2 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.

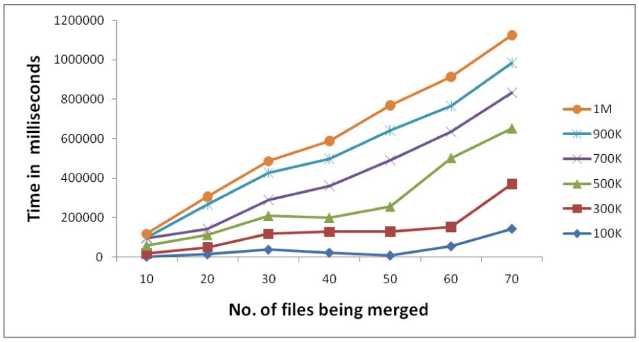


Fig. 6. Time to merge log files.

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 (i.e., submitting his username to the JAR), it averages at 1.2 minutes.

8.2.3 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 300 ms (per record) and is seemingly unrelated from the log size.

8.2.4 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. The time was averaged over 10 repetitions. We tested the time to merge up to 70 log files of 100 KB, 300 KB, 500 KB, 700 KB, 900 KB, and 1 MB each. The results are shown in Fig. 6. 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 100 KB log files at 59 ms, while the time to merge 70 1 MB files was 2.35 minutes.

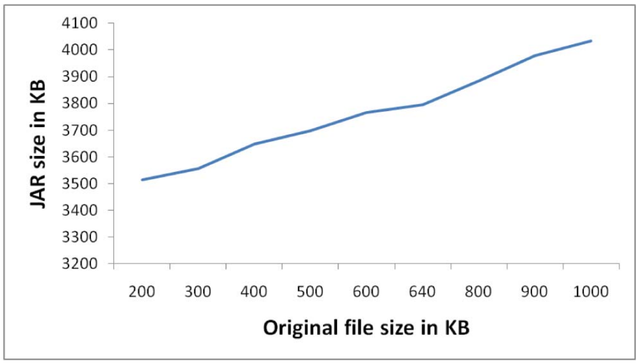


Fig. 7. Size of the logger component.

8.2.5 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 3,500 to 4,035 KB when the size of content items changes from 200 KB to 1 MB. 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 5 KB), so as to focus on the overhead added by having multiple content files in a single JAR. Results are in Fig. 7.

8.2.6 Overhead Added by JVM Integrity Checking

We investigate the overhead added by both the JRE installation/repair process, and by the time taken for computation of hash codes.

The time taken for JRE installation/repair averages around 6,500 ms. This time was measured by taking the system time stamp at the beginning and end of the installation/repair.

To calculate the time overhead added by the hash codes, we simply measure the time taken for each hash calculation. This time is found to average around 9 ms. The number of hash commands varies based on the size of the code in the code does not change with the content, the number of hash commands remain constant.

9 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.

In the future, we plan to refine our approach to verify the integrity of the JRE and the authentication of JARs [23]. For example, we will investigate whether it is possible to leverage the notion of a secure JVM [18] being developed by IBM. This research is aimed at providing software tamper resistance to Java applications. In the long term, we plan to design a comprehensive and more generic object-oriented approach to facilitate autonomous protection of traveling content. We would like to support a variety of security policies, like indexing policies for text files, usage control for executables, and generic accountability and provenance controls.

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.

[21] R. Kailar, “Accountability in Electronic Commerce Protocols,” *IEEE Trans. Software Eng.*, vol. 22, no. 5, pp. 313-328, May 1996.

[22] W. Lee, A. Cinzia Squicciarini, and E. Bertino, “The Design and Evaluation of Accountable Grid Computing System,” *Proc. 29th IEEE Int’l Conf. Distributed Computing Systems (ICDCS ’09)*, pp. 145-154, 2009.

[23] J.H. Lin, R.L. Geiger, R.R. Smith, A.W. Chan, and S. Wanchoo, *Method for Authenticating a Java Archive (jar) for Portable Devices,* US Patent 6,766,353, July 2004.

[24] F. Martinelli and P. Mori, "On Usage Control for Grid Systems," *Future Generation Computer Systems,* vol. 26, no. 7, pp. 1032-1042, 2010.

[25] T. Mather, S. Kumaraswamy, and S. Latif, *Cloud Security and Privacy: An Enterprise Perspective on Risks and Compliance (Theory in Practice),* first ed. O' Reilly, 2009.

[26] M.C. Mont, S. Pearson, and P. Bramhall, "Towards Accountable Management of Identity and Privacy: Sticky Policies and Enforceable Tracing Services," *Proc. Int'l Workshop Database and Expert Systems Applications (DEXA),* pp. 377-382, 2003.

[27] S. Oaks, *Java Security.* O'Really, 2001.

[28] J. Park and R. Sandhu, "Towards Usage Control Models: Beyond Traditional Access Control," *SACMAT '02: Proc. Seventh ACM Symp. Access Control Models and Technologies,* pp. 57-64, 2002.

[29] J. Park and R. Sandhu, "The Uconabc Usage Control Model," *ACM Trans. Information and System Security,* vol. 7, no. 1, pp. 128-174, 2004.

[30] S. Pearson and A. Charlesworth, "Accountability as a Way Forward for Privacy Protection in the Cloud," *Proc. First Int'l Conf. Cloud Computing,* 2009.

[31] S. Pearson, Y. Shen, and M. Mowbray, "A Privacy Manager for Cloud Computing," *Proc. Int'l Conf. Cloud Computing (CloudCom),* pp. 90-106, 2009.

[32] A. Pretschner, M. Hilty, and D. Basin, "Distributed Usage Control," *Comm. ACM,* vol. 49, no. 9, pp. 39-44, Sept. 2006.

[33] A. Pretschner, M. Hilty, F. Schuötz, C. Schaefer, and T. Walter, "Usage Control Enforcement: Present and Future," *IEEE Security & Privacy,* vol. 6, no. 4, pp. 44-53, July/Aug. 2008.

[34] A. Pretschner, F. Schuötz, C. Schaefer, and T. Walter, "Policy Evolution in Distributed Usage Control," *Electronic Notes Theoretical Computer Science,* vol. 244, pp. 109-123, 2009.

[35] NTP: The Network Time Protocol, http://www.ntp.org/, 2012.

[36] S. Roman, *Coding and Information Theory.* Springer-Verlag, 1992.

[37] B. Schneier, *Applied Cryptography: Protocols, Algorithms, and Source Code in C.* John Wiley & Sons, 1993.

[38] T.J.E. Schwarz and E.L. Miller, "Store, Forget, and Check: Using Algebraic Signatures to Check Remotely Administered Storage," *Proc. IEEE Int'l Conf. Distributed Systems,* p. 12, 2006.

[39] A. Squicciarini, S. Sundareswaran, and D. Lin, "Preventing Information Leakage from Indexing in the Cloud," *Proc. IEEE Int'l Conf. Cloud Computing,* 2010.

[40] S. Sundareswaran, A. Squicciarini, D. Lin, and S. Huang, "Promoting Distributed Accountability in the Cloud," *Proc. IEEE Int'l Conf. Cloud Computing,* 2011.

[41] Eucalyptus Systems, http://www.eucalyptus.com/, 2012.

[42] Emulab Network Emulation Testbed, www.emulab.net, 2012.

[43] Q. Wang, C. Wang, J. Li, K. Ren, and W. Lou, "Enabling Public Verifiability and Data Dynamics for Storage Security in Cloud Computing," *Proc. European Conf. Research in Computer Security (ESORICS),* pp. 355-370, 2009.

[44] D.J. Weitzner, H. Abelson, T. Berners-Lee, J. Feigen-baum, J. Hendler, and G.J. Sussman, "Information Accountability," *Comm. ACM,* vol. 51, no. 6, pp. 82-87, 2008.

[45] *Reed-Solomon Codes and Their Applications,* S.B. Wicker and V.K. Bhargava, ed. John Wiley & Sons, 1999.

[46] M. Xu, X. Jiang, R. Sandhu, and X. Zhang, "Towards a VMM-Based Usage Control Framework for OS Kernel Integrity Protection," *SACMAT '07: Proc. 12th ACM Symp. Access Control Models and Technologies,* pp. 71-80, 2007.

Smitha Sundareswaran received the bachelor's degree in electronics and communications engineering in 2005 from Jawaharlal Nehru Technological University, Hyderabad, India. She is currently working toward the PhD degree in the College of Information Sciences and Technology at the Pennsylvania State University. Her research interests include policy formulation and management for Distributed computing architectures.

Anna C. Squicciarini received the PhD degree in computer science from the University of Milan, Italy, in 2006. She is an assistant professor at the College of Information Science and Technology at the Pennsylvania State University. During the years of 2006-2007, she was a postdoctoral research associate at Purdue University. Her main interests include access control for distributed systems, privacy, security for Web 2.0 technologies and grid computing. She is the author or coauthor of more than 50 articles published in refereed journals, and in proceedings of international conferences and symposia. She is a member of the IEEE.

Dan Lin received the PhD degree from National University of Singapore in 2007. She is an assistant professor at Missouri University of Science and Technology. She was a postdoctoral research associate from 2007 to 2008. Her research interests cover many areas in the fields of database systems and information security.