Secure Logging As a Service—Delegating Log Management to the Cloud

Indrajit Ray, Kirill Belyaev, Mikhail Strizhov, Dieudonne Mulamba, and Mariappan Rajaram

***Abstract*—Securely maintaining log records over extended periods of time is very important to the proper functioning of any organization. Integrity of the log files and that of the logging process need to be ensured at all times. In addition, as log files often contain sensitive information, confidentiality and privacy of log records are equally important. However, deploying a secure logging infrastructure involves substantial capital expenses that many organizations may find overwhelming. Delegating log management to the cloud appears to be a viable cost saving measure. In this paper, we identify the challenges for a secure cloud-based log management service and propose a framework for doing the same.**

***Index Terms*—Cloud computing, logging, privacy, security.**

I. Introduction

A LOG IS a record of events occurring within an organization's system or network [1]. Logging is important because log data can be used to troubleshoot problems, fine tune system performance, identify policy violations, investigate malicious activities, and even record user activities. Log records play a significant role in digital forensic analysis of systems. Regulations such as HIPAA [2], Payment Card Industry Data Security Standard [3], or Sarbanes-Oxley [4] often require forensically sound preservation of information. To comply with these regulations, evidence produced in a court of law, including log records, must be unbiased, nontampered with, and complete before they can be used.

Since log files contain record of most system events including user activities, they become an important target for malicious attackers. An attacker, breaking into a system, typically would try not to leave traces of his or her activities behind. Consequently, the first thing an attacker often does is to damage log files or interrupt the logging services. Furthermore, the sensitive information contained in log files often directly contributes to confidentiality breaches. An example of this is when logs contain database transaction data. Frequently, log information can be helpful to an attacker in gaining unauthorized access to system. One example of this is the case when a user mistakenly enters her password in the username field while logging into a system. Logging programs will store the password as the user-id to record the information that a user has failed to log in. Last, but not least, information in log file can also be used to cause privacy breaches for users in the system since the log file contains record of all events in the system.

In light of the above observations, it is very important that logging be provided in a secure manner and that the log records are adequately protected for a predetermined amount of time (maybe even indefinitely). Traditional logging protocols that are based on syslog [5] have not been designed with such security features in mind. Security extensions that have been proposed, such as reliable delivery of syslog [6], forward integrity for audit logs [7], syslog-ng [8], and syslog-sign [9], often provide either partial protection, or do not protect the log records from end point attacks. In addition, log management requires substantial storage and processing capabilities. The log service must be able to store data in an organized manner and provide a fast and useful retrieval facility. Last, but not least, log records may often need to be made available to outside auditors who are not related to the organization. Deploying a secure logging infrastructure to meet all these challenges entails significant infrastructural support and capital expenses that many organizations may find overwhelming.

The emerging paradigm of cloud computing promises a low cost opportunity for organizations to store and manage log records in a proper manner. Organizations can outsource the long-term storage requirements of log files to the cloud. The challenges of storing and maintaining the log records become a concern of the cloud provider. Since the cloud provider is providing a single service to many organizations that it will benefit from economics of scale. Pushing log records to the cloud, however, introduces a new challenge in storing and maintaining log records. The cloud provider can be honest but curious. This means that it can try not only to get confidential information directly from log records, but also link log record related activities to their sources. No existing protocol addresses all the challenges that arise when log storage and maintenance is pushed to the cloud.

In this paper, we propose a comprehensive solution for storing and maintaining log records in a server operating in a cloud-based environment. We address security and integrity issues not only just during the log generation phase, but also during other stages in the log management process, including

Manuscript received October 8, 2011; revised March 21, 2012; accepted July 26, 2012. Date of publication January 9, 2013; date of current version April 17, 2013.

I. Ray, K. Belyaev, M. Strizhov, and D. Mulamda are with the Department of Computer Science, Colorado State University, Fort Collins, CO 80523 USA (e-mail: indrajit@cs.colostate.edu; kirill@cs.colostate.edu; strizhov@cs.colostate.edu; mulamba@cs.colostate.edu).

M. Rajaram is with Compuware Corporation, Detroit, MI 48226 USA (e-mail: r.mariappan@gmail.com).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/JSYST.2012.2221958

log collection, transmission, storage, and retrieval. The major contributions of this paper are as follows. We propose an architecture for the various components of the system and develop cryptographic protocols to address integrity and confidentiality issues with storing, maintaining, and querying log records at the honest but curious cloud provider and in transit. We also develop protocols so that log records can be transmitted and retrieved in an anonymous manner over an existing anonymizing infrastructure such as Tor [16]. This successfully prevents the cloud provider or any other observer from correlating requests for log data with the requester or generator. Finally, we develop a proof-of-concept prototype to demonstrate the feasibility of our approach and discuss some early experiences with it. To the best of our knowledge, ours is the first work to provide a complete solution to the cloud-based secure log management problem.

The remainder of this paper is organized as follows. Section II identifies the security properties that must be ensured in a cloud-based log management system. Section III discusses some of the existing secure logging protocols in light of these properties. In Section IV, we present the architecture for our cloud-based secure logging facility and describe the services provided by each component. The threat model is discussed in V. Section VI presents the main cryptographic techniques used to protect the log records in transit and at the cloud provider. We reason why our protocol guarantees the desirable security properties. Section VII describes the protocols used to anonymously store and retrieve the log files from the cloud. We present a brief discussion of the implementation in Section VIII and some performance study that we have undertaken. We conclude this paper with a discussion of our future work in Section IX.

II. Desirable Properties of Secure Logging As A Service

We begin by summarizing the desirable properties that we seek from a secure log management service based on the cloud computing paradigm. We will subsequently analyze our framework against these properties.

1) *Correctness*: Log data is useful only if it reflects true history of the system at the time of log generation. The stored log data should be correct, that is, it should be exactly the same as the one that was generated.
2) *Tamper Resistance*: A secure log must be tamper resistant in such a way that no one other than the creator of the log can introduce valid entries. In addition, once those entries are created they cannot be manipulated without detection. No one can prevent an attacker who has compromised the logging system from altering what that system will put in future log entries. One cannot also prevent an attacker from deleting any log entries that have not already been pushed to another system. The goal of a secure audit log in this case is to make sure that the attacker cannot alter existing log entries (i.e., the precompromise log entries) and that any attempts to delete or alter existing entries (truncation attack [10]) will be detected.
3) *Verifiability*: It must be possible to check that all entries in the log are present and have not been altered. Each entry must contain enough information to verify its authenticity independent of others. If some entries are altered or deleted, the ability to individually verify the remaining entries (or blocks of entries) makes it possible to recover some useful information from the damaged log. Moreover, the individual entries must be linked together in a way that makes it possible to determine whether any entries are missing (forward integrity, [7].
4) *Confidentiality*: Log records should not be casually browseable or searchable to gather sensitive information. Legitimate search access to users such as auditors or system administrators should be allowed. In addition, since no one can prevent an attacker who has compromised the logging system from accessing sensitive information that the system will put in future log entries, the goal is to protect the precompromised log records from confidentiality breaches.
5) *Privacy*: Log records should not be casually traceable or linkable to their sources during transit and in storage.

III. Related Work

A number of approaches have been proposed for logging information in computing systems. Most of these approaches are based on *syslog* which is the de facto standard for network wide logging protocol (see RFC 3164 [5]). The syslog protocol uses UDP to transfer log information to the log server. Thus, there is no reliable delivery of log messages. Moreover, syslog does not protect log records during transit or at the end-points.

Syslog-ng [8] is a replacement that is backward compatible with syslog. Some of its features include support for IPv6, capability to transfer log messages reliably using TCP, and filtering the content of logs using regular expressions. Syslog-ng prescribes log record encryption using SSL during transmission so as to protect the data from confidentiality and integrity breaches while in transit. However, syslog-ng does not protect against log data modifications when it resides at an end-point. Syslog-sign [9] is an another enhancement to syslog that adds origin authentication, message integrity, replay resistance, message sequencing, and detection of missing messages by using two additional messages—“signature blocks” and “certificate blocks.” Unfortunately, if signature blocks associated with log records get deleted after authentication, tamper evidence and forward integrity is only partially fulfilled. Syslog-sign also does not provide confidentiality or privacy during the transmission of data or at the end points.

Syslog-pseudo [11] proposes a logging architecture to pseudonymize log files. The main idea is that log records are first processed by a pseudonymizer before being archived. The pseudonymizer filters out identifying features from specific fields in the log record and substitutes them with carefully crafted pseudonyms. Thus, strictly speaking, this protocol does not ensure correctness of logs. That is, the log records that are stored are not the same as the ones that are generated. The other problem with this paper is that while the protocol anonymizes each log record individually it does not protect log records from attacks that try to correlate a number of anonymized records. Our objective, on the other hand, is precisely this. Moreover, privacy breaches that can occur from scenarios such as the user erroneously typing the userid in a password field (as discussed earlier) or identifying information available in fields that are not anonymized, are also not addressed in this paper. The Anonymouse log file anonymizer [12] performs a similar anonymization of identifying information by substitution with default values or more coarse values. However, the problem with this paper is that if the original values are needed in investigation, they cannot be restored. Neither syslog-pseudo nor anonymouse log file anonymizer protects log records from confidentiality and integrity violations and other end-point attacks.

Reliable-syslog [6] aims to implement reliable delivery of syslog messages. It is built on top of the blocks extensible exchange protocol (BEEP [13]) which runs over TCP to provide the required reliable delivery service. The Reliable-syslog protocol allows device authentication and incorporates mechanisms to protect the integrity of log messages and protect against replay attacks of log data; however it does not prevent against confidentiality or privacy breaches at the end-points or during transit.

The notion of *forward-integrity* of log records was proposed by Bellare and Yee [7] to protect precompromise log data from postcompromise insertion, deletion, modification, and reordering. Forward integrity is established by a secret key that becomes the starting point of a hash-chain. The hash-chain is generated by a cryptographically strong one-way function in which the key is changed for every log record. Schneier and Kelsey [14] also proposed a logging scheme that ensures forward integrity. This scheme is based on forward-secure message authentication codes and one-way hash chains similar to that suggested by the Bellare-Yee protocol. However, the major problem of both schemes is that both require online trusted servers to maintain the secret key and verification of log records. If the trusted server is attacked or compromised, it breaks the security of the logging scheme. Holt [15] improves upon the Schneier-Kelsey protocol by incorporating public verifiability of log records. However, this scheme being a public-key based scheme, the overhead is significantly more. None of these three schemes consider the privacy concerns of storing and retrieving log records. In addition, all these three scheme suffer from truncation attacks where an attacker deletes a contiguous subset of log records from the very end. This attack can be launched on log records that have yet to be pushed to the trusted server where they are finally stored. Ma and Tsudik address this problem in [10]. They use the notion of forward-secure sequential aggregate authentication in which individual signatures are folded into one single aggregated signature and all other signatures are deleted. An attacker cannot recreate this signature without knowing all previous signatures. The concern with Ma and Tsudik's scheme is that it is very expensive to verify only a single log record (or block, depending on what is considered a unit of log entries over which the aggregated message authentication code is generated). The whole chain of log records over which the accumulated signature has been generated, needs to be verified to achieve this. Further, Ma and Tsudik's scheme does not address the confidentiality and privacy problems with log file storage and retrieval.

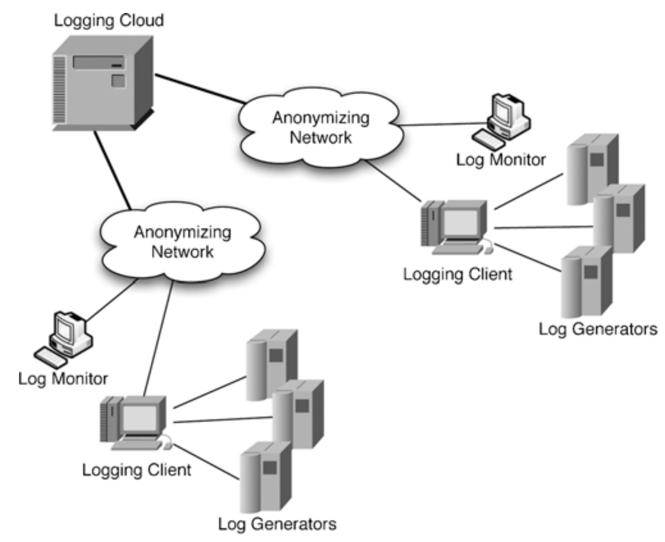


Fig. 1. System architecture for cloud-based secure logging.

IV. System Architecture

The overall architecture of the cloud based secure log management system is shown in Fig. 1. There are four major functional components in this system.

1) *Log Generators*: These are the computing devices that generate log data. Each organization that adopts the cloud-based log management service has a number of log generators. Each of these generators is equipped with logging capability. The log files generated by these hosts are not stored locally except temporarily till such time as they are pushed to the logging client.
2) *Logging Client or Logging Relay*: The logging client is a collector that receives groups of log records generated by one or more log generators, and prepares the log data so that it can be pushed to the cloud for long term storage. The log data is transferred from the generators to the client in batches, either on a schedule, or as and when needed depending on the amount of log data waiting to be transferred. The logging client incorporates security protection on batches of accumulated log data and pushes each batch to the logging cloud. When the logging client pushes log data to the cloud it acts as a logging relay. We use the terms logging client and logging relay interchangeably. The logging client or relay can be implemented as a group of collaborating hosts. For simplicity however, we assume that there is a single logging client.
3) *Logging Cloud*: The logging cloud provides long term storage and maintenance service to log data received from different logging clients belonging to different organizations. The logging cloud is maintained by a cloud service provider. Only those organizations that have subscribed to the logging cloud's services can

upload data to the cloud. The cloud, on request from an organization can also delete log data and perform log rotation. Before the logging cloud will delete or rotate log data it needs a proof from the requester that the latter is authorized to make such a request. The logging client generates such a proof. However, the proof can be given by the logging client to any entity that it wants to authorize.

4) *Log Monitor*: These are hosts that are used to monitor and review log data. They can generate queries to retrieve log data from the cloud. Based on the log data retrieved, these monitors will perform further analysis as needed. They can also ask the log cloud to delete log data permanently, or rotate logs.

We assume that the organization maintains the log generators and the logging client. The log monitor can be maintained by the same organization or can be a separate entity. The logging client can also play the role of a log monitor. We develop our model assuming that the log monitor is a separate entity that is trusted by the logging client. Since the logging client and log monitor operate independent of each other, they can communicate only in an asynchronous manner. This means that if a logging client wants to send some data to the log monitor (or vice versa), the sender cannot expect the receiver to be online to receive the data. As a result the sender has to publish the data in some location and the receiver needs to retrieve the data from there when needed. The logging cloud facilitates this communication by receiving and servicing appropriate requests.

The logging client and the log monitor communicates with the external world, over an unencrypted network that provides anonymous full-duplex communication. Our proof-of-concept prototype gets this service from the Tor network [16], [17]. Attacks on the Tor network that breach anonymity of communicating parties, have been well studied and solutions proposed. Protecting the anonymous communication channel is beyond the scope of this paper.

V. Threat Model

The organization's network is well monitored using security tools such as intrusion detection systems and firewalls. We do not assume that the organization's security cannot be breached. However, we do assume that all log generators and logging client(s) cannot be compromised at the same time. Only some predetermined fraction of them can be compromised simultaneously. In addition, if such a compromise takes place it can be identified rapidly. The logging client and log generators communicate with each other over a synchronous, encrypted, and authenticated network guaranteeing reliable delivery of messages and supporting the BEEP protocol.

We assume a Dolev–Yao attacker model [18] for the system which allows the attacker the following capabilities.

1) The attacker can intercept any message sent over the Internet.
2) The attacker can synthesize, replicate, and replay messages in his possession.
3) The attacker can be a legitimate participant of the network or can try to impersonate legitimate hosts.

In addition, the attacker can attempt to read, delete, and modify data stored in the logging cloud. Given this attack model, the different types of attacks scenarios that we seek to protect against are as follows.

1) *Replay of Log Messages*: The attacker records a set of log messages "$\mathcal{M}$" sent by a logging client to the logging cloud. Later, the attacker attacks the logging client and, in order to hide evidence about the attack, sends "$\mathcal{M}$" to the logging cloud.
2) *Integrity of Transmitted and Stored Log Data*: The attacker has access to the communication medium and can modify data during the transmission.
3) *Authenticity of Logging Client*: The attacker impersonates as the logging client and begins sending log messages to the logging cloud.
4) *Confidentiality of Log Messages*. During transmission the attacker intercepts and reads log messages. In addition, the attacker reads log data stored at the logging cloud.
5) *Privacy of Logging Client, Log Monitor, or Log Generators*: The attacker can actively try to correlate log messages or network traffic to associate these messages with specific logging client, log monitor or log generators, causing privacy breaches.

We assume an honest but curious threat model for the logging cloud. This means that the cloud provider is always available and correctly provides the services that are expected. However, it may try to breach confidentiality or privacy of any data that is stored locally.

VI. Log File Preparation for Secure Storage

Central to our log file preparation protocol is the creation and management of three sets of keys—A_i and X_i which are keys for ensuring log record integrity, and K_i which is a key for ensuring log record confidentiality. These keys are derived in sequential manner starting with three master keys A_0, X_0, and K_0. We use a secret key cryptosystem to provide integrity and confidentiality. We do not rely on a single trusted entity to store and manage keys. Instead, we propose using a secret-sharing scheme (such as the ones by Shamir [19] or Blakley [20]) to distribute the keys A_0, X_0, and K_0 across several hosts. The idea is that given a secret S, and n and q two nonnegative integers such that $0 < q \leq n$, we would like n entities to share the secret S such that: 1) no single entity holds the complete secret; 2) any subgroup of entities of size $\geq q$ can collectively recreate or recover the secret S; and 3) no subgroup of entities of size $t < q$ can re-create or recover the secret.

One of the problems of the secret sharing schemes proposed by Shamir or Blakley is that an attacker can successively compromise each host till it has compromised q hosts. In such a case, the attacker obtains the secret keys. To protect against such a possibility we use a proactive secret-sharing scheme [21], [22]. The idea behind these schemes is that at the end of a fixed period of time, the shares stored at each host change

TABLE I
SYMBOLS USED IN LOG PREPARATION PROTOCOL DISCUSSIONS

Symbol Used	Interpretation
$E_p[M]$	Encryption of message M with some secret key p
$M_1 \mid\mid M_2$	Concatenation of messages M_1 and M_2
$H_k^i[M]$	Cryptographic hash of message M with key k and a hard to invert hash function H with the hash performed i number of times
$H^n[M]$	Message M hashed n number of times
TS	A global timestamp

although the original secret stays the same. In this way, the window of opportunity for the attacker to compromise all q shares is significantly reduced.

A. Forward Integrity and Secrecy for Log Records

Table I summarizes the symbols used in the discussion on log preparation protocol.

The protocol starts with the logging client randomly generating three master keys—A_0 and X_0 for ensuring log integrity, and K_0 for ensuring log confidentiality. These keys need to satisfy the requirements of the chosen proactive secret-sharing scheme. It then embarks upon preparing the log records. Log data arrives at the logging client as a series of messages $L_1, L_2, \ldots, L_n$. Each L_i contains a group of log records generated by a log generator. We assume that these messages are transmitted to the logging client over an authenticated network. The logging client uploads prepared log records in batches of n. The value n is determined randomly at the beginning of each log batch preparation.

1) Before any log data arrives at the logging client, the logging client creates a special first log entry $L_0 = \langle \text{TS, log-Initialization, n} \rangle$. It then encrypts this log entry with the key K$_0$ and computes the message authentication code $\text{MAC}_0 = H_{A_0}[E_{K_0}[L_0]]$ for the encrypted entry with the key A$_0$. The client adds the resulting first log entry for the current batch—$\langle E_{K_0}[L_0], \text{MAC}_0 \rangle$—to the log file.
2) The logging client then computes new set of keys $A_1 = H[A_0]$, $X_1 = H[X_0]$ and $K_1 = H[K_0]$, securely erases the previous set of keys and waits for the next log message to arrive.
3) When the first log message L_1 arrives, the logging client creates a record $M_1 = L_1 \mid\mid H_{A_0}[E_{K_0}[L_0]$. It encrypts M_1 with the key K_1, and creates a message authentication code for the resulting data as $\text{MAC}_1 = H_{A_1}[E_{K_1}[M_1]$. It also computes an aggregated message authentication code $\text{MAC}'_1 = H^n_{X_1}[\text{MAC}_0 \mid\mid \text{MAC}_1 \mid\mid n]$. The log batch entry is $\langle E_{K_1}[M_1], \text{MAC}_1 \rangle$. It then creates the next set of keys $A_2 = H[A_1]$, $X_2 = H[X_1]$ and $K_2 = H[A_1]$ and securely deletes A_1 and K_1.
4) For every new log data L_i that the logging client subsequently receives, it creates log file entries $\langle E_{K_i}[M_i], \text{MAC}_i \rangle$, where $M_i = L_i \mid\mid \text{MAC}_{i-1}$ and $\text{MAC}_i = H_{A_i}[E_{K_i}[M_i]]$. It also creates the aggregated message authentication code $\text{MAC}'_i = H^{n-i+1}_{X_i}[\text{MAC}_{i-1} \mid\mid \text{MAC}_i] \mid\mid n - i + 1$. Once MAC'_i has been generated, MAC^{i-1} is securely deleted. The client finally creates new keys $A_{i+1} = H[A_i]$, $X_{i+1} = H[X_i]$ and $K_{i+1} = H[K_i]$ and securely erases the keys A_i, X_i and K_i.
5) After the client creates the last log entry M_n for the current batch from the last log data L_n, it creates a special log close entry $LC = \langle E_{K_{n+1}}[\text{TS, log-close} \mid\mid \text{MAC}_n], H_{A_{n+1}}[E_{K_{n+1}}[\text{TS, log-close} \mid\mid \text{MAC}_n]] \rangle$, and an aggregated message authentication code MAC'_{n-1}. It then securely erases the three keys used in this step and uploads the resulting log batch and aggregated message authentication code to the logging cloud as one unit. (The protocols for performing this upload is discussed in Section VII.)

B. Analysis

The above protocol ensures four of the five desirable properties that we have identified earlier in Section II. The only property that it does not ensure so far is privacy. The last property is important only when log files are stored or distributed externally. Following is a brief analysis of the protocol. We refer to the protected log batch shown in Fig. 2 for this discussion.

1) *Correctness*: Since we use a reliable delivery mechanism of log records from log generators to the logging clients (which, though not explicitly indicated, can be augmented by cryptographically strong message digests and authentication of end points of communication), the records generated by a generator are the ones received by the logging client and used in the log batch preparation. The logging client does not modify the log records in any manner except adding message authentication codes and encryption. Thus, correctness is ensured.
2) *Tamper Resistance and Verifiability*: Assuming that the master keys A_0, X_0, and K_0 have not been compromised, the protocol ensures tamper resistance of log records as follows. To validate any batch of log records, the validator needs to start with the master key A_0, X_0 and generate the key sequences $A_0, \ldots, A_{n+1}$ and $X_0, \ldots, X_{n+1}$.

   We first consider the case where a complete batch has been created with a log close record and such a batch has been tampered with. This is the scenario when an attacker tries to tamper with a log batch stored at the cloud or on transit on the network. If any log record between L_0 and L_n, say L_i, has been tampered with (modified or replaced), this will result in failure of validation for the message authentication code MAC_i. The same will be the case if the log close record has been tampered with. If any single or sequence of log records has been deleted, say $L_{i+1}, \ldots, L_t$, this will be reflected during validation of the message authentication code MAC_{t+1}. If the log close record is deleted this is evident from the nonexistence of such a record.

   We next consider the case when a partial batch of log records is tampered with. This is the scenario when an attacker breaks into the logging client say at time

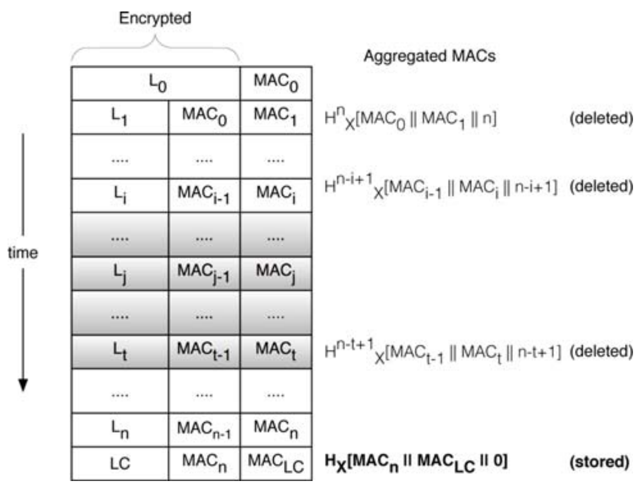

Encrypted
Aggregated MACs
time
L_0 | MAC_0
L_1 | MAC_0 | MAC_1 | $H^n{}_X[MAC_0 \| MAC_1 \| n]$ (deleted)
L_i | MAC_{i-1} | MAC_i | $H^{n-i+1}{}_X[MAC_{i-1} \| MAC_i \| n-i+1]$ (deleted)
L_j | MAC_{j-1} | MAC_j
L_t | MAC_{t-1} | MAC_t | $H^{n-t+1}{}_X[MAC_{t-1} \| MAC_t \| n-t+1]$ (deleted)
L_n | MAC_{n-1} | MAC_n
LC | MAC_n | MAC_{LC} | $H_X[MAC_n \| MAC_{LC} \| 0]$ (stored)

Fig. 2. Log batch uploaded on cloud with aggregated message authentication code.

$(i+1)$. We assume that the break-in is detected at time t. That is, records $L_{i+1}, \ldots, L_t$ are potentially affected. In this case, precompromise records $L_0, \ldots, L_i$ cannot be tampered with without being detected. This is because the message authentication codes $\mathrm{MAC}_0, \ldots, \mathrm{MAC}_i$ has been generated using keys $A_0, \ldots, A_i$ that are not available to the attacker. Consequently, the attacker cannot forge these message authentication codes. The attacker can insert spurious records for the period $(i+1), \ldots, t$. For this to occur, the attacker needs to either disregard an existing aggregated authentication code or forge a new aggregated authentication record. If an attacker disregards an existing aggregatd authentication code, the insertion is detected. If the attacker decides to forge values, it cannot do this without subsequently being detected. This is because the attacker does not know the value of n.

Our scheme can also detect truncation attacks. If the attacker deletes some trailing log records on the logging client, the existing aggregated message authentication code will be rendered invalid. In addition, since the attacker does not know the value n (that is, the number of times the current master keys A_0, X_0 and K_0 can be used), it cannot forge a new aggregated message authentication code. Note that, as opposed to the scheme proposed in [10], we can still validate a single log entry without computing the aggregated code.

Since log messages from log generators to the logging client are sent over authenticated channels, it ensures reliable delivery. An adversary that is masquerading as a logging client will be detected by the security in place at the organization. Replay of log batches as they are being sent to the logging cloud will be discussed in the next section.

3) *Confidentiality*: Since the log records are protected by the encryption keys $K_0, \ldots, K_n$ they cannot be deciphered without having the relevant keys. If an attacker compromises the logging client, it can at most obtain a key K_i that is relevant for log records yet to arrive. It cannot break precompromise log records.

The storage requirement for the log entries is as follows. Each log entry L_i is concatenated with a message authentication code and then encrypted. A message authentication code is generated on the resulting entry and appended to it. To provide reasonable long-term security guarantees, a 256-b message authentication code is sufficient under current technology. Thus, for each prepared log entry a total of 512 bits of overhead is incurred. If n number of log entries form a batch then the overhead for the batch is $512n$. The aggregated message authentication code involves another extra 256 bits. Thus the total space overhead for a batch of n entries is linear in the number of log entries in the batch. This is comparable to the overhead in the Schneier–Kelsey scheme [14]. Moreover, if the size of each L_i is large the overhead for storing the authentication tags is negligible.

VII. Anonymous Upload, Retrieval and Deletion of Log Data

The logging client uploads data in batches where each batch is delimited by a start-of-log record and an end-of-log record. The cloud provider will accept log records only from its authorized clients. Thus, during upload a logging client has to authenticate to the logging cloud to prove that the client had obtained prior authorization from the logging cloud to use the latter's services. However, we do not want the identity of the logging client to be linked to any of its transactions including the authentication process. To balance privacy and accountability, we propose using the k-times anonymous authorization protocol [23] for authentication by the logging client to the logging cloud.

We develop four different protocols for anonymous upload, retrieval and deletion of log data. We assume the existence of an anonymizing network such as Tor to support the anonymous message exchanges in these protocols. We also assume that the logging client knows the public key of the entity that has valid reason to access the log files created by the logging client (the log monitor in our case). This entity also knows the public key of the client. These public keys may be known to the logging cloud or other adversaries.

1) *Anonymous Upload-Tag Generation*: To later retrieve it, an uploaded log batch of log records needs to be indexed by a unique key value. However, we need to ensure that this key value cannot be traced back to the logging client that uploaded the data nor the log monitor that seeks the data. For this purpose, the log data is stored at the cloud indexed by an anonymously generated upload-tag. This upload-tag is created by the logging client in cooperation with the log monitor. It has the property that it is created by publicly available information. However, a given upload-tag cannot be linked either to the corresponding logging client or a log monitor. To retrieve log data from the cloud, the log monitor sends a retrieve request to the logging cloud using an upload tag. The upload-tag is not sent in an encrypted manner. Thus, any adversary can use the upload-tag to retrieve the corresponding log data. However, the log data can be deciphered if and only if the corresponding decryption key is available.

2) *Anonymous Upload*: The logging client authenticates itself to the logging cloud in an anonymous manner. Once successfully authenticated, the logging client sends a formatted message containing the upload-tag, a delete-tag and a batch of previously prepared log data. The delete-tag is used later to delete or rotate the log data (if the logging client, or another entity authorized the logging client needs to do so). Since all messages to the logging cloud are over anonymous channels, any entity can potentially ask the logging cloud to delete a log data. To prevent this from happening, the logging cloud challenges the requester of such a delete operation to prove that it has the necessary authorization. Of course, this can be achieved trivially by the logging client since it can be anonymously authenticated by the logging cloud. However, we allow the deletion to be performed by any entity authorized by the logging client. For this reason, we use the notion of a delete-tag.
3) *Anonymous Retrieve*: This protocol is straightforward. The entity that needs to download log data (most of the time the log monitor), sends a retrieve request (anonymously) together with the upload-tag corresponding to the desired log data. The logging cloud gets the data from its storage and sends it over the anonymous channel to the requester. The cloud provider does not have to authenticate the requester. This is because, by virtue of the log batches being encrypted, the retrieved data is useful only to those who have the valid decryption keys.
4) *Anonymous Delete*: To delete log data, the delete requester sends an appropriate delete message to the logging cloud. In response, the logging cloud throws a challenge to the requester. The requester proves authorization to delete by presenting a correct delete tag.

In the following, we discuss these protocols in detail. We assume that any communication between the logging client and the log monitor (or other entities as applicable) occurs in an asynchronous manner. In other words, when either the logging client or the log monitor wants to send a message to the other party, that entity may not be online. Thus, the message has to be sent to a (sort of) mailbox from where the receiver can later retrieve it. We assume that the logging cloud provides this service. We use the symbols used in Table II for these protocols.

The logging cloud and the log monitor send formatted messages to the logging cloud in order to upload or retrieve any piece of information. The structure of these messages is as follows:

The Tag field will contain information that is used by the logging cloud to index the current message. For log data in particular, this field will contain the upload-tag discussed earlier. The TS field contains time stamp values. The Delete-Tag field either contains the null value N, or the delete-tag related information as discussed earlier. When log data is uploaded the EncryptedData field of the message will contain the relevant data that has been prepared by the logging client as in Section VI. In this case, the OptionalKeyInfo field will be kept blank. If the logging client needs to share any key the EncryptedData field will contain the key suitably protected. The OptionalKeyInfo field in that case, will contain information that will allow the key to be retrieved. Finally, the digest field contains a message digest of the whole message. For the remaining discussion we do not show the digest field explicitly.

TABLE II
SYMBOLS USED IN PROTOCOL DISCUSSIONS

Symbol Used	**Interpretation**
$E_x^{pub}[M]$	Encryption of message M with public key of entity X
$E_k[M]$	Encryption of message M with secret key k
$M_1 \mid\mid M_2$	Concatenation of messages M_1 and M_2
$H_k[M]$	Cryptographic hash of message M with key k using a hard to invert hash function
$rand_k(\cdot)$	A cryptographically strong pseudo-random number generator using a secret key k that is unique to each entity
$ID(X)$	Publicly known identifier of entity X
TS	A global timestamp
N	The null value
$Sig_x[M]$	Message M signed with private key of entity X
$H[M]$	Message digest of message M generated using a hard to invert hash function
$H^n[M]$	Message M hashed n number of times using hash function H

A. Anonymous Upload-Tag Generation

An upload-tag is essentially an instance of a hashed Diffie–Hellman key. The anonymous upload-tag generation protocol (shown in Fig. 3) proceeds as follows.

1) The logging client and the log monitor agrees on a finite cyclic group G having a generating element $\alpha \in G$. They choose a large prime number p such that $p >> \alpha$ and α is a primitive root of p. These steps can be done in an offline manner or the values can be chosen from publicly available information. We assume that all adversaries know that the logging client and the log monitor have chosen these specific values for α and p.
2) The logging client, A, generates a random number r_A and keeps it secret. The log monitor, B, also chooses independently another random number r_B and keeps it secret. A creates its half of the upload-tag as $T_A = Sig_A[\alpha^{r_A} \bmod p]$. It then uploads a message on to the cloud: $ID(A), TS, N, E_B^{pub}[k_1], E_{k_1}[T_A] \mid\mid E_{k_1}[m]$. This value acts as one half of an anchor upload-tag that will be generated. The value m is used as a counter for the number of times the logging client plans to use the anchor for generating upload-tags. Similarly, the log monitor also generates one half of the anchor upload-tag as $T_B = Sig_B[\alpha^{r_B} \bmod p]$ and uploads the message $ID(B), TS, N, E_A^{pub}[k_2], E_{k_2}[T_B]$. Note that we do not require any kind of anonymity for these values uploaded to the cloud. We allow any adversary to tie these values to their sources. In fact, we explicitly have the identities of A and B associated with each of these tag segments.
3) At some point, both A and B retrieve the other party's anchor upload-tag portion. Each of them independently computes the Diffie–Hellman key $\alpha^{r_A r_B} \bmod p$.

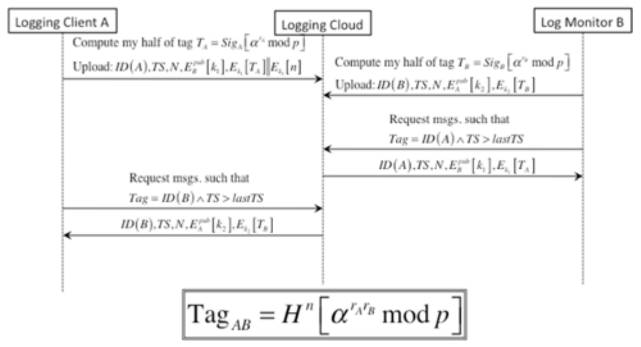

Logging Client A
Logging Cloud
Log Monitor B
Compute my half of tag $T_A = Sig_A[\alpha^{r_A} \bmod p]$
Upload: $ID(A), TS, N, E_B^{pub}[k_1], E_{k_1}[T_A] \| E_{k_1}[n]$
Compute my half of tag $T_B = Sig_B[\alpha^{r_B} \bmod p]$
Upload: $ID(B), TS, N, E_A^{pub}[k_2], E_{k_2}[T_B]$
Request msgs. such that
$Tag = ID(A) \wedge TS > lastTS$
$ID(A), TS, N, E_B^{pub}[k_1], E_{k_1}[T_A]$
Request msgs. such that
$Tag = ID(B) \wedge TS > lastTS$
$ID(B), TS, N, E_A^{pub}[k_2], E_{k_2}[T_B]$
$\text{Tag}_{AB} = H^n[\alpha^{r_A r_B} \bmod p]$

Fig. 3. Upload-tag generation.

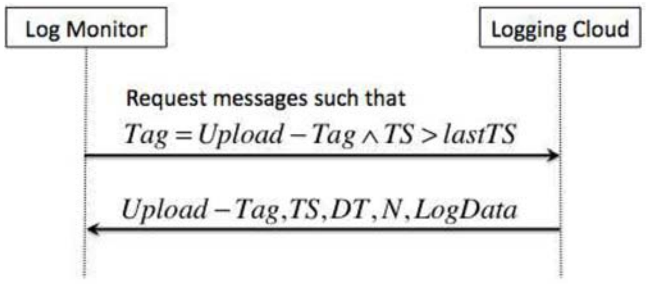

Log Monitor
Logging Cloud
Request messages such that
$Tag = Upload - Tag \wedge TS > lastTS$
$Upload - Tag, TS, DT, N, LogData$

Fig. 4. Protocol for log upload.

4) To create the ith upload-tag, the logging client hashes the Diffie–Hellman key $(m - i)$ times. That is, $upload - tag_i = H^{m-i}[\alpha^{r_A r_B} \bmod p]$. The log monitor does not interact with the logging client in this phase.

The Diffie–Hellman key $\alpha^{r_A r_B} \bmod p$ has the property that given just the public parameters and without the secret values r_A or r_B no adversary can create the key. Since the hash function $H(\cdot)$ is hard to inverse, one cannot generate $\alpha^{r_A r_B} \bmod p$ from $H^{n-i}[\alpha^{r_A r_B} \bmod p]$. Thus, by knowing an upload-tag and adversary cannot trace it back to the logging client that has generated it or the log monitor that has used it.

B. Anonymous Upload and Retrieval of Log Records

The anonymous data upload protocol proceeds as follows (see Fig. 4).

1) For every log data that needs to be uploaded to the logging cloud, the logging client creates a unique upload-tag as discussed earlier. It also creates a unique DeleteTag as DeleteTag $= H^n[rand_k(\cdot)]$ using a cryptographically strong random number. The DeleteTag is encrypted with the public key of the logging cloud—$DT = E^{pub}_{Cloud}[\text{DeleteTag}]$.
2) Using the anonymous communication channel, the logging cloud sends the LogData to the logging cloud as a message $Upload - Tag$, TS, DT, N, $LogData$.

Note that none of the values in the upload message individually or in a group can be tied to the logging client.

To retrieve log data based on a specific upload-tag an entity sends a retrieve request message specifying the upload-tag (see Fig. 5). To account for possible duplicate values on the upload-tag the entity may send some constraint based on time stamp values. Note that the probability that two different logging clients generate the same upload-tag is negligible. It will almost always be the case that these duplicate upload-tags are either instances of replay-attacks launched by some adversary or a logging client intentionally re-using an upload-tag. Hence, some constraint based on time stamps is suggested.

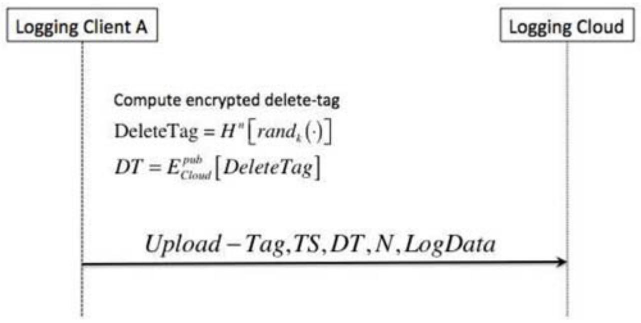

Logging Client A
Logging Cloud
Compute encrypted delete-tag
$\text{DeleteTag} = H^n[rand_k(\cdot)]$
$DT = E^{pub}_{Cloud}[DeleteTag]$
$Upload - Tag, TS, DT, N, LogData$

Fig. 5. Protocol for log retrieval.

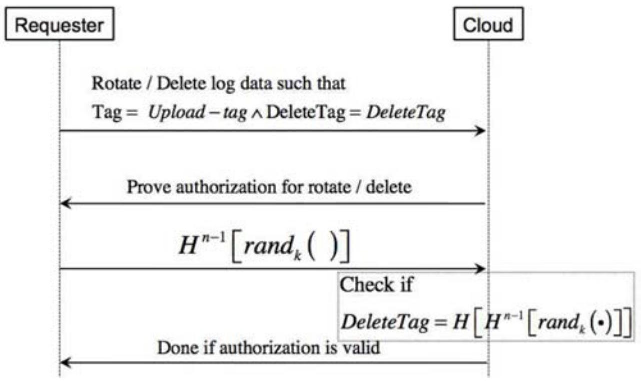

Requester
Cloud
Rotate / Delete log data such that
Tag = $Upload - tag \wedge \text{DeleteTag} = DeleteTag$
Prove authorization for rotate / delete
$H^{n-1}[rand_k(\ )]$
Check if
$DeleteTag = H[H^{n-1}[rand_k(\bullet)]]$
Done if authorization is valid

Fig. 6. Protocol for log delete or rotation.

C. Anonymous Deletion or Rotation of Log Records

The log rotate or log delete operation can be requested only by authorized entities. Evidence of possession of the DeleteTag proves the necessary authorization. If an entity other than the logging client initiates the log delete request, it has to obtain the DeleteTag from the logging client and the relevant Upload-tag. Note that deletion or rotation of log records is a critical operation and consequently needs to be undertaken after considerable deliberation. We propose that a DeleteTag be stored securely by splitting it across n hosts using the same proactive secret sharing scheme as is used in Section VI. For a delete requestor to obtain the DeleteTag, it has to convince some $q \leq n$ number of these hosts to provide a relevant portion of the DeleteTag. Organizational policy will determine what needs to be done by the delete requestor to convince these q hosts. The requestor can then recreate the DeleteTag. The steps for deletion (rotation) are then as follows (see Fig. 6).

1) The requester sends a delete or rotate request to the logging cloud over an anonymous channel. The request message is for log data that satisfies Tag = $Upload - tag \wedge$ DeleteTag = $DeleteTag$.
2) The logging cloud will ask the entity to prove that it is authorized to delete.

3) The entity provides the proof of authorization as $H^{n-1}[rand_k(\cdot)]$.
4) The logging cloud verifies that $DeleteTag = H[H^{n-1}[rand_k(\cdot)]]$. If verified, logging cloud takes the necessary action and responds accordingly.

D. Analysis

The major objective of these protocols is to protect the anonymity of the logging client by preventing the linking of some log record stored on the cloud with the logging client that generated it. (The real objective is to maintain the anonymity of the organization whose log records are being stored on the cloud. We assume that this is equivalent to protecting the anonymity of the logging client.) To get assurance of the anonymity we must ensure that: 1) no single party has enough information to link log records uploaded by the logging client to it, and 2) it is not possible for all parties to collude and get this information.

The adversaries in our protocol are then the honest but curious cloud provider and any attacker that can attack the network and the cloud provider. We assume that all encryption is resistant to cryptanalysis and that they do not leak information.

1) The logging cloud by itself does not have sufficient information to breach anonymity of a logging client. First of all since all messages to the logging cloud from the logging client or log monitor are over the Tor network, these messages cannot directly be tied to their sources. The logging cloud can hold of the following messages—identity of the logging cloud and log monitor when the partial halves of the anchor upload-tag were being generated and the public parameters used to generate these halves. It cannot, however, tie a single half of the anchor to either the logging client or log monitor that generated it. It can also get hold of an upload-tag. However, the upload-tag cannot be tied to the anchor that generated it because of the hash function used. Moreover, by virtue of the upload-tag being derived from a Diffie–Hellman key, an upload-tag cannot be tied to the parameters that were used to generate it.
2) Any adversary that eavesdrop up on the Tor network cannot link any observed message to either a logging client or a log monitor. By attacking the logging cloud it can at most get the same information as that the logging cloud has. Thus, by previous analysis the attacker cannot breach privacy of the logging client or the log monitor.
3) Similar reasoning indicates that by collusion among an attacker and the logging cloud privacy cannot be breached.

Assuming that the anonymous authentication protocol used by the logging cloud is sufficiently robust, an attacker cannot successfully masquerade as a valid logging client. If an attacker replays messages the logging cloud can detect that because every log upload is associated with a unique time stamp. If the cloud colludes with the attacker and does not flag such messages, there will just be duplicate log data in the cloud.

In short, by combination of the log preparation protocol and log upload, retrieval and delete protocols, all the desirable security properties for a cloud based log management protocol are satisfied,

VIII. Early Experiences

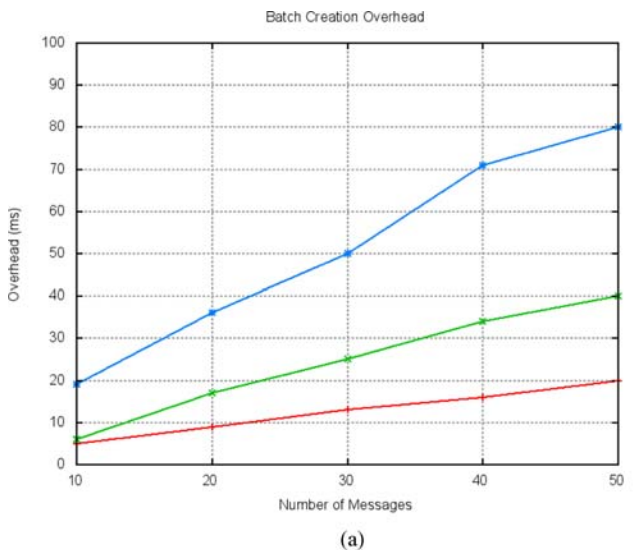
We have developed and implemented a proof-of-concept prototype of our log outsourcing solution. The log client, log monitor, secret-shares repositories, and the log cloud infrastructure have all been developed in Java. Our implementation include 38 Java classes with a total of 3600 lines of code. The log generators used in this prototype are syslog-ng servers. Each machine generating log records is configured with a syslog-ng server to forward the log records to the log client residing on a different machine. All the components of the network topology are located on separate machines within the same network.

The log client is implemented as a multithreaded server application that receives log messages from syslog-ng log generators and puts them in a log batch. It creates a Java object for a log batch and uploads the batch in a special packet as soon as the batch is filled. All the inter-node communications are carried out in the form of packet of different types depending on the operation (such as send batch, retrieve batch, delete batch). The protocol at the log client side starts by generating an initial set of keys that are distributed to the remote shares repositories (implemented on the same machines as the log generators) to satisfy the Shamir secret-sharing scheme. The batch is uploaded to the cloud with appropriate upload and delete tags included in the packet object.

The implementation of the log monitor at present is somewhat rudimentary. It is able to perform the batch retrieval and delete operations (since they are supported by the cloud). We plan to incorporate a more sophisticated query interface into it by providing a shell where the user is able to issue queries over the specified log batches (provided that the corresponding Upload or Delete Tags are also specified in the query).

The actual cloud server is implemented as a multithreaded server that accepts simultaneous connecting log clients and log monitors. Our implementation is such that it can be easily deployed on any existing cloud system. We have used a MySQL database engine to store log data. The choice of MySQL is justified by the small batch table schema (only a few columns) that takes advantage of the extremely fast MySQL tuple fetch capabilities. We store data in the remote MySQL database as byte representation of Java objects via JDBC connection. The batches are stored in the cloud database indexed by the upload-tag. We assume that in actual deployment scenario the database back-ends will reside on separate physical machines (regardless of virtualization) separate from the client serving cloud application so as to alleviate the file I/O load on the cloud application machine (which could be also clustered on multiple physical nodes).

It is worth noting that objects retain their actual Java object properties while being stored as binary objects in a conventional MySQL table. This means that any subsequent



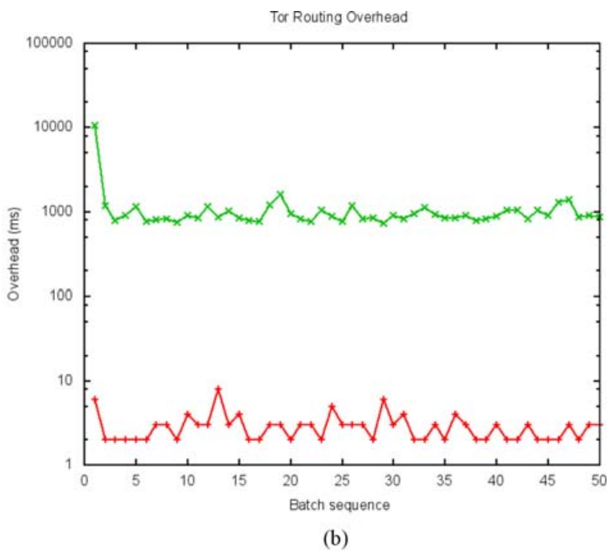


Fig. 7. Figures showing results of experiments. (a) Batch creation overhead. (b) Tor routing overhead.

retrieve operation on the cloud side fetches the Java object from the table, and delivers the same to the log monitor for extraction. The only way for the log monitor to retrieve the actual batch object encapsulated in the packet, is to execute the appropriate method within the object retrieved from the cloud database. This model follows the distributed object-oriented database paradigm employed in many contemporary object databases [24], [25]. The cloud is also responsible for providing any asynchronous communication that is needed by any communicating parties. In particular, the cloud provides the intermediate key storage used for generating the upload-tag between log client and log monitor. It is currently implemented in main memory but is subject to migration into the database back-end.

In our evaluation we wanted to answer the following questions: 1) how does encryption of log records affect the overall performance of our solution? and 2) how does anonymous communication affect the performance of our solution?

To answer the first question, we designed a set of simulations that allows us to analyze the effectiveness of different encryption settings in our prototype. We created test scenarios to understand the time (overhead) that is introduced by the log record preparation process. In our experiments, we used batch sizes of 10, 20, 30, 40, and 50 encrypted log records. For each log batch, we measured three different log-preparation settings. First, we measured the time that it takes to fill up the log batch with log records without any security related preparation. Thus, the first experiment gives us the best achievable performance in our prototype. Second, we measured an overhead for filling up log batch with partial security related log preparation. This includes unencrypted log records (no confidentiality) with calculated MAC and aggregated MAC functions (correctness, tamper resistance and verifiability assurances). Third, we calculated an overhead for security preparation that includes encrypted log records with corresponding MAC and aggregated MAC functions (correctness, tamper resistance, verifiability and confidentiality assurances).

Our experiments involved two identical machines with a 2.13 GHz Intel Xeon E7 CPU and 128 GB memory. Both machines were running Fedora 15 64-bit Linux with 2.6.42.7 kernel and were stationed on the same LAN. The first machine was running the logging client application, while the second machine was running the syslog-ng application that was configured to create fixed-size (100 bytes) log records. The results are shown in Fig. 7(a). Bottom, middle and top lines correspond to the overhead of no security, partial and full security settings. From Fig. 7(a) the overhead in adopting our complete security preparation is about twice as much as no security.

To answer the second question, we designed a set of experiments that measured the performance of anonymous communication. The main objective of this experiment was to study whether our solution was able to scale up in an anonymous network environment. We performed our experiments with 5000 log records from our department's DHCP [26] server. We kept the log batch size fixed to 100 encrypted log records. In our experiments, we compared the time it takes a log batch to traverse between the logging client and logging cloud. Fig. 7(b) shows results from our experiments. The top line corresponds to the overhead from Tor network and the bottom line shows the local network overhead. From Fig. 7(b), we find that Tor network has a fairly constant overhead. Also, we found interesting the fact that it takes about 10 s to send the first log batch to the cloud via Tor network. We think that this spike could be a result of Tor circuit connection setup among the Tor nodes. From our experience with Tor network, we found that it provides reliable data transmission and thus could be easily incorporated into our prototype.

IX. Conclusion and Future Work

Logging plays a very important role in the proper operation of an organization's information processing system. However, maintaining logs securely over long periods of time is difficult and expensive in terms of the resources needed. The emerging paradigm of cloud computing promises a more economical alternative. In this paper, we proposed a complete system to securely outsource log records to a cloud provider. We reviewed existing solutions and identified problems in the current operating system based logging services such as syslog and practical difficulties in some of the existing secure logging techniques. We then proposed a comprehensive scheme that addresses security and integrity issues not just during the log generation phase, but also during other stages in the log management process, including log collection, transmission, storage and retrieval. One of the unique challenges is the problem of log privacy that arises when we outsourced log management to the cloud. Log information in this case should not be casually linkable or traceable to their sources during storage, retrieval and deletion. We provided anonymous upload, retrieve and delete protocols on log records in the cloud using the Tor network. The protocols that we developed for this purpose have potential for usage in many different areas including anonymous publish-subscribe.

Current implementation of the logging client is loosely coupled with the operating system based logging. In the future, we plan to refine the log client implementation so that it is tightly integrated with the OS to replace current log process. In addition, to address privacy concerns current implementation allows access to log records that are indirectly identified by upload-tag values. We plan to investigate practical homomorphic encryption schemes that will allow encryption of log records in such a way that the logging cloud can execute some queries on the encrypted logs without breaching confidentiality or privacy. This will greatly reduce the communication overhead between a log monitor and the logging cloud needed to answer queries on logs.

There are some alternative approaches to Tor for providing nonlinkability and nontraceability of network communications, such as JAP [27], Ultrasurf [28], and FreeGate [29]. Among these, Tor is a more mature implementation and one of the most popular. More importantly, Tor can be easily integrated with any TCP based protocol that can be SOCKS-ified. This was one of our requirements and why we evaluated the performance over Tor. It remains to be seen whether the other anonymizing protocols can be so adapted and, if so, how they perform compared to Tor. This is part of our future work.

REFERENCES

[1] K. Kent and M. Souppaya. (1992). *Guide to Computer Security Log Management, NIST Special Publication 800-92* [Online]. Available: http://csrc.nist.gov/publications/nistpubs/800-92/SP800-92.pdf

[2] U.S. Department of Health and Human Services. (2011, Sep.). *HIPAA—General Information* [Online]. Available: https://www.cms.gov/hipaageninfo

[3] PCI Security Standards Council. (2006, Sep.) *Payment Card Industry (PCI) Data Security Standard—Security Audit Procedures Version 1.1* [Online]. Available: https://www.pcisecuritystandards.org/pdfs/pci_audit_procedures_v1-1.pdf

[4] Sarbanes-Oxley Act 2002. (2002, Sep.). *A Guide to the Sarbanes-Oxley Act* [Online]. Available: http://www.soxlaw.com/

[5] C. Lonvick, *The BSD Syslog Protocol*, Request for Comment RFC 3164, Internet Engineering Task Force, Network Working Group, Aug. 2001.

[6] D. New and M. Rose, *Reliable Delivery for Syslog*, Request for Comment RFC 3195, Internet Engineering Task Force, Network Working Group, Nov. 2001.

[7] M. Bellare and B. S. Yee, "Forward integrity for secure audit logs," Dept. Comput. Sci., Univ. California, San Diego, Tech. Rep., Nov. 1997.

[8] BalaBit IT Security (2011, Sep.). *Syslog-ng—Multiplatform Syslog Server and Logging Daemon* [Online]. Available: http://www.balabit.com/network-security/syslog-ng

[9] J. Kelsey, J. Callas, and A. Clemm, *Signed Syslog Messages*, Request for Comment RFC 5848, Internet Engineering Task Force, Network Working Group, May 2010.

[10] D. Ma and G. Tsudik, "A new approach to secure logging," *ACM Trans. Storage*, vol. 5, no. 1, pp. 2:1–2:21, Mar. 2009.

[11] U. Flegel, "Pseudonymizing unix log file," in *Proc. Int. Conf. Infrastructure Security*, LNCS 2437. Oct. 2002, pp. 162–179.

[12] C. Eckert and A. Pircher, "Internet anonymity: Problems and solutions," in *Proc. 16th IFIP TC-11 Int. Conf. Inform. Security*, 2001, pp. 35–50 .

[13] M. Rose, *The Blocks Extensible Exchange Protocol Core*, Request for Comment RFC 3080, Internet Engineering Task Force, Network Working Group, Mar. 2001.

[14] B. Schneier and J. Kelsey, "Security audit logs to support computer forensics," *ACM Trans. Inform. Syst. Security*, vol. 2, no. 2, pp. 159–176, May 1999.

[15] J. E. Holt, "Logcrypt: Forward security and public verification for secure audit logs," in *Proc. 4th Australasian Inform. Security Workshop*, 2006, pp. 203–211.

[16] R. Dingledine, N. Mathewson, and P. Syverson, "Tor: The second-generation onion router," in *Proc. 12th Ann. USENIX Security Symp.*, Aug. 2004, pp. 21–21.

[17] The Tor Project, Inc. (2011, Sep.) *Tor: Anonymity Online* [Online]. Available: http://www.torproject.org

[18] D. Dolev and A. Yao, "On the security of public key protocols," *IEEE Trans. Inform. Theory*, vol. 29, no. 2, pp. 198–208, Mar. 1983.

[19] A. Shamir, "How to share a secret," *Commun. ACM*, vol. 22, no. 11, pp. 612–613, Nov. 1979.

[20] G. R. Blakley, "Safeguarding cryptographic keys," in *Proc. Nat. Comput. Conf.*, Jun. 1979, p. 313.

[21] R. Ostrovsky and M. Yung, "How to withstand mobile virus attack," in *Proc. 10th Ann. ACM Symp. Principles Distributed Comput.*, Aug. 1991, pp. 51–59.

[22] A. Herzberg, S. Jarecki, H. Krawczyk, and M. Yung, "Proactive secret sharing or: How to cope with perpetual leakage," in *Proc. 15th Ann. Int. Cryptology Conf.*, Aug. 1995, pp. 339–352.

[23] I. Teranishi, J. Furukawa, and K. Sako, "*k*-times anonymous authentication (extended abstract)," in *Proc. 10th Int. Conf. Theor. Appl. Cryptology Inform. Security*, LNCS 3329. 2004, pp. 308–322.

[24] D. L. Wells, J. A. Blakeley, and C. W. Thompson, "Architecture of an open object-oriented database management system," *IEEE Comput.*, vol. 25, no. 10, pp. 74–82, Oct. 1992.

[25] K. Nørvåg, O. Sandstå, and K. Bratbergsengen, "Concurrency control in distributed object oriented database systems," in *Proc. 1st East-Eur. Symp. Adv. Databases Inform. Syst.*, Sep. 1997, pp. 32–32.

[26] R. Droms, *Dynamic Host Configuration Protocol*, Request for Comment RFC 2131, Internet Engineering Task Force, Network Working Group, Mar. 1991.

[27] Project: AN.ON—Anonymity Online. (2012, Mar.). *JAP Anonymity and Privacy* [Online]. Available: http://anon.inf.tu-dresden.de/index_en.html

[28] Ultrareach Internet Corporation. (2012, Mar.). *Ultrasurf—Privacy, Security, Freedom* [Online]. Available: http://ultrasurf.us/index.html

[29] Global Internet Freedom Consortium. (2012, Mar.). *FreeGate* [Online]. Available: http://www.internetfreedom.org/FreeGate

Indrajit Ray received the B.E. degree in computer science and technology from B.E. College, Shibpur, India, in 1988, the M.E. degree in computer science and engineering from Jadavpur University, Kolkata, India, in 1991, and the Ph.D. degree in information technology from George Mason University, Fairfax, VA, in 1997.

He is currently an Associate Professor with the Department of Computer Science, Colorado State University, Fort Collins. His current research interests include computer and network security, database security, security and trust models, privacy, and computer forensics.

He is on the editorial boards of several journals. He has served or is serving on program committees of a number of international conferences and on proposal reviewing committees. He is a member of the IEEE Computer Society, ACM, ACM SIGSAC, IFIP WG 11.3, and IFIP WG 11.9. He was the first Chair of the IFIP Working Group 11.9 on Digital Forensics.

Kirill Belyaev received the M.S. degree in computer science (Fulbright Alumni 2009–2011) from Colorado State University (CSU), Fort Collins, in 2011, with thesis in implementing high-load BGP XML stream broker with routing streams integration (Stanford STREAM Data-Stream Management System). His M.S. thesis work has been turned into the open-source spin-off TeleScope project that is available at http://sourceforge.net/projects/telescopecq.

He is currently a Graduate Student with the Department of Computer Science, CSU. He is currently working in the field of distributed database research, focusing on embedded database replication and query processing in asynchronous environments with the pub-sub broker model (ad-hoc network environments). His current research interests include databases, computer networks, and distributed database models.

Mikhail Strizhov received the B.S. and M.S. degrees in applied mathematics and physics from Samara State Aerospace University, Samara, Russia, in 2008 and 2010, respectively, and the M.S. degree in computer science from the Department of Computer Science, Colorado State University, Fort Collins, in 2012, where he is currently pursuing the Ph.D. degree in computer science.

His current research interests include computer and network security, privacy and anonymity, secure connection protocols, large-scale distributed systems, and cloud computing.

Dieudonne Mulamba received the B.S. degree in mathematics from the University of Kinshasa, Kinshasa, Democratic Republic of Congo, and the M.S. degree in computer science (Fullbright Alumni 2009–2011) from Colorado State University, Fort Collins, in 2007 and 2012, respectively. His M.S. thesis work was in the area of secure and anonymous publish-subscribe protocols. He is currently pursuing the Ph.D. degree with the Department of Computer Science, Colorado State University.

His current research interests include security models and protocols.

Mariappan Rajaram received the B.E. degree in computer science from Bharathiar University, Coimbatore, India, and the M.S. degree from Colorado State University, Fort Collins, in 2009, majoring in computer security.

He is currently a Technical Solution Engineer with Compuware Corporation, Detroit, MI. He leads a small team and works in the field of application performance management. Prior to joining Compuware Corporation, he was an Operations Engineer with Partners HealthCare's Research Computing Division, Boston, MA. His current research interests include computer security and performance engineering.