ABLS - An Attribute Based Logging System for the Cloud

Christopher A. Wood
Department of Computer Science
caw4567@rit.edu

ABSTRACT

User-based non-repudiation is an increasingly important property of cloud-based applica-tions. It provides irrefutable evidence that ties system behavior to specific users, thus enabling strict enforcement of organizational security policies. System logs are typically used as the basis for this property. Thus, the effectiveness of system audits based on log files reduces to the problem of maintaining the integrity and confidentiality of log files. In this project, we study the problem of building secure log files. We investigate the benefits of ciphertext-policy attribute-based encryption (CP-ABE) to solve a variety of log design issues. In addition, we also present the architecture and a preliminary analysis for a proof-of-concept system that fulfills the confidentiality and integrity requirements for a secure log.

1. INTRODUCTION

User-based non-repudiation is a system security property that provides indisputable evidence that links specific actions to individual users (or entities) that trigger such actions. Cryptographically speaking, non-repudiation requires that the integrity and origin of all data should be provable. In essence, this enables system audits to be conducted that can identify data misuse (and thus, potential security policy violations) by comparing the sources of system events with all entities authorized to invoke these events. Therefore, treating non-repudiation as a required system quality attribute in the architecture is likely to become a common trend in the commercial, government, and even more specifically, the health-care domain.

System audits typically use log files to determine the cause and effect of events that took place during the system's lifetime. In order to provide accurate information for non-repudiation purposes, it is often necessary to place some amount user-sensitive data in these log files that can be used to trace data back to its origin. As such, logs of events generated by a client that is being served must maintain data confidentiality and integrity should the system be compromised. These goals are commonly achieved using a combination of encryption and signature techniques [2]. However, traditional approaches to encryption and signature generation and verification are becoming less effective in the context of cloud applications. Furthermore, naive approaches to log security that are based on tamper-resistant hardware and maintaining continuous secure communication channels between a log aggregator and end user are no longer useful in the context of cloud-based applications [3].

Symmetric-key and public-key encryption of log entries are very common confidentiality techniques proposed in the literature. However, in cloud-based applications, these schemes are becoming less useful. There is a need for a robust access control mechanism that enables dynamic user addition and revocation with minimal overhead (i.e. re-encrypting a subset of the log database should be avoided). Both symmetric- and public-key cryptosystems lack in that access policies must be tied directly to keys used for encryption and decryption. If the access policy for a set of log messages needs to be changed, then both the keys used to encrypt and decrypt such log entries will need to be regenerated and distributed, and the entries must also be re-encrypted. Both of these tasks can be very expensive.

In addition, symmetric-key cryptosystems require keys to be shared among users who need access to the same set of logs, which requires a secure and comprehensive key management and distribution policy. From a storage perspective, public-key cryptosystems (e.g. RSA and ElGamal) suffer from the extra data transfer and storage requirements for large cryptographic keys and certificates. There may be insufficient resources to maintain a public-key infrastructure (PKI) for managing keys and digital certificates for all users.

In terms of log file integrity, aggregate signature schemes that support forward secrecy through the use of symmetric- and public-key cryptosystems are also becoming outdated [4]. Symmetric-key schemes may promote high computational efficiency for signature generation, but they do not directly support public verifiability for administrators and auditors. This means that robust key distribution schemes or the introduction of a trusted third party (TTP) are needed to ensure all required parties can access the necessary log information. Such schemes also suffer from high storage requirements and communication overhead. Public-key schemes have similar issues, as the increased key size leads to even larger storage requirements and less computational efficiency.

Collectively, we see that a balance between encryption and signature generation and verification performance is needed to support the unique scalability and resource usage requirements for cloud-based applications. Attribute-based encryption (ABE), a new cryptographic scheme that uses user attributes (or roles, in certain circumstances) to maintain the confidentiality of user-sensitive data, has an appealing application to logging systems maintained in the cloud and is capable of satisfying the aforementioned confidentiality requirements. In addition, authenticated hash-chains have been shown to be effective at enforcing log file integrity in numerous logging schemes [3].

ABLS, an attribute-based logging system that supports ciphertext-policy attribute-based encryption (CP-ABE) [1] and authenticated hash-chain constructions for log file confidentiality and integrity, respectively, was recently designed and implemented by the primary author. The preliminary ABLS architecture and design was weak with regards to the scalability of encryption operations under heavy traffic loads, the relational database schema to store log-data and other sensitive information, and the interactions with database servers. To address these issues, we propose a set of extensions that modify ABLS in the following ways:

1. The CP-ABE encryption scheme will be replaced with a hybrid cryptosystem in which the Advanced Encryption Standard (AES), the standardized symmetric-key encryption algorithm, is used to encrypt user session data by a key that is protected with CP-ABE encryption. In the event that a user generates log messages with varying sensitivity levels during a single session, multiple symmetric keys will be produced to maintain the confidentiality of information in different security classes.

2. The relational database storage scheme will either be changed to include a masking column in the appropriate log table to hide user identities or replaced entirely with a document- or object-based database. Both of these modern database systems are similar in semantics (if documents are conceptually treated as serialized objects), so a product and literature survey will be conducted prior to selecting an adequate replacement that satisfies the necessary security and performance requirements. Though, based on preferences and architectural experience, a document-based database such as MongoDB will be the likely candidate. Also, since the ABLS architecture is highly structured around a relational schema to store data, this change will require modifications made in the logging, attribute authority, and auditing modules in order to maintain functional correctness.

3. Database security mechanisms, including fine-grained access policies for protected databases and encryption of data-in-transit, will be implemented. This is particularly important for the databases that store cryptographic keys. Also, since preference is given to document-based databases such as MongoDB, third-party services such as zNcrypt (provided by Gazzang) that are specifically tailored to this DBMS will likely be used to enforce access control to sensitive databases.

4. The auditing module will be extended to include automated audits of database operations, including changes to the database structure and both successful and unsuccessful client connections with the database. Currently, the auditing module is only designed (and partially implemented) to support strategy-based audits on database contents. However, in this type of application, database performance and security are just as critical. Thus, with this change, there will be two audit techniques that can roughly be equated to data and policy inspections, both of which will only be accessible to users with the appropriate privileges. This access control will be likely be enforced using username and password credentials.

Altogether, the security features of database authentication, encryption, and auditing will be further expanded upon in the existing ABLS architecture. With these changes, we will then re-evaluate the ABLS at an architectural, security, and performance perspective to determine its usefulness in cloud-based settings.

2. LOG DESIGN

In this section we present our log generation and verification schemes. They are influenced by past work done by Schneier et al [3], Bellare et al. [**?**], Ma et al. [2], and Yavuz et al. [4].

2.1 Log Construction

Log integrity is achieved through hash chains and message-authentication codes. Each log entry is a five-tuple element that contains the generating source information, the encrypted payload of the entry, a hash digest that provides a link between the current and previous hash chain entries, and an authentication tag for this digest. In the proof-of-concept system implementation, Keccak is used as the standalone hash function H and the $HMAC$ function is built using SHA-512. Formally, each log entry L_i is built using the following protocol (as depicted in Figure 1):

$$\begin{aligned} X_i =& H(X_{i-1}, E_{SK}(D_i)) \\ Y_i =& HMAC_{EpK_j}(X_i, Ep_j) \\ L_i =& (U_{ID}, S_{ID}, E_{SK}(D_i), X_i, Y_i) \end{aligned}$$

In this scheme the X_i elements are used to link together consecutive entries in the hash chain. Similarly, the Y_i elements are used to provide authentication for the X_i element using an authentication tag that is computed from X_i and the previous epoch digest Ep_{j-1}.

In this context, an epoch Ep_j simply corresponds to a fixed-size set of log entries that are being processed (i.e. an epoch window). For example, if the epoch size is n entries, then the log generation scheme will cycle after n log entries have been constructed and begin working on a new set of log entries. After a cycle is completed, a context block for the most recent epoch is created and inserted into an epoch chain (similar to the log chain). These log generation cycles create frames (or windows) in the entire log chain in which the scheme generates log entries using a single epoch block and key Ep_j and EpK_j, respectively. More specifically, Y_i is the authentication tag that is built using only the entries within the current epoch window.

Context blocks for epoch windows are stored in the same way as log chains. Each epoch chain entry Ep_j is built as follows:

$$\begin{aligned} Ep_j =& HMAC_{EpK_j}(Ep_{j-1}, L_l) \\ Ep_0 =& HMAC_{EpK_0}(0) \end{aligned}$$

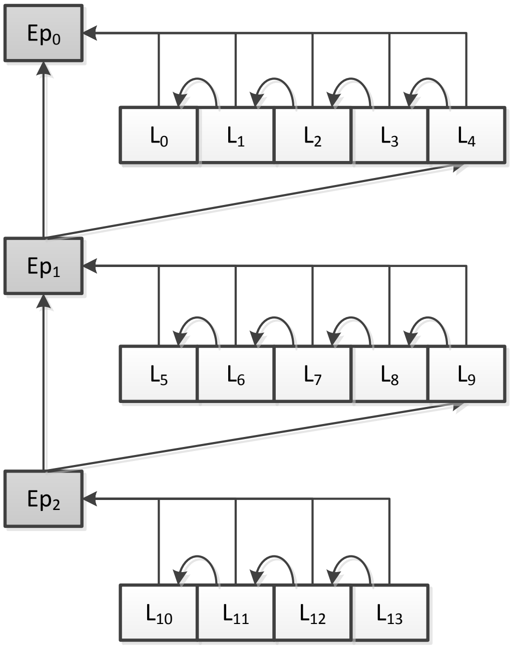

Figure 1: A visual depiction of the hash chain construction scheme. In this case, the epoch window is 5 log entries, as shown by the epoch cycle after 5 consecutive log entries.

In this context, L_l is the last log chain entry for the previous epoch Ep_{j-1}. Thus, each epoch chain entry maintains the integrity of the log chain at each epoch cycle by linking the most recent log chain entry to the previous epoch context block. The key EpK_j that is used to compute the Y_i authentication tag is based on the current epoch Ep_j, and only evolves when the epoch window cycles. This update is done with a pseudorandom function H (which, in our case, is simply the Keccak hash function), and is defined as follows:

$$EpK_{j+1} = H(EpK_j)$$

The initial epoch key EpK_0 is a secret that is initialized when a session is started. Corruption of this key can enable a determined attacker to reconstruct the log chain and epoch chain without detection. Without this information, however, such modifications are always detectable. We refer to Section **??** for a more detailed description of this issue.

Finally, as a third layer of integrity, a single digest for the entire log T_i chain is stored as the log chain is iteratively constructed. Formally, T_i is built as follows:

$$T_i = HMAC_{T_{K_i}}(L_i, T_{i-1})$$
$$T_0 = HMAC_{T_{K_0}}(L_i, 1)$$

The secret key T_{K_0} for the entire log chain is another secret that is initialized when the session is started. Similar to the epoch key, it is evolved with a pseudorandom function H as follows:

$$T_{K_{i+1}} = H(T_{K_i})$$

A visual representation of this protocol is shown in Figure 2. It is important to note that a chain of T_i elements is not maintained. Instead, only the most recent element is persisted to the database. This is critical to prevent truncation attacks.

2.2 Log Generation Rationale

The construction of each log entry satisfies the following properties:

1. Each log entry payload is encrypted and only viewable by those with the appropriate attributes.
2. The integrity of the log chain is ensured through the links generated by X_i elements, which are verifiable by the Y_i authentication tags.
3. The integrity of the entire log chain is guaranteed with the T_i element. This protects the log chain against truncation attacks.
4. The epoch-based log generation enables the logger to control the frequency of data sent to the log server, which helps with load balancing and protects against wiretapping attacks.
5. The epoch window is assumed to be a fixed (constant) size, and thus it requires $\mathcal{O}(1)$ time to verify. The log chain can be verified in $\mathcal{O}(n)$ time, where n is the length of the log chain. These results are discussed in Section 2.2.1.
6. There are three different modes of verification that can be performed (weak, normal, and strongest). These modes are described in Section 2.2.2.

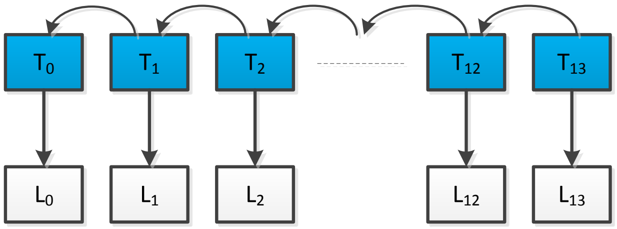

Figure 2: A visual depiction of the protocol used to build the log chain authentication tag.

7. The log is searchable by the user identity and session numbers, which leads to very fine-grained database queries.

2.2.1 *Computational Complexity*

In the worst case scenario, a single log construction event must generate the newest log entry L_i and update the epoch context block chain to cycle into a new epoch window Ep_j. Based on the definition of the log entries, a combination of one hash computation, one $HMAC$ computation, and one entry payload encryption is required. Furthermore, based on the previous definitions, only a single $HMAC$ computation is required to generate each epoch context block.

Under the assumption that each log entry payload will be of a fixed size (in the number of bytes used to represent the data), then we will have a constant time complexity for each encryption operation. Furthermore, since each hash and $HMAC$ operation is parameterized by fixed-length inputs, these will also yield constant time complexities. Thus, we have the following time complexities for each of these operations:

- Hash - $\mathcal{O}(1)$
- $HMAC$ - $\mathcal{O}(1)$
- Encryption - $\mathcal{O}(1)$

Therefore, it is easy to see that log generation has a constant time complexity, which is necessary for system scalability and performance under heavy traffic loads. However, the constant can vary significantly depending on the underlying cryptographic parameters. We will revisit this issue in the evaluation results, presented in Section **??**.

2.2.2 *Verification Modes*

Our log construction scheme enables three different modes of verification to be implemented, each of which has different integrity and performance guarantees. Table 1 provides an overview of the differences between these modes of verification. Algorithms 1 and 2 provide a description of two of these verification modes.

Although the weak verification mode does not provide the strongest integrity guarantees, it is useful for offline analysis of log files by users. In such situations, although the users cannot be entirely positive that the log file has not been tampered with, they can at least view some of their own data. For automated verifiers that run concurrently with the logging system, the strong verification mode enables them to walk the database to perform integrity checks on log chains for user sessions. This provides the users and system administrators with confidence that the log database is correct. We discuss the role of automated verifiers in Section **??**.

Algorithm 1 Strongest verification procedure

```
Require: $UID$, $SID$, $\mathcal{L} = \{L_0, L_1, ..., L_n\}$, $ChainDigest, Ep_k, L_k, EpochSize$
$payload \leftarrow UID|SID|0|L_0[3]|0$
$epoch \leftarrow HMAC(Ep_k, 0)$
$x_1 \leftarrow Keccak(payload)$
Return $FAIL$ if $x_1 \neq L_0[4]$
$y_1 \leftarrow HMAC(Ep_k, epoch|x_1)$
Return $FAIL$ if $y_1 \neq L_0[5]$
$ed \leftarrow HMAC(L_k, x_1)$
$L_k \leftarrow HMAC(L_k,$ ”constant value”$)$
for $i = 1 \rightarrow n$ do
    $payload \leftarrow UID|SID|i|L_i[3]|L_{i-1}[4]$
    $x_i \leftarrow Keccak(payload)$
    Return $FAIL$ if $x_i \neq L_i[4]$
    if $i|EpochSize$ then
        $newKey \leftarrow Keccak(Ep_k)$
        $Ep_k \leftarrow newKey$
        $payload \leftarrow epoch|L_{i-1}[4]$
        $ed = HMAC(newKey, payload)$
    $y_i \leftarrow HMAC(Ep_k, epoch|x_i)$
    Return $FAIL$ if $y_i \neq L_i[5]$
    $ed \leftarrow HMAC(L_k, x_1)$
    $L_k \leftarrow HMAC(L_k,$ ”constant value”$)$
Return $FAIL$ if $ed \neq ChainDigest$
Return $PASS$
```

Algorithm 2 Weak verification procedure

```
Require: $UID, SID, \mathcal{L} = \{L_0, L_1, ..., L_n\}$
$payload \leftarrow UID|SID|0|L_0[3]|0$
$x_1 \leftarrow Keccak(payload)$
Return $FAIL$ if $x_1 \neq L_0[4]$
for $i = 1 \rightarrow n$ do
    $payload \leftarrow UID|SID|i|L_i[3]|L_{i-1}[4]$
    $x_i \leftarrow Keccak(payload)$
    Return $FAIL$ if $x_i \neq L_i[4]$
Return $PASS$
```

2.3 Searchability

Table 1: The integrity and performance differences for the three log verification modes supported by the log generation scheme.

Verification Mode	**Visibility**	**Integrity Guarantees**	**Worst-case Performance**
Weak	Public	Weak	1 Keccak computation
Normal	Private	Medium	2 Keccak and 2 $HMAC$ computations
Strong	Private	Strong	2 Keccak and 4 $HMAC$ computations

Based on the definitions provided in Section 2.1, it is easy to see that log searchability is done by querying the database with known user identities and optional session identifiers. This was a tradeoff that we made to support reasonable auditing performance. It has been shown that in some application domains the presence of any relevant information pertaining to users is a violation of security policies. However, since the entire entry payload is kept confidential to unauthorized users through encryption, we do not anticipate the presence of this information causing many problems.

The database schema for our logging system is shown in Figure **??**. We emphasize that, if the log server were compromised, the only information that a determined attacker could retrieve from the log database is a collection of user identities, which do not directly correspond to "real" user identities (i.e. database user identities are GUIDs that are generated whenever a user account is created by the host application).

3. COMPLETED WORK

1. Symmetric key management
2. User and session attribute columns masked and encrypted
3. Deployment scheme set up, including database systems
4. Initial audit design implemented
5. Generated test data for practical applications
6. Extended the test driver program to provide a more robust interface to the client
7. Researched methods of client authentication

4. DEPLOYMENT STRATEGY

TODO: insert figure

5. FUTURE WORK

TODO

6. REFERENCES

[1] J. Bethencourt, A. Sahai, and B. Waters. Ciphertext-policy attribute-based encryption. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, SP '07, pages 321–334, Washington, DC, USA, 2007. IEEE Computer Society.

[2] D. Ma. Practical forward secure sequential aggregate signatures. In *Proceedings of the 2008 ACM symposium on Information, computer and communications security*, ASIACCS '08, pages 341–352, New York, NY, USA, 2008. ACM.

[3] B. Schneier and J. Kelsey. Secure audit logs to support computer forensics. *ACM Transactions on Information and System Security (TISSEC)*, 2(2):159–176, 1999.

[4] A. A. Yavuz and P. Ning. Baf: An efficient publicly verifiable secure audit logging scheme for distributed systems. In *Proceedings of the 2009 Annual Computer Security Applications Conference*, ACSAC '09, pages 219–228, Washington, DC, USA, 2009. IEEE Computer Society.