# ABLS: Working Towards Attribute-Based Log Security and Automated Auditing in 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 applications. 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 without sacrificing the usefulness of the data in these log files. In an ideal setting, automated audits would be possible on encrypted log data that defines audit trails. Furthermore, since useful log messages may contain sensitive information, access control for log data should be implemented so as to restrict access to only those parties that need to view it (i.e. generating users, colleagues of generating users, auditors, system administrators, etc). ABAC has been a common technique used to satisfy this requirement.

In this paper we address all of the aforementioned issues with ABLS, an attribute-based logging system designed to support automates audits of encrypted audit trails (log data) based on user-defined security policies. Access to sensitive log information is enforced using ciphertext-policy attribute-based encryption (CP-ABE) with a minimal number of log-related roles, and thus a small number of attributes, to avoid the problem of increasing encryption computational complexity with attribute explosion. We present the preliminary design of ABLS and discuss how audit trails are constructed, automated audit tasks are defined and specified, and how the system may be used in practical applications.

## 1. INTRODUCTION

User-based non-repudiation is a system security property that provides indisputable evidence linking 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 contextual information of system events (e.g. source user, time of the event, etc) with all entities authorized to invoke such 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 "who, what, when, and how" 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. Historical approaches to the problem of log security are based on tamper-resistant hardware and maintaining continuous secure communication channels between a log aggregator and end user [7]. However, such solutions are not applicable in the context of cloud-based applications.

Recent approaches have relied on combinations of encryption and signature techniques [5]. Symmetric-key and public-key encryption (and verification) of log entries are very common confidentiality techniques proposed in the literature. Unfortunately, these schemes are becoming less useful in cloud-based applications. There is a need for robust access control mechanisms that enable dynamic user addition and revocation with minimal overhead. In other words, continuously re-encrypting a subset of the log database should be avoided. Both symmetric- and public-key cryptosystems suffer 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 re-generated 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. This requires a secure and comprehensive key management and distribution scheme and supporting policy. In a similar vein, 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

[8]. 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 that all required parties can verify the necessary log data. Such schemes also suffer from relatively 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. Also, public-key schemes introduce the need for a trusted certificate authority to grant certificates for all parties that sign log information. One time-tested technique for supporting log file integrity is the use of authenticated hash-chains [7], which will be the focus of this paper.

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. Furthermore, the selected cryptographic primitives to encrypt, sign, and verify data must not exacerbate the problem of dynamically changing access control policies and user privileges. Role-based Access Control (RBAC), which first gained popularity in the mid 1990s [6] [2] and was later proposed as a standard for the National Institute of Standards and Technology in 2001[3], is an increasingly popular access control policy that enables users to be associated with roles that change less frequently. In the context of maintaining the confidentiality of log messages generated by many users, RBAC surpasses traditional mandatory and discretionary access control (MAC and DAC) [1].

More recently, attribute-based access control (ABAC) [**?**] has been developed to provide fine-grained access control to sensitive data. It is common practice to specify user roles as attributes in this access control scheme, thus enabling the benefits of RBAC with fine-grained access control. Attribute-based encryption (ABE) [4], a new cryptographic scheme that uses user attributes (or roles, in this context) 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 this paper we address all of the aforementioned issues with ABLS, an attribute-based logging system designed to support automates audits of encrypted audit trails (log data) based on user-defined security policies. Access to sensitive log information is enforced using ciphertext-policy attribute-based encryption (CP-ABE) with a minimal number of log-related roles, and thus a small number of attributes, to avoid the problem of increasing encryption computational complexity with attribute explosion. We present the preliminary design of ABLS and discuss how audit trails are constructed, automated audit tasks are defined and specified, and how the system may be used in practical applications.

The paper is organized in a top-down fashion, starting with the structure of log data and corresponding ability to define automated audit tasks. Using this foundation, we then introduce the relational data model used to persist log information, followed by the cryptographic access control mechanisms used to maintain the confidentiality of log data and audit trails. Finally, we conclude with a practical use case for ABLS in the context of healthcare organizations.

## 2. STRUCTURED LOG DATA AND AUTOMATED AUDITS

A major motivating factor for our log data structure comes from realistic security policies. In this context, we make the assumption that a security policy can be stated as a set of *negative* requirements. For example, one such requirement might be that a doctor is not allowed to change their patient's address. In order to conduct an automated audit for violations of this policy, we first translate this semantically-rich requirement into a language whose structure can be easily mapped to a relational data model. This enables us to leverage the power of structured query languages (i.e. SQL) to search for policy violations.

One solution for parsing security policy requirements into relational data is to define a grammar for producing requirement strings from a set of non-terminals that correspond to relations. Using the NIST RBAC model of access control as motivation [3], we specify this set of non-terminals and relations to be the set identifiers USER, OBS, OPS, and AFFECTEDUSERS. These finite sets are minimal enough to allow the specification of most security policies, thus making it suitable for our needs.

LAudit, a simple context-free grammar that is built on these relations, is shown below.

$$\begin{aligned}\text{LAudit} \ ::= \ & \text{USER OPS} \\ | \ & \text{USER OPS OBS} \\ | \ & \text{USER OPS OBS USER}\end{aligned}$$

In this context, USER, OPS, and OBS are all finite sets composed of the users, operations, and objects of a system, as specified by the NIST RBAC model [**?**]. While simple, this language effectively captures the "who" and "what" of log events. ABLS is capable of appending a timestamp to every that it receives, which rounds off the log event with "when" information.

ABLS clients must submit log messages according to a predefined schema that captures all of the information in LAudit. A JSON schema that can be used for constructing log messages is shown below.

```
[
    {
        user : int,
        session : int,
        action : int (or String),
        object : int (or String),
        affectedUsers : [int]
    }
]
```

In this section we present our log generation and verification schemes. They are influenced by past work done by Schneier et al [7], Bellare et al. [**?**], Ma et al. [5], and Yavuz et al. [8].

### 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:

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

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:

$$\begin{aligned} T_i &= HMAC_{T_{K_i}}(L_i, T_{i-1}) \\ T_0 &= HMAC_{T_{K_0}}(L_i, 1) \end{aligned}$$

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 **??**. 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 **??**.

6. There are three different modes of verification that can be performed (weak, normal, and strongest). These modes are described in Section 2.2.1.

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

### *2.2.1 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 **??** 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 **??**.

## 2.3 Searchability

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,

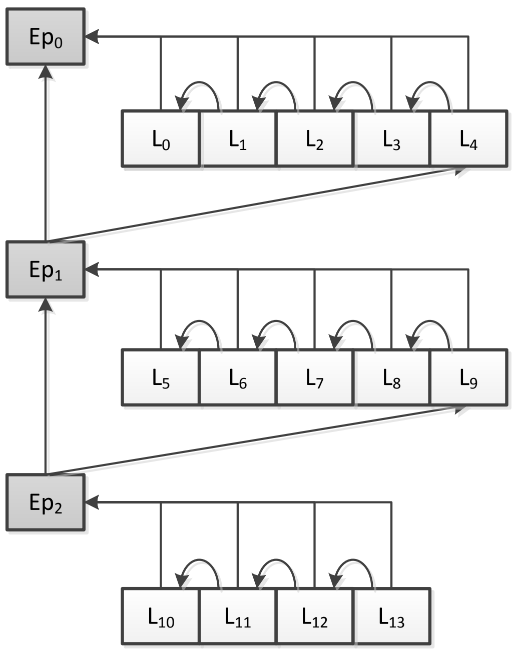

Figure 1: A visual depiction of the hash chain construction scheme. In this case, the epoch window is $5$ log entries, as shown by the epoch cycle after $5$ consecutive log entries.

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 |

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. RELATIONAL MODEL

In order to capture the audit information in a relational model to enable efficient and automated queries, the events, actions, objects, and affected users are all coupled to the incoming log data. The resulting relational model is shown in Figure TODO

a new Event table was added to the database schema. There is a one-to-one correspondence between Event and Log records, and the security of such Event and Log information is maintained using the same hash chain construction techniques as in the preliminary design. However, for simplicity, the notion of hash chain epochs was removed. Also, Action, Object, and AffectedUserGroup tables were added to the database schema to store relevant information about log events as they are received from the log proxy. A high-level depiction of this new relational model is shown in Figure 2.

In this model, all Action and Object records are stored in plaintext. These tables store elements of a finite set, and encrypting them would not deter a determined attacker. However, all information about affected users is encrypted (masked) using the same technique discussed in Section 4.6. As such, an attacker can infer information about what types of objects were operated upon, but they cannot determine the specifics of these actions or the users who performed them without compromising the ABLS master key $M_k$. We feel as though this strikes a good balance between robust audit specification, reasonable measures of audit and log efficiency, and overall log security.

## 4. LOG ACCESS CONTROL

The primary means of access control for log data is enforced using ciphertext policy attribute-based encryption (CP-ABE), a new encryption scheme that supports complex access policies that specify which secret keys can be used for decryption. In CP-ABE, secret keys are analogous to sets of attributes, and access policies are defined using tree-like ac-

**Algorithm 1** Strongest verification procedure

```
Require: $UID$, $SID$, $\mathcal{L} = \{L_0, L_1, ..., L_n\}, ChainDigest, Ep_k, L_k, EpochSize$
 1: $payload \leftarrow UID|SID|0|L_0[3]|0$
 2: $epoch \leftarrow HMAC(Ep_k, 0)$
 3: $x_1 \leftarrow Keccak(payload)$
 4: Return $FAIL$ if $x_1 \neq L_0[4]$
 5: $y_1 \leftarrow HMAC(Ep_k, epoch|x_1)$
 6: Return $FAIL$ if $y_1 \neq L_0[5]$
 7: $ed \leftarrow HMAC(L_k, x_1)$
 8: $L_k \leftarrow HMAC(L_k, \text{"constant value"})$
 9: for $i = 1 \rightarrow n$ do
10:     $payload \leftarrow UID|SID|i|L_i[3]|L_{i-1}[4]$
11:     $x_i \leftarrow Keccak(payload)$
12:     Return $FAIL$ if $x_i \neq L_i[4]$
13:     if $i|EpochSize$ then
14:         $newKey \leftarrow Keccak(Ep_k)$
15:         $Ep_k \leftarrow newKey$
16:         $payload \leftarrow epoch|L_{i-1}[4]$
17:         $ed = HMAC(newKey, payload)$
18:     $y_i \leftarrow HMAC(Ep_k, epoch|x_i)$
19:     Return $FAIL$ if $y_i \neq L_i[5]$
20:     $ed \leftarrow HMAC(L_k, x_1)$
21:     $L_k \leftarrow HMAC(L_k, \text{"constant value"})$
22: Return $FAIL$ if $ed \neq ChainDigest$
23: Return $PASS$
```

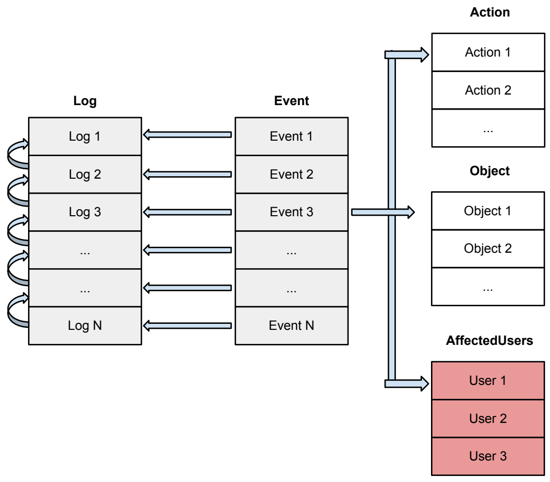

Figure 2: A high-level depiction of the new relational data model that supports automated audit tasks.

cess structures of logical AND and OR gates, where each leaf in the tree is an attribute [?]. Implementations of CP-ABE schemes are usually based on the construction of a bilinear mapping between two elliptic curve groups [?] [?]. We define both of these terms in the following sections.

## 4.1 Mathematical Foundations

**Definition** Let $\mathbb{F}_p$ be a finite field where $p > 3$ is a prime, and $a, b \in \mathbb{F}_p$ such that

$$4a^3 + 27b^2 \neq 0 \mod p \in \mathbb{F}_p$$

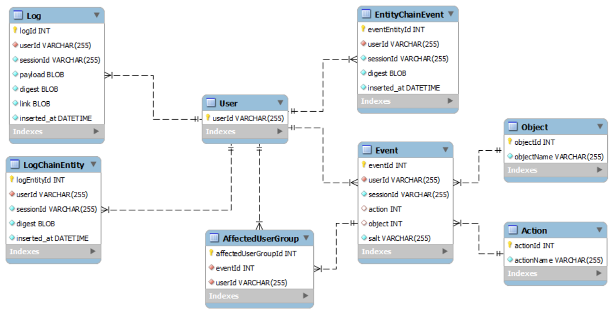

Figure 3: TODO

An *ellptic curve* $E[\mathbb{F}_p]$ is the set of solutions $(x, y)$ to the equation

$$y^2 = x^3 + ax + b \mod p \in \mathbb{F}_p[x],$$

together with the point at infinite 0.

**Definition** Let $G_1$ and $G_2$ be cyclic groups of prime order $p$ and $g$ a generator of $G_1$. We say that $e$ is a *bilinear map* defined as $e : G_1 \times G_2$, where $|G_1| = |G_2| = p$. This bilinear map satisfies the following properties:

- Bilinearity: For all $u, v \in G_0$ and $a, b \in \mathbb{Z}_p$, we have $e(u^a, v^b) = e(u, v)^{ab}$
- Non-degeneracy: $e(g, g, ) \neq 1$
- Computability: Both $G_1$ and $G_2$ are efficiently computable

## 4.2 Ciphertext Policy Attribute-Based Encryption

In the original construction of the CP-ABE scheme, Bethencourt et al. [?] defined five different procedures used in the cryptosystem: *Setup*, *Encrypt*, *KeyGeneration*, *Derypt*, and *Delegate*. We define each of these procedures as follows:

- **Setup** - This procedure takes the implicit security parameter as input and outputs the public and master keys $PK$ and $MK$.
- **Encrypt**(**PK**, M, $\mathbb{A}$) - This procedure will encrypt $M$, a plaintext message, to produce a ciphertext $CT$ such that only a user that possesses a set of attributes that satisfies the access structure $\mathbb{A}$ will be able to decrypt the message. The encryption process embeds $\mathbb{A}$ into the ciphertext.
- **KeyGeneration**(**MK**, $\mathbb{S}$) - This procedure generates a private key $SK$ using the master key $MK$ and set of attributes $S$ that describe the private key.
- **Decrypt**(**PK**, **CT**, **SK**) - This procedure decrypts the ciphertext $CT$ using the provided secret key $SK$ to return the original message $M$. Decryption is only successful if the set $S$ of attributes, which is associated with the key $SK$, satisfies the access policy embedded within the ciphertext (which is part of the access structure $\mathbb{A}$).
- **Delegate**(**SK**, $\tilde{\mathbb{S}}$) - This procedure outputs a secret key $\tilde{}SK$ for the set of attributes $\tilde{}S$, where $\tilde{}S \subset S$, the set of attributes associated with the secret key **SK**.

## 4.3 Access Policy Definition

A major component of ABLS is the policy engine, which maps access policies defined on an event basis to the corresponding access tree used for encryption. For example, an access policy might state that only User XYZ, or physician assistants or nurse practitioners from Medical Group A, are allowed to access data associated with an event $E$. The corresponding access tree for this policy is shown in Figure 4.

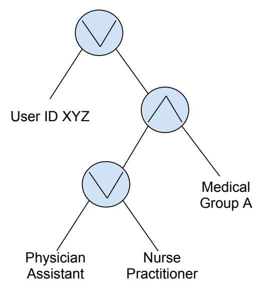

Figure 4: Access tree for a policy that only enables access to user XYZ, Medical Group A physician assistants, and Medical Group A nurse practitioners.

Our logging scheme makes the assumption that events, and the access policies for all data associated with such events, are well defined. With this assumption, we can represent the behavior of the policy engine by the system users and their associated data, events, and policy rules for specific events. In this way, policy rules are coupled to events in that policies for access are generated based on the type of event that occurred and the user who is requesting access to such information. Policy rules can be thought of as functions similar to the one described by `RuleE1` and `RuleE2` (Algorithms 2 and 3, respectively). In this example, we see that the rule for event E1 takes an `EventInformation` structure, which contains all of the information generated for an event. A prospective definition for an `EventInformation` structure is shown below.

```
struct
{
    User SourceUser;
    int EventId;
}
EventInformation;
```

The information contained within this structure is specific to the implementation of the system. For our purposes, we only require the source user from which the event was generated, as well as the event identifier. The policy engine would then query the user database to determine the secret identity of the source user and embed this in the resulting attribute list. Optionally, depending on the implementation of `RuleE1`, the policy engine would append additional attributes to the list in disjunctive normal form. The simplicity of these policy implementations makes changing them an effortless task should the organization's security policy change.

## 4.4 Key Generation and Management

By default, the secret keys used for decryption are never cached in the system's local memory. Since it is expected that

**Algorithm 2** `RuleE1` policy implementation

```
Require: An EventInformation object instance $e$.
Ensure: The access policy for the event corresponding to the
    information in $e$
1: attributeId $\leftarrow$ database.queryUserId(e.SourceUser)
2: Return ('attributeID' OR 'Colleague of attributeID' OR
    'System Administrator')
```

**Algorithm 3** `RuleE2` policy implementation

```
Require: An EventInformation object instance $e$.
Ensure: The access policy for the event corresponding to the
    information in $e$
1: attributeId $\leftarrow$ database.queryUserId(e.SourceUser)
2: Return ('attributeID' OR 'System Administrator')
```

log entries will be read much less frequently than they will be written, such keys are generated on demand by querying the appropriate database. Furthermore, the key generation process can be done in two ways. For policy rules that limit the access to only the generating user, only a single query to the attribute database is required to establish the user's secret key and then decrypt the data for all log entries corresponding to that rule. With this key the user may decrypt these entries offline without the need to query the policy engine for the appropriate access rights.

Conversely, for access policy rules that embed conditions for colleagues or other users related to the source user, the policy engine must first query the user database to ensure the requesting user meets the relationship criteria set in place by the policy. Then, if this is successful, the policy engine will grant the appropriate secret key to the requesting user. The tradeoff is that, while an online TTP is needed for such colleagues to access the log entry contents, it is significantly easier for the system administrators to manage who has access to specific log entries aside from the original source user. Simply modifying the user's relationships in the system database is sufficient to revoke access from certain colleagues.

In order to maintain the security of the system at runtime, it is necessary to cycle the master and public keys associated with the encryption scheme. Our current system does not support this feature, but there are two ways that it could be implemented. The first way is to persist the old master and public keys to a safe location that could be called during auditing and verification if needed. The second way is to re-encrypt the entire log database with the new master and public key. Unfortunately, this would not only require the system to be brought offline during the update (in order to avoid synchronization issues with live traffic), but it would also mean that the new master and public key serve as a single point of failure for the entire database if compromised. Therefore, future releases of this system plan to implement the first approach to manage keys. It would be best to determine the key cycle lifetime based on empirical data associated with the growth of the log database. Intuitively, in order to maintain auditing and verification efficiency, the cycle frequency should be defined as a monotonically increasing function that is proportional to the growth of the database.

## 4.5 Hybrid Key Management

Pairing-based cryptography is computationally expensive, and under the assumption that ABLS might be subject to very heavy traffic loads at any particular time, the overhead of encrypting data to be stored in the database should be as minimal as possible. Therefore, each unique policy that is needed to encrypt a log message is associated with an AES-256 symmetric key, which is in turn encrypted using CP-ABE and then serialized to be stored in the key database. The Charm crypto package allows all cryptographic objects (which tend to be nested Python dictionaries and other complicated data structures) to be serialized to byte representations for database persistence. This design enhancement enables increased throughput without sacrificing the level of confidentiality granularity that is needed for each log entry. However, should an unencrypted policy key for a given user's session become compromised, the remaining entries in that log database are at risk of being compromised.

The basic procedure for encrypting a log entry is shown in Algorithm 4. Once encrypted, the ciphertext is stored in the database with the rest of the information necessary to continue the log chain for a given user's session.

**Algorithm 4** Log entry encryption

```
Require: An unencrypted log entry $L_i$ for session $S_j$ of user $U_k$
1: Let $P$ be the access control policy for the message of $L_i$, as determined by the PolicyEngine
2: if The symmetric key $K$ for $(U_k, S_j)$ has not been generated for $P$ then
3:     Generate $K$ and encrypt it with the CP-ABE encryption module using $P$, yielding $K_E$
4:     Persist $K_E$ to the key database
5: else
6:     Query the database for $K_E$, the encrypted key for policy $P$.
7:     Decrypt $K_E$ using the attributes of user $U_k$, yielding $K$
8: Encrypt $L_i$ with AES-256 using $K$, yielding $E(L_i, K)$
9: Persist $E(L_i, K)$ to the log database
```

In order to improve the performance of the logger, the per-policy symmetric keys for a user session are kept in memory until the session has been closed. This avoids the need for the logger to query the database for the key when a new log message arrives.

Also, in order to ensure that every encryption module (cipher) contained within loggers, verifiers, and database shims uses the same master key $M_k$, a key manager singleton object was implemented and shared among all ABLS entities that require the master key. Upon creation, an encryption module will register itself with the key manager in order to receive any changes made to the master or public key.

### 4.6 Database Design

As shown in section 5, there are five main databases that must be maintained by ABLS: the log, key, user, audit_user, and policy database. The log database maintains all information in the log chain for every single user and session pair. The key database stores the cryptographic keys that were used to construct such log chains. The user, audit_user, and policy databases store user information and policy rules for ABLS, respectively. In the current prototype of ABLS, the policy database is not used internally. Instead, all event rules are hard-coded into the `PolicyEngine`.

In order to link the entries in the log tables to their corresponding verification and encryption keys in the key database, common user and session IDs are used (though not as the primary key for the tables since they do not satisfy the uniqueness property). However, storing user and session information in plaintext may lead to a privacy violation if the database is compromised. Therefore, using a technique similar to the "onion encryption" design in CryptDB [?], this information is now deterministically encrypted before being stored in the database.

This procedure works by encrypting the user and session attributes with a symmetric key generated from the logger's master key salted by the target table identifier. In mathematical terms, the encrypted user and session IDs, $[U_i]$ and $[S_j]$, stored in table $T$ are generated as follows.

$$[U_i] = E(M_k||H(T), U_i)$$
$$[S_j] = E(M_k||H(T), S_j)$$

In this context, $M_k$ is the master key for the logger. Using the table identifier as a salt to the master key enable ensures that tables do not share any common information about the user, which helps prevent against inference attacks in the event that the database servers are compromised. Furthermore, this enables verifiers, who will have access to $M_k$ through the key manager, to decrypt log entries and recover the user identifier so that they may check the contents of the other databases as needed.

Also, to better support audits that use the log database, a timestamp field was added to each table as a required attribute. Not only does such information capture the exact timing of critical system events, it acts as a protection mechanism in the event that log messages are inserted into the database out of order. Also, it is important to note that each and every timestamp for a log message is generated when a database entry is generated to be sent to the log collector.

## 5. DEPLOYMENT

ABLS is designed to be a centralized logging system backed by a set of distributed databases. A context diagram for the ABLS deployment scheme is shown in Figure 5.

Based on the purpose of each piece of data used in the log, it is best to physically separate databases that store data of different security classes rather than rely on a single, segregated database that uses MAC with polyinstantiation to protect data of different security classes. Of course, access control and authentication mechanisms for all of the database servers is to be enforced at the operating system level, thus prohibiting immediate access to all unauthorized users other than the internal tasks (i.e. logger, verifier, policy engine, etc) within an ABLS instance.

## 6. ANALYSIS

The prototype ABLS system was written in Python, utilizing the Charm cryptography package [?] for pairing-based cryptographic primitives. A major concern for this new architecture was its scalable performance while processing large amounts of log data.

Therefore, we experimented with the performance overhead incurred by the log encryption process and corresponding storage

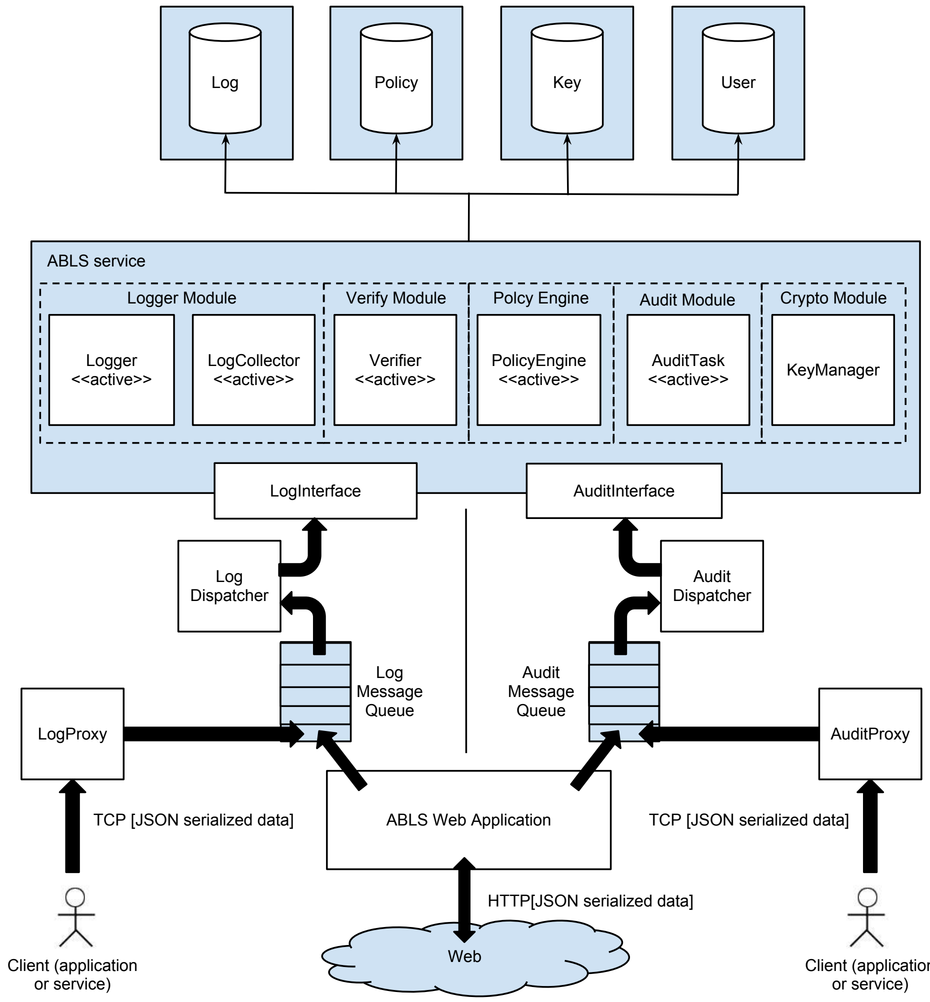

Figure 5: A high-level depiction of the deployment for an ABLS instance, where each box represents a unique runtime environment (i.e. a unique server).

12K (fresh) 134K (after 100 log1) 147K Feb 9 23:48 log.db (log3) 149K Feb 9 23:49 log.db (log4)

approximately 1.2KB per log entry...

TODO: estimated storage overhead and performance results (grab plots tonight and throw into the report)

## 7. APPLICATIONS

TODO: Electronic Health Record Systems

## 8. REFERENCES

[1] M. Abrams, K. Eggers, L. LaPadula, and I. Olson. A generalized framework for access control: An informal description. In *Proceedings of the 13th National Computer Security Conference*, pages 135–143, 1990.

[2] F. David and K. Richard. Role-based access controls. In *Proceedings of 15th NIST-NCSC National Computer Security Conference*, volume 563. Baltimore, Maryland: NIST-NCSC, 1992.

[3] D. Ferraiolo, R. Sandhu, S. Gavrila, D. Kuhn, and R. Chandramouli. Proposed nist standard for role-based access control. *ACM Transactions on Information and System Security (TISSEC)*, 4(3):224–274, 2001.

[4] V. Goyal, O. Pandey, A. Sahai, and B. Waters. Attribute-based encryption for fine-grained access control of encrypted data. In *Proceedings of the 13th ACM conference on Computer and communications security*, pages 89–98. ACM, 2006.

[5] 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.

[6] R. Sandhu, E. Coyne, H. Feinstein, and C. Youman. Role-based access control models. *Computer*, 29(2):38–47, 1996.

[7] 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.

[8] 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.