An Architecture Supporting Automated Audits Over Encrypted Log Data in the Cloud

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

Abstract

User-based non-repudiation is an increasingly important property of cloud-based systems. It provides irrefutable evidence that ties system behavior to specific users, thus enabling strict enforcement of organizational security policies, such as HIPAA. System logs are typically used to construct audit trails of user behavior, which can then be examined to enforce this property. Thus, the efficacy 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. The effectiveness of this approach can be further improved with automated audits. Unfortunately, as log data stored in the cloud needs to be encrypted, autonomous audits that query such data for information about system behavior are becoming increasingly difficult to implement. In essence, the system needs to support the ability to query over encrypted data. In this work we present the architecture for ABLAS (Attribute-Based Logging and Auditing System), a proof-of-concept auditing system that simultaneously supports robust access control to log data with attribute-based encryption and the ability perform automated audits over encrypted log data using user-defined security policies.

1 Introduction

User-based non-repudiation is a security property that provides indisputable evidence linking system behavior to individual users. Cryptographically speaking, non-repudiation requires that the integrity and origin of all data should be undeniable and 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.

Audits typically use log files to determine the contextual information of events (i.e. the "who, what, when, and how") that take place during a 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.

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. source users, colleagues of source users, auditors, system administrators, etc). ABAC is a common technique used to satisfy this requirement. In an ideal setting, automated audits on encrypted log data would also be possible so as to support manual inspections done to comply with federal regulations or compliance laws.

Automating the audit process is another ideal feature for cloud-based logging systems, especially if the granularity of data collected is very small. Under such circumstances, checking log data in real-time as it comes in from clients is not feasible. Furthermore, given the copious amount of data stored in the log database, manual searches for events of interest can be an exhaustive and time-consuming process. Therefore, automated audits

that asynchronously examine the log data to determine if system security policies have been violated are very ideal in these scenarios. Unfortunately, as the log data is encrypted in the cloud, such automated audits require that these autonomous auditing tasks have the ability to query over encrypted data to determine if events of interest have occurred.

In this paper we address all of the aforementioned issues with a proof-of-concept system called ABLAS, an attribute-based logging system designed to support automated 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) [7] 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 initial design of ABLAS and discuss how audit trails are constructed from log data to ensure confidentiality and integrity, the design through which automated audit tasks are defined and specified, a preliminary analysis of the performance and storage overhead incurred by the system, and, finally, how the system may be used in practical applications.

2 Related Work

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 [19]. However, such solutions are not appropriate in the context of cloud-based applications.

Recent approaches have relied on combinations of encryption and signature techniques [14]. Symmetric-key and public-key encryption of log entries are very common confidentiality techniques proposed in the literature. Unfortunately, these schemes are becoming less useful in modern corporate environments with dynamically changing users and access policies. 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 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. 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. In some systems 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 very popular solutions in the literature [21]. However, while symmetric-key schemes 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 [19], which will be a focus of this paper.

Collectively, 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 [17] [9] and was later proposed as a standard for the National Institute of Standards and Technology in 2001[10], 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) [20] [3] [22] has been developed to provide more 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) [11], a new cryptographic primitive 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.

3 Secure Logging Requirements

Some of the fundamental requirements for a secure logging system are that it provides log data integrity and confidentiality. In the context of a secure logging system, integrity is based on the forward-secure stream integrity model [6], which guarantees that log data cannot be forged or rearranged within the stream of log messages. This model also specifies that the logging system is resilient against attacks that try to recover old keys after a machine has been compromised (hence, forward-secure).

Research into this problem has since revealed that a variety of other realistic requirements also exist, including a resilience to truncation and delayed detection attacks, minimal reliance on an online server for log storage and verification, and storage efficiency. For completeness, truncation and delayed detection attacks are defined below, based on explanations presented in [15]. These attacks assume the presence of an untrusted log server $\mathscr{L}$ and a trusted machine $\mathscr{T}$.

- **Truncation attack** - An attack in which a set of log entries residing on an untrusted log server is truncated, or shortened so as to remove suspicious events, without being detected by synchronization with the trusted log server.
- **Delayed detection** - An attack in which log entries on an untrusted log server $\mathscr{L}$ can be modified by an attacker who possesses the authentication key A_i after compromising the system between log entries L_i and L_{i+1}. With this key, the attacker can easily change new entries in the log. However, once the trusted machine $\mathscr{T}$ synchronizes with $\mathscr{L}$ to check the integrity of the log messages, this attack is immediately detected. Of course, significant damage and log data modification could have already been done during this time window (i.e. between the time when $\mathscr{L}$ was compromised and when it was synchronized with $\mathscr{T}$).

4 Logging Scenario

Before describing the logging scheme for ABLAS, we must first explain the typical scenario in which the log data will be generated, stored, and accessed. Effective log servers operate asynchronously with software applications that generate log data. Users of such applications do not directly submit log themselves. Instead, the application is developed such that, depending on the user's interaction with the application, the appropriate log message is sent to the server. This type of user-based logging promotes an injective assignment between system log messages and users of the application, thus helping to promote user-based non-repudiation.

Furthermore, users interact with applications in distinct sessions. For example, in the context of an Electronic Health Record (EHR) system, a user will typically securely login, perform some actions or view their current health records, and then log out. This behavior constitutes a single session, and is an important distinction for ABLAS. Log data sent to the ABLAS server must correspond to a user session, not simply a user. Since ABLAS stores structured log data, rather than a "dump" of log strings, this additional metadata for log information is necessary to help organize the information in a useful way. Formally, in the context of ABLAS, a session is any continuous timeframe where a client authenticates with the system, interacts with the system and generates some set of corresponding log data over an arbitrary amount of time, and then terminates their connection.

Finally, ABLAS log servers are deployed in the cloud, not on a local environment (in actuality, there is no technical constraint prohibiting this option). This deployment scheme means that log data should inherently be encrypted to prevent disclosure to unauthorized personnel.

5 Log File Reliability

Log file reliability, which reduces to log integrity, can be achieved through hash chains and message-authentication codes (MACs). This log construction scheme was first introduced by Schneier et al. [19] and motivated by Bellare et al. [6]. Each log entry L_i is a five-tuple element that contains the generating source information (i.e. the user U and session identifier S), the encrypted payload C of the log data

D, a hash digest X that provides a link between the current and previous hash chain entries, and an authentication tag Y for the digest X. This iterative construction is formalized by the following equations:

$$X_i = H(X_{i-1}, E_K(D_i))$$
$$Y_i = HMAC_{A_i}(X_i)$$
$$L_i = (U_i, S_i, E_{PK}(D_i), X_i, Y_i)$$

In this scheme the X_i elements are used to link together consecutive entries in the hash chain. Similarly, the Y_i elements serve as authentication tags for the X_i element. Also, the elements K and A_i are the policy key (which is discussed later) and MAC key for this hash chain. The initial value for the MAC key A_0 is randomly generated when a user session is created, and is continually evolved every time it is used with a forward-secure pseudorandom function H as follows:

$$A_{i+1} = H(A_i)$$

In order to prevent truncation and deletion attacks, a single entity T_i for the entire log chain is updated as the log chain is iteratively constructed. Formally, T_i is computed as follows:

$$T_i = HMAC_{B_i}(L_i, T_{i-1})$$
$$T_0 = HMAC_{B_0}(L_i, 1)$$

B_0 is a secret key that that is randomly generated when the log chain for a user session is initialized. Similar to the key A_i, this key is evolved as follows:

$$B_{i+1} = H(B_i)$$

This entity creation and hash chaining scheme could be replaced with a publicly-verifiable Forward Secure Sequential Aggregate (FssAgg) signature scheme backed by a trusted certificate authority (CA) to support forward-secure stream integrity. However, since we do not assume the presence of such a CA, we did not make this modification to our logging scheme. We refer the reader to [14] for a more thorough treatment of FssAgg aggregate signature schemes and their role in updating the single log chain entity.

5.1 Log Verification

The ABLAS log construction scheme enables two different modes of verification to be implemented, each of which has different integrity and performance guarantees. The first mode of verification requires any entity, trusted or untrusted, to walk the log chain, computing the digest X_i and comparing it to the value stored in the database. However, this method does not guarantee the integrity of the log chain because an attacker may easily modify the X_i values of the hash chain if they compromise the log server.

The second mode of verification requires a trusted verifier task $\mathscr{V}$ to use the initial hash chain key A_0 and entity key B_0 to walk n log entries, computing both X_i and Y_i, for all $0 \leq i \leq n$, and comparing them against the values stored in the database. At the end of this traversal, $\mathscr{V}$ will compare the final entity value B_n against the value stored in the database and only accept the log chain if these values are equal. While this mode does not lend itself to public verifiability, it guarantees the integrity of the log chain if a forward-secure MAC is used and the keys A_0 and B_0 are protected.

6 Log Access Control

Access control for all of the log data is enforced using ciphertext policy attribute-based encryption (CP-ABE), a pairing-based cryptographic primitive that embeds robust access policies in ciphertext which specify the secret keys that can be used for decryption [7]. In CP-ABE, secret keys are analogous to sets of attributes, and access policies are defined using tree-like access structures of logical AND and OR gates, where each leaf in the tree is an attribute corresponding to a subject requesting access control. Implementations of CP-ABE schemes are usually based on the construction of a bilinear mapping between two elliptic curve groups [7] [12].

In the original construction of the CP-ABE scheme, Bethencourt et al [7] defined five different procedures used in the cryptosystem: *Setup*, *Encrypt*, *KeyGeneration*, *Derypt*, and *Delegate*. For completeness, we define each of these procedures below:

- **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 $\mathbb{A}$ embedded within the ciphertext.
- **Delegate**(**SK**, $\tilde{S}$) - 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**.

In our system, the delegate procedure is not used, though if we need to support delegation of access to log files we can easily do so. The procedure for encrypting log database entries is shown in Algorithm 1.

Algorithm 1 Log entry encryption

Require: An unencrypted piece of log data D_i for session S_j of user U_k

1: Let P be the access control policy for the data D_i, as determined by the policy engine
2: **if** The symmetric key PK for (U_k, S_j) has not been generated for P **then**
3: Randomly generate PK 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 PK
8: Encrypt D_i with PK, yielding $E_{PK}(D_i)$
9: Persist $E_{PK}(D_i)$ to the log database

6.1 Access Policy Definition

A major component of ABLAS is the policy engine, which maps access policies for events to a 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 1.

ABLAS is unique in that temporal attributes can be embedded in the access policy trees. For example, an attribute that states the requesting user is a

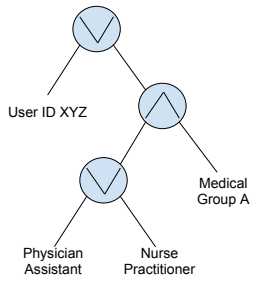

Figure 1: Access tree for a policy that only enables access to user XYZ, Medical Group A physician assistants, and Medical Group A nurse practitioners.

"colleague of User XYZ" can be added to the access policy. As will be discussed in the next section, the addition of this type of temporal, or context-sensitive, attribute enables flexible addition and revocation of access from users who do not own log data.

Our logging scheme makes the assumption that events, and the access policies for all data associated with such events, are well defined, which is often the case with organizations that must comply with federal regulations like HIPAA [5]. With this assumption, administrators are required to define access policies for events of interest based on system-wide attributes (e.g. user attributes, data attributes, etc). In this way, policy rules are coupled to events so that policies for access are generated based on the type of event that occurred and the user who is requesting access to such information. System administrators may define these policies using the standard eXtensible Access Control Markup Language [4].

7 Structured Relational Data

The volume of log data generated by modern EHR systems that are properly configured to record comprehensive data for auditing purposes makes real-time policy enforcement a difficult task. An alternate solution is for the auditing system to periodically, asynchronously, and autonomously compare the contents of log files against some pre-defined audit policy to check for violations. Our system addresses this solution by leveraging the robust policy specification language of XACML [4] and power of relational database query languages (e.g. SQL) to construct an efficient and usable framework for automated audits.

Policies are at the core of the auditing system. The schema of the XACML specification language

Table 1: XACML-to-Relation mapping. In the ABLAS data model, an Event relation, which contains the logging information, also wraps entries for all of these individual relations that are derived from XACML.

XACML Element	*Data Model Relation*
Target	NA
Rule	NA
Subjects	User
Resources	NA
Actions	NA
Environment	NA

is mapped to a relational data model that enables autonomous auditing tasks to periodically query the contents of the database in search of policy violations. The mapping of the XACML schema to the relational model is based on the policy targets and rules. Policies used in the context of access control mechanisms also leverage the Obligation and Rule Combination Algorithm elements, but for the purposes of specifying general security policies, these elements are not necessary.

In XACML, a target element consists of subject, resource, action, and environment elements that are used to determine whether the parent policy is applicable to incoming requests. Since we are not using this as a form of access control specification, we do not need to consider requests. Therefore, in our system, targets serve to enumerate generic policies that can be defined in terms of subjects, resources, actions, and the system environment. An example policy might be that no subject should be allowed to modify patient lab results (resource) after they've been persisted in the system. The XACML equivalent description of this policy might then take the form shown in Figure 2.

With the ability to express general security policies using the XACML policy specification language, we mapped XACML elements to relations in our data model. This mapping is shown in Table 1. The data model is thus highly coupled to the policy specification language to facilitate query processing and automated audits, as described in the following section.

8 Querying Over Encrypted Data

In order to execute efficient queries over encrypted we make the following comments about the data model. First, all attribute values are encrypted using the CP-ABE scheme discussed in Section 6. This enables log payloads to be safely and securely transferred from the log server for offline processing. The data is only at risk in the event that the attribute authority or a valid user's keys are compromised.

Second, each relation maintains pairs of attributes to store the encrypted payload of the attribute and the forward-secure hash of the payload, respectively. This is done to enable the ABLAS proxy to generate queries that can be executed on the encrypted log data. To compute these hash digests, the attribute value to be stored in the database table is salted with the ABLAS master key and a randomly generated initialization vector. More specifically, the hash digest tag $T_{i,j}$ for an attribute element D_j associated with the log event entry L_i is computed as follows:

$$T_{i,j} = H(D_i||M_k),$$

where M_k is the master key for the ABLAS system and $||$ is the concatenation operator. With a cryptographically strong, forward-secure hash function H, this derivation enables only the entity who holds the master key M_k to generate the correct hash digest and infer data from the database. In this way, should the database be compromised, the attacker would not be able to infer any information from the encrypted log entries or hash digests so long as the master key M_k is securely stored. A concrete subset of the data model containing all of the log information is shown in Section 5. The data tags are then stored in the database, resulting in log entries similar to those shown in Table 2.

While this approach does not allow for modern SQL queries such as LIKE and SUBSTRING, we argue that such queries are not needed given how log data will be used. First, common audit policies can be enforced at the level of abstract user roles (e.g. doctors, patients), system actions (e.g. add, delete, modify), and affected objects (e.g. address, test result, etc). The specific private information about such events, such as a patient's social security number or the actual test results are not required to enforce security policies. Therefore, since abstract roles constitute a finite set of elements in the XACML specification, it is enough to perform exact matches on data entry tags when querying over the encrypted data.

As an example, consider the policy that states "No one should be able to change a patient's lab results." Encoded in XACML, as shown in Figure 2, ABLAS would parse and retrieve the resource `LabResult` and action `change`. Then, using the query manipulation technique previously

```
<Policy PolicyId="SamplePolicy">
   <Target>
     <Subjects>
       <AnySubject/>
     </Subjects>
     <Resources>
       <ResourceMatch MatchId="function:string-equal">
         <AttributeValue DataType="string">LabResult</AttributeValue>
       </ResourceMatch>
     </Resources>
     <Actions>
       <ActionMatch MatchId="string-equal">
          <AttributeValue DataType="string">change</AttributeValue>
         </ActionMatch>
     </Actions>
   </Target>
</Policy>
```

Figure 2: XACML equivalent policy for specifying the action, "No one should be able to modify a patient's lab results."

Table 2: Subset of the Event relation as stored in the ABLAS database. Hex values correspond to encrypted entries and hash digests, respectively.

Event	User	Session	Log	X_i	Y_i	Action	ActionTag	Resource	ResourceTag	Salt
0	0x3a	0x89	0x31	0x97	0x12	0x81	0x12	0x9f	0x66	0x11

discussed, the ABLAS proxy would then compute $T_0 = H('LabResult'||M_k)$ and $T_1 = H('change'||M_k)$, where T_0 corresponds to the resource tag and T_1 corresponds to the action tag. Then, the ABLAS would submit the following query to the ABLAS engine for execution:

```
SELECT * FROM Event ResourceTag='T0'
          AND ActionTag='T1'
```

The resulting set of event tables would be returned to the ABLAS proxy or sent to some other auditing interface for further processing.

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 inspired by the "onion encryption" design in CryptDB [16], 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 M_k 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.

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

Using the table identifier as a salt to the master key 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.

9 ABLAS System Architecture

ABLAS is designed to be a centralized logging system backed by a set of distributed databases. A ma-

jor goal for the architecture was horizontal scalability, which is achieved through separation of the log and audit consumption and production processes. Message queues are used to collate all data to be processed by an ABLAS instance from an arbitrary number of clients, which means that the only bottleneck in the system is how fast these messages can be served by the ABLAS instance. If deployed in the cloud (e.g. Amazon AWS), ABLAS can be granted a large enough worker thread pool to optimize log and audit message processing. A graphical context diagram for the ABLAS system architecture that highlights these design elements is shown in Figure 3.

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 ABLAS instance.

10 Performance Analysis

The prototype ABLAS system was written in Python, utilizing the Charm cryptography package [2] for pairing-based cryptographic primitives. Since a major concern for the ABLAS architecture was its scalable performance while processing large amounts of log data, we conducted experiments to measure the encryption overhead from events that have uniform access policies, as well as the encryption overhead from events that have different access policies. These two experiments were conducted to gauge the overhead that results from log encryption and querying the policy engine for generating new policies. The results for these two experiments are shown in Figures 4 and 5. Based on the resulting data, it was shown that a single log message requires approximately 180ms and 190ms to encrypt if the data uses a pre-cached and newly-generated encryption key. Therefore, we conclude that the encryption procedure has a more significant impact on the ABLAS performance than the presence of the policy engine does. This is an indication that future optimizations would yield significantly better performance metrics.

To measure the storage overhead for the log data we compared the physical size of an empty SQLite database against one that contained log data for three different policies, each with different numbers of attributes. The results of this comparison are shown in Table 3. Based on this analysis, we concluded that each log entry contributed approximately 1.2Kb of information to the log database, which is largely due to the highly conservative data types in the relational model. This overhead could be drastically reduced if the model was adjusted to match the cryptographic primitives used in the logging scheme and the use of BLOB attribute types was removed.

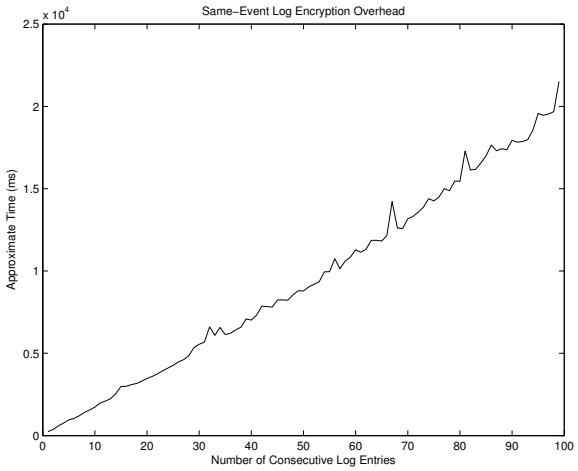

Figure 4: Log encryption performance for data in a single user session that all have the same access policy.

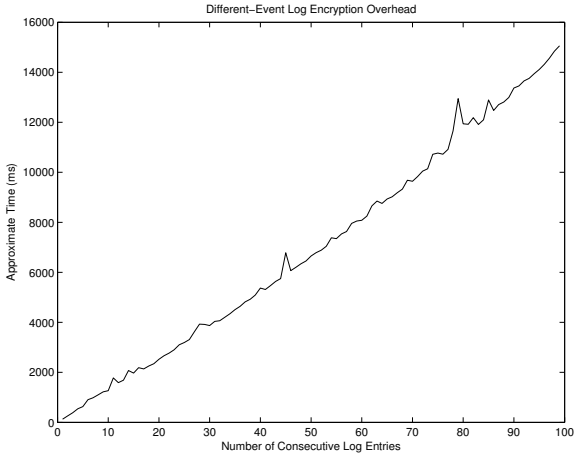

Figure 5: Log encryption performance for data in a single user session that all have different access policies.

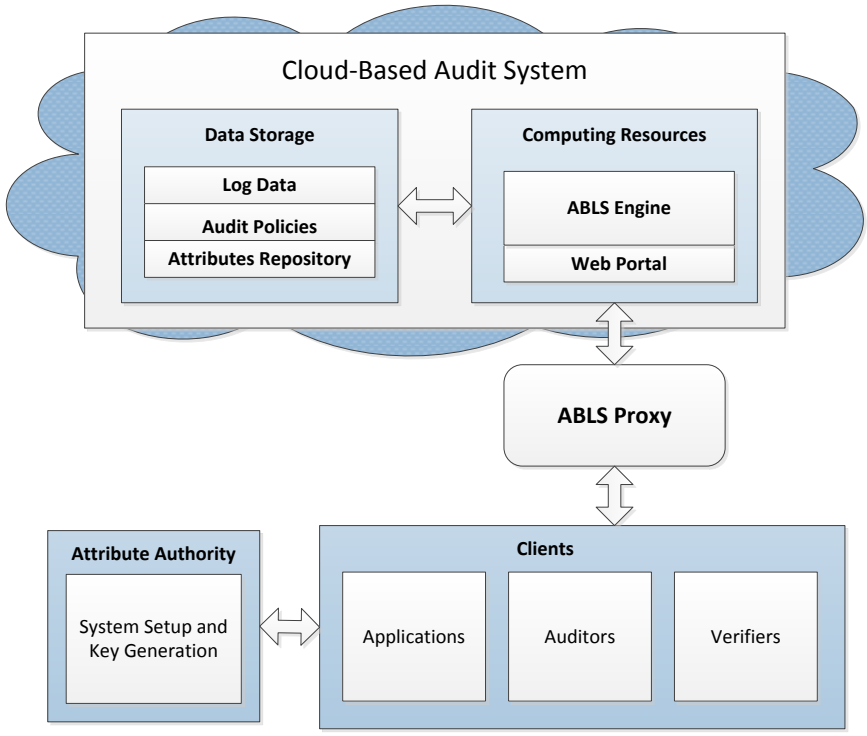

Figure 3: The cloud deployment architecture for ABLAS. Similar in spirit to CryptDB, the secure ABLAS proxy is responsible for query manipulation to support querying over encrypted data.

Table 3: Storage overhead for storing the log data

Database Contents	*Size (Kb)*
None	12
100 log entries - minimal attributes	134
100 log entries - medium attributes	147
100 log entries - maximum attributes	149

11 Future Research

While the current ABLAS design provides efficient automated audits to help support non-repudiation, there are several fundamental design changes that can be explored for further improvement. One such change is the application of publicly-verifiable FssAgg signature generation algorithms to generate entity tags for log chains. This would reduce the storage overhead for log information by ultimately removing the need to use hash chains altogether and reduce the signature generation (aggregation) time without sacrificing the complexity of signature verification. However, these schemes require a trusted CA for public verification, which may or may not be available in the environments where ABLAS would be deployed.

Another avenue for future research is the removal of the online server dependence for log verification and temporal (time-sensitive) log decryption. The author does not know how this design problem would be solved, but it is worth investigating in the future.

12 Conclusion

In this paper we presented ABLAS, a secure logging system for the cloud that supports automated audits and attribute-based log file encryption. We discussed the log integrity and confidentiality properties supported by ABLAS, and how they map to the underlying system architecture and relational data model. We then presented the current architecture and deployment scheme for an ABLAS instance in the cloud, which included the client interfaces for submitting log data and executing audit queries. We finished with a performance analysis of the current ABLAS prototype and discussion of employing

ABLAS in the healthcare domain.

References

[1] ABRAMS, M., EGGERS, K., LAPADULA, L., AND OLSON, I. A generalized framework for access control: An informal description. In *Proceedings of the 13th National Computer Security Conference* (1990), pp. 135–143.

[2] AKINYELE, J., GREEN, M., AND RUBIN, A. Charm: A framework for rapidly prototyping cryptosystems.

[3] ALIPOUR, H., SABBARI, M., AND NAZEMI, E. A policy based access control model for web services. In *Internet Technology and Secured Transactions (ICITST), 2011 International Conference for* (2011), IEEE, pp. 472–477.

[4] ANDERSON, A. A comparison of two privacy policy languages: Epal and xacml.

[5] ANNAS, G. Hipaa regulationsa new era of medical-record privacy? *New England Journal of Medicine 348*, 15 (2003), 1486–1490.

[6] BELLARE, M., AND YEE, B. S. Forward integrity for secure audit logs. Tech. rep., 1997.

[7] BETHENCOURT, J., SAHAI, A., AND WATERS, B. Ciphertext-policy attribute-based encryption. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2007), SP '07, IEEE Computer Society, pp. 321–334.

[8] COMMITTEE, P. I. T. A., ET AL. Revolutionizing health care through information technology. *Arlington, VA: National Coordination Office for Information Technology Research and Development* (2004).

[9] DAVID, F., AND RICHARD, K. Role-based access controls. In *Proceedings of 15th NIST-NCSC National Computer Security Conference* (1992), vol. 563, Baltimore, Maryland: NIST-NCSC.

[10] FERRAIOLO, D., SANDHU, R., GAVRILA, S., KUHN, D., AND CHANDRAMOULI, R. Proposed nist standard for role-based access control. *ACM Transactions on Information and System Security (TISSEC) 4*, 3 (2001), 224–274.

[11] GOYAL, V., PANDEY, O., SAHAI, A., AND WATERS, B. Attribute-based encryption for fine-grained access control of encrypted data. In *Proceedings of the 13th ACM conference on Computer and communications security* (2006), ACM, pp. 89–98.

[12] JUNOD, P., AND KARLOV, A. An efficient public-key attribute-based broadcast encryption scheme allowing arbitrary access policies. In *Proceedings of the tenth annual ACM workshop on Digital rights management* (New York, NY, USA, 2010), DRM '10, ACM, pp. 13–24.

[13] KING, J., SMITH, B., AND WILLIAMS, L. Modifying without a trace: General audit guidelines are inadequate for electronic health record audit mechanisms. *Proceedings of ACM IHI 2012* (2012).

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

[15] MA, D. Practical forward secure sequential aggregate signatures. In *Proceedings of the 2008 ACM symposium on Information, computer and communications security (ASIACCS08)* (2008), pp. 341–352.

[16] POPA, R. A., REDFIELD, C. M. S., ZELDOVICH, N., AND BALAKRISHNAN, H. Cryptdb: processing queries on an encrypted database. *Commun. ACM 55*, 9 (Sept. 2012), 103–111.

[17] SANDHU, R., COYNE, E., FEINSTEIN, H., AND YOUMAN, C. Role-based access control models. *Computer 29*, 2 (1996), 38–47.

[18] SANDHU, R., FERRAIOLO, D., AND KUHN, R. The nist model for role-based access control: towards a unified standard. In *Symposium on Access Control Models and Technologies: Proceedings of the fifth ACM workshop on Role-based access control* (2000), vol. 26, pp. 47–63.

[19] SCHNEIER, B., AND KELSEY, J. Secure audit logs to support computer forensics. *ACM Transactions on Information and System Security (TISSEC) 2*, 2 (1999), 159–176.

[20] SHEN, H., AND HONG, F. An attribute-based access control model for web services. In *Parallel and Distributed Computing, Applications and Technologies, 2006. PDCAT'06. Seventh International Conference on* (2006), IEEE, pp. 74–79.

[21] YAVUZ, A. A., AND NING, P. Baf: An efficient publicly verifiable secure audit logging scheme for distributed systems. In *Proceedings of the 2009 Annual Computer Security Applications Conference* (Washington, DC, USA, 2009), ACSAC '09, IEEE Computer Society, pp. 219–228.

[22] ZHU, J., AND SMARI, W. Attribute based access control and security for collaboration environments. In *Aerospace and Electronics Conference, 2008. NAECON 2008. IEEE National* (2008), IEEE, pp. 31–35.