Streamforce: Outsourcing Access Control Enforcement for Stream Data to the Clouds

Dinh Tien Tuan Anh  
School of Computing  
National University of Singapore  
13 Computing Drive, Singapore  
dinhhta@comp.nus.edu.sg

Anwitaman Datta  
School of Computer Engineering  
Nanyang Technological University  
50 Nanyang Avenue, Singapore  
anwitaman@ntu.edu.sg

ABSTRACT

In this paper, we focus on the problem of data privacy on the cloud, particularly on access controls over stream data. The nature of stream data and the complexity of sharing data make access control a more challenging issue than in traditional archival databases. We present Streamforce — a system allowing data owners to securely outsource their data to an untrusted (curious-but-honest) cloud. The owner specifies fine-grained policies which are enforced by the cloud. The latter performs most of the heavy computations, while learning nothing about the data content. To this end, we employ a number of encryption schemes, including deterministic encryption, proxy-based attribute based encryption and sliding-window encryption. In Streamforce, access control policies are modeled as secure continuous queries, which entails minimal changes to existing stream processing engines, and allows for easy expression of a wide-range of policies. In particular, Streamforce comes with a number of secure query operators including Map, Filter, Join and Aggregate. Finally, we implement Streamforce over an open-source stream processing engine (Esper) and evaluate its performance on a cloud platform. The results demonstrate practical performance for many real-world applications, and although the security overhead is visible, Streamforce is highly scalable.

Categories and Subject Descriptors

H.2.0 [Database Management]: General — Security, integrity and protection

Keywords

access control, stream processing, outsourced databases, cloud computing

1. INTRODUCTION

An enormous amount of data is being generated everyday, and it has become increasingly common to process data as they arrive in continuous streams. Examples range from high-frequency streams such as generated from stock or network monitoring applications, to low-frequency streams originated from weather monitoring, social network or health monitoring applications. The variety and abundance of data, combined with the potential of social interactivity, mash-up services and data sciences, has brought data sharing into the foreground. A crucial problem with sharing data is security concerning the question of who gets access to which aspects of the data (fine-grained access control) and under which context (data privacy). This paper studies the former question, which we believe to be more challenging for stream data than for archival data because of three reasons. First, traditional archival data systems enforce access control by pre-computing views, which is not possible with stream data because of its infinite size. Second, access control over stream is inherently data-driven (triggered by arrival of specific data values) as opposed to user-driven, and it often involves temporal constraints (sliding windows). Third, many of the sharing activities take place in collaborative settings with large numbers of users and even larger numbers of policies.

At the same time, more businesses and individual users are leveraging the cloud for its instantly available and virtually unbounded computing resources provided by various vendors at competitive prices. Many enterprise systems are migrating their infrastructure to the cloud, while a plethora of small-to-medium size systems are being developed and deployed on the cloud regularly. In the context of stream data sharing, cloud computing emerges as an ideal platform for two reasons. First, data can be hosted and managed by a small number of cloud providers with unlimited resources, which is important since data streams are of infinite sizes. Second, data co-location makes it easy to share and to perform analytics. On the other hand, the cloud is a potential adversary since it may obtain unauthorized access to the data, or inadvertently leak data due to external attacks or software vulnerability. Thus, when considering the cloud for sharing data, it becomes imperative to design a mechanism which facilitates sharing while guaranteeing data confidentiality even from the cloud.

In this paper, we consider data streams being outsourced to untrusted (curious-but-honest) clouds. The challenge is then to protect data confidentiality from the cloud, while at the the same time leveraging the latter for fine-grained access control. We present Streamforce — a fine-grained access control system for stream data over untrusted clouds.
Streamforce is designed with three goals. First, it supports specification and enforcement of fine-grained access control policies. Second, data is outsourced to the cloud where access control policies are enforced, with the latter learning nothing about the data content. Third, the system is efficient, in the sense that the cloud handles most of the expensive computations. The last two goals require the cloud to be more active than being merely a storage and transit facility. To realize these goals, Streamforce uses a number of encryption schemes: deterministic encryption, proxy-based attribute based encryption, and a sliding-window based encryption. While encryption is necessary to protect data confidentiality against the cloud and against unauthorized access, we believe that directly exposing encryption details to the system entities (data owner, user and cloud) is not the ideal abstraction when it comes to access control. Instead, Streamforce models and enforces access control policies using secure query operators: secure Map, Filter, Join and Aggregate. These operators are at higher level and more human-friendly than raw encryption keys. Since existing stream processing engines are very efficient at executing continuous queries made from similar query operators, they can be leveraged by the cloud without major changes.

Streamforce occupies an unique position in the design space of outsourced access control. It considers untrusted (semi-honest) clouds, which is different to [4]. Systems such as Plutus [13] and CryptDb [17] assume untrusted clouds, but they support only coarse-grained policies over archival data. Recent systems utilizing attribute-based encryption [11, 19] achieve more fine-grained access control on untrusted clouds, but they do not support stream data. Furthermore, the cloud is not fully utilized as it is used mainly for storage and distribution. To the best of our knowledge, Streamforce is the first system that allows secure, efficient outsourcing of fine-grained access control for stream data to untrusted clouds. It is not catered for applications demanding high throughput, but it presents important first steps towards supporting them. Our contributions are summarized as follows:

- We present a system and formal security model for outsourcing access control of stream data to untrusted clouds. We discuss different security levels that query operators can achieve.
- We present details and analyze security properties of different encryption schemes used for enforcing fine-grained access control, including a new scheme supporting sliding window aggregation.
- We show how to combine these encryption schemes to construct secure query operators: secure Map, secure Filter, secure Join and secure Aggregate.
- We implement a prototype of Streamforce over Esper and benchmark it on Amazon EC2. The results indicate practical performance for many applications. Although the cost of security is evident, we show that it can be compensated by the system’s high scalability.

Next we present the system and security model, followed by the constructions of the encryption schemes. We then describe how to construct secure query operators. Prototype implementation and evaluation is presented in Section 5. Related work follows in Section 6, before we draw conclusion and discuss future work.

2. SYSTEM AND SECURITY MODEL

2.1 System Model

2.1.1 Overview.

There are three types of entities: data owners (or owners), data users (or users) and a cloud. Their interactions are illustrated in Fig. 1[a]: the owners encrypt their data and relay them to the cloud, which performs transformation and forwards the results to the users for final decryption. We do not consider how the owner determines access control policies, and we assume that the negotiation process (in which the owner grants policies to the user) happens out-of-band. The system goals are three-folds:

1. The owner is able to express flexible, fine-grained access control policies.
2. The system ensures data confidentiality against untrusted cloud, and access control against unauthorized users (as elaborated later).
3. Access control enforcement is done by the cloud. Decryptions at the user are light-weight operations compared to the transformations at the cloud.

2.1.2 Data Model.

A data stream $S$ has the following schema:

$$S = (TS, A_1, A_2, ..., A_n)$$

where $TS = \mathbb{N}$ is the timestamp, and all data attributes $A_i$ are of integer domains. A data tuple at time $ts$ is written as $d_{ts} = (ts, v_{A_1},..., v_{A_n})$. Queries over data streams are continuous, i.e. they are triggered when new data arrives. Each query is composed from one or more query operators, which take one or more streams as inputs and output another stream. We focus on four operators: Map, Filter, Join and Aggregate.

- Map: outputs only the specified attributes.
Filter: outputs tuples satisfying a given predicate.

Join: takes as inputs two streams \((S_1, S_2)\), two integers \((w_{s1}, w_{s2})\) and a join attribute. Incoming data are added to the queues of size \(w_{s1}\) and \(w_{s2}\), from which they are joined together.

Aggregate: outputs the averages over a sliding window. A sliding window is defined over the timestamp attribute, with a window size \(w_s\) and an advance step \(\Delta\).

Note that the assumption of the data attributes being in the integer domain does not exclude data types such as float or string, since they can be converted to integer representation. Although our system falls short of supporting arbitrary type (some query operators may not make sense over some data types), the current data model nevertheless can cover a wide range of real-life applications.

2.1.3 Access Control via Queries.

As in traditional (relational) databases, access control in Streamforce is defined via views that are created by querying the database. This is facilitated by the use of the abstract operators, instead of exposing cumbersome encryption details to the system entities. Specifically, the access control process involves two steps. First, the owner specifies a policy by mapping it into a continuous query. Second, the query is registered to be executed by the cloud, whose outputs are proactively tried to access unauthorized data. To this end, users are considered dishonest, in the sense that they may intentionally try to access stale data. Security against such attacks is crucial for many applications, but it is out of the scope of this paper. Data integrity, launch denial of service attacks, or compute using incorrect inputs are the number of calories burned, the activity and the location from health monitoring devices, where

\[
\begin{align*}
S_{vs} &= (TS, RTime, Name, HR, BP) \\
S_{fn} &= (TS, RTime, Name, Cal, Act, Loc)
\end{align*}
\]

\(S_{vs}\) contains owner’s vital signs as produced by health monitoring devices, where \(RTime, HR, BP\) are the real time, heart rate and blood pressure respectively. \(S_{fn}\) contains fitness information, where \(Cals, Act, Loc\) are the number of calories burned, the activity and the owner’s location respectively. Data users could be friends from social network, research institutes or insurance companies. For a friend, the owner may want to share vitals data when they exceed a certain threshold \((Q_0)\), or average fitness information every hour \((Q_1)\). A research institute may be given a joined view from both streams in order to monitor the individual’s vitals during exercises \((Q_1)\).

2.2 Security Model

2.2.1 Adversary Model.

Streamforce is designed assuming that the cloud is not fully trusted, in that it may try to learn content of the outsourced data, however it still follows the protocol correctly. This curious-but-honest (semi-honest) adversary model is standard in the literature, since it reflects the cloud’s incentives to gain benefits from user data while being bound by the service level agreements and market forces. We do not consider malicious cloud, which may try to break data integrity, launch denial of service attacks, or compute using stale data. Security against such attacks is crucial for many applications, but it is out of the scope of this paper. Data users are considered dishonest, in the sense that they may proactively try to access unauthorized data. To this end, they may collude with each other and also with the cloud.

To meet both fine-grained access control and data confidentiality requirements, we use three different encryption schemes. Proxy attribute based encryption is used for Map and Filter operators. Second, Join operator is realized via deterministic encryption. Aggregate is supported by sliding-window encryption. This section provides formal definition of these schemes and their security properties. Detailed constructions and proofs of security are presented in Section 3.

Deterministic encryption scheme.

\(E_i = (Gen, Enc, Dec)\) is a private-key encryption scheme, where \(Gen(\kappa)\) generates secret key \(SK\) using security parameter \(\kappa\), \(Enc(m, SK)\) encrypts message \(m\) with \(SK\), and \(Dec(CT, SK)\) decrypts the ciphertext. Security of \(E_s\) (Det-CPA) is defined via the security game detailed in Appendix A.

Proxy Attribute-Based Encryption scheme.

Attribute-Based Encryption (ABE) is a public-key scheme that allows for fine-grained access control: ciphertexts can only be decrypted if the security credentials satisfy a certain predicate. Two types of ABE [11] exist: Key-Policy (KP-ABE) and CipherPolicy (CP-ABE). We opt for the former, in which the predicate is embedded in user keys and the ciphertext contains a set of encryption attributes. KP-ABE and CP-ABE can be used interchangeably, but the former is more data-centric (who gets access to the given data), while the latter is more user-centric (which data the given user has access to).

ABE’s encryption and decryption are expensive operations. Proxy Attribute Based Encryption [12] (or proxy ABE) is design to aid the decryption process by letting a third party transform the original ABE ciphertexts into a simpler form. It consists of five algorithms \(E_p = (Gen, KeyGen, Enc, Trans, Dec)\). \(Gen(\kappa)\) generates public parameters \(PK\) and master key \(MK\). \(KeyGen(MK, P)\) creates a transformation key \(TK\) and a decryption key \(SK\) for the predicate \(P\). \(Enc(m, PK, A)\) encrypts \(m\) with the set of encryption attributes \(A\). \(Trans(TK, CT)\) partially decryots the ciphertext using \(TK\). Finally, \(Dec(SK, CT)\) decrypts the transformed ciphertext using the decryption key.

Security of \(E_p\) is defined in [12] via a game in the selective-set model. The highest level of security achievable is RCCA, when the adversary is allowed to query the decryption oracles. In this paper, however, we use a lower level of security: CPA security in the selective model, achieved when the adversary has no access to the decryption oracle.

Sliding-window encryption scheme (SWE).

The scheme consists of three algorithms: \(E_w = (Gen, Enc, Dec)\), which allows an user to decrypt only the aggregate of a window of ciphertexts, and not the individual ciphertexts. Let \(s(M, ws)[i]\) and \(p(M, ws)[i]\) be the sum and product of the \(i^{th}\) window sliding windows (size \(ws\) and advance step \(\Delta\)) over a sequence \(M\). \(Gen(\kappa)\) generates the public parameters and the private keys. \(Enc(M = \langle m_0, m_1, ..., m_{n-1}, W\rangle)\) encrypts \(M\) using a set of window sizes \(w_s\), whose result is \(CT = \langle c_0, c_1, c_2, ..\rangle\). \(Dec(ws, CT, SK_{ws})\) decrypts \(CT\) for the window size \(ws\) using the private key \(SK_{ws}\). The result is the aggregates of the sliding window, i.e. \(s(M, ws)[i]\) for all \(i\).

Security of \(E_w\) is defined via a selective-window security game consisting of four phases: Setup, Corrupt, Challenge, Guess.
...but our model requires only the windows (sub-sequences) of information must be revealed to the latter. There exists a trade-off between security and functionality of the query operators.

Res-CEW guarantees access control under weak collusion (except from what can be derived from its own window). They cannot learn the plaintexts. In addition, Res-CEW the unauthorized user have access to decryption keys, hence control against unauthorized users. Neither of cloud nor confidentiality against untrusted cloud, and providing access capabilities. CPA security allows for meaningful computations (aggregate) over ciphertexts. Res-CEW is secure against a weak form of security guarantee, as it protects data confidentiality only

Security of the sliding-window scheme is related to that of secure multi-party computation, which ensures that no other information is leaked during the computation of a function except from the final output. Our model is similar, but stronger than the aggregator oblivious model proposed in [18], since the security game allows for more types of adversarial attacks. More specifically, [18] requires the two message sequences M₀ and M₁ to have the same aggregate, but our model requires only the windows (sub-sequences) of M₀ and M₁ to have the same aggregate. Both Res-CEW and CW security allow for meaningful computations (aggregate) over ciphertexts. Res-CEW is secure against a weak form of collusion (between users with access to window sizes which are multiples of each others), whereas CW is not.

2.2.4 Access control via Encryption.

Encryption plays two roles in our system: protecting data confidentiality against untrusted cloud, and providing access control against unauthorized users. Neither of cloud nor the unauthorized user have access to decryption keys, hence they cannot learn the plaintexts. In addition, Res-CEW and CW security ensure that given access to a window size ws, the user cannot learn information of other window sizes (except from what can be derived from its own window). Res-CEW guarantees access control under weak collusion among dishonest users.

For access control to be enforced by the cloud, some information must be revealed to the latter. There exists a trade-off between security and functionality of the query operators that make up the policies. For Map and Filter policies, the cloud must be able to check if certain attributes are included in the ciphertexts, which is allowed by CPA security. For Join, the cloud needs to be able to compare if two ciphertexts are encryptions of the same message, which requires the encryption to be deterministic (or Det-CEA secure). For Aggregate, a homomorphic encryption is required, which in our case means the highest security level is Res-CEW.

3. ENCRYPTION SCHEMES

This section details the constructions of three encryption schemes defined in the previous section. Except for the proxy ABE construction which we take directly from the original work, the other encryption schemes are specifically designed for Streamforce. Detailed security proofs are omitted due to space constraint, and will be provided as an accompanying extended technical report if/when this paper is published.

3.1 Deterministic Encryption

Let \( G \) be a multiplicative group of prime order \( p \) and generator \( g \). Let \( F : \mathbb{Z}_p \times \{0,1\}^* \to G \) be a pseudorandom permutation with outputs in \( G \). The scheme \( \mathcal{E}_d \) is constructed as follows. \( \text{Gen}(k) \to \mathcal{E}_d = (k_1, k_2) \in \mathbb{Z}_p^\times \times \mathbb{Z}_p \). \( \text{Enc}(m, SK) \to F(k_1, m)^{k_2} \). Finally, \( \text{Dec}(CT, SK) \to F^{-1}(k_1, CT^{\frac{k_2}{k_1}}) \).

Assuming that \( F \) is a pseudorandom permutation, \( \mathcal{E}_d \) is Det-CEA secure.

3.2 Proxy ABE Construction

We use the CPA-secure construction as presented in [12]. Recall that \( \mathcal{E}_p \) can actually achieve higher security level (R-CCA), but our system concerns confidentiality only, hence the CPA-secure construction suffices.

3.3 Sliding-Window Encryption

We propose three alternative constructions of the sliding-window encryption scheme: \( \mathcal{E}^1_w, \mathcal{E}^2_w, \mathcal{E}^3_w \).

The naive construction, \( \mathcal{E}^1_w \), masks the plaintext with random values whose sum over the sliding window is the user key. Encrypting a plaintext value produces \( W \) ciphertexts, where \( W \) is the set of all possible window sizes. The second construction, \( \mathcal{E}^2_w \), employs an auxiliary scheme \( \mathcal{E}_aux \) to encrypt the window aggregates directly. In particular, at timestamp \( ts \), data owner computes and encrypts the aggregates of all windows ending at \( ts \). Finally, \( \mathcal{E}^3_w \) masks the plaintext with random values whose sum over the sliding window is encrypted with another scheme \( \mathcal{E}_aux \). It can be considered as a generalized version of \( \mathcal{E}^3_w \), but it places no restriction over the random values. As shown in Appendix C (Agg-3), one can select \( \mathcal{E}_aux \) scheme to minimize space overhead. In the following, we present details of \( \mathcal{E}^2_w \) (other constructions are described in Appendix C). We compare properties of \( \mathcal{E}^1_w, \mathcal{E}^2_w, \mathcal{E}^3_w \) later in Section 4.4.

Let \( W \) be the set of all possible window sizes, \( \mathcal{E}_aux = (\text{Gen}, \text{Enc}, \text{Dec}) \) be a CPA-secure public encryption scheme. \( \mathcal{E}_w \) is implemented as follows:

- \( \text{Gen}(k) \): let \( \mathcal{E}_aux = (\text{Gen}, \text{Enc}, \text{Dec}) \) be a CPA-secure asymmetric encryption scheme. For all \( ws \in W \), invokes \( \mathcal{E}_aux, \text{Gen}(k) \) to generate a key pair \( (PK_{aux}, SK_{aux}) \).
- \( \text{Enc}(M, W) \) : \( CT = \bigcup_{ws \in W} \mathcal{E}_aux, \text{Enc}(PK_{aux}, s(M, ws)[i]) \).
MK returns data tuples containing attributes in a set 
(3) how the transformation at the cloud is done. We also describe the 
provide the underlying security assurance for Streamforce.

4. SECURE QUERY OPERATORS
The encryption schemes discussed in previous sections 
formed using a combination of proxy ABE scheme di-
user decryption key is generated by 
Es.KeyGen(MK, P-Filter([(A, k, op)]), in which 
P-Filter([(A, k, op)]) = \bigcup_{A \in FA} D(A, k, op)

When the ciphertext CT arrives at the cloud, the latter transforms it using Es.Trans(TK, CT) and forwards the result to the user.

Similar to Map, this operator uses proxy ABE scheme directly, 
the size of \( \mathbb{P} \). The bigger the size of \( \mathbb{P} \), the more policies of the type ‘mod p’ can be supported, but at the expense of more storage overhead. Notice that values of filtering attributes are exposed to the cloud in the form of encryption attributes, thus the data owner should only use non-sensitive attributes, such as TS, for the set FA.

4.3 Join
Let J be the join attributes of two streams S1, S2. We assume that the join operator returns all data attributes (more complex cases are discussed in Section 4.5). We use a combination of proxy ABE scheme Es and deterministic scheme Ed. Initially, the two owners of S1, S2 invoke Ed.Gen(\cdot) in a way that satisfies two conditions: (1) both end up with the same group G and pseudorandom function F; (2) SK1 = (k1,1, k1,2) and SK2 = (k2,1, k2,2) are the two secret keys such that k1,1 = k2,1.

The user decryption for stream i is 
(ki,2, Ei.KeyGen(MK, P-Join(J))), in which 
P-Join(J) = ‘att = J’

The owner encrypts using: 
Enc-Join(dts, J) = (U, V) = \left( Es.Enc(d, ‘att = J’), Ed.Enc(v_J) \right)

The user who received both k1,2 and k2,2 computes (z1 = \frac{r_{1,1}}{r_{1,2}}, z2 = \frac{r_{2,1}}{r_{2,2}}) where s \in \mathbb{Z}_{p} and sends it to the cloud. When two ciphertexts (U1, V1) and (U2, V2) arrive at the cloud, it checks if V1^z1 = V2^z2. If true, the ciphertexts can be joined. The cloud then performs Es.Trans(TK1, U1), Ed.Trans(TK2, U2) and forwards the results to the user.

Because Ed is Det-CPA secure, the cloud can learn if the encryption of v_J is the same as in both streams, but only if given the values z1 and z2. In other words, the cloud cannot perform joining unless requested by the user. Other attributes in dts are protected with CPA security by Es. The storage requirement is \( O(\log(p)) \) bits per data tuple, because Es.Enc(\cdot) produces a group element and Es encrypts the entire data tuple with only one encryption attribute.
4.4 Aggregate (Sliding Window)

In Streamforce, sliding windows are based on timestamp attribute TS, and they are non-overlapping, i.e. advance steps are the same as the window sizes. Let $A_s$ be the aggregate attribute, over which the sums are computed. We propose three implementations based on three constructions of the sliding window encryption schemes, namely $Agg-1$, $Agg-2$, $Agg-3$. In $Agg-1$, the data is first encrypted with $E_w$, then the result is encrypted with $E_p$. In $Agg-2$, window aggregates are computed and encrypted with $E_p$. In $Agg-3$, random values are added to the data which is then encrypted with $E_p$, and the sum of the random values is also encrypted with $E_p$. In this section, we describe $Agg-2$ which, as we later show, incurs low space and computation overhead in most cases.

$Agg-2$ uses $E_w^2$ with $E_p$ as the auxiliary encryption scheme. The owner itself computes the window aggregates and encrypts the result using $E_p$. User decryption key is $E_p, \text{KeyGen}(MK, P-\text{Agg2}(ws, A_p))$, where:

$$P-\text{Agg2}(ws, A_p) = \{\text{att} = A_p \cap \text{window} = ws\}$$

To encrypt $d_{ts}$, the owner first executes $\text{Enc-2}((s, d_{ts}, E_w, W) \rightarrow CT)$ as shown in Fig. 2, then the ciphertext is computed as:

$$\text{Enc-2}(d_{ts}) = \left(E_p, \text{Enc}(d_{ts}, \{\text{window} = 1\}), CT\right)$$

At the cloud, the ciphertexts for a window aggregate are of the same form as for a normal data tuple. The cloud simply invokes $E_p, \text{Trans}(CT_{ws}, TK_{ws})$ and forwards the results to the user.

Discussion.

Unlike $\text{Map}, \text{Filter}$ and $\text{Join}$, the $\text{Aggregate}$ operator requires more effort from the cloud, i.e. multiplication of ciphertexts. Since it uses $E_w$ as the final layer of encryption, it achieves CPA security with respect to the cloud. In all three implementations, the transformed ciphertexts received by the user are data encrypted with $E_w^1$, $E_w^2$ or $E_w^3$. As discussed in the previous section, these schemes achieve Res-CEW security, therefore the user learns nothing more than the aggregate values.

$Agg-1$ and $Agg-2$ support a fixed set of window sizes. $Agg-3$ is more flexible: it supports all window sizes whose prime factors are in $P$, and it allows for arbitrary starting positions of the sliding windows. The storage cost of $Agg-1$ and $Agg-2$ is $O(|W| \cdot \log(p))$ bits per data tuple, but it is for the average case with $Agg-1$, and for the worst-case for $Agg-2$. For $Agg-3$, the average cost is $O(|P| \cdot \log(p))$. There is a trade-off between flexibility and storage overhead. When the owner wishes to support a small number of windows, $Agg-2$ is a better choice among the three. However, when more flexible windows are required, $Agg-3$ may have a better trade-off between flexibility and storage cost. Our experimentation with Streamforce in Section 5 suggests that this is indeed the case.

4.5 Combining Multiple Operators

Map and Filter.

The user decryption key is generated by combining the $\text{Map}$ and $\text{Filter}$ key, i.e. $E_p, \text{Gen}(MK, P-\text{Map}(B), P-\text{Filter}([A, k, op]))$. The owner encrypts using:

$$\text{MF-Enc}(d_{ts}) = \left((s, \{v \in A \in FA\}, E_p, \text{Enc}(v_{A_1}, A_1^t), E_p, \text{Enc}(v_{A_2}, A_2^t), \ldots)\right)$$

where $A_i^t = \{\text{att} = A_i^t\} \cup \bigcup_{A \in FA} AS(A, v_a)$. This operator is CPA secure, and the storage cost is $O(|A| \cdot |FA| \cdot |P| \cdot \log(p))$ bits per data tuple.

Map, Filter and Join.

This operator allows the cloud to join two encrypted streams only when filter conditions on each stream are met. The user decryption key is made up of the $\text{Map-Filter}$ key and the $\text{Join}$ key, i.e.:

$$\left(k_{i, 2}, E_p, \text{KeyGen}(MK, P-\text{Join}(J)), E_p, \text{Gen}(MK, P-\text{Map}(B), P-\text{Filter}([A, k, op]))\right)$$

This operator is Det-CPA secure, and its storage cost is dominated by the cost of $\text{MF-Enc}$, which is $O(|A| \cdot |FA| \cdot |P| \cdot \log(p))$ bits per data tuple.

Filter and Aggregate.

We assume that each sliding window contains only continuous elements, i.e. $\{d_{ts}, d_{ts+1}, \ldots, d_{ts+ws-1}\}$. Therefore, combining $\text{Filter}$ and $\text{Aggregate}$ only applies to filtering conditions of the form $(TS, k, \geq \cdot)$ for $k \in \mathbb{Z}$. Using $Agg-2$, the user decryption key is:

$$\left(E_p, \text{KeyGen}(MK, P-\text{Agg2}(ws, A_p)), E_p, \text{KeyGen}(MK, P-\text{Filter}([TS, k, \geq \cdot]))\right)$$

To encrypt $d_{ts}$, the owner simply invokes $\text{Enc-2}(d_{ts})$. Thus, this operator has the same security level as that of $Agg-2$: CPA security against the cloud and Res-CEW security against users. The memory cost is $O(|W| \cdot \log(p))$ bits per data tuple.

4.6 Limitation

Streamforce does not support arbitrary combination of operators. For example, if a filter operator does not return continuous elements, it is not possible to link it with an aggregate operator. Another limitation is that the set of filter $FA$ and join attributes $J$ is pre-defined, hence filtering on attributes not in $FA$ or joining on an attribute $J' \neq J$ is not allowed.

5. PROTOTYPE AND EVALUATION
### 5.1 Implementation and Benchmark

We implement a prototype of Streamforce over Esper\(^1\) — an open source stream processing engine capable of processing millions of data items per second. One can register a continuous query to Esper, then implement a listener that processes the output stream. In Streamforce, policies are translated into queries (Table 1), and transformations for each policy are done at the corresponding listener. We leverage Esper to manage policies and data flow between operators, as well as to handle the complex join operation. We use OpenSSL’s AES implementation for deterministic encryption scheme, while proxy ABE and sliding window schemes are implemented by extending the KP-ABE library [1].

We create a benchmark containing stock market data of the scheme:

\[
\text{StockEvent} = (TS, \text{hour}, \text{stockId}, \text{price}, \text{volume})
\]

in which \(\text{hour}\) values are in \([0, 24]\) while \(\text{price}, \text{volume}\) values are in \([0, 100]\). Each stream is identified by its \(\text{stockId}\). The benchmark data contains 1 million encrypted data tuples belonging to 100 streams\(^2\). We generate different types of policies, as listed in Table 1 which also shows how the policies are translated into Esper queries. Notice that when \(\text{Agg-1}\) or \(\text{Agg-3}\) implementation is used, the query involves Esper’s window operator because we rely on Esper to maintain the window’s buffer. In contrast, \(\text{Agg-2}\) requires no window since the cloud only transforms individual ciphertexts. Join policies use Filter-Join operators (the Filter conditions are similar to those of \(\text{T2}, \text{T3}, \text{T4}\) policies), and involves two steps: the first transforms the input stream into \(\text{StockJoinEvent}\) stream containing the deterministic encryption of the join attribute, the second takes two \(\text{StockJoinEvent}\) streams and produces join outputs.

We first benchmark individual cost of various operations at the owner, the cloud and the user by measuring their execution time. Next, we evaluate system performance in terms of throughput and latency. Throughput is quantified by the number of unique data tuples processed by the system per second. For join policies, however, it is measured as the number of join outputs processed per second. Latency is determined from the time a data tuple enters Streamforce to the time it is sent to the user. This metric includes both queuing time and transformation time. Our experiments were carried out on Amazon’s EC2 instances, with 8 window sizes (\([2, 4, 8, \ldots, 256]\)) and maximum of 100 policies (mixture of all different types) per stream.

<table>
<thead>
<tr>
<th>Policy</th>
<th>Description</th>
<th>Esper query</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>select certain stock</td>
<td>select * from StockEvent(stockId=x)</td>
</tr>
<tr>
<td>T2</td>
<td>stock within timestamp range</td>
<td>select * from StockEvent(stockId=x, y &lt; ts &lt; z)</td>
</tr>
<tr>
<td>T3</td>
<td>stock within time interval</td>
<td>select * from StockEvent(stockId=x, y &lt; hour &lt; z)</td>
</tr>
<tr>
<td>T4</td>
<td>stock every fixed interval</td>
<td>select * from StockEvent(stockId=x, ts(y) = y)</td>
</tr>
<tr>
<td>T5</td>
<td>aggregate (Agg-1)</td>
<td>select price('ws=l'), volume('ws=l') from StockEvent(stockId=x).win-length_batch(y)</td>
</tr>
<tr>
<td>T6</td>
<td>join price</td>
<td>select * from StockEvent(stockId=x, y &lt; ts &lt; z) //output StockJoinEvent stream select * from StockJoinEvent(policyId=p).win-length(y) as s1, StockJoinEvent(policyId=p).win-length(l2) as s2 where s1.price('det') = s2.price('det')</td>
</tr>
</tbody>
</table>

Table 1: Access control policies

![Figure 3](image-url) Transformation for aggregate policies

We start with a simple workload consisting of one stream and one \(\text{T1}\) policy. We run the workload on different types of EC2 instances with different capacity, including (from small to large): \(m1.large\), \(m2.large\) and \(m3.large\). We vary the data rate, and observe the system performance at saturation point. \(m3.large\) achieves the best performance, with throughput of 249 (tuples/sec) and latency of 367ms (at 99th percentile). In contrast, \(m1.large\) and \(m2.large\) have lower throughputs at 125 and 160 (tuples/sec), and higher latency at 781ms and 628ms. The results presented below are from experiments running on \(m3.large\) instances. When the system first starts, the owners and users have to initialize the cryptographic sub-systems (running \(\text{Gen}(\cdot)\), among other things). This one-off cost consists of a constant cost for pre-computing discrete logarithms, and a variable cost depending on the number of encryption attributes. Even with 1024 encryption attributes, this initialization process takes less than 3.5s.

Fig. 3[a] shows the cost of encryption per data tuple at the owner. If the owner does not allow for aggregate policies, it is relatively constant at approximately 0.5s. The cost for supporting \(\text{Agg-1}\) is the largest (over 4s), since the owner has to encrypt the data multiple times (one for each window size). \(\text{Agg-3}\) is also more expensive, since two extra columns are encrypted for each tuple. The cost of \(\text{Agg-2}\) stays low for most of the time (its maximum value is still as high as that of \(\text{Agg-1}\)). This agrees with our analysis in Section 4, i.e. most of the time the owner incurs no extra encryption per tuple, but in the worst case it has to do 8 encryptions per tuple. Fig. 3[b] compares the transformation costs at the cloud for different implementations of the aggregate operator. It can be seen that for \(\text{Agg-2}\) the cost is constant, whereas for others it is linear with the size of the window. It is because for \(\text{Agg-1}\) and \(\text{Agg-3}\), the cloud needs to transform many ciphertexts and multiply them to get the average.
The cost to generate and initialize different types of policies are depicted in Fig. 4[a]. Generating a new policy at the owner involves creating new transformation and decryption key for the corresponding predicate, which varies with the policy complexity. T2 policies, for example, contain many bag-of-bit attributes that make up complex predicates, and therefore they take longer. The cost of initializing policies at the cloud depends on key sizes, hence it is roughly the same for all types of policies, except for Join (which involves 2 keys from the two input streams). Fig. 4[b] shows the transformation cost at the cloud versus decryption cost at the user, in which the former is an order of magnitude bigger. This illustrates that heavy computations are being outsourced to the cloud. The highest throughput is for T1 policies, at 250 (tuples/sec). We remark that many current stream applications, such as fitness and weather monitoring, have very low data arrival rate (in order of minutes), therefore our throughput can sufficiently accommodate many streams at the same time.

Fig. 5 illustrates system throughputs for more complex workloads consisting of multiple policies, multiple streams and join policies. We create mixed workloads containing different types of policies. Fig. 5[a] shows that increasing the number of policies decreases the throughput, which is heavily influenced by the number of T2 policies (the workload of 2 and 4 policies contain only 1 T2 policy). This makes sense because each tuple has to be matched with (and transformed for) more policies, and because T2’s transformation cost is the highest. When there are multiple streams but only one matching policy, communication overheads can reduce the throughput. But as Fig. 5[b] indicates, having more matching policies for every stream helps maintain the overall throughput (r, nP, nS are the data rate per stream, number of matching policies and number of streams respectively). The similar pattern is found for Join policies, as shown in Fig. 5[c]. It can be observed that throughput of join depends on the similarity of the two joining streams. Specifically, when two Filter conditions are of type T2 (y < ts < z), the output streams (for joining) have more matches and therefore are more similar (throughput of 60) than when one filter condition is of type T4 (ts%x = y) where throughput is at 40 tuples/sec.

Finally, Fig. 6 illustrates how the system performance improves when more servers are utilized. We create a workload consisting of 16 streams and 320 policies. 4 of these streams incur expensive load with 4 T2 policies per stream. When there are more than one servers at the cloud, we consider two ways of distributing the workload: simple — each stream occupies one machine, and balanced — expensive policies are distributed evenly among the machine. The latter may result in one stream occupying multiple servers. Fig. 6[a] shows that the throughput increases linearly with the number of servers, which is as expected. Also, the balanced distribution achieves lower throughputs, because in the simple distribution, the servers handling light workload gets very high throughputs, whereas with the balanced distribution all servers get low throughputs. However, at 16 servers, the balanced distribution outgrows the simple distribution, but this throughput is obtained over duplicate tuples. This is because at 8 and more servers, there are streams being processed by multiple servers. Fig. 6[b] shows the latency distributions which clearly demonstrates the benefit of having more servers. The maximum latency using 1 server is over 100s, but is reduced to below 14s using 16 machines. The balanced distribution achieves lower maximum latency and lower variance, since all servers incur a similar load (as opposed to a few servers incurring much heavier loads than the others).

To understand this latency distribution, we examine the breakdown of execution time at the cloud. The latency per tuple consists of queuing time and processing time (transformation, or crypto time). Fig. 6[c] illustrates this breakdown for 16 servers. It can be seen that adding more servers does not benefit the crypto time. More importantly, the queuing time (overall time minus the crypto time) is an order of magnitude larger than crypto time. Adding more servers reduces the queue size at each node, therefore the queuing time and subsequently the overall execution time can be greatly improved.

Discussion.

Our experiments with a single cloud server showed that processing at the cloud (transformation of ciphertexts) achieves the highest throughput of only 250 tuples/sec. That this result compares poorly against Esper’s reported performance of over 1 million tuples/sec may cause one to question the reason to outsource data to the cloud and subsequently the very practicability of Streamforce. However, we stress that our system model lends itself naturally to data outsourcing for two reasons. First, stream data is potentially infinite in size, hence the cost of storage and management can be greatly reduced by using the cloud. Second, our system follows the multi-client model (one stream is accessed by many different clients), and the stream processing can be computationally expensive. In such settings, the saving of network bandwidth and computation justify the moving of data to the cloud [6]. As demonstrated in Fig. 6, the overall throughput can be improved considerably by adding more servers at the cloud, so Streamforce is amenable to heavier workload by massive distribution and parallelization of the tasks. Finally, while there is room for improvement both in the set of features and functionalities, Streamforce makes an important first step in providing security assurance over streaming data outsourced to an untrusted cloud.

6. RELATED WORK

The design space concerning access control enforcement on a cloud environment can be characterized using three properties [7]: policy fine-grainedness, cloud trustworthiness and cloud/client work ratio. The last property specifies how much work the cloud and user has to perform in relation to
7. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented a system providing fine-grained access control for stream data over untrusted clouds. Our system — Streamforce — allows the owners to encrypt data before relaying them to the cloud. Encryption ensures both confidentiality against the cloud and access control against dishonest users. Streamforce uses combinations of three encryption schemes: a deterministic scheme, a proxy ABE scheme and a sliding-window scheme. We have showed how the cloud can enforce access control over ciphertexts by transforming them for authorized user, without learning the plaintexts. In Streamforce, the cloud handles most of the heavy computations, while the users are required to do only simple, inexpensive decryptions. We have implemented Streamforce on top of Esper, and carried out a benchmark study of the system. The security cost is large enough to hinder the system from achieving very high throughputs (as compared to the maximum throughput of Esper). However, we believe the current throughput is sufficient for many real-life applications in which data arrives at low rate. Furthermore, we have shown that employing more servers in the cloud can substantially improve the overall performance, since it is possible to a certain extent to parallelize the workload in Streamforce.

We believe that our work has put forth the first secure system for outsourcing the enforcement of fine-grained access control for stream data. Streamforce occupies an unique position in the design space, and also opens up a wide avenue for future work. There exists classes of applications that require much higher throughput than currently possible in Streamforce. We acknowledge that more effort is required to satisfy both security and demand for performance. However, Streamforce provides a crucial first step.

Our immediate plan is to make Streamforce scale better. To this end, we are investigating to implement Streamforce over cloud-based stream processing engines, such as Storm [14] and S4 [2]. These systems provide an easy way for users to specify how to process query graphs over a given number of servers. Supporting elasticity for Streamforce (spawning new servers to balance the load during runtime) is an interesting and challenging venue for future work, especially since processing states at the servers may not be partitionable [9]. At the current stage, getting the data into Streamforce is the main bottleneck: each ciphertext is over 100KB in size. We are exploring techniques to reduce the

---

(a) 1 stream, multiple policies
(b) multiple streams, multiple policies
(c) join

Figure 5: System throughput for complex workloads

(a) Throughput
(b) Latency
(c) Breakdown

Figure 6: Workload distribution
ciphertext sizes and to improve the (incoming) data throughput.

Although Streamforce supports a wide range of policies, this range can still be improved. As stated in [7], policies involving more complex functions such as granularity and similarity policies are useful in many applications. Supporting these functions over ciphertext requires more powerful homomorphic encryptions, such as [10]. However, one must be careful to strike the balance between security and performance. Our current encryption schemes do not support revocation, nor do they support negative and hidden attributes. In particular, hidden attributes are necessary when the owner wishes to hide more information from the cloud. We plan to explore if and how existing proposals for these features [3, 16, 15] can be implemented in our system. Furthermore, we would like to relax the current adversary model which is semi-honest. A malicious adversary may compromise data integrity, skip computation or compute using stale data. We believe that detecting and recovering from these attacks are important for outsourced database systems, but they may come at heavier cost of performance. Finally, in Streamforce we have assumed that owners know which data to share, while recommendations are not easy to make. Differential privacy [8] could help reasoning about which data to share, while recommendation techniques [5] can help determining the appropriate policies.

Acknowledgement

This work was supported by A*Star SERC grant no. 102 158 0038 for the project ‘pCloud: Privacy in data value chains using peer-to-peer primitives’, and by the Singapore NRF’s CRP grant no. NFR-CRP-2011-08 for the project ‘Design and Development of a Comprehensive Information Technology Infrastructure for Data-Intensive Applications and Analysis’. The first author did part of this work while he was a research fellow at NTU Singapore.

8. REFERENCES


APPENDIX

A. DET-CPA SECURITY DEFINITION

Security of $E_d$ is defined via the security game consisting of three phases: Setup, Challenge, Guess.

- Setup: the challenger runs $\text{Gen}()$.
- Challenge: the adversary sends to the challenger two messages: $M_0 = (m_{0,0}, m_{0,1}, \ldots)$ and $M_1 = (m_{1,0}, m_{1,1}, \ldots)$, such that $|M_0| = |M_1|$ and $m_{i,j}$ are all distinct. The challenger chooses $b \leftarrow \mathcal{R} \{0, 1\}$, runs $\text{Enc}(M_b, SK)$ and returns the ciphertext to the adversary.
- Guess: the adversary outputs a guess $b' \in \{0, 1\}$.

$E_d$ is said to be secure with respect to deterministic chosen plaintext attacks, or DET-CPA secure, if the adversary advantage, defined as $\text{Adv}^{\text{CPA}} = |\Pr[b = b'] - \frac{1}{2}|$, is negligible.

B. SLIDING-WINDOW ENCRYPTION

Let $W$ be the set of all possible window sizes, $G$ be a multiplicative group of prime order $p$ and generator $g$. $d\text{Log}(x)$ computes the discrete log of $x$ in $G$ (we assumed that the plaintext domain is small).

B.1 Construction 1: $E_w^1$

- $\text{Gen}(\kappa)$: for all $w \in W$, $SK_w \leftarrow \mathcal{R} Z_p$.
- $\text{Enc}(M, W)$: for each $w \in W$, let $R = (r_0, r_1, \ldots, r_{|M|})$ such that $r_i \leftarrow \mathcal{R} Z_p$ and $s(R, w[i]) = SK_w$. The ciphertext is $CT = \bigcup_{w \in W} CT_w$, where $CT_w = (g^{m_0 + r_0}, g^{m_1 + r_1}, \ldots)$. $\text{Dec}(w, CT, SK_w)$: extracts $CT_w$ from $CT$ and computes $s(M, w[i]) = d\text{Log}(\frac{E(CT_w)}{SK_w})$.
Figure 7: Implementation of aggregate operator

B.2 Construction 2: $\mathcal{E}_w^2$

- Gen($\kappa$): let $\mathcal{E}_w$ be a CPA-secure symmetric encryption scheme. For all $w \in W$, invokes $\mathcal{E}_w.Gen(\kappa)$ to generate a key pair $(PK_w, SK_w)$.

- Enc($M, W$): $CT = \bigcup_{w \in W} \mathcal{E}_w.Enc(PK_w, s(M, w)[i])$.

- Dec($w$, $CT$, $SK_w$): extracts $CT_w$ from $CT$, then computes $s(M, w)[i] = \mathcal{E}_w.Dec(SK_w, CT_w[i])$

B.3 Construction 3: $\mathcal{E}_w^3$

- Gen($\kappa$): the same as in $\mathcal{E}_w^2$.

- Enc($M, W$): let $R = (r_0, r_1, \ldots, r_{|M|-1})$ where $r_i \leftarrow \mathbb{Z}_p$, let $CT_0 = (g^{m_0 + r_0}, g^{m_1 + r_1}, \ldots)$. For all $w \in W$, let $CT_w[i] = \mathcal{E}_w.Enc(PK_w, s(R, w)[i])$. Finally, $CT = CT_0 \cup \bigcup_{w \in W} CT_w$.

- Dec($w$, $CT$, $SK_w$): extracts $CT_w$ from $CT$, then computes $s(M, w)[i] = dLog \left( \frac{p^{\mathcal{E}_w.Dec(CT_w, CT_w[i])}}{g^{\mathcal{E}_w.Enc(CT_w, CT_w[i])}} \right)$

C. AGGREGATE SECURE OPERATOR

Agg-1.

The owner first encrypts data using $\mathcal{E}_w^1$, the ciphertext is then encrypted with $\mathcal{E}_p$. The user decryption key is $\mathcal{E}_p.Gen(\cdot)$, and $P$-Agg1$(w, A_p) = \{'att = A_q', \{'window = ws'\}\}$.

To encrypt $d_{ws}$, the owner first executes $\mathcal{E}_w^1(\{CT_{ws}\})$ as shown in Fig. 7, then computes:

$\mathcal{E}_p.Enc(d, \{'window = 1'\})$

For every window size $ws$, the cloud maintains a buffer of size $ws$. The incoming ciphertext $CT$ is transformed using $\mathcal{E}_p.Trans(\cdot)$, and the result is added to the buffer. Once the buffer is filled, the cloud computes the product of its elements, sends the result to the user and clears the buffer.

Agg-2.

This implementation uses $\mathcal{E}_w^2$ with $\mathcal{E}_p$ as the auxiliary encryption scheme. The owner itself computes the window aggregates and encrypts the result using $\mathcal{E}_p$. User decryption key is $\mathcal{E}_p.KeyGen(MK, P$-Agg2$(ws, A_p))$, where:

$P$-Agg2$(ws, A_p) = \{'att = A_q', \{'window = ws'\}\}$

To encrypt $d_{ws}$, the owner first executes $\mathcal{E}_w^2(\{CT_{ws}, TK_{ws}\}) \rightarrow CT$ as shown in Fig. 7, then the ciphertext is computed as:

$\mathcal{E}_p.Enc(d_{ws}, \{'window = 1'\})$

At the cloud, the ciphertext for a window aggregate are of the same form as for a normal data tuple. The cloud simply invokes $\mathcal{E}_p.Trans(CT_{ws}, TK_{ws})$ and forwards the results to the user.

Agg-3.

This implementation uses $\mathcal{E}_w^3$ with $\mathcal{E}_p$ as the auxiliary encryption scheme. The user key is $\mathcal{E}_p.KeyGen(MK, P$-Agg3$(ws, A_p))$, where:

$P$-Agg3$(ws, A_p) = \{'att = A_q', \{'window = ws'\}\}$

To encrypt $d_{ws}$, the owner first computes $\mathcal{E}_w^3(d_{ws}) \rightarrow (U, V)$ as shown in Fig. 7 where $s^* \leftarrow \mathbb{Z}_p$ is a public parameter. The ciphertext is:

$\mathcal{E}_p.Enc(U, \{'att = A_q'\})$,

$\mathcal{E}_p.Enc(V, \{'window = 1'\})$

The cloud maintains a $ws$-size buffer, and a variable $X$ whose initial value is $g^{s^*}$. 1. For the incoming ciphertext $CT = (ts, U, V, Z)$, the cloud performs $\mathcal{E}_p.Trans(TK, V)$ and adds the result to the buffer. Once the buffer is filled (at index $ts$), the cloud computes the product $U'$ of the buffer elements and clears the buffer. Next, it computes $V' \leftarrow \mathcal{E}_p.Trans(TK, Z)$, and then assign $X \leftarrow V'$. Finally, it sends $(U', V')$ to the user, at which the sum is decrypted as:

$\mathcal{E}_p.Dec(SK, U')$