PolyStream: Cryptographically Enforced Access Controls for Outsourced Data Stream Processing

Cory Thoma, Adam J. Lee, Alexandros Labrinidis
Department of Computer Science
University of Pittsburgh
(corythoma, adamlee, labrinid)@cs.pitt.edu

ABSTRACT

With data becoming available in larger quantities and at higher rates, new data processing paradigms have been proposed to handle high-volume, fast-moving data. Data Stream Processing is one such paradigm wherein transient data streams flow through sets of continuous queries, only returning results when data is of interest to the querier. To avoid the large costs associated with maintaining the infrastructure required for processing these data streams, many companies will outsource their computation to third-party cloud services. This outsourcing, however, can lead to private data being accessed by parties that a data provider may not trust. The literature offers solutions to this confidentiality and access control problem but they have fallen short of providing a complete solution to these problems, due to either immense overheads or trust requirements placed on these third-party services.

To address these issues, we have developed *PolyStream*, an enhancement to existing data stream management systems that enables data providers to specify attribute-based access control policies that are cryptographically enforced while simultaneously allowing many types of in-network data processing. We detail the access control models and mechanisms used by PolyStream, and describe a novel use of security punctuations that enables flexible, online policy management and key distribution. We detail how queries are submitted and executed using an unmodified Data Stream Management System, and show through an extensive evaluation that PolyStream yields a 550x performance gain versus the state-of-the-art system StreamForce in CODASPY 2014, while providing greater functionality to the querier.

1. INTRODUCTION

With more devices connecting to the Internet, the amount and speed of data being generated is ever-increasing, and processing it is becoming progressively more challenging. Data is being generated by a more diverse set of instruments ranging from sensors embedded into natural environments to monitor earthquakes and tsunamis, to sensors embedded in the human body to monitor personal well-being, to an increasing array of sensors built into smartphones and other wearables, to social media which is constantly updating and evolving [20]. This increase in data quantity and diversity, coupled with the real time nature of most monitoring applications, has brought about the paradigm of Data Stream Processing.

In a streaming environment, queries are long-running and process transient data flowing through the system. Stream processing is especially well-suited for early detection of anomalous events and for long-term monitoring through the use of Data Stream Management Systems (DSMS). Streaming environments separate the *provider* of the data from the *consumer*, and often leverage third-party *computational nodes* for processing their continuous queries. Unlike traditional database systems, this separation leads to data sources having little control over *how* their data is handled or *who* has access to it. Given this separation of the data provider and the eventual data consumer, it becomes difficult to reason about how a data provider can protect their private data. For a system to guarantee the confidentiality of the provider's data once it has been emitted, it must provide an access control framework that allows a data provider to easily describe who has access to their data. To ensure data remains confidential, it should be encrypted to prevent unauthorized users from learning any information about the underlying data once it has left the data provider. To accommodate this encryption, there must be a protocol for an online key management system which can dictate who gets access and how they should be granted access. Furthermore, modern systems are ever-changing with users changing their preferences, leaving and entering the system, or changing their demand on the system. Over time, a data provider may wish to change their access control policies to match changes in the system or their personal preferences. To add a final complication to the problem of enforcing access controls over streaming data, modern systems often make use of outsourced third party systems to cheaply and easily manage their continuous queries. Adding access controls and encryption should not limit a data consumers ability to outsource computation or author meaningful and useful queries over data for which they have been granted access, nor should it greatly impact the performance of these queries either when outsourced or executed locally.

The current state-of-the-art system to solve this problem is Streamforce [5] and although it addresses many of the issues in enforcing access controls, it incurs prohibitive overheads, and limits the types of queries that can be issued to the system. A system like CryptDB [29] addresses a similar problem for outsourced databases but does not provide the dynamic online access control and key management protocol required for an ever-changing streaming environment.

To fully address these issues, we have designed the PolyStream framework, which considers data confidentiality and access controls as first-class citizens in distributed DSMSs (DDSMS) while supporting a wide range of query processing primitives and flexi-

SACMAT'16, June 05-08, 2016, Shanghai, China

DOI: http://dx.doi.org/10.1145/2914642.2914660

ble key distribution and policy management. Unlike previous work in this space, PolyStream runs on top of an unmodified DDSMS platform; supports a wide range of attribute-based, user-specified cryptographic access controls; allows dynamic policy updates and online, in-stream key management; and enables queriers to submit arbitrary queries using a wide range of in-network processing options. More precisely, in developing PolyStream, we make the following *contributions*:

- *PolyStream allows users to cryptographically enforce access controls over streaming data and alter their policies in real time.* In PolyStream, access control is based upon a data consumer's cryptographically-certified attributes. PolyStream supports Attribute-Based Access Controls (specifically, a large fragment of ABAC_α [22]) and Attribute-Based Encryption (ABE) to enable data providers to write and enforce flexible access control policies over data at the column, tuple, or stream levels.
- *PolyStream provides a built-in scheme for distributing and managing cryptographic keys using ABAC.* PolyStream utilizes a modified version of Security Punctuations [25] (SPs) to enforce ABAC policies. SPs are typically used to allow data providers to communicate access control policies to the trusted servers on which users run queries over their streaming data. Prior work in the cryptographic DDSMS space has largely ignored the subject of key management and changes to policy by relying on separate offline systems to handle key and policy distribution. By contrast, PolyStream uses SPs to both communicate the policies protecting the contents of a given stream, as well as to provide a key distribution channel for decryption keys that are protected by Attribute-Based Encryption enforcement of ABAC policies. This enables a flexible, online key management and policy update infrastructure, even for stateful continuous queries.
- *PolyStream allows data consumers to submit a wide range of queries.* To the best of our knowledge, no streaming system has allowed in-network processing of arbitrary queries over protected data streams handled by an untrusted infrastructure. In systems supporting user-specified queries, the data processing servers are typically assumed to be trusted [3, 13, 14, 25, 26]. In systems processing data over untrusted infrastructure, cryptographic protections are enforced such that data consumers have only limited query processing abilities [5]. By contrast, PolyStream's key management infrastructure allows untrusted compute nodes to process equality, range, and aggregate queries, and also has limited support for in-network joins.
- *Finally, PolyStream functions as a stand-alone access control layer on top of an underlying DSMS.* PolyStream is not, itself, a DDSMS. Rather, it provides an access control service layer on top of another DDSMS. SPs are processed by PolyStream and are obtained via long-running selection queries on the underlying DDSMS. Queries are submitted via PolyStream, rewritten, and deployed using operations already available from the underlying DDSMS, thereby requiring no changes to the system.

We survey related work in Section 2, and describe our system and threat models in Section 3. Section 4 describes the design and implementation of PolyStream, which is then experimentally analyzed in Section 5. Finally, we present our conclusions and directions for future work in Section 6.

2. RELATED AND PRELIMINARY WORK

This section outlines the related work as well as the primitives necessary for understanding PolyStream.

2.1 Related Work

Many streaming systems have been proposed and studied to date. The most notable modern stream processing systems are the Aurora [2], its distributed version Borealis [1], and STREAM [9]. To protect the users of streaming systems from having their private data leaked or stolen, several access control techniques have been proposed. These techniques can be classified into two main categories: those that trust an outsourced third party to enforce access controls over their data, and those that do not.

FENCE [26] is a streaming access control system that trusts third parties to enforce access controls. Nehme et al. introduced the concept of a *Security Punctuation* for enforcing access control in steaming environments [25]. A Security Punctuation (SP) is a tuple inserted directly into a data stream that allows a data provider to send access control policies and updates to the stream processing server(s) where access controls are to be enforced.

Carminati et al. provide access control via enforcing Role Based Access Control (RBAC) and secure operators [13–15]. Operators are replaced with secure versions which determine whether a client can access a stream by referencing an RBAC policy. Their work assumes a trusted and honest server that enforces their access control policies. In [13], the authors extend this work to interface with any stream processing system through the use of query rewriting and middleware, as well as a wrapper to translate their queries into any language accepted by a DSMS.

Ng et al. [27] allow the data provider to author policies over their data. The system uses the principles of limited disclosure and limited collection to limit who can access and operate on data streams, requiring queries to be rewritten to match the level at which they can access the data. Their system requires changing the underlying DSMS and therefore is not globally applicable, and it also requires a trusted server to rewrite the queries.

Linder and Meier [24] focus on securing the Borealis Stream Engine [1]. They introduce a version of RBAC called owner-extended RBAC, or OxRBAC which operates over different levels of stream objects. OxRBAC allows for each object to have an owner, as well as allowing for rules and permissions. Owners are allowed to set RBAC policies over their objects, which the system will enforce. Objects include schemas, streams, queries, or systems. Users are limited to RBAC policies and must trust the server to enforce their policies as well as see their data in plain text.

Unlike the aforementioned work, Streamforce [5] does not trust the stream processing infrastructure to enforce access control and instead relies on cryptography. Streamforce assumes an untrusted, honest-but-curious DDSMS and utilizes Attribute-Based Encryption (ABE) to enforce access control. The data provider will encrypt their data based on what attributes they desire a potential data consumer to possess. Streamforce is able to enforce access control over encrypted data through the use of their main access structure, *views*. Views are submitted by the *data provider* to the (untrusted) server as a query and only those results are returned to the data consumer. The use of views in this system requires the data provider to be directly involved in the querying process, which has the consequence of limiting what a querier can do with the permissions they were given. Streamforce's use of ABE results in large decryption times depending on the number of attributes. In order to reduce the cost on the data consumer's end, Streamforce outsources decryption to the server [16, 17]. However, even with outsourced decryption, Streamforce reports up to 4,000x slowdown compared to an unmodified system due to their extensive use of ABE. Streamforce also requires the *data provider* to execute all aggregates locally, which may not be feasible since the provider may be a system of sensors, or simply a publish/subscribe system. Finally, Streamforce

requires an offline key management solution which makes it hard to reason about key revocation and policy updates.

CryptDB [29] allows computation over encrypted data on an untrusted honest-but-curious relational DBMS. CryptDB's primary goal is not access control, but rather allowing computation over encrypted data stored on an untrusted third-party database system. Essentially, CryptDB offers protection from honest-but-curious database administrators through the use of encryption, but does not offer fine grained access controls over the data stored on the system, nor does it offer a key management mechanism since the data owner is in direct control of who can access their data and can change keys at will. CryptDB utilizes specialized encryption techniques for allowing queries to operate on untrusted servers over encrypted data. Specifically, CryptDB employs Deterministic, Order-Preserving, Homomorphic, Specialty Search, Random, and Join encryption techniques to enable many different queries to operate. CryptDB uses *onion* structures to store data, in which data is encrypted under multiple keys: the outer layer of the onion is the most secure, and successive layers provide more functionality (i.e., allow for queries to be executed), but may leak some data. The use of onions as tuples in a streaming system would lead to unnecessary encryptions and decryptions as not all encryption levels are required (cf. Section 5). MONOMI [32] extends CryptDB to allow the querier to also processes queries to provide a broader range of queries to the users.

2.2 Cryptographic Primitives

We now overview the basic encryption techniques that will be used in the coming sections. We use two main types of encryption: computation-enabling and attribute-based encryption.

2.2.1 Attribute-Based Encryption

Attribute-Based Encryption (ABE) is used to encrypt data such that only entities with the proper certified attributes can decrypt a given ciphertext. In an ABE system, an Attribute Authority (AA) holds a master key that can be used to generate decryption keys tied to an individual's attributes (e.g., *Professor* or *Orthopedist*). Encryption requires only public parameters released by the AA and a logical policy p in addition to the data to be encrypted, while decryption requires attribute-based decryption keys provided by the AA. The following functions comprise an ABE system:

- **GenABEMasterKey()**: Generates a master key MK.
- **GenABEPublicParamaters(MK)**: Generates the public parameters pa_p needed for encryption.
- **GenABEDecryptionKey(UA_{user}, MK)**: generates a decryption key k_d for *user* based on their set of attributes UA_{user}.
- $\mathbf{Enc}_{ABE}(pa_p, p, d)$: generates an ABE encrypted ciphertext c with the public parameters, a logical policy p, and the data d.
- $\mathbf{Dec}_{ABE}(p, k_d, c)$: recovers the data d using the ABE decryption key k_d, the policy p, and the ciphertext c.

Note that the first three functions are executed by the AA. On the other hand, $\mathbf{Enc}_{ABE}(pa_p, p, d)$ can be executed by any entity, as it relies only on public information, while $\mathbf{Dec}_{ABE}(p, k_d, c)$ can be executed by any entity with an ABE decryption key k_d.

2.2.2 Computation-Enabling Encryption

The computation-enabling encryption techniques allow a user to perform some sort of computation over the encrypted data and therefore allow for outsourced data to be processed without leaking plaintext data. However, each technique does leak some metadata about the underlying plaintext.

- *Random Encryption (RND)* uses a block cipher (e.g., AES in CBC mode) to encrypt fields so that no two fields are encrypted to the same value, and does not leak information regarding the correspondence of actual values. Ciphertexts are different even when RND is given the same input for any given value.
- *Deterministic Encryption (DET)* ensures that multiple encryptions of the same value result in the same ciphertext. DET is implemented using a standard cipher (e.g., AES) with some small alterations. Values less than 64 bits are padded, and any value greater than 128 bits is encrypted in CMC mode [18] since CBC mode leaks prefix equalities. This enables equality checking over encrypted values.
- *Order-Preserving Encryption (OPE)* enforces the relationship that $x < y$ iff $OPE(x) < OPE(y)$. The OPE scheme used in our system is adapted from Boldyreva et al. [11], where the authors present Order-Preserving Symmetric Encryption. This enables range queries over encrypted data, but only has IND-OCPA (indistinguishability under ordered chosen-plaintext attack) security and therefore can leak the ordering of tuples [12].
- *Homomorphic Encryption (HOM)* enforces the relationship that $HOM(x) * HOM(y) = HOM(x+y)$. This allows the execution of summation (and by extension average) queries on untrusted servers without leaking field data values or the summation value. PolyStream uses the Paillier [28] encryption scheme. This enables in-network aggregation of encrypted data without leaking individual data values, but comes at the cost of increasing the computational load of this aggregation. An adversary does learn a relationship for the sliding window, since the encrypted sum for the sliding window's worth of tuples is revealed. Note that a sliding window is simply the range of tuples used to generate a result (i.e. 3 minutes, 100 tuples) over the life of the stream.

As noted above, these four techniques make it possible for untrusted computational infrastructure to execute certain query processing functionalities over encrypted data. Deterministic, Order-Preserving, and Random cryptosystems are parameterized by a similar set of functions:

- $\mathbf{GenKey}_{DET,OPE,RND}()$: Generates a symmetric key k corresponding to the technique used.
- $\mathbf{Enc}_{DET,OPE,RND}(k, d)$: Encrypts data d with key k.
- $\mathbf{Dec}_{DET,OPE,RND}(k, c)$: Decrypts ciphertext c with key k.

The Pallier homomorphic cryptosystem does not rely on a single key, but rather a pair of (public) encryption and (private) decryption parameters. For the purposes of this paper, we represent this functions parameterizing this cryptosystem as follows, and refer the reader to [28] for more information:

- $\mathbf{GenKey}_{HOM}()$: Generates a encryption parameter pa_{HOM} and a private parameter pp_{HOM}.
- $\mathbf{Enc}_{HOM}(pa_{HOM}, d)$: Encrypts data d with key pa_{HOM}.
- $\mathbf{Dec}_{HOM}(pp_{HOM}, c)$: Decrypts ciphertext c with key pp_{HOM}.

3. SYSTEM AND THREAT MODEL

3.1 System Model

PolyStream provides an API that sits between end users (i.e., data providers and data consumers) and an underlying Distributed Data Stream Management System (DDSMS). No changes to the underlying DDSMS are required for PolyStream to work. Instead, PolyStream makes uses of common functions provided by all DDSMS. Specifically, DDSMSs provide the user with an $optimizeQuery$ func-

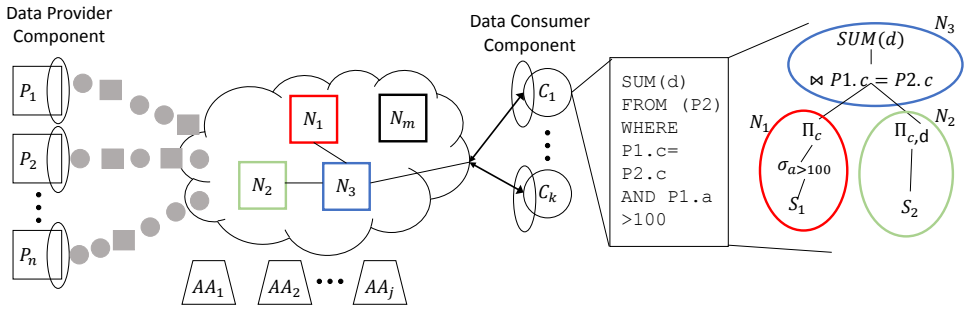


Figure 1: PolyStream system model. An example query is given and represented in the cloud. Node N_1 executes part of the query as represented by the dotted line, and similarly N_2 and N_3 are responsible only for what is represented in the execution tree depicting the query.

tion that takes a CQL query, verifies it, optimizes it, and returns a query plan, and a function *submitQuery* that places the query. DDSMSs also provide a *results* function that yields the result of a query. PolyStream provides functionality to support the three key players in a DDSMS, and one new component (cf., Figure 1):

- *Data Providers* (squares labeled "P_i") create and distribute data streams. Data providers do not necessarily trust all other parties in the DDSMS, which requires creating and updating access control policies for their streams. To aid data providers in creating policies, a trusted third-party *Attribute Authority* verifies the identities and attributes of other parties in the system.
- *Attribute Authorities* (trapezoids labeled "AA_i") verify and certify the attributes of system components. The scope of an AA may vary: while one AA may exist to certify the job titles or roles of employees within a company, others may certify attributes that cross-cut many organizations (e.g., ABET accredits many universities). One system can have many AAs, and individuals may choose which AAs they trust.
- *Compute & Route Nodes* (CRN, squares labeled "N_i") are tasked with executing queries on data streams. Data consumers place query operators on CRNs, which then process incoming tuples and produce output tuples that flow either to other CRNs or to the data consumer.
- *Data Consumers* (circles labeled "C_i") submit continuous queries. Depending on permissions, the query operators resulting from these queries are either submitted to CRNs via PolyStream or executed locally. Queries are submitted using a declarative language, such as CQL [7], and are optimized by the data consumer. PolyStream allows streaming operators to be spread across multiple CRNs with varying levels of trust.

A sample query is given in Figure 1. In this simple example, Data from P_1 and P_2 are combined from different machines running selection and projection operators. The results are joined and summed on a different node, and returned to the data consumer. PolyStream assumes that tuples arrive in order. This is easy to accomplish by utilizing sequence numbers from the data provider and enforcing that only the next tuple can be processed.

3.2 Threat Model

The main goal of PolyStream is to provide a mechanism that data providers can use to author and enforce access control policies over their own data streams. Access control policies are specified using Attribute-Based Access Controls (i.e., a fragment of ABAC_α), and enforced using Attribute-Based Encryption (ABE). *Attribute Authorities* (AAs) are trusted to correctly issue attributes to entities within the system. There may be many AAs in the system, which can vary in the scope of attributes that they will certify. AAs are the master secret key holders of the ABE system and, as such, are responsible for creating ABE decryption keys for the entities whose attributes they certify. AAs are also responsible for the revocation of attributes once a user's attributes have changed, which is outside the scope of this paper. The literature has explored attribute revocation [19,21,34] and interested readers are encouraged to explore further. *Data providers* are trusted by data consumers to correctly emit the data streams that they advertise. *Compute and Routing Nodes* (CRNs) may not be trusted by data providers; as such, data streams may be encrypted to hide information from these parties. Similarly, PolyStream avoids placing consumers' query operators on CRNs not trusted to execute these operators. CRNs are assumed to behave in the honest-but-curious model: they will not maliciously alter data that they process, but may attempt to infer information from the tuples that they process. *Data consumers* may or may not be trusted by data providers, who can make use of ABE to enforce ABAC policies protecting their data from unauthorized consumers. Finally, we assume that all entities can establish and communicate over pairwise private and authenticated channels (e.g., using SSL/TLS tunnels).

4. PolyStream

We now overview the PolyStream system. First, we introduce the access control framework that a data provider can use to describe and author policies. We then detail the online policy distribution and cryptographic key management channel used to communicate and enforce the access control policies. We also detail how data consumers' queries are handled in the PolyStream system.

4.1 Access Control Model and Mechanism

Given the dynamic nature of real-time data stream processing systems, data providers, data consumers, and compute and route nodes are likely to join and leave the system over time. This inhibits a data provider's ability to have a full understanding of every entity acting in the system. As such, PolyStream makes use of attribute-based policies to help data providers protect their sensitive data in a more generalizable manner.

Access Control Model. Attribute-based access controls allow a data provider to *describe* authorized consumers of their data, rather than listing them explicitly. PolyStream makes use of a large fragment of the ABAC_α [22] model. An ABAC_α system is comprised of the following state elements:

- Sets U, S, and O of users, subjects, and objects
- Sets UA, SA, and OA of user attributes, subject attributes, and object attributes

Furthermore, ABAC_α makes use of the following grammar for specifying policies:

$$
\begin{aligned}
p ::= \;& p \wedge p \;\mid\; p \vee p \;\mid\; (p) \;\mid\; \neg p \;\mid \\
& \text{set}\;\; setcompare \;\; \text{set} \;\mid\; \text{atomic} \in \text{set} \;\mid \\
& \text{atomic} \;\; automiccompare \;\; \text{atomic} \\
set ::= \;& set_{sa} \subseteq SA | set_{oa} \subseteq OA | set_{ua} \subseteq UA \\
setcompare ::= \;& \subset \mid \subseteq \mid \not\subset \\
atomic ::= \;& attribute \in SA | attribute \in OA | attribute \in UA \\
atomiccompare ::= \;& < \mid = \mid \leq \\
attribute ::= \;& < string >
\end{aligned}
$$

In PolyStream, the set U is comprised of all entities acting in the system (i.e., data providers, consumers, and CRNs). The set O contains pairs (t,ℓ) containing all tuples t being processed by the underlying DDSMS (i.e., data fields or streams), and the access level ℓ at which they should be protected. PolyStream supports four such access levels, corresponding to the type of in-network processing that will be allowed: NONE (no in-network access), SJ (in-network selection and join), RNG (in-network range queries), and AGG (in-network aggregation). While there are no explicit subjects in PolyStream, queries issued by a data consumer can be given access to a limited set of the issuing user's attributes. As such, S is comprised of the long-running queries submitted by data consumers.

Data producers use the ABAC_α policy grammar to author protections over the data that they supply to the DDSMS. In this paper, we will use the shorthand $(q \wedge r) \vee s$ to express a policy of the form $(q \in UA \wedge r \in UA) \vee s \in UA)$, since all policies are written as constraints over the set UA of user attributes that must be possessed by an authorized data consumer (and thus by the query subject operating on their behalf). Note also that PolyStream does not make use of the atomic operators for $<$ and $\leq$, since our underlying ABE library supports only string attributes.

Enforcement Mechanism. Unlike most stream processing systems, in PolyStream, CRNs are not trusted to correctly enforce data provider access controls. As such, we enforce ABAC_α policies cryptographically by encrypting data prior to introducing it to the DDSMS. Recall that PolyStream supports four access permissions: NONE, SJ, RNG, and AGG. We now describe each in more details, and discuss how cryptography can assist in the enforcement of these permissions.

- **NONE**. This permission prevents all in-network processing. To enforce the NONE permission for a tuple t, we simply encrypt t using a randomized cryptosystem (e.g., AES in CBC mode) prior to transmission. That is, given a session key k, we transmit ciphertext $c = \mathbf{Enc}_{RND}(\text{k,t})$ to the DDSMS. Intermediate CRNs cannot glean any information about the contents of this ciphertext, but authorized consumers can decrypt it upon receipt.
- **SJ**. This permission allows in-network selection and joins of streams sent by the same data producer. To enforce the SJ permission for a tuple t, we encrypt t using a deterministic cryptosystem (e.g., AES in CMC mode) prior to transmission. Given a session key k, we transmit the ciphertext $c = \mathbf{Enc}_{DET}(k,t)$ to the DDSMS. Since the same plaintext value will always encrypt to the same ciphertext value, untrusted CRNs can carry out selection on static values or join two streams whose join attributes are encrypted under the same key.

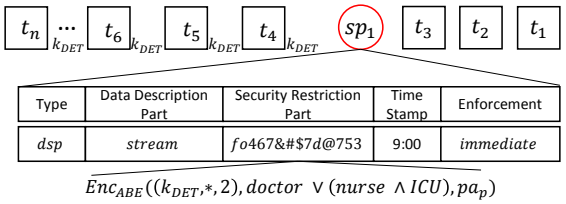


Figure 2: A typical Security Punctuation with an example use case.

- **RNG**. This permission allows in-network processing of range queries. To enforce the RNG permission for a tuple t, we use an order-preserving encryption scheme (e.g., [11]) and a session key k to transmit the ciphertext $c = \mathbf{Enc}_{OPE}(k,t)$ to the DDSMS. Given (encrypted) range bounds $l = \mathbf{Enc}_{OPE}(k,v_1)$ and $h = \mathbf{Enc}_{OPE}(k,v_2)$, an untrusted CRN can check whether $l \leq c \leq h$ without learning v_1, v_2, or t.
- **AGG**. This permission allows in-network processing of aggregate queries. Enforcement of the AGG permission uses an additively homomorphic cryptosystem (e.g., Pallier [28]) to enable in-network aggregation. Given tuples $t_1, t_2, \ldots, t_n$ and a public/private key pair $\langle k, k^{-1} \rangle$, we compute and transmit $c_1 = \mathbf{Enc}_{HOM}(k,t_1), c_2 = \mathbf{Enc}_{HOM}(k,t_2), \ldots, c_n = \mathbf{Enc}_{HOM}(k,t_n)$ to the DDSMS. An untrusted CRN can then compute $c_1 \times c_2 \times \ldots \times c_n = \mathbf{Enc}_{HOM}(k, s = t_1 + t_2 + \ldots + t_n)$ without learning s or any t_i.

Table 1 summarizes how each permission can be enforced cryptographically, as well as the DDSMS operations enabled by the permission. Note that PolyStream only supports encrypted joins on streams that are DET-encrypted under the same key. In principle, this likely means that joins are only possible over streams published by the same data producer. Supporting a richer variety of joins is left to future work.

Although the above constructions enable in-network processing, they do not enable attribute-based control of these access permissions. To cryptographically enforce ABAC_α policies over objects in PolyStream, we make use of attribute-based encryption to ensure that the session keys used above can only be recovered by authorized data consumers. In particular, consider an ABAC_α policy p authored over attributes issued by some authority AA_i whose public parameters are pa_i, and a session key k used to enforce one of the above four access permissions over some data tuple. In this case, the data producer can transmit $\mathbf{Enc}_{ABE}(pa_i, p, k)$ to authorized data consumers. Authorized consumers can then decrypt the session key k, which can be used to access protected data tuples. The exact mechanics of this policy distribution and key management process will be discussed next.

4.2 Policy Distribution

In a DDSMS, data providers do not control the paths taken by their data. As such, distributing, updating, and enforcing policies protecting that data take some effort, particularly if the infrastructure itself is only semi-trusted. Security Punctuations (SP) [25] address this issue by providing a mechanism for distributing policy along *with* data. A SP is simply a tuple injected into a provider's data stream (represented as a circle in Figure 2) that describes an access control policy over some set of protected data. For PolyStream, a SP dictates the ABE-enforced ABAC policy or policies protecting a stream to potential consumers. SPs are comprised of the five fields below (and the top box in Figure 2):

Permission	Scheme	Type of Queries	Supported operators	Information Gained by Adversary
NONE	RND	None	None	Nothing
SJ	DET	Equality	Equality Select, Project, Join, Count, Group By, Order by	Equality of attributes
RNG	OPE	Range	Equality Select, Range Select, Join, Count	A partial to full order of tuples
AGG	HOM	Summations	Aggregates over summations	Encrypted Sum for sliding window

Table 1: Summary of what types of queries and operators are supported by each encryption scheme, as well as what each scheme could reveal to a potential adversary.

- *Type*: Indicates that the SP originated from a data provider.
- *Data Description Part*: Indicates the schema fields (e.g., "heart rate") within a tuple that are protected by this policy. This may be as broad as an entire stream, or as specific as an individual field.
- *Security Restriction Part*: Describes the policy being enforced.
- *Timestamp*: The time at which the tuple was generated.
- *Enforcement*: Either *immediate* or *deferred*. Immediate enforcement applies the new policy to tuples in buffers, whereas deferred enforcement applies the new policy only to tuples timestamped after the SP.

While prior work has used SPs to distribute plaintext policies for enforcement by a trusted DDSMS, PolyStream makes use of SPs as a policy and key distribution mechanism, but relies on cryptography for policy enforcement. This means that while the *type*, *data description part*, *timestamp*, and *enforcement* fields are straightforward, the structure of the *security restriction part* (SRP) requires greater explanation. PolyStream uses the SRP field to transmit a tuple $\langle c, p \rangle$ where p is the ABAC_α policy protecting access to the fields listed in the data description part, and c is an ABE ciphertext generated by encrypting the following three pieces of information:

- *Access Type:* The type of in-network permission (i.e., NONE, SJ, RNG, or AGG) allowed by this policy
- *Index:* The position(s) of the data field(s) being protected by the policy, listed in the DDP.
- *Decryption Key:* The symmetric key k used to recover data protected at the NONE, SJ, or RNG levels, or the private key k^{-1} used to recover data protected at the AGG level.

Note that the above index information is needed due to the fact that a given stream may include several copies of a given schema field. For instance, if one policy on a stream grants AGG access to "heart rate" to some individuals while providing other individuals with SJ access, two copies of the "heart rate" field will be transmitted: one encrypted using Pallier (for AGG access) and one encrypted with AES in CMC mode (for SJ access).

Given an SP with an SRP containing the pair $\langle c, p \rangle$, a data consumer can inspect p to determine whether they possess the attributes needed to decrypt c. If so, decrypting c provides the data consumer with a description of the in-network processing allowed by the policy, the indexes upon which this processing can occur, and the (symmetric or private) key needed to decrypt result tuples. This is enough information to facilitate query planning (*Which queries can I run?*), operator placement (*How can I place physical operators for these queries in the CRN network?*), and results analysis (*How can I decrypt the results that I receive?*).

Key revocation in PolyStream is as simple as updating the access control policy (even the same one again) so that a new key is generated. A key is therefore revoked when a user no longer possesses the proper attributes to satisfy the ABAC policy to get the new key. Data providers can develop their own policy for updating and refreshing keys to satisfy their own needs. Key revocation does not include the revocation of attributes. Attribute Authorities (AA) are responsible for the revocation of attributes so when a user loses possession of an attribute, a new ABE decryption key is issued. This could lead to a time where data consumers can have unauthorized accesses due to a loss of an attribute but still have the key from the last Security Punctuation. This can be protected against by data providers periodically updating the keys that they use to protect their streams, via the Security Punctuation mechanism described previously.

Algorithm 1 SubmitQuery

```
1: Submit query q to DSMS query Optimizer for Plan p
2: for Operation o in qurery q do
3:     if no entry in schemaTable then
4:         Return permission denied
5:     else
6:         retrieve Schema Key k from schemaTable
7:         Encrypt Attribute with k
8:         if o is Filter or Count then
9:             if Filtering on Equality AND
           permissionTable contains "SJ" then
10:                Encrypt value in o with key in permissionTable
11:            else if Filter on range AND
           permissionTable contains "RNG" then
12:                Encrypt value in o with key in permissionTable
13:            else if permissionTable contains "NONE" then
14:                Operator Executes Locally, exit
15:        if o is Sum then
16:            if permissionTable contains "AGG" then
17:                Change o to Multiplication
18:            else
19:                Operator Executes Locally, exit
20:        if o is Average then
21:            if permissionTable contains "AGG" then
22:                Create operations Count, SUM
23:                Create local operation Division for sum/count
24:            else
25:                Operator Executes Locally, exit
26:        if o is Join then
27:            if permissionTable contains "SJ" or "RNG"
           for the same provider then
28:                Encrypt value in o with key in permissionTable
29:            else
30:                Operator Executes Locally, exit
31:        if Other operator then
32:            Operator Executes Locally, exit
33: submit(Q)
```

4.3 Query Processing

This section overviews how a data consumer can submit and change queries based on policy updates from the data providers they are are interested in. When a data consumer authors a query, they submit it to PolyStream, which follows the steps outlined in Algorithm 1. Recall that PolyStream sits between an unmodified DDSMS and the data providers/consumers. First, PolyStream submits the query to the underlying DDSMS's query optimizer using the DDSMS's own *optimizeQuery* function, and receives back the generated query plan (in the form of a physical operator graph). Using this plan, PolyStream iterates through each operation of the

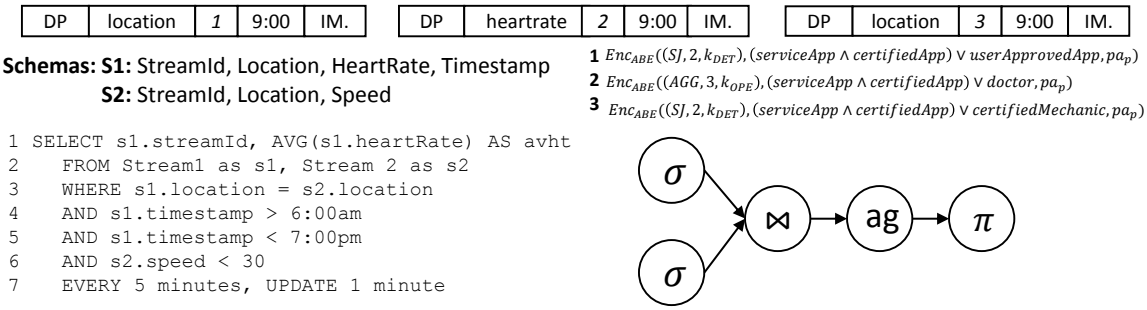


Figure 3: Motivating Example - RoadRageReducer App.

Algorithm 2 handleSecurityPunctuation

1: *SRP* = Security Restriction Part of *SP*
2: *DDP* = Data Description Part of *SP*
3: Have pa_{pr} ABE Decryption Key from AA
4: **if** pa_{pr} decrypts *SRP* **then**
5: **for** Fields f in *DDP* **do**
6: Associate f with permission p at index i from *SRP*

plan starting at the data source nodes in the graph (line 2) and determines where the operation must be placed.

Using data extracted from the SRP field of SPs received by the consumer from the data providers, PolyStream iteratively checks each operation to see if in-network processing has been enabled by the data provider (lines 8,15,20,26,31) and if access has been granted to the data consumer (lines 9,11,13,16,21,27). If so, this operation can be submitted to a CRN for in-network processing. If in-network processing is not possible, but the consumer has access to the field(s) being operated upon, the operator executes locally on the data consumer's device, where local decryption is possible (lines 14,19,25,30,32). We note that once *any* operator is placed on the data consumer's machine, all subsequent operations are placed on the data consumer's machine as well to avoid unnecessary network round trips. Once all operator placement decisions have been made, the query is submitted using the *submitQuery* function provided by the DDSMS and results are processed using the *results* function.

PolyStream provides a large number of operations that can be executed by CRNs over encrypted data, including operations that require multiple encrypted streams. For instance, a data consumer who is interested in aggregates over multiple encrypted streams can simply execute an aggregate separately over each encrypted stream and combine the results on their trusted machine once they are decrypted. A data consumer can also preform a join on two encrypted streams so long as they are encrypted with the same DET or OPE key. When the streams are encrypted under different keys, or processing on the CRN is otherwise not possible, PolyStream provides functionality similar to MONOMI [32] in that operations are executed on the data consumer's trusted machine after data is decrypted, so long as the data consumer possesses the proper decryption keys. Ultimately, this allows the consumer to issue *any* query for which they at least have decryption capabilities.

There exist alternative approaches to executing a multi-provider joins on the data consumer's trusted machine. One approach is for the data consumer to deploy a trusted node in the CRN network that simply decrypts multiple streams that are to be joined and re-encrypts each using a single symmetric key. This will allow later nodes in the CRN network to handle in-network joins, while minimizing the computational impact on the data consumer. Another approach is to use proxy re-encryption to compute on data even when it is encrypted with different keys [33]. Proxy re-encryption will enable one stream to be joined with another simply by re-encrypting one (still in its encrypted form) so that it is encrypted form matches the other. These techniques are being considered in our ongoing work

Example. Consider the scenario presented in Figure 3 with a single data provider, a city commuter, who is producing two data streams. The first stream contains health and location data being produced by a fitness watch linked to a phone, while the second contains location and travel data from her car's on-board computer. Stream 1 is protected by SP_1, which enables in-network SJ processing on the Location field for entities satisfying the policy $p_1 = (serviceApp \wedge certifiedApp) \vee userApprovedApp$ to recover the resulting data. Stream 1 is also protected by SP_2 enabling in-network AGG processing on the HeartRate field for anyone satisfying $p_2 = (serviceApp \wedge certifiedApp) \vee doctor$. Stream 2 is protected by SP_3, which enables in-network SJ processing on the Location field for entities satisfying the policy $p_3 = (serviceApp \wedge certifiedApp) \vee certifiedMechanic$ to recover the resulting data.

A data consumer, a mobile app called RoadRageReducer (a certified service app), wishes to execute the query shown in Figure 3. This query determines if the commuter has road rage by checking whether they are in their car while their average heart rate is elevated. To reduce the overall workload of the query, only high traffic driving times at low speeds are considered. Optimizing this query using the underlying DDSMS produces the operator graph shown in Figure 3. Given the information recovered from SP_1, SP_2 and SP_3, each of these operators can be placed on the CRN network since (i) the initial selection operates over unprotected fields (Speed and Timestamp), (ii) the join combines both streams using the SJ protected Location field, (iii) the averaging operator aggregates over the AGG protected HeartRate field, and (iv) the only input to the projection operator is a field index. Once the query is processed and a result is returned, the RoadRageReducer app can then use its Paillier decryption key to decrypt the resulting average over the HeartRate field.

5. EXPERIMENTAL EVALUATION

Like many other confidentiality enforcement systems, PolyStream exposes a tradeoff between performance and confidentiality. To better understand this tradeoff, we examined many different configurations/workloads on an experimental system comprised of a cluster of 10 small instances on Amazon EC2, which implements PolyStream as described above. All network communications occur over SSL/TLS tunnels. We also compared PolyStream with the current state-of-the-art Streamforce [5].

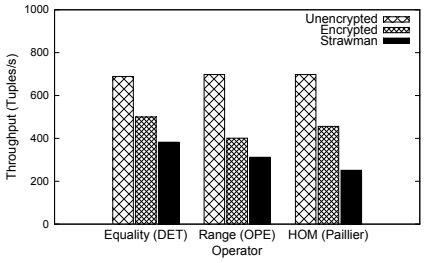


Figure 4: Throughput for each of the different operations supported for both unencrypted and encrypted streams.

5.1 Experimental Setup and Platform

Our system is built on top of the Storm distributed computing platform [30], as is the case for many other distributed DSMS prototypes/evaluations [4, 6]. Given that we do not use any functionality unique to Storm, we fully expect that PolyStream could be trivially ported to other distributed computing platforms like Spark Streaming [35] and Twitter Herron [23]. Storm provides a communication layer that guarantees tuple delivery. Storm accepts user-defined topologies that direct how components are networked. The main components of Storm are *spouts* and *bolts*. Spouts provide data to the system and therefore assume the role of data provider, and bolts compute on the data and take the role of data consumer or CRN. To better control experiments, a special scheduler was implemented to dictate which machines handled which components.

Tests were run on Amazon EC2 using small instances. All components were programmed in Java and packaged as JAR files. Each data consumer was assigned a set of attributes from a bolt *Central Authority*. One EC2 instance was devoted to controlling Storm's required libraries as well as assigning tasks and was not used in experimentation; leaving nine that were used as CRNs with data consumers on them. Data providers were generated from outside machines and fed into the cluster so that data generation would not alter the state and load of each machine. Tests involved between one and eight data providers; 1,000 and 8,000 tuples per second input rates; two and 20 data consumers; and two and eight CRNs. All CP-ABE functionality was provided by the Advanced Crypto Software Collection library [10], and the HOM key size was 1024 bytes.

5.2 Workload Description

For our experiments, we used simulated Twitter-like data from a workload generator which provided control over distribution and frequency of keywords as input data. This generator is capable of forming both text and numerical data. Values can be controlled in either a fine or course-grained fashion. Fine-grained control allows us to define a small dictionary and assign a distribution over the occurrence of each value in the dictionary. Course-grained support simply sets a desired amount of data and desired selectivities (as to control selectivity for windowed and one-shot queries). We chose not to the Linear Road [8] benchmark for two main reasons. First, adding encryption and policy changes to arbitrary values adds overheads to the actual benchmark and requires altering it, which could undermine the intentions behind the data and queries. Secondly, Linear Road requires compatibility with a traditional database system. In the PolyStream model, the database system may reside on the server (colocated with the data) which can leak data since it would be required to remain in plaintext. A system like CryptDB could be used in this regard, but that says nothing for the methods of PolyStream which focuses on data streams.

5.3 Overhead for Computation Functionality

To better understand the effect that each operator has on the overall throughput, we compared *unencrypted* versus *encrypted* processing using one encryption type. We also included a *Strawman* approach where all data is routed to the data consumer for processing under the RND encryption scheme.

Configuration One data provider with one stream distributed the data to a single CRN with a single data consumer. This data consumer posed one query to the stream corresponding to the given encryption scheme (e.g., DET encryption matched to equality select and OPE mapped to range queries). One field was encrypted for each operator. For equality queries, Range, and summation queries DET, OPE, and HOM were used, respectively.

Results (Figure 4 and Table 2) On average, deterministic encryption only incurs 12% overhead, whereas HOM decreases throughput by 49% on average. This large difference in HOM is attributed to its use of Homomorphic encryption, which involves costly homomorphic additions running on each CRN. We also evaluated a more secure scheme (implemented on the same system) in which RND encryption is used and all tuples are sent back to the data consumer for processing. This requires *every* tuple to be decrypted and the operation computed over the plaintext value. Since every tuple is encrypted, the overall cost of execution is hindered by the cost of decrypting each tuple before processing. The overhead incurred by each encryption scheme originates either from the encryption or the decryption phase of the algorithm. Table 2 shows exactly how much time is spent during each phase of encryption. Note that a summation for the HOM scheme itself takes on average .015ms, and the key size (modulus size) for HOM plays a significant role in its encryption and decryption time. It is also important to note that the system will always pay the encryption cost for every tuple, but may not pay the decryption cost for each tuple depending on the selectivities.

Takeaway Compared to an unmodified DSMS, PolyStream's overhead is a modest 28% in supporting access control on honest-but-curious CRNs. In contrast, the overhead of the state-of-the-art Streamforce [5] is 4,000x, according to the authors.

Mode	RND	OPE	DET	H-1024	H-2048	H-4096
Encrypt	8.2	13.1	12.5	18.1	70.2	151.8
Decrypt	8.2	13.2	12.3	12.9	21.6	36.5

Table 2: The encryption and decryption times (in ms) for each of the schemes used by our system (H-xxxx = HOM at that key size).

5.4 Effects on Latency per Encryption Type

To explore the perceived effect on waiting for a result based on an incoming tuple, this experiment compared the latency of PolyStream to that of the baseline Storm-based DDSMS without any encryption.

Configuration This experiment used only one EC2 small instance. One query was used to test each encryption type, and each query was simply a selection (i.e. on comparison) or addition wherein one addition or one comparison needed to be made. The input rate of tuples remained constant. Each query was tested five times with the average reported for 1,000 tuples per trial. Finally, experiments were carried out in succession with the same system setup and background. Results are reported in milliseconds (ms).

Results (Table 3) Table 3 shows the latencies for each type. The main differences between PolyStream and the baseline DSMS is in the decryption time on the data consumer. The actual computation on the CRN is roughly the same (with the exception of HOM) since

System	RND	OPE	DET	HOM
PolyStream	425	413	326	1,144
Baseline	356	357	308	485

Table 3: The latency (ms) of each encryption when used in a query.

Tuples/second	Encrypt	Decrypt	CP-ABE	Compute	Transmit	Idle
2,000	3.8	4.0	6.0	**41.2**	9.5	35.6
4,000	3.9	5.3	6.2	**61.7**	10.3	12.6
6,000	3.4	7.2	6.3	**69.0**	12.2	1.6
8,000	3.4	8.6	5.8	**76.2**	16.0	0.0

Table 4: The percentage of system time spent on a task based on the input rate. CP-ABE represents the time spent passing keys and managing attribute-based encryptions.

the operators are only comparing larger integers or strings. HOM, however, takes longer to compute since the integers are larger and require multiplication as opposed to simple summations. Note that the HOM latency is calculated as the arrival of the first tuple in a window until the time the resulting summation is outputted. For this experiment, the window size was five tuples.

Takeaway Summation or averaging queries incur larger delays due to the need for multiplying larger numbers (a costlier operation) to homomorphically sum tuples using the Paillier [28] scheme.

5.5 Total System Overview

Given that each encryption scheme yields an overhead, it is worth exploring exactly what percentage of system time is devoted to doing a given task. We consider six main tasks when examining where the system spends its time: encrypting, decrypting, attribute alterations (CP-ABE), computing, transmitting, and waiting.

Configuration The results are based on an hour-long simulation where over 40,000 tweets were generated, and 600 changes in policy were assigned. The worker nodes were in a wheel configuration (Figure 5) with each leaf sending data to a sink (a bolt which receives and deletes data) to emulate retransmission. We used a mixed query workload, consisting of equality, range, and summation queries (33% for each type). Since the workload depends largely on the input rate, results are given for different input rates. In the event that the machines became overwhelmed, a typical simple load shedding technique was used [31].

Results (Table 4) For all experiments, over 70% of the time was spent on computation or idling if the workload was light. The time spent on attribute alterations and the time spent encrypting stays relatively constant throughout the simulation. The system spends more time decrypting as the workload increases since more tuples are sent. The wait time of 0.0%, for the 8,000 tuples/sec case, indicated a system saturated with tuples and, as such, some tuples were dropped (4.9% of tuples).

Takeaway PolyStream spends on average 15-17% of the total time in encryption, decryption, and key management.

5.6 SP Frequency vs. Throughput

A change in policy can occur at any time. Next, we evaluate how the frequency of policy changes effects the overall latency.

Configuration We used two machines, each with two data consumers. The frequency of policy changes is determined by the frequency of inserting SPs into the stream. We compare against using ABE encryption for all tuples, similar to the state-of-the-art [5]. The number of attributes was fixed at five, with a mixed query workload of equality, range, and summation queries (33% each).

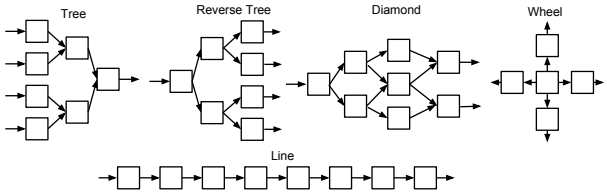


Figure 5: Configurations used to test how network topology affects PolyStream.

Results (Figure 6a) Results are depicted in Figure 6a. Note that the per-tuple ABE uses outsourced decryption. These experiments show that PolyStream was better than per-tuple ABE for all cases except the degenerate case of one SP per data tuple. PolyStream performed well when changes in policy are infrequent. It is clear that PolyStream outperforms the state-of-the-art [5] in even the simple case of one policy update for every two tuples, while providing more flexibility in submitting queries.

Takeaway Given a ratio of 1/100 (data tuples to SPs), PolyStream outperforms Streamforce by over 40x.

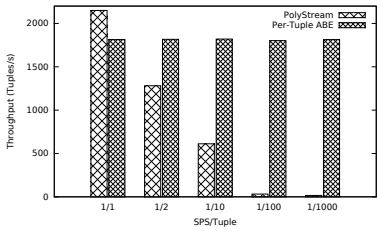
5.7 Tuple vs. Punctuation Level ABE

The implementation of the current state-of-the-art, Streamforce [5], uses decryption outsourcing techniques from Green et al. [17] to outsource decryption of Attribute-Based Encryptions to the cloud. Through the use of a transformation key, the server (CRN) is able to aid in decryption by doing most of the decryption, leaving only a small decryption operation to the data consumer. For *every tuple* selected by the system, however, a full attribute-based decryption must be done, which is costly regardless of whether or not it is done on a server. This means the number of ABE decryptions in Streamforce is large when compared to PolyStream which only uses ABE for policy updates (SPs). To test the effects of outsourcing attribute-based decryption to the cloud, we implemented the scheme used by Streamforce to compare our key distribution approach with their attribute-based approach.

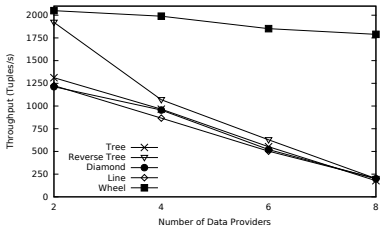
Configuration Streamforce used four different queries ranging from simple selections to summations. To compare their results with ours, the total decryption time is taken as the transformation time plus the decryption time performed on the data consumer. The total decryption time for PolyStream is simply the faster cryptographic scheme decryption time, which averages to 13.2 seconds, as mentioned above. Since PolyStream only uses ABE to share keys (i.e. only when a Security Punctuation is issued and processed), it does not pay the cost of ABE decryption on every tuple; instead, it only pays the cost once, as described above. One query was used, along with one stream on one machine. Note that the ABE decryption time depends on the number of attributes, so results are given for different numbers of attributes. Also, note that in this experiment the only comparison drawn between Streamforce and our work is Streamforce's use of ABE for each tuple. Streamforce relies on the data provider to do aggregates rather than the server, and the deterministic encryption and summations are the same as the ones used in PolyStream, so they were excluded.

Results (Figure 6d) Even with the smallest number of attributes, outsourced ABE is 4x slower than the PolyStream approach, and at one point it is nearly 550x slower depending on the number of attributes. These results are in line with initial results from Green et al. [17], which were on similar, yet better, hardware.

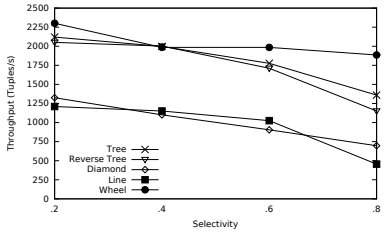
Takeaway By using ABE only for key management (i.e. not for every tuple), PolyStream incurs up to 550x less overhead per tuple than Streamforce.



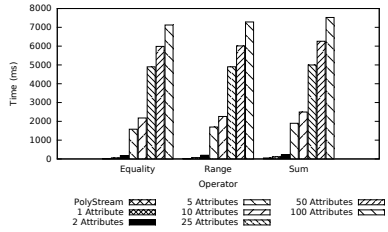
(a) Effect on average throughput by altering the frequency of Security Punctuations.



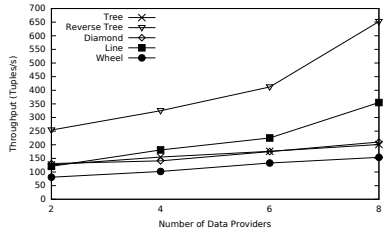
(b) Total throughput for increased load with selectivity .8, two clients per CRNs, and all encryption types used.



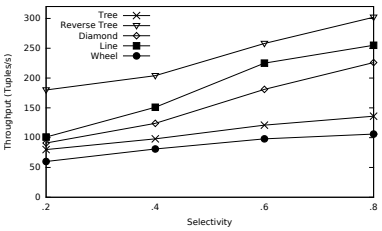
(c) Total throughput for increasing selectivity where there is one data source, two clients per CRN, with only DET used over selection queries.



(d) Outsourced ABE decryption for different operators with different numbers of attributes. Note that PolyStream decryption are included as the first column per set.



(e) Total latency for increased load with selectivity .8, two clients per CRNs, and all encryption types used.



(f) Total latency for increasing selectivity where there is one data source, two clients per CRN, with only DET used over selection queries.

Figure 6: Network Configuration, SPS frequencey, and Encryption Technique Effects on Throughput and Latency

5.8 Network Effect on Throughput & Latency

Configuration (Figure 5) Storm enables the user to describe the configuration of the network interconnecting the worker nodes. To better see how network connections affect the system, we tested five configurations with different input rates, data consumers, selectivities, and CRNs. These five configurations consisted of a tree, a reverse tree, a line, a diamond, and a wheel.

Throughput Results (Figures 6b, 6c) The first network experiment measured the throughput with respect to the workload. Each configuration had all of the worker nodes running. Figure 6b depicts the results. As the workload increases for each configuration, there is a corresponding drop in throughput. The wheel configuration is less affected as there is no single bottleneck whereas each other configuration has at least one bottleneck where multiple streams meet at a CRN. The throughput is not just a factor of the workload, it is also a factor of selectivity and the number of worker nodes. Only deterministic selection queries were used in this experiment. Figure 6c shows the effects of selectivity on throughput for each configuration. The results are similar to the increase in workload, but the trees have a higher throughput since they reduce the number of tuples at each stage due to changes in selectivity.

Latency Results (Figures 6e, 6f) Figure 6e shows that the reverse tree incurs the highest latency. Again, the output node becomes the bottleneck, causing delays to compound as the number of tuples increases. The wheel configuration preforms the best since there is no delay getting data consumers. Figure 6f shows the effects on latency when the selectivity of operatiors increases. Networks that reduce the number of CRNs as data flows tend to do worse as the workload increases. This verifies that PolyStream does not incur unnecessary overheads that would not appear otherwise.

Takeaway Network configurations have an impact on the latency and throughput of PolyStream since delays compound depending on the encryption types and selectivities.

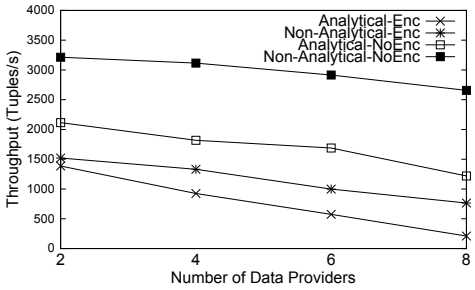


Figure 7: Effects of encrypted analytical workloads versus encrypted non-analytical workloads.

5.9 Overhead of Analytical Queries

Analytical queries can be more costly than regular queries when summation is involved. We explore these next.

Configuration For analytical queries, we used an equal mix of 100 range and summation queries. Range queries had a selectivity of 0.5. Ten queries were registered to each of ten data consumers who were assigned two per machine in a wheel pattern (see Figure 5). The same data was used for the non-analytical queries, but all query types were included to show how throughput was affected. Analytical queries were simply summations over a fixed window and filters over a fixed window, whereas non-analytical queries were equality filters and plain-text joins.

Results (Figure 7) Figure 7 shows the throughput for an analytical query-heavy workload and a non-analytical-query-heavy workload. Analytical queries must use the Paillier [28] encryption scheme, which requires large integer computations to be done on the server, resulting in the slowdown depicted in Figure 7.

Takeaway Analytical queries require multiplication of large numbers and will incur larger overheads than simpler queries.

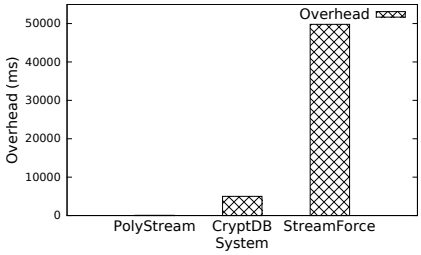


Figure 8: Comparisons between PolyStream, CryptDB (in a streaming environment), and Streamforce.

5.10 Encryption Overhead Comparisons

Configuration Here, we introduce CryptDB [29] as adapted for a streaming environment. CryptDB [29] and PolyStream utilize many of the same tools to accomplish their goals, although they were designed for very different system needs: CryptDB operates on traditional Database Management Systems, whereas PolyStream operates on DDSMSs. CryptDB's primary goal is not access control for all parties, but rather eliminating unwanted access by third-party storage systems by allowing computation over encrypted data on the untrusted third-party database. CryptDB utilizes specialized encryption techniques for allowing queries to operate on untrusted servers over encrypted data. Specifically, CryptDB employs Deterministic, Order-Preserving, Homomorphic, Specialty Search, Random, and Join encryption techniques to enable many different queries. Each technique leaks a different level of information (discussed in Section 4.3) but allows for different levels of functionality. These different techniques are structured in "Onions" in which the outer layer contains the most secure encryption technique. Removal of layers allows more functionality (i.e. going from RND to DET), but leaks some sensitive data.

When considered for use in a DDSMS, CryptDB encounters a few limitations. First, the data consumer no longer has control of the data source, meaning they do not control the encryption being used, or the accesses being given (including whether they themselves have access). This requires an online key management system as well as knowledge of what types of encryption are required for each potential data consumer, and an access control mechanism for different end users. In the system model described above, one data provider can have many data consumers digesting their data. Each data consumer may require a different level of encryption for processing.

We implemented a micro-benchmark to show the average overhead incurred by using onions in a streaming environment. This benchmark consisted of three onions (all those from CryptDB minus searches and joins) for a simple schema of four fields: Name, HeartRate, StepsTaken, and Glucose. Each field was onion-encrypted, resulting in 12 fields. Between 2 and 12 fields were chosen at random to be decrypted to a random level, for 10,000 tuples.

Results (Figure 8) The average overhead from decryption was 51.5ms per tuple for the stream adaptation of CryptDB. This means a DSMS that could handle 10,000 tuples per second would be reduced to 194 tuples per second, hindering the useful work being done by 98%, and causing an increase in encryption overhead of nearly 5,000%. These overheads and the need for an access control element limit the use of CryptDB in a streaming environment. PolyStream avoids these overheads by simply encrypting data at one level and by avoiding re-encryption. Also note that using CryptDB for a streaming application would cause greater overheads, due to a large number of insertions into the database and frequent query re-execution to get up to the date results. Both of these overheads are not explored here.

In addition to these overheads, recall that the encryption overhead for Streamforce causes a 4,000x slowdown on an unaltered system (as claimed in [5]). Our experiments from Section 5.6 show that a workload with just 5 attributes would incur at least 49,000% overhead for every tuple. Note also that from Section 5.6, PolyStream with a relatively low policy update rate can incur as little as 12% overhead attributed to encryption, but will average roughly 56%. These overheads are displayed in Figure 8.

Takeaway PolyStream incurs very little overhead versus the closest related work.

6. CONCLUSION

Modern data streaming applications, which separate the source of data from its eventual consumer, make it difficult for data providers to author and enforce effective access controls. Access control frameworks for DDSMSs must allow data providers the ability to easily author policies, while supporting policy changes over time as the system evolves. To ensure data confidentiality from (potentially) untrusted third-party compute nodes, these policies should be enforced cryptographically, which requires an online key management system. A key challenge is enforcing these protections without incurring undue performance or utility degradation.

In this paper, we introduced PolyStream to address the above problems via cryptographically enforced access controls over streaming data. Through the use of various cryptographic schemes, PolyStream allows untrusted third-party infrastructure to compute on encrypted data, allowing in-network query processing and access control enforcement with minimal impact on system utility. PolyStream uses a combination of security punctuations, attribute-based encryption, and hybrid cryptography to enable flexible (ABAC) access control policy management and key distribution with minimal overheads. We have performed an extensive experimental evaluation on a real system (using Storm) and showed that PolyStream provides an excellent tradeoff between confidentiality and performance. Compared to the state-of-the-art, PolyStream performed up to 550x faster in our evaluation.

7. ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers and our paper shepherd, Sherman Chow, for their helpful and constructive feedback. This work was partially supported by the National Science Foundation under awards CNS-1228697, CAREER CNS-1253204, CAREER IIS-0746696, and OIA-1028162.

8. REFERENCES

[1] D. Abadi et al. The design of the borealis stream processing engine. In *CIDR*, 2005.

[2] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB Journal-The International Journal on Very Large Data Bases*, 12(2):120–139, 2003.

[3] R. Adaikkalavan and T. Perez. Secure shared continuous query processing. In *ACM SAC*, pages 1000–1005, 2011.

[4] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. Millwheel: fault-tolerant stream processing at internet scale. *Proceedings of the VLDB Endowment*, 6(11):1033–1044, 2013.

[5] D. T. T. Anh and A. Datta. Streamforce: outsourcing access control enforcement for stream data to the clouds. In *Proceedings of the 4th ACM conference on Data and application security and privacy*, pages 13–24, 2014.

[6] L. Aniello, R. Baldoni, and L. Querzoni. Adaptive online scheduling in storm. In *Proceedings of the 7th ACM DEBS*, pages 207–218. ACM, 2013.

[7] A. Arasu, S. Babu, and J. Widom. The cql continuous query language: semantic foundations and query execution. *The VLDB Journal—The International Journal on Very Large Data Bases*, 15(2):121–142, 2006.

[8] A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. S. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts. Linear road: a stream data management benchmark. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pages 480–491. VLDB Endowment, 2004.

[9] S. Babu and J. Widom. Continuous queries over data streams. *ACM Sigmod Record*, 30(3):109–120, 2001.

[10] J. Benthencourt, A. Sahai, and B. Waters. Advanced crypto software collection: Ciphertext-policy attribute-based encryption. 2011.

[11] A. Boldyreva, N. Chenette, Y. Lee, and A. O'Neill. Order-preserving symmetric encryption. In *Eurocrypt*, pages 224–241. Springer, 2009.

[12] A. Boldyreva, N. Chenette, and A. O'Neill. Order-preserving encryption revisited: Improved security analysis and alternative solutions. In *Advances in Cryptology–CRYPTO 2011*, pages 578–595. Springer, 2011.

[13] B. Carminati, E. Ferrari, J. Cao, and K. L. Tan. A framework to enforce access control over data streams. *ACM Transactions on Information and System Security (TISSEC)*, 13(3):28, 2010.

[14] B. Carminati, E. Ferrari, and K. L. Tan. Enforcing access control over data streams. In *Proceedings of the 12th ACM symposium on Access control models and technologies*, pages 21–30, 2007.

[15] B. Carminati, E. Ferrari, and K. L. Tan. Specifying access control policies on data streams. In *Advances in Databases: Concepts, Systems and Applications*, pages 410–421. Springer, 2007.

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

[17] M. Green, S. Hohenberger, and B. Waters. Outsourcing the decryption of abe ciphertexts. In *USENIX Security Symposium*, 2011.

[18] S. Halevi and P. Rogaway. A tweakable enciphering mode. In *CRYPTO 2003*, pages 482–499. Springer, 2003.

[19] J. Hur and D. K. Noh. Attribute-based access control with efficient revocation in data outsourcing systems. *Parallel and Distributed Systems, IEEE Transactions on*, 22(7):1214–1221, 2011.

[20] H. V. Jagadish et al. Big data and its technical challenges. *Communications of the ACM*, 57(7):86–94, Jul 2014.

[21] S. Jahid, P. Mittal, and N. Borisov. Easier: Encryption-based access control in social networks with efficient revocation. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 411–415. ACM, 2011.

[22] X. Jin, R. Krishnan, and R. Sandhu. A unified attribute-based access control model covering dac, mac and rbac. In *Data and applications security and privacy XXVI*, pages 41–55. Springer, 2012.

[23] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja. Twitter heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 239–250. ACM, 2015.

[24] W. Lindner and J. Meier. Securing the borealis data stream engine. In *Database Engineering and Applications Symposium, 2006. IDEAS'06. 10th International*, pages 137–147. IEEE, 2006.

[25] R. Nehme, E. A. Rundensteiner, and E. Bertino. A security punctuation framework for enforcing access control on streaming data. In *IEEE 24th International Conference on Data Engineering (ICDE)*, pages 406–415, 2008.

[26] R. V. Nehme, H.-S. Lim, and E. Bertino. Fence: Continuous access control enforcement in dynamic data stream environments. In *Proceedings of the third ACM conference on Data and application security and privacy*, pages 243–254, 2013.

[27] W. S. Ng, H. Wu, W. Wu, S. Xiang, and K.-L. Tan. Privacy preservation in streaming data collection. In *Proceedings of the 2012 IEEE 18th International Conference on Parallel and Distributed Systems*, pages 810–815, 2012.

[28] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Proc. of Eurocrypt*, pages 223–238, 1999.

[29] R. A. Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan. Cryptdb: protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 85–100, 2011.

[30] StormProject. Storm: Distributed and fault-tolerant realtime computation. http://storm.incubator.apache.org/documentation/Home.html, 2014.

[31] N. Tatbul, U. Çetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker. Load shedding in a data stream manager. In *Proceedings of the 29th international conference on Very large data bases-Volume 29*, pages 309–320, 2003.

[32] S. Tu, M. F. Kaashoek, S. Madden, and N. Zeldovich. Processing analytical queries over encrypted data. In *Proceedings of the 39th international conference on Very Large Data Bases*, pages 289–300, 2013.

[33] B. Wang, M. Li, S. S. Chow, and H. Li. A tale of two clouds: Computing on data encrypted under multiple keys. In *Communications and Network Security (CNS), 2014 IEEE Conference on*, pages 337–345. IEEE, 2014.

[34] S. Yu, C. Wang, K. Ren, and W. Lou. Attribute based data sharing with attribute revocation. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, pages 261–270. ACM, 2010.

[35] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, volume 10, page 10, 2010.