Data Science and Engineering
https://doi.org/10.1007/s41019-019-00100-5

Latency-Aware Secure Elastic Stream Processing with Homomorphic Encryption

Arosha Rodrigo[1] · Miyuru Dayarathna[2] · Sanath Jayasena[1]

Received: 18 May 2019 / Revised: 20 August 2019 / Accepted: 22 August 2019

Abstract
Increasingly organizations are elastically scaling their stream processing applications into the infrastructure as a service clouds. However, state-of-the-art approaches for elastic stream processing do not consider the potential threats of exposing their data to third parties in cloud environments. We present the design and implementation of an Elastic Switching Mechanism for data stream processing which is based on homomorphic encryption (HomoESM). The HomoESM not only elastically scales data stream processing applications into public clouds but also preserves the privacy of such applications. Using a real-world test setup, which includes an E-mail Filter benchmark and a Web server access log processor benchmark (EDGAR), we demonstrate the effectiveness of our approach. Experiments on Amazon EC2 indicate that the proposed approach for homomorphic encryption provides a significant result which is 10–17% improvement in average latency in the case of E-mail Filter benchmark and EDGAR benchmark, respectively. Furthermore, EDGAR add/subtract operations, multiplication, and comparison operations showed up to 6.13%, 7.81%, and 26.17% average latency improvements, respectively. Finally, we evaluate the potential of scaling the homomorphic stream processor in the public cloud. These results indicate the potential for real-world deployments of secure elastic data stream processing applications.

Keywords Cloud computing · Elastic data stream processing · Compressed event processing · Data compression · IaaS · System sizing and capacity planning

1 Introduction

Data stream processing has become one of the main paradigms for data analytics in recent times [6, 7]. Various different applications of stream processing can be found in different domains such as transportation [20], telecommunications [8, 29], disaster management [10], and environmental monitoring [16]. The economies of scale introduced by cloud computing platforms consistently indicate the importance of migrating stream processing applications to cloud. This has resulted in data stream processors which run as managed cloud services (e.g., [13, 18]) as well as hybrid cloud services (e.g., Striim [28]).

It is a common observation that data stream processors face resource limitations during their operation due to unexpected loads [3, 9]. There are multiple possible solutions for these issues. Elastically scaling into an external cluster [19, 25], load shedding, approximate query processing [24], etc., are some examples. Out of these, elastic scaling has become a key choice because approaches such as load shedding and approximate computing have to compromise accuracy which is not accepted by certain categories of applications. The previous work has been there which used data compression techniques to optimize the network connection between private and public clouds [25]. However, current elastic scaling mechanisms for stream processing do not consider the problem of preserving the privacy of the data sent to the public cloud.

Preserving the privacy of stream processing becomes a key question to be answered when scaling into a public cloud. Sending the data unencrypted to the server definitely

✉ Miyuru Dayarathna
miyurud@wso2.com

Arosha Rodrigo
uom.arosha@gmail.com

Sanath Jayasena
sanath@cse.mrt.ac.lk

1 Department of Computer Science and Engineering, University of Moratuwa, Moratuwa, Sri Lanka

2 WSO2, Inc., 787 Castro Street, Mountain View, CA 94041, USA

exposes them to prying eyes of the eavesdroppers. Sending data encrypted over the network and decrypting them to get original values at the server may also expose sensitive information. Multiple works have recently been conducted on privacy-preserving data stream mining. Privacy of patient health information has been a serious issue in recent times [23]. Fully homomorphic encryption (FHE) has been introduced as a solution [12]. FHE is an advanced encryption technique that allows data to be stored and processed in encrypted form. This gives cloud service providers the opportunity for hosting and processing data without even knowing what the data are handling. However, current FHE techniques are computationally expensive needing excessive space for keys and ciphertexts. However, it has been shown with some experiments done with HElib [15] (an FHE library) that it is practical to implement some basic applications such as streaming sensor data to the cloud and comparing the values to a threshold.

In our previous work, we present elastic scaling in a private/public cloud (i.e., hybrid cloud) scenario with privacy-preserving data stream processing [26]. We design and implement a privacy-preserving Elastic Switching Mechanism (HomoESM) over private/public cloud system. Homomorphic encryption scheme of HElib has been used on top of this switching mechanism for compressing the data sent from private cloud to public cloud. Application logic at the private cloud is implemented with Siddhi event processing engine [20]. We designed and developed two real-world data stream processing benchmarks called E-mail Processor and HTTP Log Processor (EDGAR benchmark) during the evaluation of the proposed approach.

In this paper, we extend our privacy-preserving data stream processing mechanism (HomoESM) with significant additional features such as the support for homomorphic multiplication operations [26]. Furthermore, we extend our HomoESM mechanism to elastic scaling with multiple VMs running in public cloud and report results. Moreover, we demonstrate that latency improvement is consistent across multiple different experiment rounds.

Using multiple experiments on real-world system setup with the stream processing benchmarks, we demonstrate the effectiveness of our approach for elastic switching-based privacy-preserving stream processing. We observe that homomorphic encryption provides significant results. It provides 10–17% of improvement in average latency in the case of E-mail Filter benchmark. EDGAR add/subtract, multiplication, and comparison operations showed 6.13%, 7.81%, and 26.17% of average latency improvements, respectively. HomoESM is the first known data stream processor which does privacy-preserving data stream processing in hybrid cloud scenarios effectively. We have released HomoESM and the benchmark codes as open source software.[1,2,3] Specifically, the contributions of our work can be listed as follows.

- *Enhanced privacy-preserving Elastic Switching Mechanism (HomoESM)* We design and develop a mechanism for conducting elastic scaling of stream processing queries over private/public cloud in a privacy-preserving manner. We enhance this to operate in public cloud with multiple virtual machine (VM) instances.
- *Homomorphic multiplication operation* We improved the stream processing functionality of HomoESM by implementing homomorphic multiplication.
- *Optimization of homomorphic operations* We optimized several homomorphic evaluation schemes such as equality and less than/greater than comparison. We also do data batching-based optimizations.
- *Evaluation* We evaluate the proposed approaches by implementing them on real-world systems. We compare the performance of homomorphic add/subtract operations as well as multiplication operations. We also evaluate the criteria for scaling into multiple VMs in public cloud.

The paper is organized as follows. Next, we provide the related work in Sect. 2. We give the brief overview to the technologies used in this paper in Sect. 3. We provide the details of system design in Sect. 4 and implementation of the HomoESM in Sect. 5. The evaluation details are provided in Sect. 6. We make a discussion of the results in Sect. 7. We provide the conclusions in Sect. 8.

2 Related Work

There have been multiple previous works on elastic scaling of event processing systems in cloud environments.

Cloud computing allows for realizing an elastic stream computing service, by dynamically adjusting used resources to the current conditions. Hummer et al. discussed how elastic computing of data streams can be achieved on top of cloud computing [17]. They mentioned that the most obvious form of elasticity is to scale with the input data rate and the complexity of operations (acquiring new resources when needed and releasing resources when possible). However, most operators in stream computing are stateful and cannot be easily split up or migrated (e.g., window queries need to

[1] https://github.com/arosharodrigo/event-publisher.

[2] https://github.com/arosharodrigo/statistics-collector.

[3] https://github.com/arosharodrigo/simple-siddhi-server.

store the past sequence of events). In HomoESM, we handle this type of queries by query switching.

Stormy is a system developed to evaluate the with ‘stream processing as service’ concept [22]. The idea was to build a distributed stream processing service using techniques used in cloud data storage systems. Stormy is built with scalability, elasticity, and multitenancy in mind to fit in the cloud environment. They have used distributed hash tables (DHTs) to build their solution. They have used DHTs to distribute the queries among multiple nodes and to route events from one query to another. Stormy builds a public streaming service where users can add new streams on demand. One of the main limitations in Stormy is it assumes that a query can be completely executed on one node. Hence, Stormy is unable to deal with streams for which the incoming event rate exceeds the capacity of a node. We address this issue in our work via the concept of data switching of HomoESM.

Cervino et al. [3] try to solve the problem of providing a resource provisioning mechanism to overcome inherent deficiencies of cloud infrastructure. They have conducted some experiments on Amazon EC2 to investigate the problems that might affect badly a stream processing system. They have come up with an algorithm to scale up/down the number of VMs (or EC2 instances) based solely on the input stream rate. The goal is to keep the system with a given latency and throughput for varying loads by adaptively provisioning VMs for streaming system to scale up/down. However, none of the above-mentioned works have investigated on reducing the amount of data sent to public clouds in such elastic scheduling scenarios. In this work, we address this issue.

Data stream compression has been studied in the field of data mining. Cuzzocrea et al. have conducted research on a lossy compression method for efficient OLAP [4] over data streams. Their compression method exploits semantics of the reference application and drives the compression process by means of the with ‘degree of interestingness.’ The goal of this work was to develop a methodology and required data structures to enable summarization of the incoming data stream. However, the proposed methodology trades off accuracy and precision for the reduced size.

Dai et al. [5] have implemented homomorphic encryption library on graphic processing unit (GPU) to accelerate computations in homomorphic level. As GPUs are more compute-intensive, they show 51 times speedup on homomorphic sorting algorithm when compared to the previous implementation. Although computation-wise it gives better speedup, when encrypting a Java String field, its length goes more than 400 KB which is too large to be sent over a public network. Hence, we used HElib as the homomorphic encryption library in our work.

Intel has included a special module in CPU, named *Software Guard eXtension (SGX)*, with its sixth generation Core i5, i7, and Xeon processors [27]. SGX reduces the trusted computing base (TCB) to a minimal set of trusted code (programmed by the programmer) and the SGX processor. Shaon et al. developed a generic framework for secure data analytics in an untrusted cloud setup with both single-user and multiuser settings [27]. Furthermore, they proposed BigMatrix which is an abstraction for handling large matrix operations in a data oblivious manner to support vectorizations. Their work is tailored for data analytics tasks using vectorized computations and optimal matrix-based operations. However, in this work HomoESM conducts stream processing which is different from the batch processing done by BigMatrix.

3 Overview

In this section, we provide a brief description of WSO2 Stream Processor which is the stream processing engine used for implementing our HomoESM. Then, we discuss about existing ESM and furthermore give introduction to homomorphic encryption concept and available libraries.

3.1 Overview of WSO2 Stream Processor

WSO2 Stream Processor (WSO2 SP) is a lightweight, easy-to-use, stream processing engine. In our work, we are using Siddhi library which is a component of the WSO2 Stream Processor [32]. It is available as open source software under the Apache Software License v2.0 [31]. WSO2 SP lets users provide queries using an SQL-like query language in order to get notifications on interesting real-time events, where it will listen to incoming data streams and generate new events when the conditions given in those queries are met by correlating the incoming data streams.

WSO2 SP uses a SQL-like Event Query Language to describe queries. For example, the following query detects the number of taxis dropped off in each cell in the last 15 min [20].

```
from Trip #window.time (15 min )
select count (medallion) as count group by cellId
insert into OutputStream
```

Listing 1 EmailFilter condition.

3.2 Elastic Switching Mechanism

The Elastic Switching Mechanism (ESM) [25] is designed to operate stream processing engines between private and public cloud environments as shown in Fig. 1. Basic idea is to have on-demand public SP engine according to the input load. This mechanism is able to maintain good QoS metrics as it can automatically scale for additional resources when required. ESM will route data between private and public stream processing engines with taking care of a QoS parameter configured by user. QoS measurements need to be taken at receiver component of ESM end, and publisher component will check for QoS level to take the decision of routing data to public stream processing engine. Current implementation will look for a pre-configured latency value as the QoS parameter.

3.3 Homomorphic Encryption

Homomorphic encryption is a type of encryption that allows computation on ciphertexts. It generates an encrypted result. When decrypted, the result matches with the result of the operations as if they had been performed on the plaintext. The purpose of homomorphic encryption is to allow computation on encrypted data [2]. Therefore, homomorphic encryption allows complex mathematical operations to be performed on encrypted data without compromising the privacy.

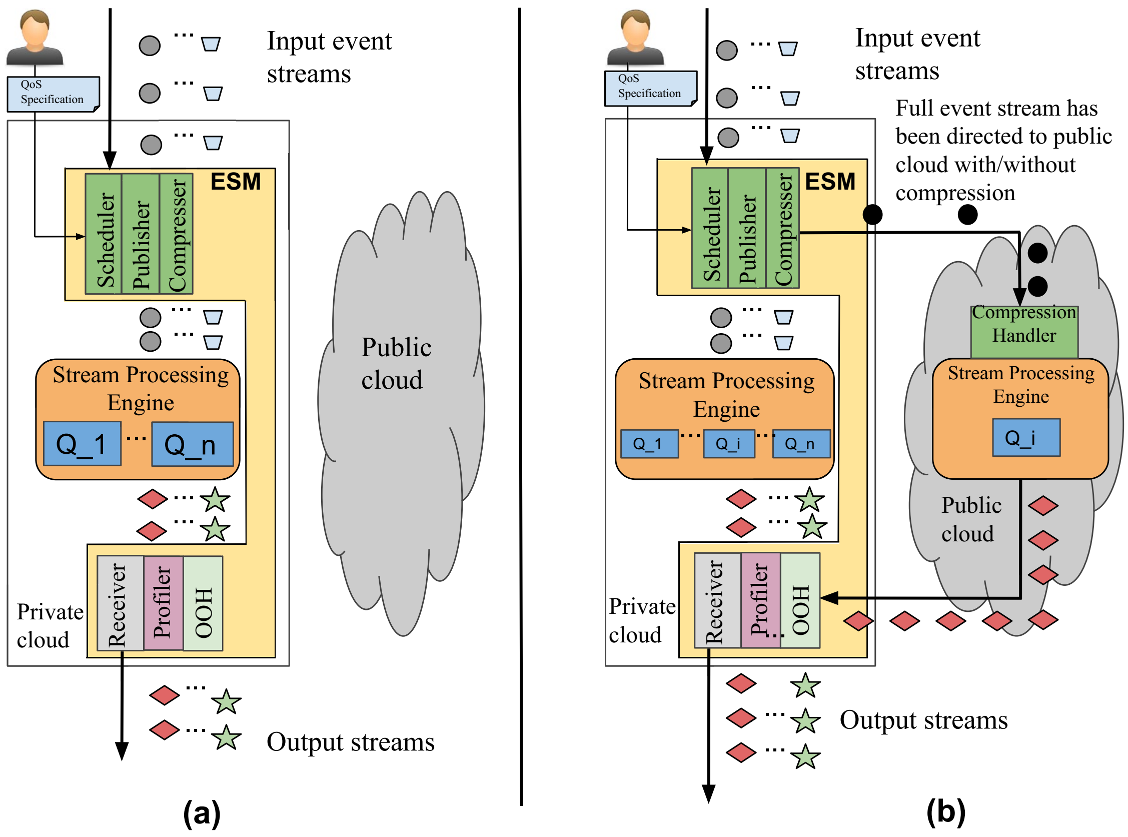


Fig. 1 Approach for elastic compressed complex event processing. System operation with single query switched to public cloud with data switching. **a** Private cloud-only mode of operation. **b** Hybrid cloud mode of operation with data switching and compression

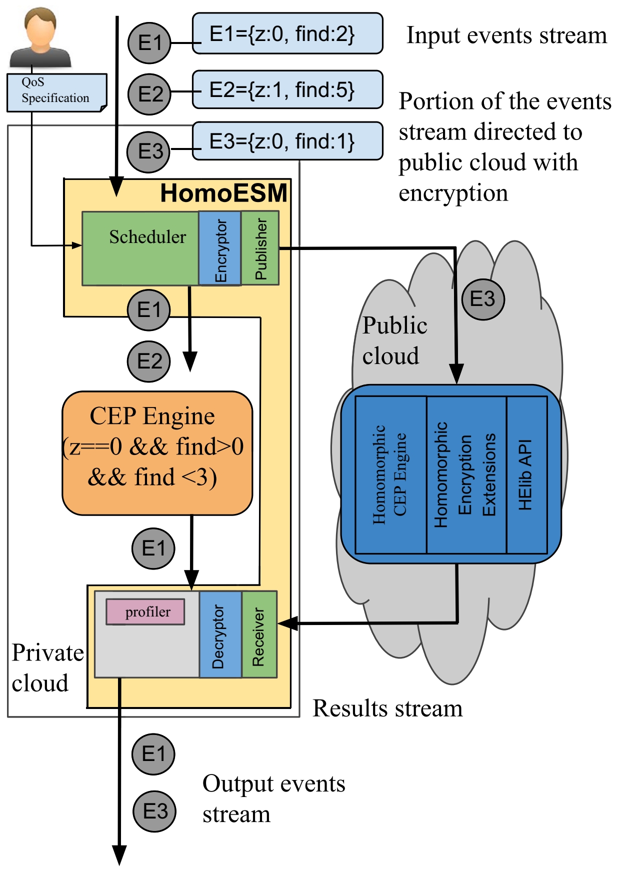


Fig. 2 System architecture of homomorphic encryption-based ESM (HomoESM) with an example of how comparison operation has been conducted

There are two main homomorphic encryption schemes: partially homomorphic encryption and fully homomorphic encryption (FHE). FHE supports arbitrary computation on ciphertexts and is far more powerful while partially homomorphic encryption supports limited computations. Fully homomorphic cryptosystems have great practical implications in the outsourcing of private computations in the context of cloud computing.

There are several implementations of homomorphic encryption. CUDA Homomorphic Encryption Library (cuHE) [5] is a GPU-accelerated library for homomorphic encryption (HE) schemes and homomorphic algorithms defined over polynomial rings.

Another popular implementation of homomorphic encryption is HElib. This library is open source on GitHub and written in C++ [14]. Unlike some earlier HE schemes, HElib uses a SIMD-like optimization known as ciphertext packing. As a result, each individual ciphertext element with which one can perform a computation (addition or multiplication) is conceptually a vector of encrypted plaintext integrals. HElib is particularly effective with problems that can benefit from some level of parallel computation. The size of this vector decides according to the settings when to initialize the HElib. HElib supports multithreaded environment, and we need to enable that feature while we install HElib on a system. It provides low-level routines such as set, add, multiply and shift. These are the reasons for why we choose HElib over other homomorphic encryption libraries to implement HomoESM.

4 System Design

In this section, we first describe the architecture of HomoESM and then describe the switching functions which determine when to start sending data to public cloud.

The HomoESM architecture is shown in Fig. 2. The components highlighted in the dark blue color correspond to components which directly implement privacy-preserving stream processing functionality.

Figure 2 shows an example scenario of comparison operation has been implemented. There are three events $E1$, $E2$, and $E3$ where $E1$ and $E3$ satisfy the stated conditions.

Table 1 Notation

Notation	Description
t	Unit time slot
L_t	Average latency measured at the receiver component of the HomoESM during the time slot t
L_s	VM Startup threshold latency. When the average latency exceeds this value, the HomoESM decides to initiate the VM start up process
L_d	Data switching threshold latency. When $L > L_d$, the HomoESM starts sending data to public cloud
τ	Tolerance period. After the unit timeslot (t) elapses, the HomoESM waits for additional τ period before it initiates the VM startup process. In the current implementation of the ESM, τ is set equal to t
L_p	Private cloud threshold latency. At least L_p amount of latency needs to be present in the private cloud for a VM to be kept running in the next unit time slot
D_t	Total amount of data received by the VM from private cloud during the time slot t
D_s	Threshold for total amount of data received by the VM from private cloud during the time slot t

However, $E2$ does not satisfy this condition. Hence, it gets filtered out.

The HomoESM Scheduler collects events from the Plain Event Queue according to the configured frequency and the timestamp field on the event. Then, it routes the events into the private publishing thread pool and to the public publishing queue, according to the load transfer percentage and the threshold values.

Receiver receives events from both private and public Siddhi. If the event is from the private Siddhi, it is sent to the Profiler. If the event is not a composite event, it is directed to the 'Composite Event Decode Worker' threads located inside the Decryptor which basically performs the decryption function. Finally, all the streams which go out from HomoESM run through Profiler which conducts the latency measurements.

In this paper, we use the same switching functions described in [25] for triggering and stopping VMs and sending data to public cloud (see Eqs. 1 and 2). Here, $\phi_{\mathrm{VM}}(t)$ is the binary switching function for a single VM and t is the time period of interest. L_{t-1} and D_{t-1} are the latency and data rate values measured in the previous time period. A time period of τ has to be elapsed in order for the VM startup process to trigger. The symbols used in the two equations are shown in Table 1.

$$\phi_{\mathrm{VM}}(t) = \begin{cases} 1, & L_{t-1} \geq L_{\mathrm{s}}, \tau \text{ has elapsed.} \\ 0, & D_{t-1} < D_{\mathrm{s}}, L_{t-1} < L_{\mathrm{p}} \quad \text{Otherwise,} \end{cases} \quad (1)$$

$$\phi_{\mathrm{data}}(t) = \begin{cases} 1, & \phi_{\mathrm{VM}}(t-1) = 1, L_{t-1} \geq L_{\mathrm{d}}, L_{\mathrm{s}} > L_{\mathrm{d}} \\ 0 & \text{Otherwise,} \end{cases} \quad (2)$$

ESM needs to take decisions on following three main scenarios,

When to start public VM Average latency measured for the last period at receiver (L_{t-1}) should be greater than VM startup threshold latency (L_{s}), and tolerance period (τ) needs to be elapsed.

When to stop public VM We do not switch off the VMs just because the charging period elapses. The decision is taken at the end of charging period if the following two conditions are satisfied.

- Data sent to public VM within the last period (D_{t-1}) should be less than threshold amount of data sent to public VM for a period (D_{s})

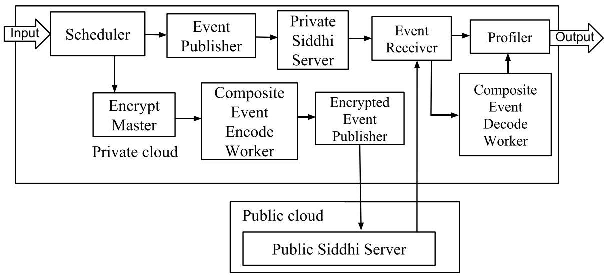


Fig. 3 Main components of HomoESM

- Average latency measured for the last period at receiver (L_{t-1}) should be less than private cloud threshold latency (L_{p})

When to send data to public VM

- Public VM should be up and running
- VM Startup threshold latency (L_{s}) should be greater than data switching threshold latency (L_{d}). Note that this condition is always true and it is maintained by ESM initial configurations.
- Average latency measured for the last period at receiver (L_{t-1}) should be greater than data switching threshold latency (L_{d}).

5 Implementation

First, we describe the implementation details of HomoESM in Sect. 5.1, and we describe the benchmark implementations in Sects. 5.2, 5.3, 5.4, and 5.5.

5.1 Implementation of HomoESM

We have developed the HomoESM on top of the WSO2 Stream Processor (WSO2 SP) software stack. As we described earlier, WSO2 SP internally uses Siddhi which is a complex event processing library [20]. Siddhi feature of WSO2 SP lets users run queries using an SQL-like query language in order to get notifications on interesting real-time events.

High-level view of the system implementation is shown in Fig. 3. Input events are received by the 'Event Publisher.' Java objects are created for each incoming event and put into a queue. Event Publisher thread picks those Java objects from the queue according to the configured period. Next, it evaluates whether the picked event needs to be sent to the private or the public Siddhi server, according to the configured load transfer percentage and threshold values. If that event needs to be sent to private Siddhi, it will mark the time and delegate the event into a thread pool which handles

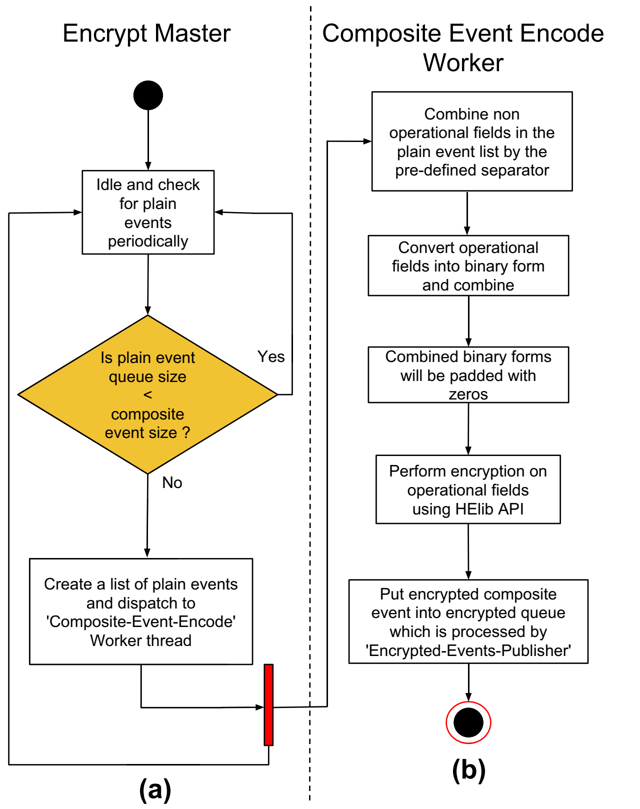

Encrypt Master
Composite Event Encode Worker
Idle and check for plain events periodically
Is plain event queue size < composite event size ?
Yes
No
Create a list of plain events and dispatch to 'Composite-Event-Encode' Worker thread
(a)
Combine non operational fields in the plain event list by the pre-defined separator
Convert operational fields into binary form and combine
Combined binary forms will be padded with zeros
Perform encryption on operational fields using HElib API
Put encrypted composite event into encrypted queue which is processed by 'Encrypted-Events-Publisher'
(b)

Fig. 4 Data encryption and the composite event creation process at the private Siddhi server. **a** Encrypt Master thread **b** composite Event Encode Worker thread

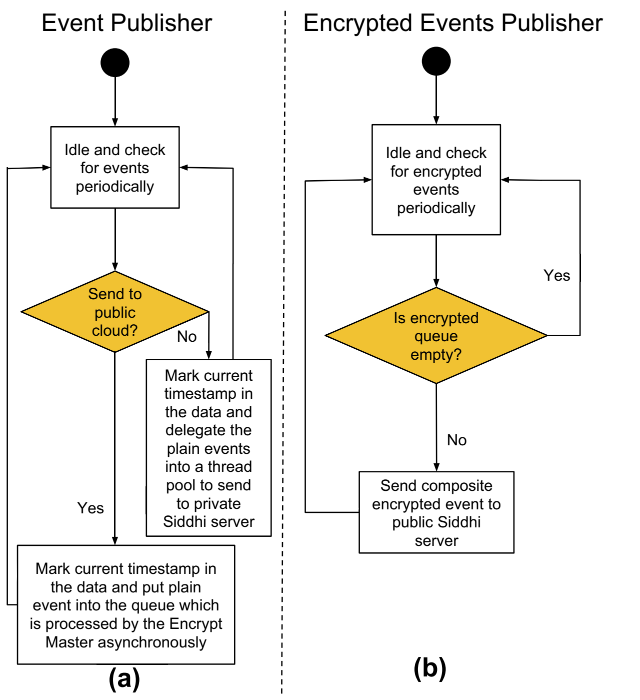

Event Publisher
Encrypted Events Publisher
Idle and check for events periodically
Send to public cloud?
No
Yes
Mark current timestamp in the data and delegate the plain events into a thread pool to send to private Siddhi server
Mark current timestamp in the data and put plain event into the queue which is processed by the Encrypt Master asynchronously
(a)
Idle and check for encrypted events periodically
Is encrypted queue empty?
Yes
No
Send composite encrypted event to public Siddhi server
(b)

Fig. 5 Operation of the Event Publisher and the Encrypted Events Publisher components. **a** Event Publisher, **b** Encrypted Events Publisher

sending to private Siddhi. If that event needs to be sent to public Siddhi, it will mark the time and be put into the queue which is processed by the Encrypt Master asynchronously.

Encrypt Master thread (see Fig. 4a) periodically checks a queue which keeps the events required to be sent to public cloud. The queue is maintained by the 'Event Publisher' (see Fig. 5a). If that queue size is greater than or equal to composite event size, it will create a list of events equal to the composite event size. Next, it delegates the event encryption and composite event creation task to the 'Composite Event Encode Worker' (see Fig. 4b).

Composite Event Encode Worker is a thread pool which handles event encryptions and composite event creations. First, it combines nonoperational fields of each plain event in the list by the pre-defined separator. Then, it converts operational fields into binary form and combines them together. Next, it pads the operational fields with zeros in order to encrypt using HElib API. Finally, it performs encryption on those operational fields and puts the newly created composite event into a queue which is processed by the 'Encrypted Events Publisher' thread (see Fig. 5b).

Firing events into the public VM is done asynchronously. Decision of how many events are sent to the public Siddhi server was taken according to the percentage we have configured initially. But the public Siddhi server's publishing flow has max limit of 1500 TPS (tuples per second). If the Event Publisher receives more than the max TPS, the events are routed back into the private Siddhi server's VM.

'Encrypted Events Publisher' thread periodically checks for encrypted events in the encrypted queue which is put by the 'Composite Event Encode Worker' at the end of the composite event creation and encryption process (see Fig. 4b). First, it combines nonoperational fields of each plain event in the list by the pre-defined separator. If there are encrypted events, it will pick those at once and send them to public Siddhi server. The encryptor module batches events into composite events and encrypts each composite message using homomorphic encryption. The encrypted events are sent to the public cloud where Homomorphic CEP Engine module conducts the evaluation.

We encrypt operand(s) and come up with composite operand field(s) in each HE function initially, in order to perform HE operations on operational fields in composite event. For example, in the case of the E-mail Filter benchmark, at the Homomorphic CEP Engine which supports homomorphic evaluations, initially it converts the constant operand into an integer (int) buffer with size 40 with a necessary 0 padding. Then, it replicates the integer buffer ten times and encrypts using HElib [14]. Finally, the encrypted value and

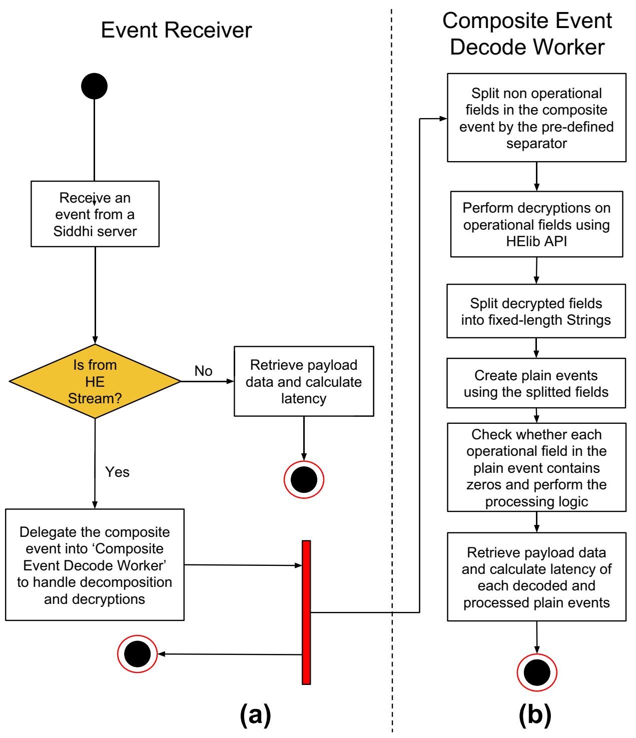

Event Receiver
Receive an event from a Siddhi server
Is from HE Stream?
No
Retrieve payload data and calculate latency
Yes
Delegate the composite event into 'Composite Event Decode Worker' to handle decomposition and decryptions
(a)
Composite Event Decode Worker
Split non operational fields in the composite event by the pre-defined separator
Perform decryptions on operational fields using HElib API
Split decrypted fields into fixed-length Strings
Create plain events using the splitted fields
Check whether each operational field in the plain event contains zeros and perform the processing logic
Retrieve payload data and calculate latency of each decoded and processed plain events
(b)

Fig. 6 Event receiving, decomposition, and decryption processes

the relevant field in the composite event are used for HElib's relevant (e.g., comparison, addition, subtraction, multiplication, etc.) operation homomorphically. The result is replaced with the relevant field in the composite event and is sent to the receiver without any decryption.

The received encrypted information is decrypted and decomposed to extract the relevant plain events. The latency measurement happens at the end of this flow. 'Event Receiver' thread checks whether the event received from the Siddhi server is encrypted with homomorphic encryption. If so, it delegates composite event into 'Composite Event Decode Worker.' If not, it will read payload data and calculate the latency (see Fig. 6a).

After receiving a composite event from the Event Receiver, the Composite Event Decode Worker handles all decompositions and decryptions of the composite event (see Fig. 6b). It first splits nonoperational fields in the composite event by the pre-defined separator. Second, it performs decryption on the operational fields using HElib API and splits the decrypted fields into fixed-length strings. Then, it creates plain events using the split fields. Next, it checks each operational field in the plain event to see whether it contains zeros and then processes the events. Finally, it calculates the latency of the decoded events.

Note that we implement the homomorphic comparison of values following the work by Togan et al. [30]. We have used Togan et al.'s methodology for implementing homomorphic comparison operation because it provides an $O(\log_2(n))$ solution which evaluates the comparison. Furthermore, according to the authors, their approach provides good results compared to other previous approaches [1]. For two single-bit numbers with x and y, Togan et al. [30] have shown that the following equations (see Eq. 3) will satisfy greater-than and equal operations, respectively.

$$\begin{aligned} &x > y \Leftrightarrow xy + x = 1 \\ &x = y \Leftrightarrow x + y + 1 = 1 \end{aligned} \tag{3}$$

Togan et al. have created comparison functions for n-bit numbers using divide and conquer methodology. In our case, we derived two-bit number comparisons as follows. x_1x_0 and y_1y_0 are the two numbers with two bits (see Eq. 4). Here, every '+' operation is for XOR gate operation and every '$\cdot$' operator is for AND gate operation.

$$\begin{aligned} &x_1x_0 > y_1y_0 \Leftrightarrow (x_1 > y_1)(x_1 = y_1)(x_0 > y_0) = 1 \\ &\Leftrightarrow (x_1 \cdot y_1 + x_1) + (x_1 + y_1 + 1)(x_0 \cdot y_0 + x_0) = 1 \\ &\Leftrightarrow x_1 \cdot y_1 + x_1 + x_1 \cdot x_0 \cdot y_0 + x_1 \cdot x_0 + \\ &y_1 \cdot x_0 \cdot y_0 + y_1 \cdot x_0 + x_0 \cdot y_0 + x_0 = 1 \\ &x_1x_0 == y_1y_0 \Leftrightarrow (x_0 + y_0 + 1) \cdot (x_1 + y_1 + 1) = 1 \\ &\Leftrightarrow x_0 \cdot x_1 + x_0 \cdot y_1 + x_0 + y_0 \cdot x_1 + y_0 \cdot y_1 + y_0 + 1 = 1 \\ &x_1x_0 < y_1y_0 \Leftrightarrow (x_1x_0 > y_1y_0) + (x_1x_0 == y_1y_0) + 1 = 1 \\ &\Leftrightarrow (x_1 \cdot y_1 + x_1 + x_1 \cdot x_0 \cdot y_0 + x_1 \cdot x_0 + y_1 \cdot x_0 \cdot y_0 + \\ &y_1 \cdot x_0 + x_0 \cdot y_0 + x_0) + (x_0 \cdot x_1 + x_0 \cdot y_1 + \\ &x_0 + y_0 \cdot x_1 + y_0 \cdot y_1 + y_0 + 1) + 1 = 1 \end{aligned} \tag{4}$$

Reason that we build up comparison functions for two-bit numbers is to apply the concept of homomorphic encryption and evaluation into the CEP engine. Even for two-bit number

Fig. 7 Architecture of E-mail Filter benchmark

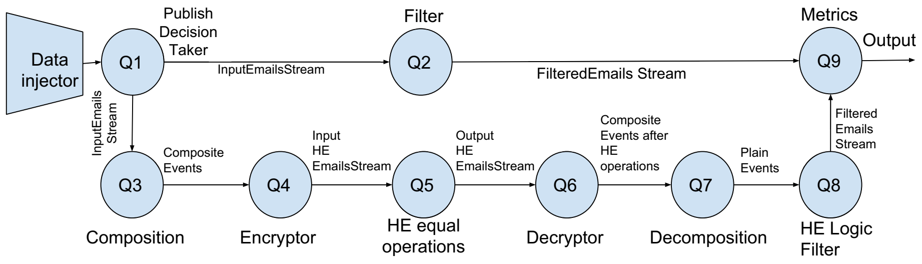
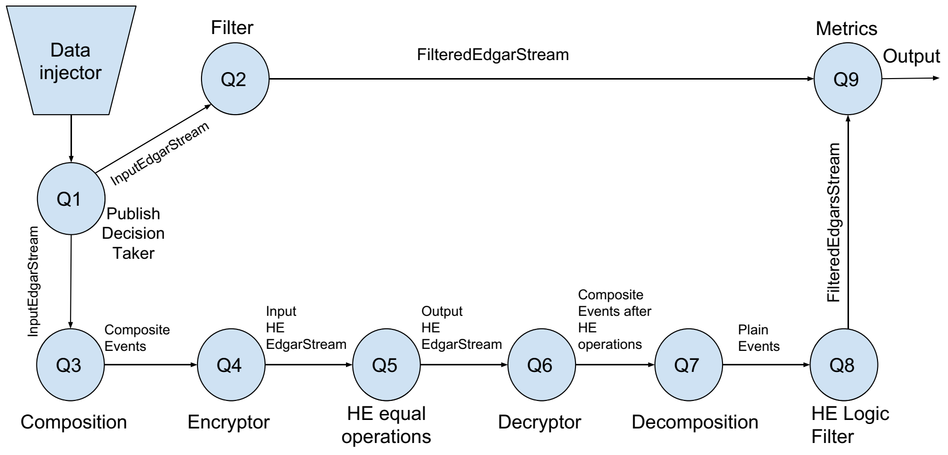


Fig. 8 EDGAR filter benchmark



comparisons, a number of XOR and AND gate evaluations need to be done as above.

After evaluating the individual HE operations at public SP, filtering using those gate operations happens at private SP. Boolean conditions are evaluated on encrypted operands using HE with above limitations for input number range, and ‘NOT,’ ‘AND,’ and ‘OR’ gate operations are evaluated at private SP after decrypting/decoding the events which come from public SP after HE evaluations.

We have evaluated the HomoESM’s functionality using five benchmark applications developed using two datasets. Next, in order to ensure the completeness of this section, we describe the implementation details of these benchmarks.

5.2 E-mail Filter Benchmark

E-mail Filter is a benchmark we developed based on the canonical Enron e-mail dataset [21]. The dataset has 517,417 e-mails with an average body size of 1.8 KB, the largest being 1.92MB. The E-mail Filter benchmark only had filter operation and was used to compare filtering performance compared to the EDGAR Filter benchmark which is described in the next subsection. The architecture of the E-mail Filter benchmark is shown in Fig. 7. The events in the input e-mails stream had eight fields *iij_timestamp, fromAddress, toAddresses, ccAddresses, bccAddresses, subject, body, regexstr* where all the fields were Strings except *iij_timestamp* which was long type. We formatted the *toAddresses* and *ccAddresses* fields to have only single e-mail address to support HElib evaluations. The criterion for filtering out e-mails was to filter by the e-mail addresses `lynn.blair@enron.com` and `richard.hanagriff@enron.com`. The filtering SiddhiQL statement can be stated as in Listing 2,

```
NOT ((fromAddress is equal to ‘lynn.blair@enron.com’) AND
 ((toAddresses is equal to ‘richard.hanagriff@enron.com’)
 OR (ccAddresses is equal to ‘richard.hanagriff@enron.com’
 )))
```

Listing 2 EmailFilter condition.

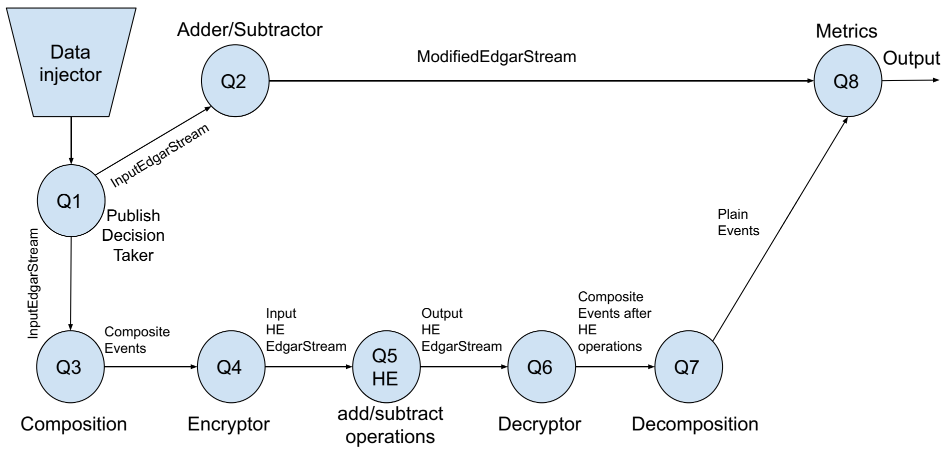


Fig. 9 EDGAR Add/Subtract benchmark

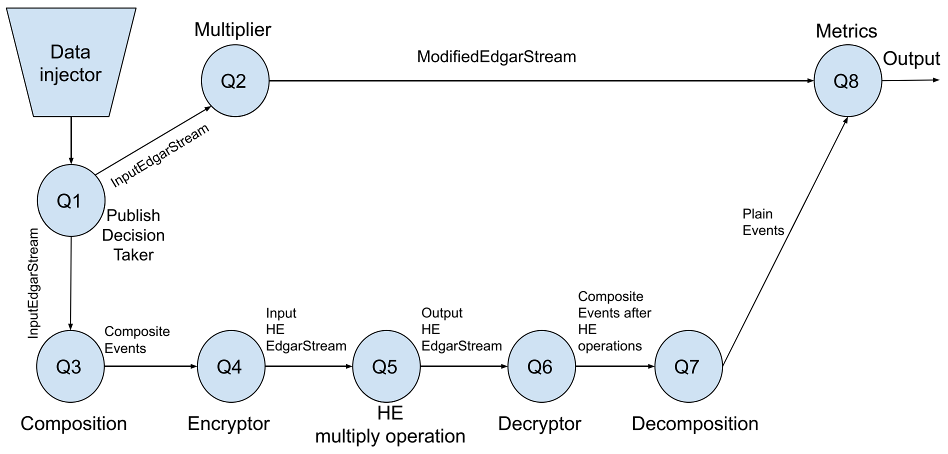


Fig. 10 EDGAR Multiply benchmark

5.3 EDGAR Filter Benchmark

We developed another benchmark based on a HTTP log dataset published by Division of Economic and Risk Analysis (DERA) [11]. The data provide details of the usage of publicly accessible EDGAR company filings in a simple but extensive manner [11]. Each record in the dataset consists of 16 different fields; hence, each event sent to the benchmark had 16 fields (*iij_timestamp, ip, date, time, zone, cik, accession, extension, code, size, idx, norefer, noagent, find, crawler,* and *browser*). Similar to the E-mail Filter benchmark, all of the fields except *iij_timestamp* were strings. Out of these fields, we used *noagent* field by adding lengthy string of 1024 characters to the existing value, in order to increase the events' size. (Note that we have done the same for all the EDGAR benchmarks described in this paper.) The architecture of EDGAR filter benchmark is shown in Fig. 8.

The EDGAR benchmark was developed with the aim of implementing filtering support. Basic criterion was to filter out EDGAR logs, which satisfies the conditions shown in Listing 3.

```
(extension == ‘v16003sv1.htm’) and (code == ‘200.0’) and
(date == ‘2016-10-01’)))
```

Listing 3 EDGAR filter condition.

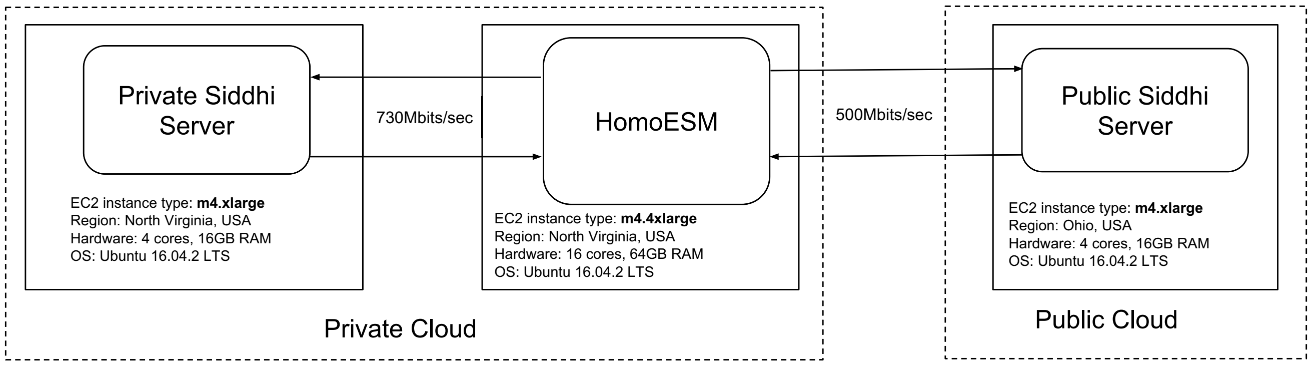

Private Siddhi Server
730Mbits/sec
HomoESM
500Mbits/sec
Public Siddhi Server
EC2 instance type: m4.xlarge
Region: North Virginia, USA
Hardware: 4 cores, 16GB RAM
OS: Ubuntu 16.04.2 LTS
EC2 instance type: m4.4xlarge
Region: North Virginia, USA
Hardware: 16 cores, 64GB RAM
OS: Ubuntu 16.04.2 LTS
EC2 instance type: m4.xlarge
Region: Ohio, USA
Hardware: 4 cores, 16GB RAM
OS: Ubuntu 16.04.2 LTS
Private Cloud
Public Cloud

Fig. 11 Experiment setup of HomoESM on Amazon EC2

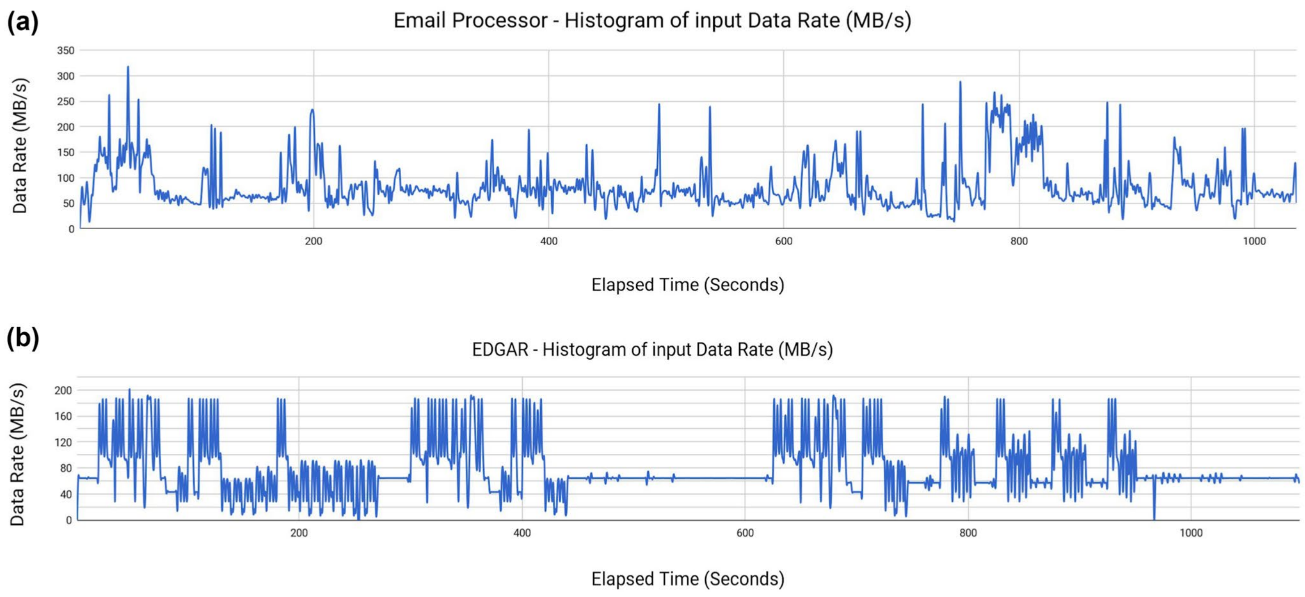


Fig. 12 Input data rate variation of the two benchmarks **a** E-mail Filter benchmark **b** EDGAR benchmarks

Most of the EDGAR log events were the same, and the logs did not have any data rate variation inherently. Therefore, we introduced varying data rate by publishing events in different TPS values according to a custom-defined function.

5.4 EDGAR Comparison Benchmark

Using the same EDGAR dataset, we developed EDGAR Comparison benchmark to evaluate the performance [10] of homomorphic comparison operation. In the EDAGR Comparison benchmark, we have changed the input format of the *zone* and *find* fields to integer (Int) in order to do comparison operations. Since we are doing only bitwise operations, we limited the HElib message space to 2, in order to use only 0s and 1s. Therefore, maximum length for encrypting field when we used message space as 2 was 168, and we used composite event size as 168 when sending to public Siddhi server. The architecture of EDGAR Comparison benchmark is similar to the topology shown in Fig. 7. Basic criterion is to filter out EDGAR logs, which satisfies the following conditions (see Listing 4).

```
(zone == 0) and (find > 0) and (find < 3)
```

Listing 4 EDGAR comparison condition.

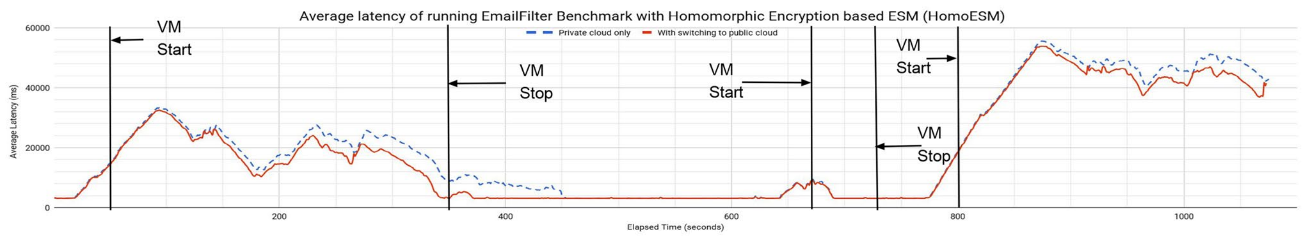


Fig. 13 Average latency of elastic scaling of the E-mail Filter benchmark with securing the event stream sent to public cloud via homomorphic encryption

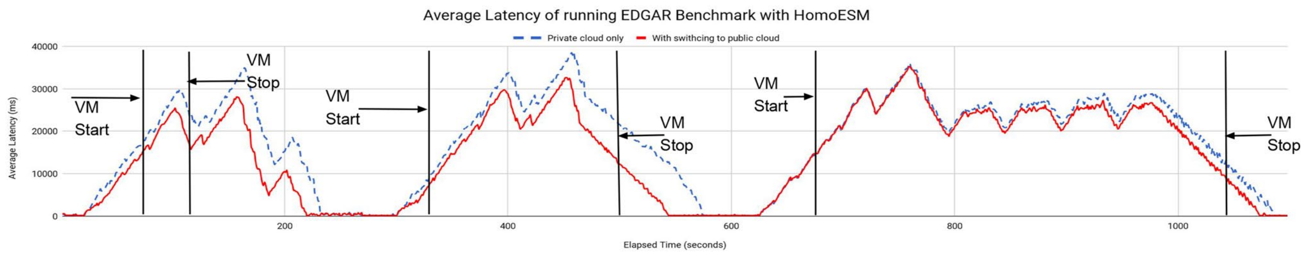


Fig. 14 Average latency of elastic scaling of the EDGAR benchmark with homomorphic filter operations

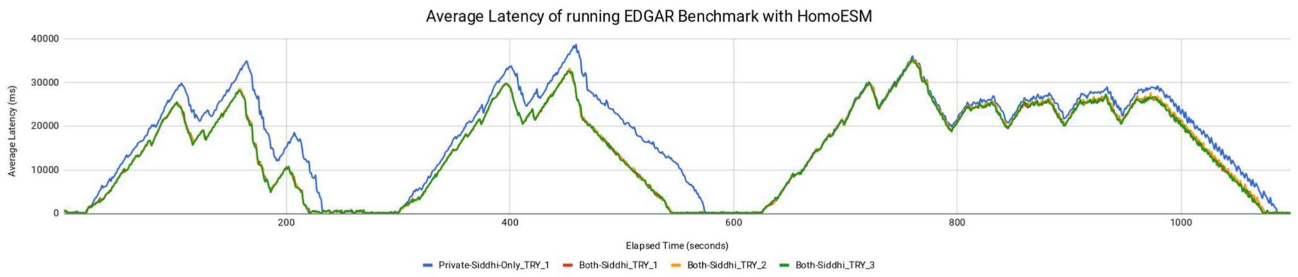


Fig. 15 Results comparison for three runs of the EDGAR benchmark with homomorphic filter operations

5.5 EDGAR Add/Subtract Benchmark

In EDGAR Add/Subtract benchmark, we have changed the input format to an Integer, for *code, idx, norefer*, and *find* fields in order to support add/subtract operations. The corresponding Siddhi query which depicts the addition and subtract operations conducted by this benchmark is shown in Listing 5.

```
@info(name = 'query5') from
 inputEdgarStream select iij_timestamp, ip, date, time,
 zone, cik, accession, extension, code-100 as code, size,
 idx+30 as idx, norefer+20 as norefer, noagent, find-10
 as find, crawler, browser insert into outputEdgarStream;
```

Listing 5 EDGAR add/subtract siddhi query.

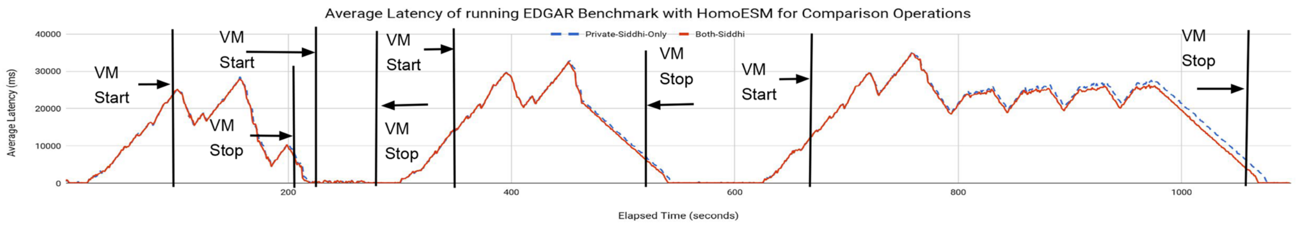


Fig. 16 Average latency of elastic scaling of the EDGAR benchmark with homomorphic comparison operations

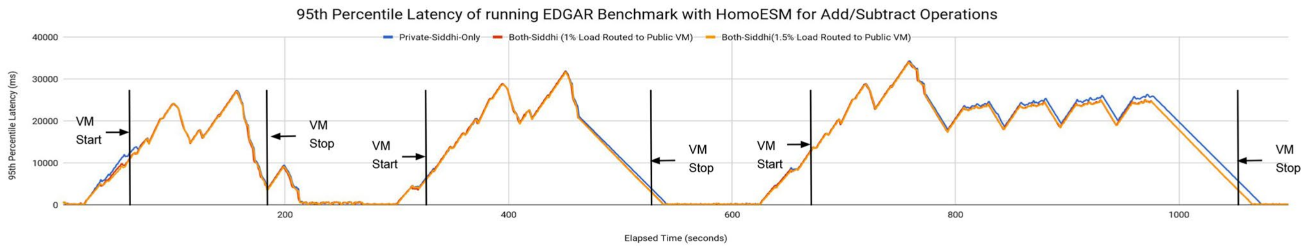


Fig. 17 Average latency of elastic scaling of the EDGAR benchmark with homomorphic add/subtract operations

The architecture of EDGAR Add/Subtract benchmark is shown in Fig. 9. Note that EDGAR Multiply benchmark also has similar architecture although Q2 and Q5 operators conduct multiply operations instead.

5.6 EDGAR Multiply Benchmark

In EDGAR Multiply benchmark, we have changed the input format to an Integer, for 'code' and 'idx' fields. As in EDGAR filter benchmark, here also we add lengthy string of 1024 characters to the existing value of 'noagent' field, in order to increase the packet size. We multiply code field by 2 and idx field by 3. The corresponding Siddhi query which depicts the multiplication operation done by this benchmark is shown in Listing 6. The architecture of EDGAR Multiply benchmark is shown in Fig. 10.

```
define stream inputEdgarStream (iij_timestamp long, ip
string, date string, time string, zone string, cik string,
accession string, extension string, code int, size string,
idx int, norefer int, noagent string, find int, crawler
string, browser string);
@info(name = 'query5') from inputEdgarStream select
iij_timestamp, ip, date, time, zone, cik, accession,
extension, code*2 as code, size, idx*3 as idx, norefer,
noagent, find, crawler, browser insert into
outputEdgarStream;
```

Listing 6 EDGAR multiply siddhi query.

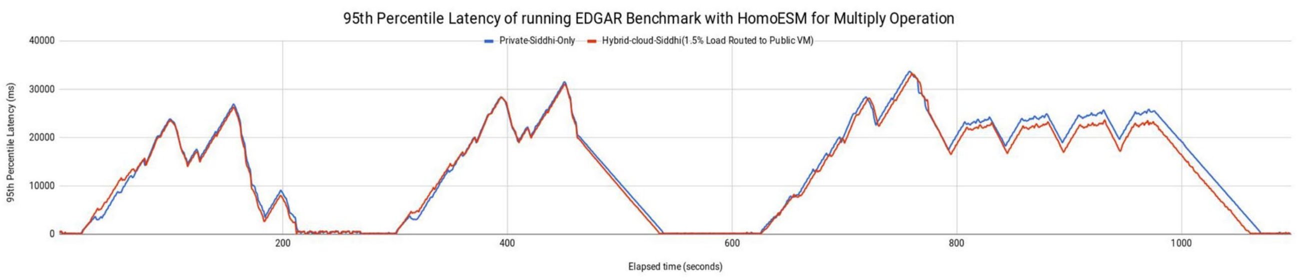


Fig. 18 Average latency of elastic scaling of the EDGAR benchmark with homomorphic multiplication operation

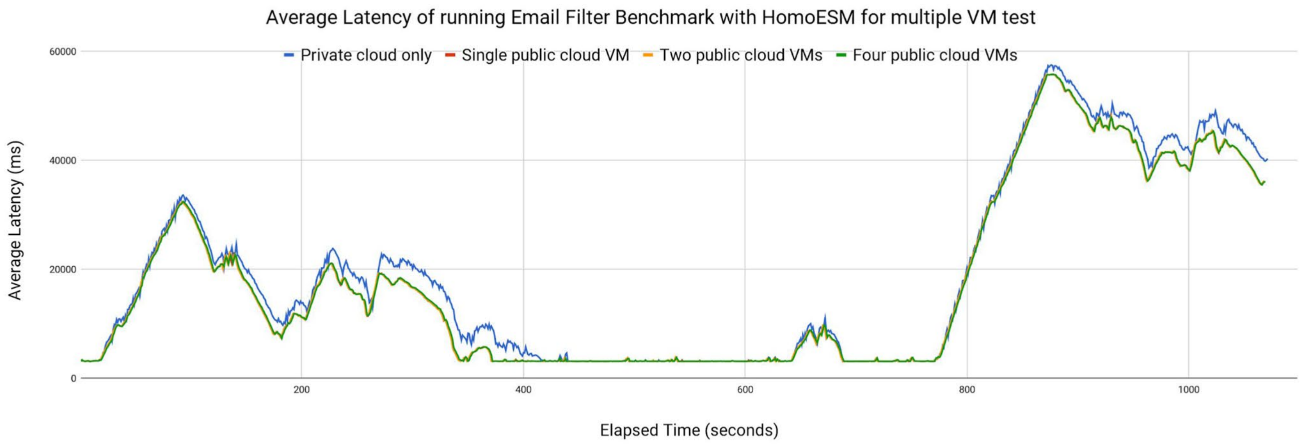


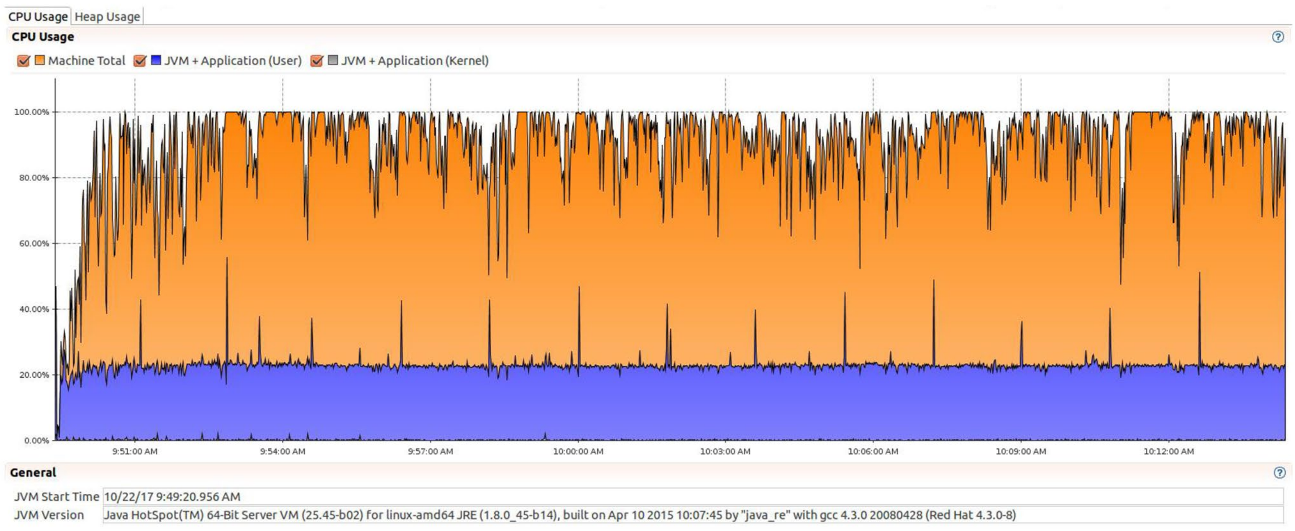
Fig. 19 Average Latency of running E-mail Filter benchmark with HomoESM for multiple VM test

6 Evaluation

We conducted the experiments using three VMs in Amazon EC2. In this experiment, two VMs were hosted in North Virginia, USA, and they were used as private cloud while the VM used as public cloud was located in Ohio, USA. We used the E-mail Filter benchmark in this experiment which does filtering of an e-mail event stream. Out of the two VMs in North Virginia, one was a *m4.4xlarge* instance which had 16 cores, 64 GB RAM while the private CEP Engine was deployed in a *m4.xlarge* instance which had four CPU cores, 16 GB RAM. In *m4.4xlarge* VM, we have deployed 'Event Publisher' (Event Publisher) and 'statistic-collector' (Event Receiver) modules. The Stream Processor engine running in the public cloud was deployed on the VM running in Ohio which was a *m4.xlarge* instance. All the VMs were running on Ubuntu 16.04.2 LTS (Long-Term Support). Using a network speed measurement tool, we observed that network speed between the two VMs in North Virginia was around 730 Mbits/s while the network speed between North Virginia and Ohio was 500 Mbits/s. Figure 11 shows the architecture of the experimental setup. The input data rate variation of the E-mail benchmark and the EDGAR benchmark datasets is shown in Fig. 12a, b respectively. The two charts indicate that the workloads imposed by the two benchmarks have significantly different characteristics.

6.1 E-mail Filter Benchmark

In the first round, we used E-mail Filter benchmark. The results of this experiment are shown in Fig. 13. The curve in the blue color (dashed line) indicates the private cloud deployment. The red color curve indicates the deployment with switching to public cloud. A clear reduction in average latency can be observed when switched to the public cloud in this setup compared to the private cloud-only deployment. With homomorphic elastic scaling, an overall average latency reduction of 2.14 s per event can be observed. This



JVM Start Time	10/22/17 9:49:20.956 AM
JVM Version	Java HotSpot(TM) 64-Bit Server VM (25.45-b02) for linux-amd64 JRE (1.8.0_45-b14), built on Apr 10 2015 10:07:45 by "java_re" with gcc 4.3.0 20080428 (Red Hat 4.3.0-8)

Fig. 20 CPU utilization at Event Publisher/statistics collector VM when sending data to public SP engine

is 10.24% improvement compared to the private cloud-only deployment. Note that in all the following charts, we have marked the times where VM start/VM stop operations have been invoked in order to start/stop the VM in the public cloud. Since VM startup and data sending times are almost similar, in this paper we assume VM startup time as the data sending time and VM stop time as the point where we stop sending data to public cloud.

6.2 EDGAR Filter Benchmark

In the second round, we used EDGAR Filter benchmark for evaluation of our technique. The results are shown in Fig. 14. Significant performance gain in terms of latency can be observed when switching to public cloud with the EDGAR benchmark. A notable fact is that EDGAR dataset had relatively smaller message size. The average message size of the EDGAR benchmark was 1.1 KB. The HomoESM mechanism was able to reduce the delay with considerable improvement of 17%. Furthermore, multiple rounds of experiments done with EDGAR Filter benchmark indicate that our approach provides consistent results (see Fig. 15).

6.3 EDGAR Comparison Benchmark

Next, we evaluated the homomorphic comparison operation. Here, we have used a slightly modified version of the EDGAR Filter benchmark to facilitate comparison operation in a homomorphic manner. Here, also we add lengthy string of 1024 characters to the existing value of 'noagent' field. The results are shown in Fig. 16.

We could see only a slight improvement in latency with EDGAR comparison benchmark. The improvement in the average latency was around 449 ms which is 3% improvement compared to the private-only deployment. Compared to equal-only operation, less-than and greater-than operations consume more XOR and AND gate operations in the homomorphic encryption (HE) level. Due to that, Siddhi engine processing throughput, when having homomorphic less-than and greater-than operations, is quite low compared to equal operation-only case. Therefore, the portion of events sent to public Siddhi is lesser than other cases. That's why we could not see much advantage (only 3%) on latency curves for both private and public Siddhi setup compared to private Siddhi-only setup. During the middle spike shown in Fig. 16, a 26.17% improvement in latency was observed.

6.4 EDGAR Add/Subtract Benchmark

We evaluated the homomorphic add/subtract operation using the EDGAR benchmark. The addition and subtraction HE operations' supported message space range is from 0 to 1201. Although 32-bit full adder circuits using HElib could increase the range further, we keep this as a further work. The overall improvement was 3.68% for the scenario where 1.5% of the load was sent to the public VM. We observed a maximum 6.13% performance improvement in the third spike shown in Fig. 17.

6.5 EDGAR Multiplication Benchmark

Next, we studied the performance behavior of the homomorphic multiplication functionality. The multiply HE operation's supported message space range is from 0 to 1201. If we need to support full-range multiply operation, we need to come up with at least 32-bit binary multiplier circuit using

HElib. But in this work we do not address this issue. The results of this experiment are shown in Fig. 18.

Multiplication had similar performance curve to the Add/Subtract benchmark. The third spike has performance improvement of 7.81%.

6.6 Multiple VM Test for E-mail Filter Benchmark

Up to this point, we had evaluated the performance improvement we could obtain using only one public cloud VM. In this experiment, we evaluated the advantage of using multiple VMs and scaling the homomorphic stream processor among those VMs. We conducted E-mail Filter benchmark performance test with two public VMs and four public VMs in two separate tests. The purpose of the experiment was to investigate whether adding more VMs in the public cloud may improve the performance. We identified that when we used a single public VM with a routing load percentage of 1.5%, the public VM is still not overloaded. Therefore, even if we increased the number of public VMs, the expected advantage could not be achieved. On the other hand, we cannot increase the routing load percentage more than 1.5% due to higher CPU utilization in Event Publisher VM instance. Therefore, we have ended up with similar latency curves for all cases with single, two, and four public VMs (Fig. 19).

7 Discussion

In this paper, we have not only implemented a mechanism for elastic privacy-preserving data stream processing but also shown considerable performance benefits on real-world experimental setups. Results comparing HomoESM to the private cloud-only deployments demonstrate 3–17% latency improvements. Furthermore, during large workload spikes, HomoESM has shown 6–26% latency improvements which is almost doubled performance improvement. Workload spikes are the key situations where HomoESM needs to be deployed which indicates HomoESM's effectiveness in handling such situations.

According to the above experiments, we can see better results only in E-mail Filter and EDGAR Filter benchmarks. These benchmarks' evaluations undergo only with single homomorphic XOR gate computations per composite event. Therefore, the complexity of computation at public SP engine is low, compared to EDGAR comparison and Add/Subtract benchmarks. This is why we see higher performance gains with the E-mail Filter and EDGAR Filter benchmarks compared to other benchmarks.

Apart from the above, EDGAR comparison and Add/Subtract benchmark experiments have limitations. EDGAR comparison benchmark experiment is performed only on two-bit numbers. This is due to the increment of circuit complexity in HElib, with the increment of no. of bits. EDGAR Add/Subtract benchmark also supports the range from 0 to 1201, which is the message space of HElib according to our selected settings. If we want to have support for larger numbers like 32-bit integers, we need to come up with HElib circuitry that will take longer time.

Although one could argue that the techniques presented in this paper are restricted due to the nature of the modern homomorphic encryption techniques, we have overcome the difficulties via batching and compressing the events, which is one of the key contributions of this paper. We have used high-performance VM instance type m4.4xlarge in the evaluations, because composite event composing and decomposing require more CPU for publisher and statistics collector. In the multi-VM experiment for example when we routed 1.5% load into the public cloud VM, the CPU utilization almost reached 100%. This is due to higher CPU consumption when performing composition and decomposition by Event Publisher and statistics collector, respectively. Java Flight Recorder (JFR) output for Event Publisher when sending data to public SP engine is shown in Fig. 20.

A limitation of FHE is that it needs prior knowledge of the data to conduct different operations on the encrypted data. Hence, HomoESM is applicable only for data streams with finite and unchanging data.

8 Conclusion

Privacy has become an utmost important barrier which hinders leveraging IaaS for running stream processing applications. In this paper, we introduce a mechanism called HomoESM which conducts privacy-preserving elastic data stream processing. We evaluated our approach using two benchmarks called E-mail Filter and EDGAR on Amazon AWS. We observed significant improvements in overall latency of 10% and 17% for E-mail Processors and EDGAR datasets with using HomoESM on equality operation. We also implemented comparison, add/subtract, and multiplication operations in HomoESM which resulted in maximum 26.17%, 6.13%, and 7.81% improvements in the average latencies, respectively. In the future, we plan to extend this work to handle more complicated streaming operations. We also plan to experiment with multiple query-based tuning for privacy-preserving elastic scaling.

References

1. Aguilar-Melchor C, Fau S, Fontaine C, Gogniat G, Sirdey R (2013) Recent advances in homomorphic encryption: a possible future for signal processing in the encrypted domain. IEEE Signal Process Mag 30(2):108–117
2. Armknecht F, Boyd C, Carr C, Gjøsteen K, Jäschke A, Reuter CA, Strand M (2015) A guide to fully homomorphic encryption. IACR Cryptol ePrint Arch 2015:1192
3. Cervino J, Kalyvianaki E, Salvachua J, Pietzuch P (April 2012) Adaptive provisioning of stream processing systems in the cloud. In: 2012 IEEE 28th international conference on data engineering workshops (ICDEW), pp 295–301
4. Cuzzocrea A, Chakravarthy S (2010) Event-based lossy compression for effective and efficient OLAP over data streams. Data Knowl Eng 69(7):678–708
5. Dai W, Sunar B (2015) cuhe: A homomorphic encryption accelerator library. Cryptology ePrint archive, report 2015/818. https://eprint.iacr.org/2015/818
6. Dayarathna M, Perera S (2018) Recent advancements in event processing. ACM Comput Surv 51(2):33:1–33:36
7. Dayarathna M, Suzumura T (2012) Hirundo: a mechanism for automated production of optimized data stream graphs. In: Proceedings of the 3rd ACM/SPEC international conference on performance engineering, ICPE ’12, pp 335–346, New York, NY, USA. ACM
8. Dayarathna M, Suzumura T (2013) Automatic optimization of stream programs via source program operator graph transformations. Distrib Parallel Databases 31(4):543–599
9. Dayarathna M, Suzumura T (2013) A mechanism for stream program performance recovery in resource limited compute clusters. Springer, Berlin, pp 164–178
10. Dayarathna M, Suzumura T (2013) A performance analysis of system S, S4, and esper via two level benchmarking. In: Joshi K, Siegle M, Stoelinga M, D’Argenio PR (eds) Quantitative evaluation of systems. QEST 2013. Lecture notes in computer science, vol 8054. Springer, Berlin, Heidelberg, pp 225–240. https://doi.org/10.1007/978-3-642-40196-1_19
11. DERA (2017). Edgar log file data set. https://www.sec.gov/dera/data/edgar-log-file-data-set.html
12. Gentry C (2009) Fully homomorphic encryption using ideal lattices. In: Proceedings of the forty-first annual ACM symposium on theory of computing, STOC ’09. ACM, New York, NY, USA, pp 169–178
13. Google (2017) Cloud dataflow. https://cloud.google.com/dataflow/
14. Halevi S (2017) An implementation of homomorphic encryption. https://github.com/shaih/HElib
15. Halevi S, Shoup V (2014) Algorithms in HElib. Springer, Berlin, pp 554–571
16. Hayes JP, Kolar HR, Akhriev A, Barry MG, Purcell ME, McKeown EP (2013) A real-time stream storage and analysis platform for underwater acoustic monitoring. IBM J Res Dev 57(3/4):15:1–15:10
17. Hummer W, Satzger B, Dustdar S (2013) Elastic stream processing in the cloud. Wiley Interdiscip Rev Data Min Knowl Discov 3(5):333–345
18. IBM (2017) Streaming analytics. https://www.ibm.com/cloud/streaming-analytics
19. Ishii A, Suzumura T (2011) Elastic stream computing with clouds. In: 2011 IEEE 4th international conference on cloud computing, pp 195–202
20. Jayasekara S, Perera S, Dayarathna M, Suhothayan S (2015) Continuous analytics on geospatial data streams with wso2 complex event processor. In: Proceedings of the 9th ACM international conference on distributed event-based systems, DEBS ’15. ACM, New York, NY, USA, pp 277–284
21. Klimt B, Yang Y (2004) Introducing the enron corpus. In: CEAS
22. Loesing S, Hentschel M, Kraska T, Kossmann D (2012) Stormy: An elastic and highly available streaming service in the cloud. In: Proceedings of the 2012 Joint EDBT/ICDT workshops, EDBT-ICDT ’12, ACM, New York, NY, USA, pp 55–60
23. Page A, Kocabas O, Ames S, Venkitasubramaniam M, Soyata T (Dec 2014) Cloud-based secure health monitoring: Optimizing fully-homomorphic encryption for streaming algorithms. In: 2014 IEEE Globecom Workshops (GC Wkshps), pp 48–52
24. Quoc DL, Chen R, Bhatotia P, Fetzer C, Hilt V, Strufe T (2017) Streamapprox: Approximate computing for stream analytics. In: Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference, Middleware ’17. ACM, New York, NY, USA, pp 185–197
25. Ravindra S, Dayarathna M, Jayasena S (2017) Latency aware elastic switching-based stream processing over compressed data streams. In: Proceedings of the 8th ACM/SPEC on international conference on performance engineering, ICPE ’17. ACM, New York, NY, USA, pp 91–102
26. Rodrigo A, Dayarathna M, Jayasena S (2019) Privacy preserving elastic stream processing with clouds using homomorphic encryption. In: Li G, Yang J, Gama J, Natwichai J, Tong Y (eds) Database systems for advanced applications. Springer, Cham, pp 264–280
27. Shaon F, Kantarcioglu M, Lin Z, Khan L (2017) Sgx-bigmatrix: A practical encrypted data analytic framework with trusted processors. In: Proceedings of the 2017 ACM SIGSAC conference on computer and communications security, CCS ’17. ACM, New York, NY, USA, pp 1211–1228
28. Striim (2017) Striim delivers streaming hybrid cloud integration to microsoft azure. http://www.striim.com/press/hybrid-cloud-integration-to-microsoft-azure
29. Theeten B, Bedini I, Cogan P, Sala A, Cucinotta T (2014) Towards the optimization of a parallel streaming engine for telco applications. Bell Labs Tech J 18(4):181–197
30. Togan M, Plesca C (2014) Comparison-based computations over fully homomorphic encrypted data. In: 2014 10th international conference on communications (COMM), pp 1–6
31. WSO2 (2018) Wso2 stream processor. https://wso2.com/analytics-and-stream-processing
32. WSO2 (2019) Stream processing and complex event processing engine. https://github.com/siddhi-io/siddhi