Privacy Preserving Elastic Stream Processing with Clouds using Homomorphic Encryption

Arosha Rodrigo[1], Miyuru Dayarathna[2], and Sanath Jayasena[1]

[1] Department of Computer Science & Engineering, University of Moratuwa
uom.arosha@gmail.com, sanath@cse.mrt.ac.lk
[2] WSO2, Inc.
miyurud@wso2.com

Abstract. Prevalence of the Infrastructure as a Service (IaaS) clouds has enabled organizations to elastically scale their stream processing applications to public clouds. However, current approaches for elastic stream processing do not consider the potential security vulnerabilities in cloud environments. In this paper we describe the design and implementation of an Elastic Switching Mechanism for data stream processing which is based on Homomorphic Encryption (HomoESM). The HomoESM not only does elastically scale data stream processing applications into public clouds but also preserves the privacy of such applications. Using a real world test setup, which includes an email filter benchmark and a web server access log processor benchmark (EDGAR) we demonstrate the effectiveness of our approach. Multiple experiments on Amazon EC2 indicate that the proposed approach for Homomorphic encryption provides significant results which is 10% to 17% improvement of average latency in the case of email filter benchmark and EDGAR benchmarks respectively. Furthermore, EDGAR add/-subtract operations and comparison operations showed 6.13% and 26.17% average latency improvements respectively. These promising results pave the way for real world deployments of privacy preserving elastic stream processing in the cloud.

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

1 Introduction

Data stream processing conducts online analytics processing on data streams [5]. Data stream processing has applications in diverse areas such as health informatics [1], transportation [16], telecommunications [24], etc. These applications have been implemented on data stream processing engines [5]. Most of the initial data stream processors were run on isolated computers/clusters (i.e., private clouds). The rise of cloud computing era has resulted in the ability of on demand provisioning of hardware and software resources. This has resulted in data stream processors which run as managed cloud services (e.g., [10][14]) as well as hybrid cloud services (e.g., Striim [23]).

Stream processing systems often face resource limitations during their operation due to unexpected loads [2][6]. Several approaches exist which could solve such an issue. Elastically scaling into an external cluster [15][21], load shedding, approximate

The final authenticated version is available online at https://doi.org/10.1007/978-3-030-18579-4_16.

query processing [20], etc. are some examples. Out of these, elastic scaling has become a key choice because approaches such as load shedding, approximate computing has to compromise accuracy which is not accepted by certain categories of applications. Previous work has been there which used data compression techniques to optimize the network connection between private and public clouds [21]. However, current elastic scaling mechanisms for stream processing do not consider a very important problem: preserving the privacy of the data sent to public cloud.

Preserving the privacy of stream processing operation becomes one of the key questions to be answered when scaling into a public cloud. Sending the data unencrypted to the server definitely 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 work has recently being conducted on privacy preserving data stream mining. Privacy of patient health information has been a serious issue in recent times [19]. Fully Homomorphic Encryption (FHE) has been introduced as a solution [9]. 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 is. However, current FHE techniques are computationally expensive needing excessive space for keys and cypher texts. However, it has been shown with some experiments done with HElib [12] (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 this paper we discuss elastic scaling in a private/public cloud (i.e., hybrid cloud) scenario with privacy preserving data stream processing. 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 [16]. We designed and developed two real world data stream processing benchmarks called EmailProcessor and HTTP Log Processor (EDGAR benchmark) during the evaluation of the proposed approach. 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 which is 10% to 17% improvement of average latency in the case of Email Filter benchmark. EDGAR comparison and add/subtract operations showed 26.17% average latency improvement. 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 [3][4][5]. Specifically, the contributions of our work can be listed as follows.

[3] https://github.com/arosharodrigo/event-publisher
[4] https://github.com/arosharodrigo/statistics-collector
[5] https://github.com/arosharodrigo/simple-siddhi-server

- *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.
- *Benchmarks* -We design and develop two benchmarks for evaluating the performance of HomoESM.
- *Optimization of Homomorphic Operations* - We optimized several homomorphic evaluation schemes such as equality, 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.

The paper is organized as follows. Next, we provide related work in Section 2. We provide the details of system design in Section 3 and implementation of the HomoESM in Section 4. The evaluation details are provided in Section 5. We make a discussion of the results in Section 6. We provide the conclusions in Section 7.

2 Related Work

There have been multiple previous work 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 [13]. 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 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 "stream processing as service" concept [18]. 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 multi-tenancy in mind to fit in the cloud environment. They have used distributed hash tables (DHT) 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. This is an issue which we address in our work via the concept of data switching of HomoESM.

Cervino *et al.* try to solve the problem of providing a resource provisioning mechanism to overcome inherent deficiencies of cloud infrastructure [2]. They have conducted some experiments on Amazon EC2 to investigate the problems that might affect badly on 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 [3] over data streams. Their compression method exploits semantics of the reference application and drives the compression process by means of the "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.* have implemented homomorphic encryption library [4] 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 speed up, when encrypting a Java String field, its length goes more than 400KB 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 6th generation Core i5, i7, and Xeon processors [22]. 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 multi-user settings [22]. 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 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 Figure 1. The components highlighted in the dark blue color correspond to components which directly implement privacy preserving stream processing functionality.

In this system architecture 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 & public Siddhi. If the event is from the private Siddhi, it is sent to the Profiler. If not the event is a composite event and 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 goes out from HomoESM run through Profiler which conducts the latency measurements.

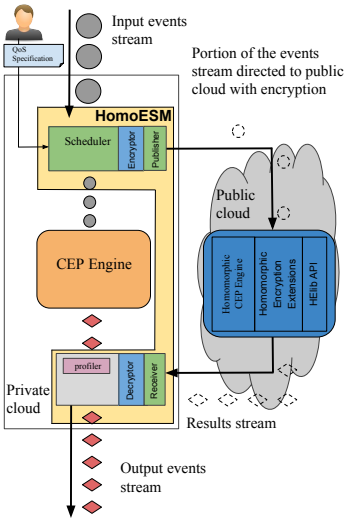

Fig. 1. The system architecture of Homomorphic Encryption based ESM (HomoESM).

In this paper we use the same switching functions described in [5] for triggering and stopping data sending to public cloud (See Equation 1). It should be noted that the main contribution of this paper is to describe the elastic privacy preserving stream functionality. Here $\phi_{VM}(t)$ is the binary switching function for a single VM, 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. D_s is the threshold for total amount of data received by the VM from private cloud.

$$\phi_{VM}(t) = \begin{cases} 1, L_{t-1} \geq L_s, \tau \; has \; elapsed. \\ \\ 0, D_{t-1} < D_s, L_{t-1} < L_p \qquad \text{Otherwise,} \end{cases} , \tag{1}$$

4 Implementation

In this Section first we describe the implementation details of HomoESM in Section 4.1 and we describe the benchmark implementations in Sections 4.2, 4.3, 4.4, and 4.5.

4.1 Implementation of HomoESM

We have developed the HomoESM on top of the WSO2 Stream Processor (WSO2 SP) software stack. WSO2 SP is an open source, lightweight, easy-to-use, stream processing engine [26]. WSO2 SP internally uses Siddhi which is a complex event processing library [16]. 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 Figure 2. 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 sending to private Siddhi. If that event needs to be sent to public Siddhi, it will mark the time and put into the queue which is processed by the Encrypt Master asynchronously.

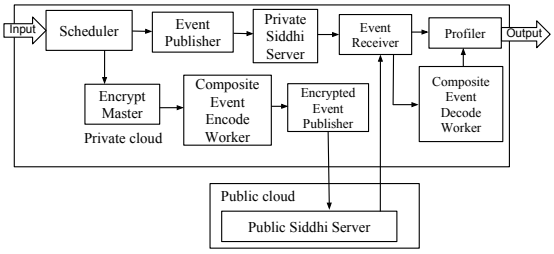

Fig. 2. Main components of HomoESM

.

Encrypt Master thread (see Figure 3 (a)) periodically checks a queue which keeps the events required to be sent to public cloud. The queue is maintained by the 'Event Publisher' (See Figure 4 (a)). If that queue size is greater than or equal to composite event size, it will create a list of events equal to the size of composite event size. Next, it delegates the event encryption and composite event creation task to the 'Composite Event Encode Worker' (see Figure 3 (b)).

Composite Event Encode Worker is a thread pool which handles event encryptions and composite event creations. First, it combines non-operational fields of each plain events 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 Figure 4 (b)).

Firing events into the public VM is done asynchronously. Decision of how many events 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 Figure 3 (b)). First, it combines non-operational fields of each plain event in the list by the pre-defined separator. If there

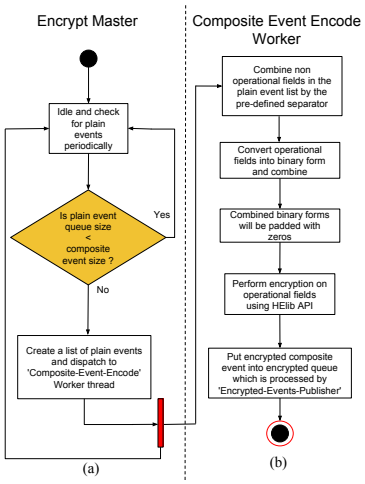

Fig. 3. Data encryption and the composite event creation process at the private Siddhi server. (a) Encrypt Master thread (b) Composite Event Encode Worker thread

are encrypted events, it will pick those at once and send 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 Email 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 10 times and encrypts using HElib [11]. Finally, the encrypted value and the relevant field in the composite event are used for HElib's relevant (e.g., comparison, addition, subtraction, division, 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 if 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 Figure 5 (a)).

After receiving a composite event from the Event Receiver the Composite Event Decode Worker handles all decomposition and decryptions of the composite event (See Figure 5 (b)). It first splits non-operational fields in the composite event by the predefined 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 splitted fields. Next, it checks each operational fields in the plain event to see

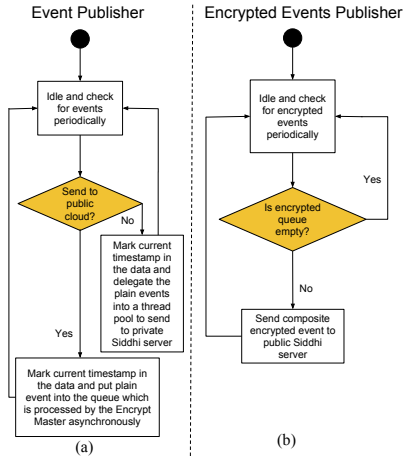

Fig. 4. Operation of the Event Publisher and the Encrypted Events Publisher components. (a) Event Publisher (b) Encrypted Events Publisher

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.* [25]. For two single bit numbers with x and y, Togan *et al.*[25] have shown that the following equations (see Equation 2) will satisfy greater-than and equal operations, respectively.

$$\begin{aligned} x > y \Leftrightarrow xy + x = 1 \\ x = y \Leftrightarrow x + y + 1 = 1 \end{aligned} \tag{2}$$

Togan *et al.* have created comparison functions for n-bit numbers using divide and conquer methodology. In our case we derived 2-bit number comparisons as follows. x_1x_0 and y_1y_0 are the two numbers with 2-bits (see Equation 3). Here every '+' operation is for XOR gate operation and every '.' operator is for AND gate operation.

$$\begin{aligned} x_1x_0 > y_1y_0 \Leftrightarrow (x_1 > y_1)(x_1 = y_1)(x_0 > y_0) = 1 \\ \Leftrightarrow (x_1.y_1 + x_1) + (x_1 + y_1 + 1)(x_0.y_0 + x_0) = 1 \\ \Leftrightarrow x_1.y_1 + x_1 + x_1.x_0.y_0 + x_1.x_0 + \\ y_1.x_0.y_0 + y_1.x_0 + x_0.y_0 + x_0 = 1 \\ x_1x_0 == y_1y_0 \Leftrightarrow (x_0 + y_0 + 1).(x_1 + y_1 + 1) = 1 \\ \Leftrightarrow x_0.x_1 + x_0.y_1 + x_0 + y_0.x_1 + y_0.y_1 + y_0 + 1 = 1 \\ x_1x_0 < y_1y_0 \Leftrightarrow (x_1x_0 > y_1y_0) + (x_1x_0 == y_1y_0) + 1 = 1 \\ \Leftrightarrow (x_1.y_1 + x_1 + x_1.x_0.y_0 + x_1.x_0 + y_1.x_0.y_0 + \\ y_1.x_0 + x_0.y_0 + x_0) + (x_0.x_1 + x_0.y_1 + \\ x_0 + y_0.x_1 + y_0.y_1 + y_0 + 1) + 1 = 1 \end{aligned} \tag{3}$$

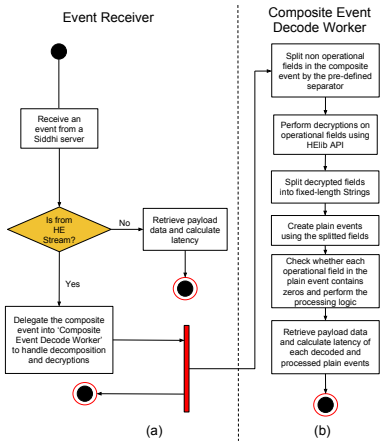

Fig. 5. Event receiving, decomposition, and decryption processes.

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 2-bit number comparisons, there are 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 evaluate at private SP after decrypting/decoding the events which comes from public SP after HE evaluations.

We have evaluated the HomoESM's functionality using four benchmark applications developed using two data sets. Next, in order to ensure the completeness of this section we describe the implementation details of the two benchmarks.

4.2 Email Filter Benchmark

Email Filter is a benchmark we developed based on the canonical Enron email data set [17]. The data set has 517,417 emails with an average body size of 1.8KB, the largest being 1.92MB. The Email 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 Email Filter benchmark is shown in Figure 6. The events in the input emails 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 email address to support HElib evaluations. The criteria for filtering out Emails was to filter by the email addresses `lynn.blair@enron.com` and `richard.hanagriff@enron.com`. The filtering SiddhiQL statement can be stated as in Listing 1.1,

```
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 1.1. EmailFilter condition.

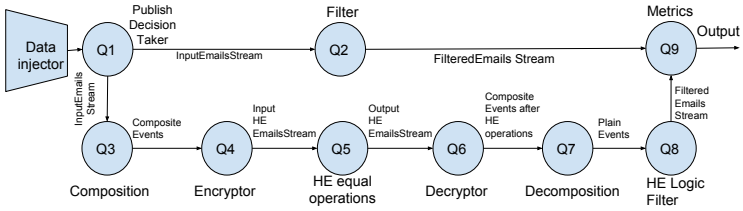

Fig. 6. Architecture of Email Filter benchmark.

4.3 EDGAR Filter Benchmark

We developed another benchmark based on a HTTP log data set published by Division of Economic and Risk Analysis (DERA) [8]. The data provides details of the usage of publicly accessible EDGAR company filings in a simple but extensive manner [8]. Each record in the data set 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 Email 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 EDGAR benchmark was developed with the aim of implementing filtering support. Basic criteria was to filter out EDGAR logs, which satisfy the conditions shown in Listing 1.2.

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

Listing 1.2. EDGAR filter condition.

Most of the EDGAR log events were 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.

4.4 EDGAR Comparison Benchmark

Using the same EDGAR data set we developed EDGAR Comparison benchmark to evaluate the performance [7] 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 Figure 6. Basic criteria is to filter out EDGAR logs, which satisfy following conditions (See Listing 1.3).

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

Listing 1.3. EDGAR comparison condition.

4.5 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 1.4.

```
@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 1.4. EDGAR add/subtract siddhi query.

5 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 Email Filter benchmark in this experiment which does filtering of an email event stream. Out of the two VMs in North Virginia one was a *m4.4xlarge* instance which had 16 cores, 64GB RAM while the private CEP Engine was deployed in a *m4.xlarge* instance which had 4 CPU cores, 16GB 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 730Mbits/sec while the network speed between North Virginia and Ohio was 500Mbits/sec. Figure 7 shows the architecture of the experiment setup. The input data rate variation of the Email benchmark and the EDGAR benchmark data sets is shown in Figure 8 (a) and (b) respectively. The two charts indicate that the workloads imposed by the two benchmarks have significantly different characteristics.

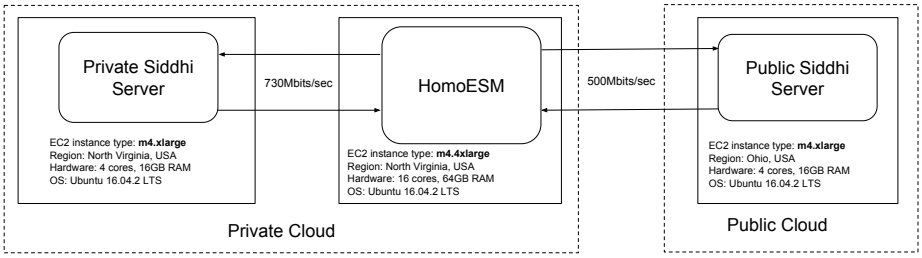

Fig. 7. Experiment setup of HomoESM on Amazon EC2.

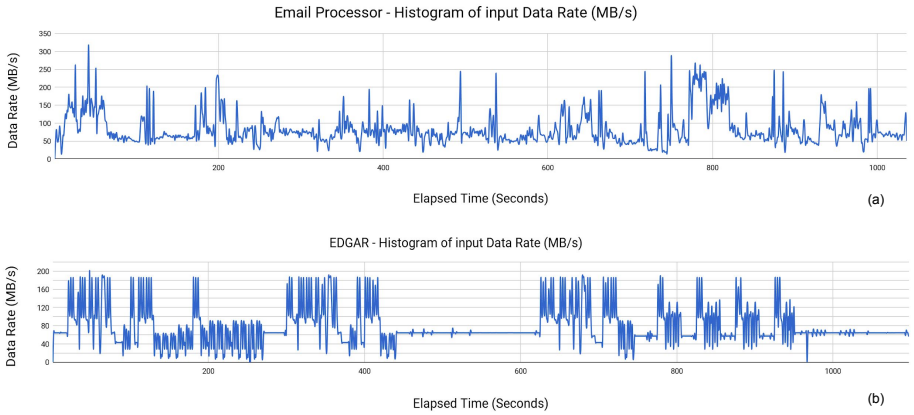

Fig. 8. Input data rate variation of the two benchmarks (a) Email Filter benchmark (b) EDGAR benchmarks.

5.1 Email Filter Benchmark

In the first round we used Email Filter benchmark. The results of this experiment is shown in Figure 9. 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. It can be observed a clear reduction of average latency 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 seconds per event can be observed. This 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.

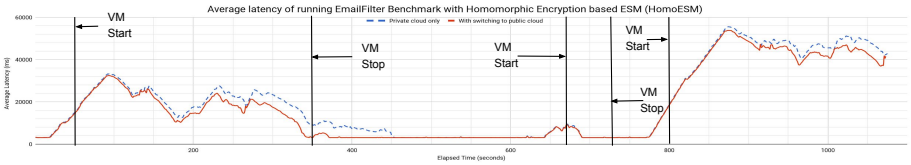

Fig. 9. Average latency of elastic scaling of the Email Filter benchmark with securing the event stream sent to public cloud via homomorphic encryption.

5.2 EDGAR Filter Benchmark

In the second round we used EDGAR Filter benchmark for evaluation of our technique. The results are shown in Figure 10. It can be observed significant performance gain in terms of latency when switching to public cloud with the EDGAR benchmark. A notable fact is that EDGAR data set 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%.

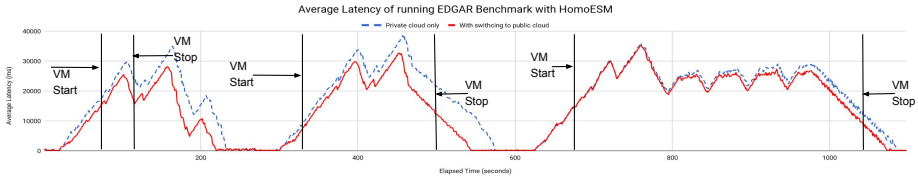

Fig. 10. Average latency of elastic scaling of the EDGAR benchmark with Homomorphic filter operations.

5.3 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 Figure 11.

We could see only a slight improvement of latency with EDGAR comparison benchmark. The improvement of the average latency was around 449 ms which is 3% improvement compared to the private only deployment. Compared to equal only operation, less-than & greater-than operations consume more XOR & AND gate operations in the Homomorphic Encryption (HE) level. Due to that Siddhi engine processing throughput, when having homomorphic less-than & 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 & public Siddhi setup compared to private Siddhi only

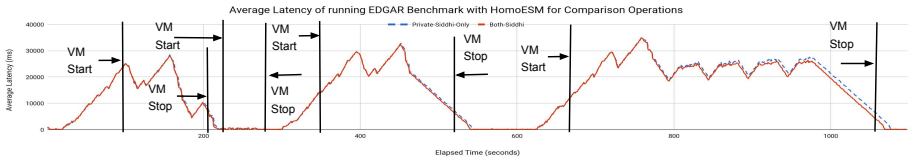

Fig. 11. Average latency of elastic scaling of the EDGAR benchmark with Homomorphic comparison operations.

setup. During the middle spike shown in Figure 11, a 26.17% improvement in latency was observed.

5.4 EDGAR Add/Subtract Benchmark

Finally, 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 Figure 12.

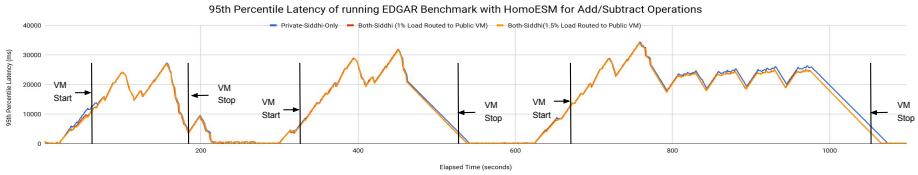

Fig. 12. Average latency of elastic scaling of the EDGAR benchmark with Homomorphic Add/-Subtract operations.

6 Discussion

Privacy preserving data mining in clouds has been an area of significant interest in recent times. However, none of the previous work on elastic stream processing has demonstrated the feasibility of conducting elastic privacy preserving data stream processing. In this paper we have not only implemented a mechanism for elastic privacy preserving data stream processing but also have shown considerable performance benefits on real world experiment 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.

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 & decomposing require more CPU for publisher and statistics collector. 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.

7 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 Email Filter and EDGAR on Amazon AWS. We observed significant improvements of overall latency of 10% and 17% for Email Processors and EDGAR data sets with using HomoESM on equality operation. We also implemented comparison and add/subtract operations in HomoESM which resulted in maximum 26.17% and 6.13% improvement in the average latencies respectively. In 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. M. Blount, M. Ebling, J. Eklund, A. James, C. McGregor, N. Percival, K. Smith, and D. Sow. Real-time analysis for intensive care: Development and deployment of the artemis analytic system. *Engineering in Medicine and Biology Magazine, IEEE*, 29(2):110–118, March 2010.
2. J. Cervino, E. Kalyvianaki, J. Salvachua, and P. Pietzuch. Adaptive provisioning of stream processing systems in the cloud. In *Data Engineering Workshops (ICDEW), 2012 IEEE 28th International Conference on*, pages 295–301, April 2012.
3. A. Cuzzocrea and S. Chakravarthy. Event-based lossy compression for effective and efficient OLAP over data streams. *Data & Knowledge Engineering*, 69(7):678 – 708, 2010.
4. W. Dai and B. Sunar. cuhe: A homomorphic encryption accelerator library. Cryptology ePrint Archive, Report 2015/818, 2015. https://eprint.iacr.org/2015/818.
5. M. Dayarathna and S. Perera. Recent advancements in event processing. *ACM Comput. Surv.*, 51(2):33:1–33:36, Feb. 2018.
6. M. Dayarathna and T. Suzumura. *A Mechanism for Stream Program Performance Recovery in Resource Limited Compute Clusters*, pages 164–178. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.
7. M. Dayarathna and T. Suzumura. A performance analysis of system s, s4, and esper via two level benchmarking. In *Quantitative Evaluation of Systems*, pages 225–240, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

8. DERA. Edgar log file data set. URL: https://www.sec.gov/dera/data/edgar-log-file-data-set.html, 2017.
9. C. Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing*, STOC '09, pages 169–178, New York, NY, USA, 2009. ACM.
10. Google. Cloud dataflow. URL: https://cloud.google.com/dataflow/, 2017.
11. S. Halevi. An implementation of homomorphic encryption. URL: https://github.com/shaih/HElib, 2017.
12. S. Halevi and V. Shoup. *Algorithms in HElib*, pages 554–571. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.
13. W. Hummer, B. Satzger, and S. Dustdar. Elastic stream processing in the cloud. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 3(5):333–345, 2013.
14. IBM. Streaming analytics. URL: https://www.ibm.com/cloud/streaming-analytics, 2017.
15. A. Ishii and T. Suzumura. Elastic stream computing with clouds. In *2011 IEEE 4th International Conference on Cloud Computing*, pages 195–202, July 2011.
16. S. Jayasekara, S. Perera, M. Dayarathna, and S. Suhothayan. 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, pages 277–284, New York, NY, USA, 2015. ACM.
17. B. Klimt and Y. Yang. Introducing the enron corpus. page 2, 01 2004.
18. S. Loesing, M. Hentschel, T. Kraska, and D. Kossmann. Stormy: An elastic and highly available streaming service in the cloud. In *Proceedings of the 2012 Joint EDBT/ICDT Workshops*, EDBT-ICDT '12, pages 55–60, New York, NY, USA, 2012. ACM.
19. A. Page, O. Kocabas, S. Ames, M. Venkitasubramaniam, and T. Soyata. Cloud-based secure health monitoring: Optimizing fully-homomorphic encryption for streaming algorithms. In *2014 IEEE Globecom Workshops (GC Wkshps)*, pages 48–52, Dec 2014.
20. D. L. Quoc, R. Chen, P. Bhatotia, C. Fetzer, V. Hilt, and T. Strufe. Streamapprox: Approximate computing for stream analytics. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, Middleware '17, pages 185–197, New York, NY, USA, 2017. ACM.
21. S. Ravindra, M. Dayarathna, and S. Jayasena. 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, pages 91–102, New York, NY, USA, 2017. ACM.
22. F. Shaon, M. Kantarcioglu, Z. Lin, and L. Khan. 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, pages 1211–1228, New York, NY, USA, 2017. ACM.
23. Striim. Striim delivers streaming hybrid cloud integration to microsoft azure. URL: http://www.striim.com/press/hybrid-cloud-integration-to-microsoft-azure, 2017.
24. B. Theeten, I. Bedini, P. Cogan, A. Sala, and T. Cucinotta. Towards the optimization of a parallel streaming engine for telco applications. *Bell Labs Technical Journal*, 18(4):181–197, 2014.
25. M. Togan and C. Plesca. Comparison-based computations over fully homomorphic encrypted data. In *2014 10th International Conference on Communications (COMM)*, pages 1–6, May 2014.
26. WSO2. Wso2 stream processor. URL: https://wso2.com/analytics-and-stream-processing, 2018.