LOW LATENCY, ELASTIC AND PRIVACY PRESERVING DATA STREAM PROCESSING

Buddhima Arosha Rodrigo

148237H

Degree of Master of Science

Department of Computer Science and Engineering

University of Moratuwa
Sri Lanka

May 2018

LOW LATENCY, ELASTIC AND PRIVACY PRESERVING DATA STREAM PROCESSING

Buddhima Arosha Rodrigo

148237H

This dissertation submitted in partial fulfillment of the requirements for the degree Master of Science in Computer Science Specializing Parallel Computing

Department of Computer Science and Engineering

University of Moratuwa

Sri Lanka

May 2018

DECLARATION

I declare that this is my own work and this dissertation does not incorporate without acknowledgement of any material previously submitted for a Degree or Diploma in any other University or institute of higher learning and to the best of my knowledge and belief it does not contain any material previously published or written by another person except where the acknowledgement is made in the text.

Also, I hereby grant to University of Moratuwa the non-exclusive right to reproduce and distribute my dissertation, in whole or in part in print, electronic or other medium. I retain the right to use this content in whole or part in future works.

Candidate

....

Buddhima Arosha Rodrigo Date

I certify that the declaration above by the candidate is true to the best of my knowledge and he has carried out research for the Masters Dissertation under my supervision.

Supervisors

..... ….....

Prof. Sanath Jayasena Dr. Miyuru Dayarathna

....

Date Date

Abstract

Prevalence of the Infrastructure-as-a-Service (IaaS) clouds has enabled organizations to utilize compute services on demand via elastic scaling of their applications. Data stream processing is one such area which is benefited by elastic scaling. The main drawback of using these IaaS clouds is the security risks on sensitive data in the aspect of data stream processing. It will be a great solution if we can preserve the privacy of data of data-sensitive applications, while using them in IaaS clouds with minimized security risks.

The aim of this research is to implement elastic scaling mechanism in a private/public cloud environment by preserving the privacy of the data in the aspect of stream processing. To enable the privacy preserving on data, we use the concept of Homomorphic Encryption (HE) which can perform computations on encrypted data. We designed and implemented several functions which support Homomorphic Encryption using a well-known library HElib. We extended existing Elastic Switching Mechanism (ESM) to support newly implemented HE based functions. This Homomorphic Encryption based Elastic Switching Mechanism (HomoESM) operates between the boundaries of a private and a public cloud while preserving data security.

Using two real-world data stream processing scenarios, which include an email data set and a web server access log processor data set (EDGAR), we derive four benchmark applications. Several experiments on those benchmarks indicate that, the proposed approach for Homomorphic Encryption based equal operation provides significant results which are 10% and 17% improvement of average latency when compared to private Stream Processor (SP) only case for the scenarios of Email Filter benchmark and EDGAR Filter benchmark respectively. The HE operations which consume more computations such as greater-than and less-than comparison operations, add and subtract operations, also provide beneficial results but not much as equal operation's results. Therefore, this HomoESM performance directly depends on the complexity of HE computations. In this work we use data batching technique in our HomoESM implementation by creating a composite event using several plain events in order to address Single-Instruction-Multiple-Data (SIMD) support given by HElib. This approach is the key advancement in our HomoESM which enables to realize the elastic stream processing with HomoESM. Mainly our work addresses the feasibility and limitations of using HE operations under the aspect of data stream processing in a private/public cloud environment.

Keywords: Cloud computing, Elastic data stream processing, Homomorphic Encryption, IaaS, System sizing and capacity planning.

ACKNOWLEDGEMENTS

This project would not have been possible without the support of many people. Specially thanks to my supervisors, Prof. Sanath Jayasena and Dr. Miyuru Dayarathna, for their valuable guidance and endless support. Many thanks to our M.Sc. research project coordinators, Dr. Shehan Perera, Dr. Dilum Bandara and Dr. Malaka Walpola, for their dedication and support. Thanks to all the lecturers at the Faculty of Computer Science and Engineering, University of Moratuwa, for their valuable advices. I also thank to my parents and my wife, Manorika Athukorala who always offering me support and love. Finally, I thank to my colleagues Prabath Weerasinghe, Oshan Deshapriya and numerous friends who endured this long process with me.

TABLE OF CONTENTS

LIST OF FIGURES

Page

LIST OF ABBREVIATIONS

Abbreviation	Description
CCTV	Closed Circuit TV
CEP	Complex Event Processor
CPU	Central Processing Unit
cuHE	CUDA Homomorphic Encryption Library
DHT	Distributed Hash Table
ESM	Elastic Switching Mechanism
FHE	Fully Homomorphic Encryption
GPU	Graphic Processing Unit
HE	Homomorphic Encryption
HomoESM	Homomorphic Encryption based Elastic Switching Mechanism
IaaS	Infrastructure as a Service
JFR	Java Flight Recorder
MDI	Metaphysical Data Independence
PaaS	Platform as a Service
QoS	Quality of Service
RFID	Radio Frequency Identification
SaaS	Software as a Service
SIMD	Single Instruction Multiple Data
SP	Stream Processor
TPS	Transactions per Second
VM	Virtual Machine

1 INTRODUCTION

This chapter gives basic background to stream processing, cloud computing, performance and quality of service concerns of stream processing and privacy concerns of cloud-based data stream processing system. Then it describes the problem and the motivation behind this work. Finally, it summarizes our contributions of the work.

1.1 Background

Stream processing is a real-time processing of data continuously, concurrently, and in a record-by-record fashion. Stream processing treats data not as static tables or files, but as a continuous infinite stream of data integrated from both live and historical sources. Streams are actually sequences of data that flow between operators in a streams processing application. Stream processing conducts online analytical processing on high-velocity and high-volume data streams with minimal latency. The main reason for that stream processing is so fast is because it analyzes the data before it stores in disk. Stream processing has applications in diverse areas such as health informatics [1], telecommunications [23], electric grids [12], transportation [14], [16] and useful for tasks like fraud detection, logistics decisions and real-time traffic controlling systems. These applications have been implemented on stream processing engines and should satisfy several performance and quality of service metrics.

Most of the initial stream processors were ran on isolated computers or clusters which are private clouds. After that people came up with distributed stream processors and there are now several distributed stream processing engines/frameworks also available publicly such as Apache Storm [34], Apache Spark [35] and Apache Flink [36]. As a result of further evolvement, distributed stream processors touch the cloud infrastructure as well. The rise of cloud computing era has resulted in the ability of on demand provisioning of hardware and software resources. There are three main types of cloud computing: Infrastructure as a service

(IaaS), Platform as a service (PaaS) and Software as a service (SaaS). Under stream processing we are interested in IaaS, as stream processors can elastically scale their infrastructure on demand. IaaS provide several benefits other than infrastructure: Good reliability, High network speed, High performance, Global scale, Good productivity and Low cost. Public cloud services are sold on demand, typically by the minute or hour, though long-term commitments are available for many services. Organizations only pay for the Central Processing Unit (CPU), memory, storage or bandwidth they consume. This has resulted in stream processors which can run as managed cloud services [3], [9], [15] as well as hybrid cloud services such as Striim [22]. A hybrid cloud is a combination of public cloud services and an on-premises private cloud, with orchestration and automation between the two. Organizations can run mission-critical workloads or sensitive applications on the private cloud and use the public cloud to handle workload bursts or spikes on demand. The goal of a hybrid cloud is to create a unified, automated, scalable environment that takes advantage of all that a public cloud infrastructure can provide, while still maintaining control over mission-critical data. Generally, IaaS providers, supply a virtual server instance and storage, as well as APIs that enable users to migrate workloads to a VM (Virtual Machine). Users have an allocated storage capacity and can start, stop, access and configure the VM and storage as desired. IaaS providers offer small, medium, large, extra-large and memory-optimized or compute-optimized instances, in addition to customized instances, for various workload needs.

Resource limitation becomes a key issue when operating only in a private cloud. Such systems often face resource limitations during their operation due to unexpected loads [2], [5] and hit with low quality of service. Depending on the requirements, systems should fulfill good quality of service and good performance metrics such as higher throughput, low latency and higher availability. For an example, if fraud detection system fails to respond with low latency, system will not be able to stop fraudulent transactions in real-time. Therefore, people came up with several approaches which could solve such issues. Elastically scaling into an external cluster [29], load shedding and approximate query processing [33] are some of the examples. Out of these, elastic scaling has become a key choice because approaches

such as load shedding, approximate computing have to compromise accuracy which is not accepted by certain categories of applications.

Data security and data privacy remain primary concerns for organizations when they use public cloud as infrastructure. The problem of privacy-preserving data mining has become more important in recent years because of the increasing ability to store personal data about users, and the increasing sophistication of data mining algorithms to leverage this information. This has led to concerns that the personal data may be misused for a variety of purposes. In order to alleviate these concerns, a number of techniques have recently been proposed in order to perform the data mining tasks in a privacy-preserving way. Public cloud service providers share their underlying hardware infrastructure between numerous consumers, as public cloud is a multi-tenant environment. This environment demands rich isolation between logical compute resources. At the same time, access to public cloud storage and compute resources is guarded by account login credentials. Many organizations bound by complex regulatory obligations and governance standards are still hesitant to place data or workloads in the public cloud for fear of outages, loss or theft. There are several reported cloud data breaches and most recent one is Equifax Data Breach [37] which lasts from mid-May through July 2017. The hackers have accessed people's names, Social Security numbers, birth dates, addresses and driver's license numbers. They also stole credit card numbers for about 209,000 people and dispute documents with personal identifying information for about 182,000 people. However, this resistance is fading, as logical isolation has proven reliable, and the addition of data encryption and various identity and access management tools has improved security within the public cloud.

1.2 Motivation

There are many applications which require privacy preserving of data and followings are some motivational scenarios.

1. Applications which use medical databases:

   Need to secure patients' health information.

2. CCTV (Closed Circuit TV) security analytic applications:

CCTV video streams are passing into these systems from homes/shops to identify thefts while those streams need to be secured.

3. Applications which uses fingerprint databases:

Using leaked fingerprints, a third party can do miscellaneous robberies and require security on fingerprint data.

Multiple works have recently been conducted on privacy preserving data stream mining. Privacy of patient health information has been serious issue in recent times [21]. FHE (Fully Homomorphic Encryption) has been introduced as a solution [8]. FHE is an advanced encryption technique that allows data to be stored and processed in encrypted form. This provides 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 [11] (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.

1.3 Problem Statement

Current elastic scaling mechanisms for stream processing do not consider, preserving the privacy of data which sent to public cloud, for scaling purposes.

1.4 Objectives

The objectives of our research are as follows.

1. Explore privacy preserving data stream processing techniques and select suitable technology for elastic scaling mechanism.
2. Implement selected technology on elastic scaling mechanism in a private/public cloud scenario by preserving privacy.
3. Evaluation of the proposed approaches.

1.5 Contributions

We design and implement a Privacy Preserving Elastic Switching Mechanism (HomoESM) over private/public cloud system. Homomorphic encryption scheme has been used on top of this switching mechanism for preserving privacy of the data sent from private cloud to public cloud. We use two real world data stream processing scenarios called Email-Processor [27], [28] and HTTP Log Processor (EDGAR) [6] during the evaluation of the proposed approach. We derive four benchmark applications using these scenarios. 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 based equal operation provides significant results which are 10% to 17% improvement of average latency in the case of Email Filter benchmark and EDGAR Filter benchmark respectively. HE (Homomorphic Encryption) operations which consume more computations such as greater-than and less-than comparison operations, add and subtract operations, also provide beneficial results but not much as equal operation's results. Here we use data batching technique in our HomoESM implementation by creating a composite event using several plain events in order to address SIMD support given by HElib. This approach is the key advancement in our HomoESM which enables single evaluation at homomorphic level corresponds to all the events embedded in composite event. To the best of our knowledge this is the first work done on elastic scaling of stream processing using privacy preserving data analytics.

Main contributions of our research are as follows.

1. 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.

2. Development of Homomorphic Operations

We developed and optimized several homomorphic evaluation schemes such as equality, less-than/greater-than comparisons and addition/subtraction operations.

1.6 Organization of the Thesis

Chapter 2 contains the literature survey with several related works done on elastic scaling and homomorphic evaluations. In addition to that, it will describe other FHE libraries which we failed to use under data stream processing. Chapter 3 gives an overview of the stream processing software and the benchmarks used in this study. Chapter 4 elaborates implementation details and it provides the details of the system design and implementation of the HomoESM. Chapter 5 depicts evaluation details including the setup of the experiments and the analysis of the results. Chapter 6 contains the conclusion and presents potential future improvements for the system.

2 LITERATURE REVIEW

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

2.1 Stream Processing

Kleiminger *et al.* studied on implementing a distributed stream processor system based on MapReduce on top of a cloud IaaS to allow it to scale up/down elastically [18]. The main use case of their work was to implement financial algorithms on their framework. They explored how a local stream processor can be deployed in cloud infrastructure to scale to keep up with the expected latency constraints. They mainly talk about two load balancing strategies to achieve this. First one is always-on (load balanced between local and cloud) and second one is adaptive load balancing (move to cloud when the capacity is not enough in local node).

Stormy is a system developed to evaluate the "stream processing as service" concept [19]. 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 DHTs (Distributed Hash Tables) 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 intent to build a public streaming service where users can add new streams on demand; that is, register and share queries, and instantly run their stream for as long as they want. One of the main limitations in Stormy is it assumes that a query can be completely executed on one node. Stormy is unable to deal with streams for which the incoming event rate exceeds the capacity of a node which is an issue.

Data stream compression has been studied in the field of data mining. Cuzzocrea *et al.* has conducted a 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 "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 in order to

finally make the usage of advanced analysis/mining tools over data streams more effective and efficient. However, the proposed methodology trades off accuracy and precision for the reduced size.

Jeffery *et al.* has tried to address shortcomings of RFID (Radio Frequency Identification) data streams by cleaning the data streams using smoothing filters, they have proposed a middleware layer between the sensors and the application which process data streams. This middleware is responsible for making physical device issues transparent to the higher-level application by correcting them at the middleware. The layer/middleware is referred to as MDI (Metaphysical Data Independence) in their work [17].

Work done in [17] is utilized in [7] to clean and compress data generated by RFID tags deployed in a book store. In this work the data stream is compressed by removing redundant data. They have taken application and deployment semantics into account to develop an efficient data compression method for that specific domain. The MDI layer presented in [17] is customized by adding application specific compression algorithms and they claim that results in better performance than employing the generic method suggested in [17]. Yanming Nie *et al.* have done a research on inference and compression over RFID data streams [3], [20]. They have developed online compression method. The compression of the data stream is mainly achieved by identifying and discarding redundant data.

2.2 Elastic Scaling

Cloud computing allows for realizing an elastic stream computing service, by dynamically adjusting used resources to the current conditions. Waldemar *et al.* discussed how elastic computing of data streams can be achieved on top of Cloud computing [13]. Features particularly interesting for elastic stream processing in the Cloud includes handling of stream imperfections, guaranteed data safety and delivery, and automatic partitioning and scaling of applications. Load shedding is a well-studied mechanism for reducing the system load by dropping certain events from the stream. Deferred processing of data that cannot be immediately handled are

stored for later processing. The assumption of deferred processing is that the overload is limited in time. 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).

Cervino *et al.* tries 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 [42] 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 (EC2 instances) based solely on 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. In contrast to [19] Cervino *et al.*'s work is focused on running a big query efficiently which is decomposed to smaller quarries which can run on different VMs. However, none of the above-mentioned works have investigated on reducing the amount of data sent to public clouds in such elastic scheduling scenarios.

2.3 Homomorphic Encryption

Dai *et al.* have implemented homomorphic encryption library [40], [44] on GPU (Graphic Processing Unit) to accelerate computations in homomorphic level. As GPUs are more compute-intensive, they show that 51 times speedup on homomorphic sorting algorithm when compared to previous implementation. Initially we tried this library for our homomorphic encryption and computation wise it gives better speed up, but when encrypt a Java String field, it length goes more than 400K. As we need to send these encrypted values through network we faced several bandwidth problems.

2.4 WSO2 Stream Processor

WSO2 SP (Stream Processor) 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. It is available as open source software under the Apache Software License v2.0 [26]. 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 minutes [16].

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

Listing 1. Query example

When WSO2 SP receives a query, it builds a graph of processors to represent the query where each processor is an operator like filter, join and pattern. Input events are injected to the graph, where they propagate through the graph and generate results at the leaf nodes. Processing can be done using a single thread or using multiple threads, where in the latter case we use LMAX Disruptor [24] to exchange events between threads. More details of the WSO2 SP are available on [26].

2.5 Elastic Switching Mechanism

The Elastic Switching Mechanism (ESM) [29] is designed to operate SP engines between private and public cloud environments as shown in Figure 2.1 [29]. Basic idea is to have on-demand public SP engine according to the input load. This kind of a mechanism able to maintain good QoS (Quality of Service) metrics as it can automatically scale for additional loads. ESM will route data between private and public stream processing engines with taken 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.

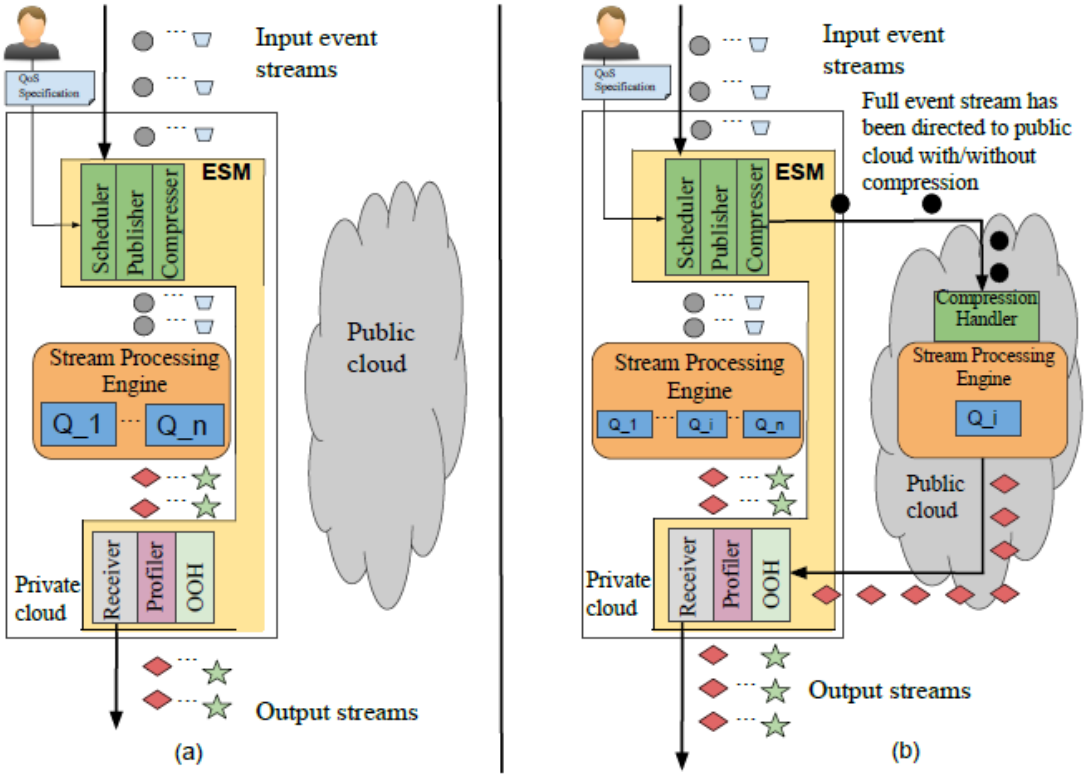

Figure 2.1: The approach for elastic compressed complex event processing. System operation with single query switched to public cloud with data switching. (a) The private cloud only mode of operation. (b) The hybrid cloud mode of operation with data switching and compression

2.6 Homomorphic Encryption and Implementations

In this section we will give introduction to homomorphic encryption and describe two main homomorphic encryption libraries.

2.6.1 Homomorphic encryption

Homomorphic encryption is a form of encryption that allows computation on cipher texts, generating an encrypted result which, when decrypted, matches the result of the operations as if they had been performed on the plaintext [39]. The purpose of homomorphic encryption is to allow computation on encrypted data. Therefore, Homomorphic encryptions allow complex mathematical operations to be performed on encrypted data without compromising the encryption.

In mathematics, homomorphic describes the transformation of one data set into another while preserving relationships between elements in both sets. The term is derived from the Greek words for *same structure*, as the data in a homomorphic

encryption scheme retains the same structure, identical mathematical operations, whether they are performed on encrypted or decrypted data will yield equivalent results.

Homomorphic encryption is expected to play an important part in cloud computing, allowing organizations to store encrypted data in a public cloud and take advantage of the cloud provider's analytic services.

There are two main homomorphic encryptions: Partially homomorphic encryption and Fully homomorphic encryption (FHE). FHE supports arbitrary computation on cipher texts 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.

To understand homomorphic encryption in a nutshell, consider a secret (s) and operation (f). If following condition in Eq. (2.1) is satisfying, 'encrypt' and 'decrypt' functions are said to be homomorphic encryption and decryption. Furthermore 'f' function is a homomorphic function.

$$\text{decrypt}(f(\text{encrypt}(s))) = f(s) \tag{2.1}$$

2.6.2 Homomorphic encryption library implementations

There are several implementations of homomorphic encryption. cuHE (CUDA Homomorphic Encryption Library) [40] is a GPU-accelerated library for homomorphic encryption (HE) schemes and homomorphic algorithms defined over polynomial rings. cuHE yields an astonishing performance while providing a simple interface that greatly enhances programmer productivity. It features both algebraic techniques for homomorphic evaluation of circuits and highly optimized code for single-GPU or multi-GPU machines.

Another popular implementation of homomorphic encryption is HElib [11]. This library is open source on github [10] and written in C++. Unlike some earlier HE schemes, HElib uses a SIMD-like optimization known as cipher text packing. As a

result, each individual cipher text element with which you can perform a computation (addition or multiplication) is conceptually a vector of encrypted plaintext integrals.

Thus, this scheme is particularly effective with problems that can benefit from some level of parallel computation. The size of this vector decides according to the settings which we given when initialize the HElib. HElib supports multi-threaded environment and we need to enable that feature while we are installing it. It provides low-level routines such as set, add, multiply, and shift.

In our case we used HElib as our homomorphic encryption library because the length of the encrypted text is around 30,000 while cuHE's encrypted text length get almost 400,000. Therefore, we have a 10 times advantage on network bandwidth when we send encrypted data into public SP.

3 OVERVIEW OF BENCHMARKS

In this section we provide short introductions to the derived benchmarks from the EmailProcessor and the EDGAR data sets which used for our experiments. These data sets consist of data fields which support our newly implemented HE functions.

3.1 Email Filter Benchmark

EmailProcessor is an application benchmark originally designed by Nabi *et al.* [27]. The benchmark is designed around the canonical Enron email dataset which is described in [28]. The data set consisted of 517,417 emails with a mean body size of 1.8KB, the largest being 1.92MB. The dataset we used had undergone an offline cleaning and staging phase where all the emails were serialized and stored within a single file with the help of Apache Avro. In our benchmark implementation [30], [38] the data injector read emails from the Avro file and de-serialized them.

The EmailFilter benchmark is a modified version of EmailProcessor benchmark which used the same dataset which EmailProcessor used. The difference is that, the EmailFilter benchmark only focus on filter operation and we need to evaluate equal comparison on string type fields. Hence, we formatted the toAddresses and ccAddresses fields to have only single email address which will contain only the first email of the list, to support HElib [10] evaluations and sent to Q1 for publishing (Figure 3.1). The criterion for filtering out Emails was to filter emails sent by "lynn.blair@enron.com" to "richard.hanagriff@enron.com". The filtering logic which used has shown in Listing 2.

```
NOT (
        (f romAddress 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. Email filter logic

Above logic which contains set of equal operations and gate operations happens at private stream processing engine using following stream and query definitions as shown in Listing 3.

define stream inputEmailsStream (iij_timestamp long, fromAddress string, toAddresses string, ccAddresses string, bccAddresses string, subject string, body string);

@info(name = 'query3') from inputEmailsStream [(str:equals(fromAddress, 'lynn.blair@enron.com') and (str:equals(toAddresses, 'richard.hanagriff@enron.com') or str:equals(ccAddresses, 'richard.hanagriff@enron.com'))) == false] select iij_timestamp, fromAddress, toAddresses as toAdds, ccAddresses as ccAdds, bccAddresses as bccAdds, subject as updatedSubject, body as bodyObfuscated insert into outputEmailsStream;

Listing 3. Email filter private VM stream processor engine definitions

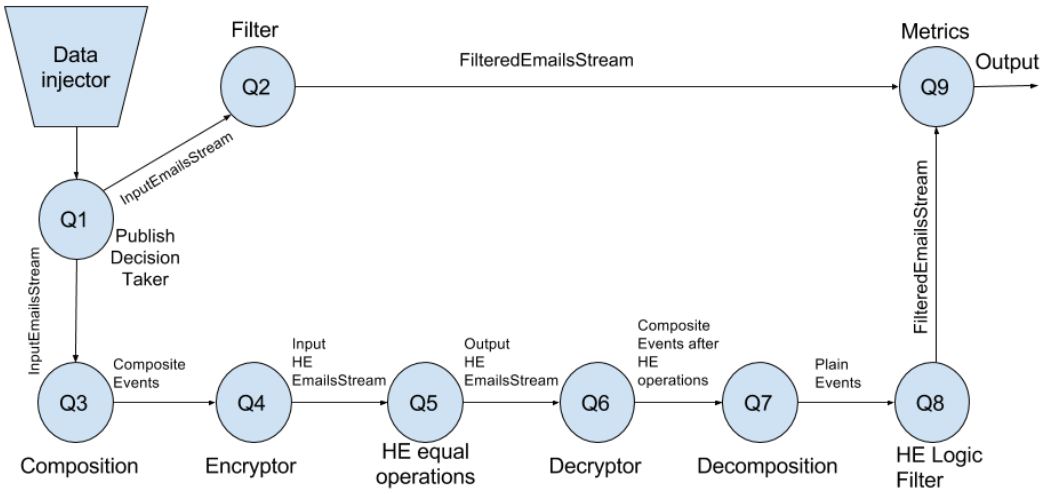

Figure 3.1: Flow diagram of Email Filter benchmarkFigure

In experiments we have fired events at 45,000 TPS (Transactions per Second) and the data rate variation is shown in Figure 3.2.

3.2 EDGAR Filter Benchmark

We developed another benchmark based on a HTTP log data set published by Division of Economic and Risk Analysis (DERA) [6]. The data provides details of usage of publicly accessible EDGAR company filings in a simple but extensive

manner [6].

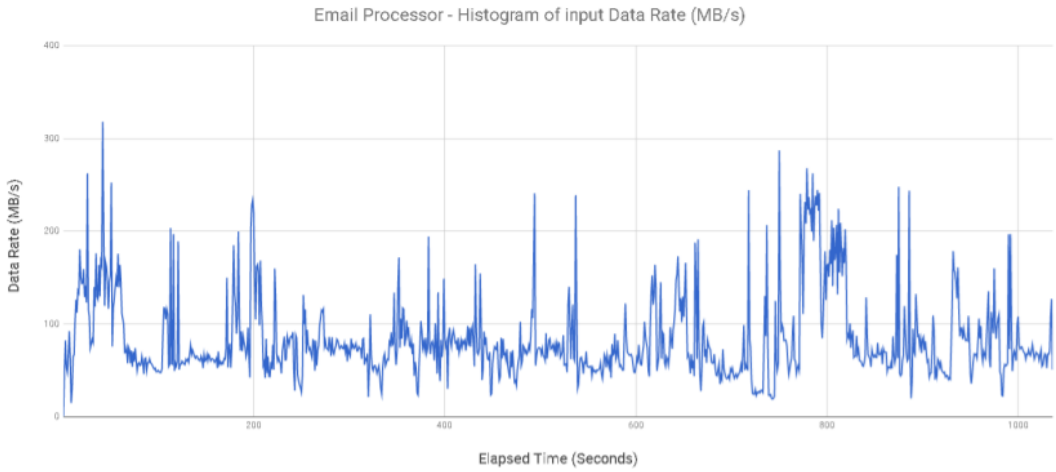

Figure 3.2: Email Processor – Histogram of input data rate

Each record in the data set consists of 15 different fields and each event we sent through the publisher had 16 fields as the benchmark application will inject the field iij_timestamp in order to track the time of event generation. Out of these fields we modified 'noagent' field by adding lengthy string of 1024 characters to the existing value, in order to increase the packet size.

Most of the EDGAR log events are same in size and it does not have any data rate variation inherently. Therefore, we introduced varying data rate by publishing events by varying TPS according to a custom defined function as in Figure 3.3.

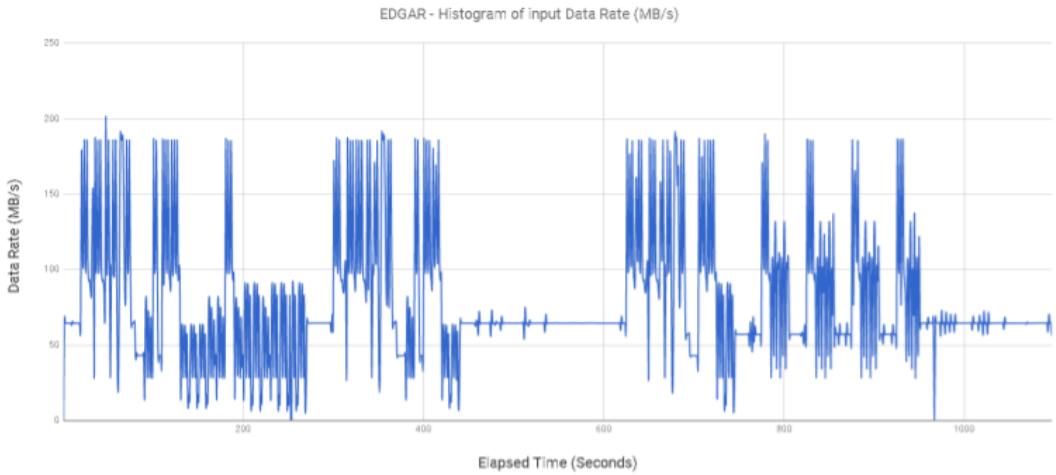

Figure 3.3: EDGAR – Histogram of input data rate

The EDGAR filter benchmark was developed with the aim of implementing filtering support. The basic criterion is to filter out EDGAR logs, which satisfy set of equal conditions as shown in Listing 4.

```
NOT (
        (extension equals ‘v16003sv1.htm’) AND
        (code equals ‘200.0’) AND
        (date equals ‘2016-10-01’)
)
```

Listing 4. EDGAR filter logic

Above logic which contains set of equal operations and gate operations happens at private SP using following stream and query definitions as shown in Listing 5.

```
define stream inputEdgarStream (iij_timestamp long, ip string, date string, time string, zone string, cik string, accession string, extension string, code string, size string, idx string, norefer string, noagent string, find string, crawler string, browser string);

@info(name = 'query5') from inputEdgarStream [(str:equals(date, '2016-10-01') and str:equals(extension, 'v16003sv1.htm') and str:equals(code, '200.0')) == false] select iij_timestamp, ip, date, time, zone, cik, accession, extension, code, size, idx, norefer, noagent, find, crawler, browser insert into outputEdgarStream;
```

Listing 5. EDGAR filter private VM stream processor engine definitions

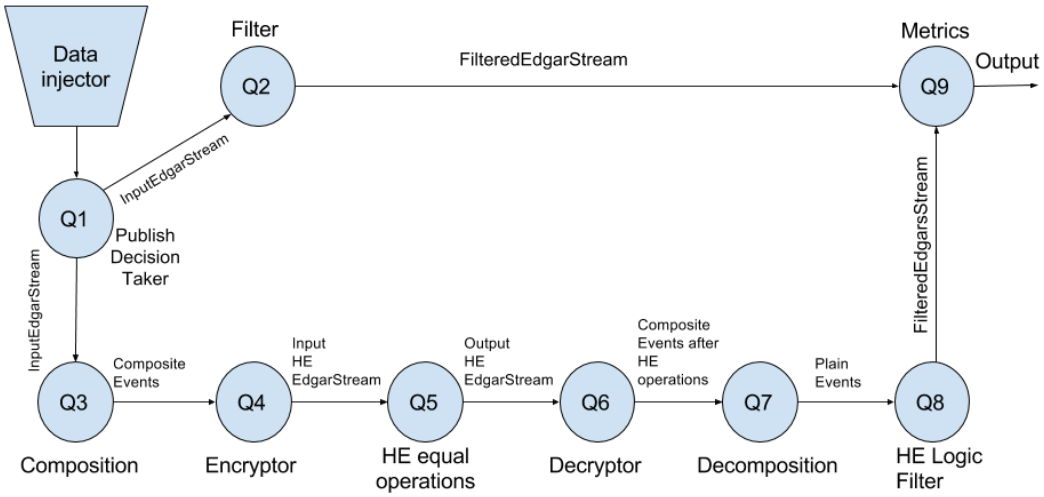

Figure 3.4: Flow diagram of EDGAR Filter benchmark

3.3 EDGAR Comparison Benchmark

EDGAR Comparison benchmark allows us to evaluate the performance of Homomorphic Comparison operation. Here also we used the same TPS varying function as previous case (Sec. 3.2) in order to have data variation with the time. In EDGAR Comparison benchmark we have changed input format to an Integer, for ‘zone’ and ‘find’ fields in order to do comparison operations. 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. The basic criterion is to filter out EDGAR logs, which satisfy following conditions.

```
NOT (
        (zone equals 0) AND (find > 0) AND (find < 3)
)
```

Listing 6. EDGAR comparison logic

Above comparison operations and gate operations happens at private SP using following stream and query definitions as shown in Listing 7.

```
define stream inputEdgarStream(iij_timestamp long, ip string, date string, time
string, zone int, cik string, accession string, extension string, code string, size string,
idx string, norefer string, noagent string , find int, crawler string, browser string);

@info(name='query5') from inputEdgarStream[((zone==0) and (find>0) and
(find<3)) == false] select iij_timestamp, ip, date, time, zone, cik, accession,
extension, code, size, idx, norefer, noagent, find, crawler, browser insert into
outputEdgarStream;
```

Listing 7. EDGAR comparison private VM stream processor engine definitions

The architecture of EDGAR Comparison benchmark has shown in Figure 3.5.

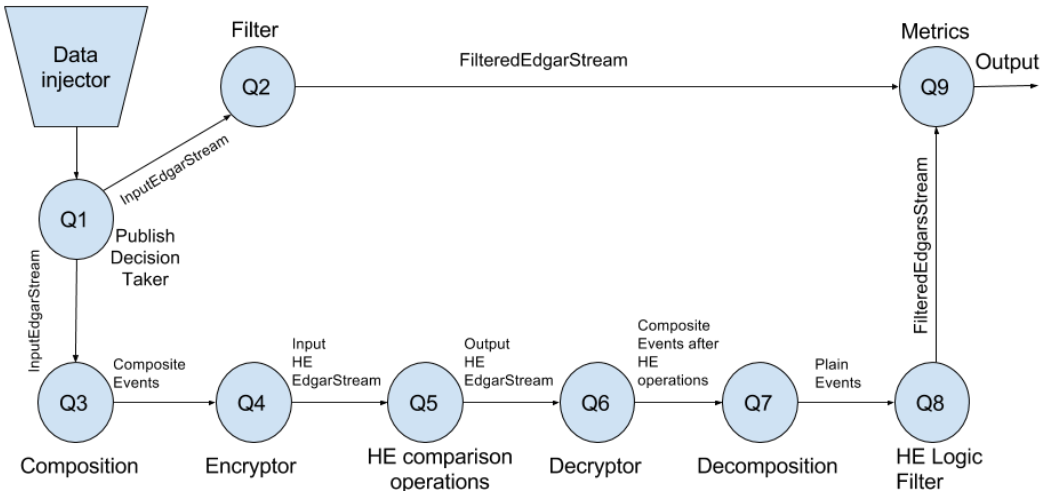

Figure 3.5: Flow diagram of EDGAR Comparison benchmark

3.4 EDGAR Add/Subtract Benchmark

EDGAR add/subtract benchmark allows us to evaluate the performance of Homomorphic add/subtract operations. Here also we used the same TPS varying function as previous case (Sec. 3.2) in order to have data variation with the time. 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. 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. The basic modifying logic we are following for this benchmark as follows for above mentioned four fields.

code = code-100
idx = idx+30
norefer = norefer+20
find = find-10

Listing 8. EDGAR add/subtract logics

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-100 as code, size, idx+30 as idx, norefer+20 as norefer, noagent, find-10 as find, crawler, browser insert into outputEdgarStream;

Listing 9. EDGAR add/subtract private VM stream processor engine definitions

Above add/subtract operations happen at private stream processing engine using following stream and query definitions as shown in Listing 9.

The architecture of EDGAR add/subtract benchmark has shown in Figure 3.6.

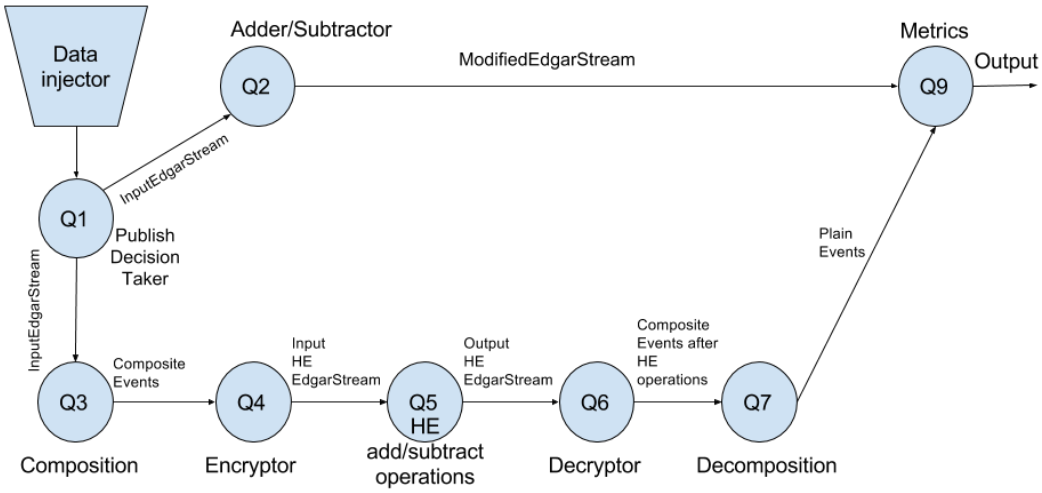

Figure 3.6: Flow diagram of EDGAR Add/Subtract benchmark

4 SYSTEM DESIGN AND IMPLEMENTATION

This chapter starts with describing the architecture of newly implemented HomoESM and its elastic scaling mechanism. Then it reveals how the encryption happens at publisher. After that it describes the different evaluation methodologies applied in public stream processing engine for four different benchmarks. Finally, it depicts how the decryption happens at receiver.

4.1 Architecture of HomoESM

In order to implement the privacy preserving streaming analytics functionality we incorporated Homomorphic encryption technique for ESM. The updated version of the ESM is known as HomoESM (Figure 4.1). The components highlighted in dark blue color correspond to newly added components which help to run privacy preserving stream processing.

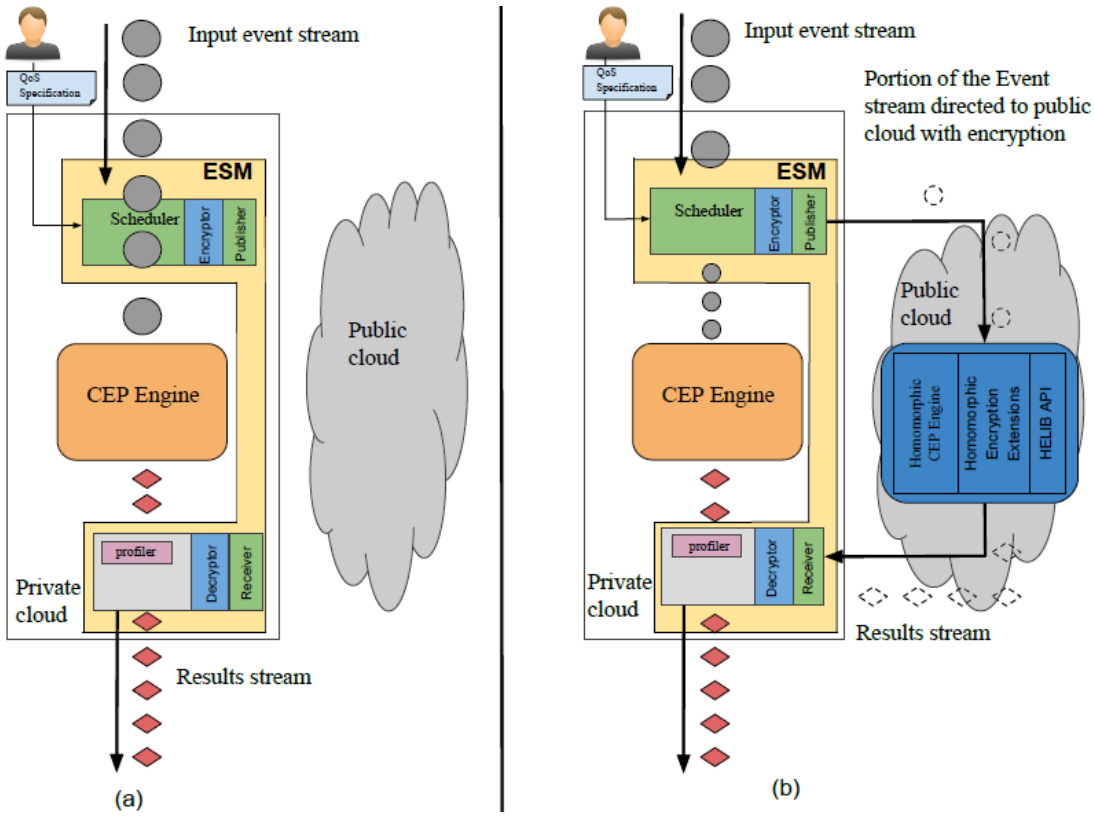

Figure 4.1: The system architecture of Homomorphic Encryption based ESM (HomoESM)

HomoESM also uses same switching functions which used by existing ESM [29]. ESM need to take decisions on following three main scenarios:

1. 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) need to be elapsed.

2. When to stop public VM –

   Anyway, VMs are not stopping in between the charging period mentioned by cloud service provider. The stopping decision will be taken at the end of charging period if following two conditions are satisfied.

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

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

3. When to send data to public VM -

   i. Public VM should up and running

   ii. 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.

   iii. Average latency measured for the last period at receiver (L_{t-1}) should be greater than Data switching threshold latency (L_d).

Following two equations [29] will summarize above conditions.

$$\emptyset_{VM}(t) = \begin{cases} 1, & L_{t-1} \geq L_s, \ \tau \ has \ elapsed. \\ 0, D_{(t-1)} < D_s, L_{t-1} < L_p & Otherwise \end{cases} \quad (4.1)$$

$$\emptyset_{data}(t) = \begin{cases} 1, \emptyset_{VM}(t-1) = 1, L_{t-1} \geq L_d, L_s > L_d \\ 0, \qquad\qquad\qquad\qquad\qquad\quad Otherwise \end{cases} \quad (4.2)$$

In next sub sections we describe main components of HomoESM as shown in Figure 4.2.

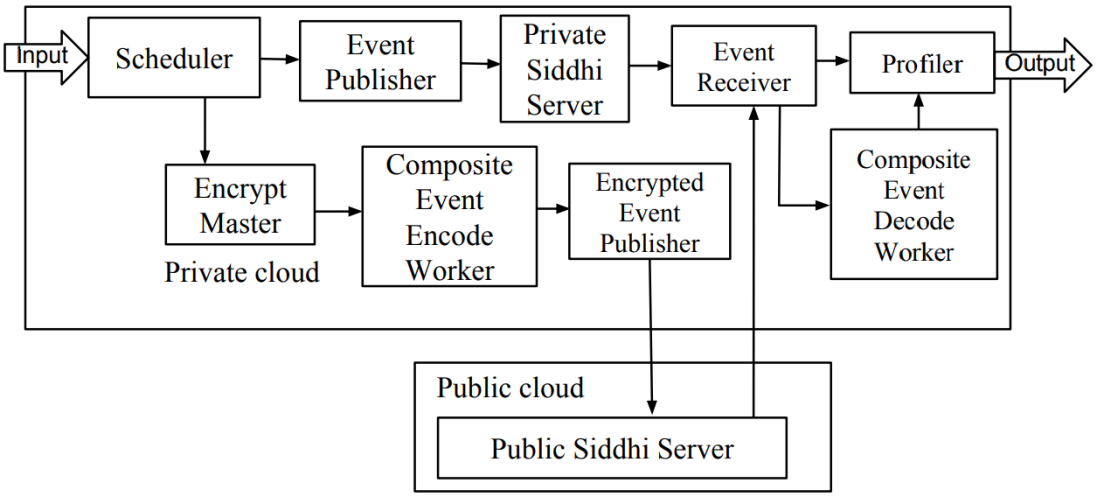

Figure 4.2: Main components of Homomorphic Encryption based ESM (HomoESM)

4.2 Encryption at Publisher

Java objects are created for each incoming event from data reader and put into the Plain Event Queue. 'Event publisher' thread picks those Java objects from the queue according to the configured frequency. Then it evaluates whether the picked event needs to be sent to private or public SP engine, according to the configured load transfer percentage and threshold values. If the event needs to send to private SP, it will mark the time and delegate the event into a thread pool which handles sending to private SP. If the event needs to send to public SP, it will mark the time and put into the public publishing queue, which is processed by Encrypt-Master asynchronously.

'Encrypt Master' (Figure 4.3) thread periodically checks on the public publishing queue which has the events required to be sent to public, put by 'Event Publisher'. 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. After that it delegates the encryption and composite event creation part of the list to 'Composite Event Encode Worker' pool.

Firing events to public VM is done in asynchronously by 'Composite Event Encode Worker' which gets triggered by the 'Encrypt Master'. Decision of how much amount send to public SP has taken according to the percentage we configure initially. But anyway, public SP publishing flow has max limit of TPS (1500 TPS for EDGAR filter benchmark, 500TPS for EDGAR comparison benchmark, 500TPS for EDGAR

add/subtract benchmark, 1000TPS for Email Filter benchmark) and if event publisher receives more than max TPS, it will be routed again into private SP VM and it will not drop any events in middle.

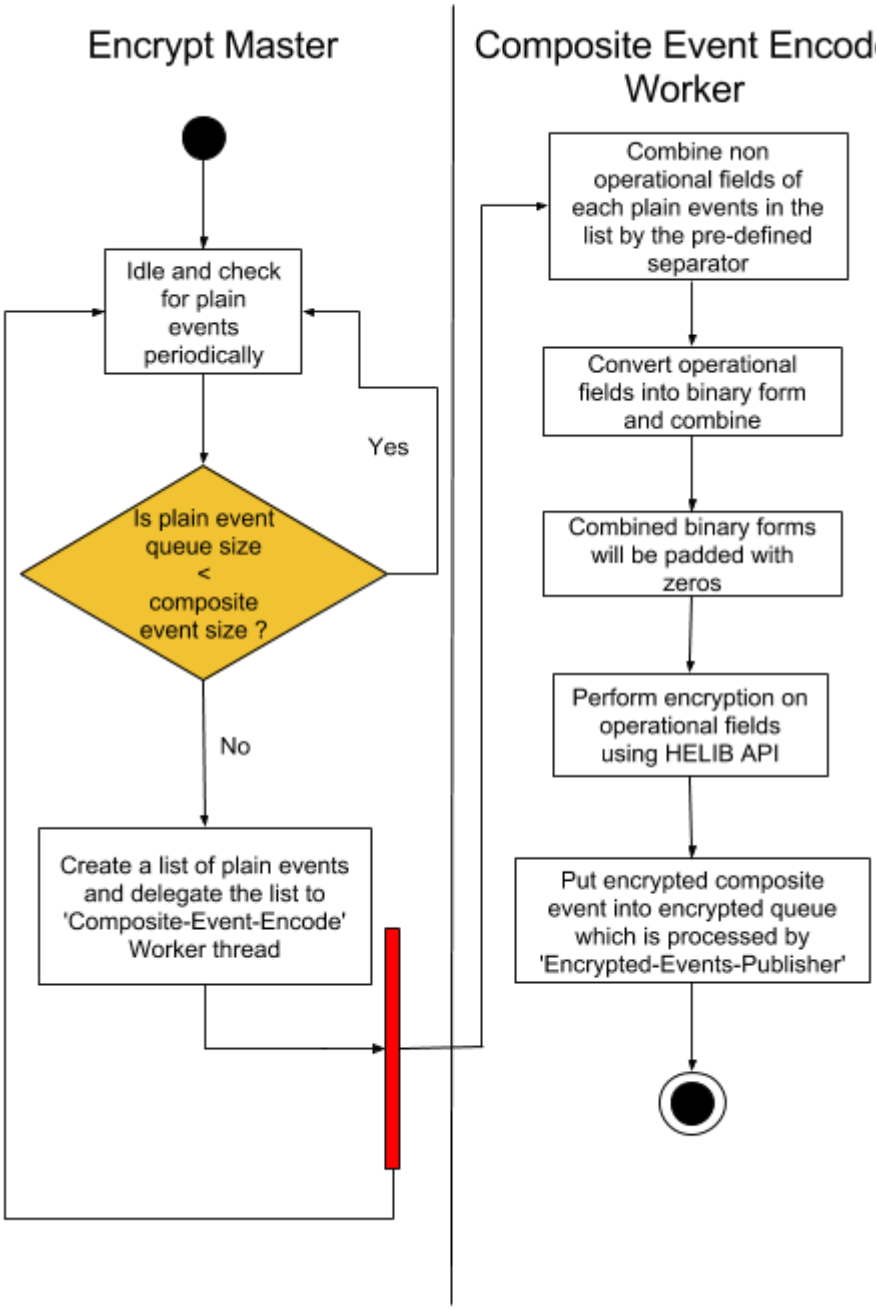

Figure 4.3: Composition and Encryption of events which need to send to public SP engine. Here red color bar depicts delegating composition and encryption to 'Composite Event Encode Worker' from 'Encrypt Master'

When firing events into public SP VM, to gain more throughput, we combine multiple plain events into a single composite event. In order to fully utilize the HElib

encrypting array we have selected different composite event sizes according to max field length for each benchmark as follows:

10 for Email filter benchmark
23 for EDGAR filter benchmark
168 for EDGAR comparison benchmark
168 for EDGAR add/subtract benchmark

There we combine the fields which will not be used for evaluations (Email filter benchmark - iij_timestamp, bccAddresses, subject, body; EDGAR equal benchmark - iij timestamp, ip, time, zone, cik, accession, size, idx, norefer, noagent, find, crawler, browser; EDGAR comparison benchmark - iij_timestamp, ip, date, time, cik, accession, extension, code, size, idx, norefer, noagent, crawler, browser; EDGAR add/subtract benchmark - iij_timestamp, ip, date, time, zone, cik, accession, extension, size, noagent, crawler, browser), with pre-defined separator and output value is a String value in composite event. For the fields which will be used for evaluations (Email filter benchmark - fromAddress, toAddresses, ccAddresses; EDGAR equal benchmark - date, extension, code; EDGAR comparison benchmark - zone, find; EDGAR add/subtract benchmark - code, idx, norefer, find) we do following transformation and encryption.

First, we convert the String value into an integer buffer according to its max length with necessary 0 padding by using ASCII value of each char. Max length will be varying according to the benchmark. Following are the max length for used benchmarks:

40 for Email filter benchmark
20 for EDGAR equal benchmark
1 for EDGAR comparison benchmark
1 for EDGAR add/subtract benchmark

Next, we combine 10 for Email filter benchmark, 23 for EDGAR equal benchmark, 168 for EDGAR comparison benchmark, 168 for EDGAR add/subtract benchmark integer buffers (according to benchmark's composite event size) into a single integer buffer and encrypt using HElib. Encrypted value will be a String with around 30K chars and composite event having those encrypted fields at the end. After that it will put composite event into the encrypted queue, which is processed by 'Encrypted Events Publisher' thread.

Private publishing pool threads directly send events to private SP server. For public SP, 'Encrypted Events Publisher' thread periodically checks for encrypted events in the encrypted queue which is put by 'Composite Event Encode Worker' after completing the composite event creation including encryption. If there are encrypted events, it will pick those at once and send to public SP server.

As a summary the Encryptor module batches events into composite events and encrypts each composite event in hormomorphic manner. The encrypted events are sent to the public cloud where Homomorphic CEP (Complex Event Processor) Engine module conducts the evaluation homomorphically. At the Homomorphic CEP engine which supports homomorphic evaluations, initially we convert constant operand into an integer buffer with size 40 for Email filter benchmark, 20 for EDGAR equal benchmark, 1 for EDGAR comparison benchmark, 1 for EDGAR add/subtract benchmark with necessary 0 padding. Then replicate the int buffer 10 times for Email filter benchmark, 23 times for EDGAR equal benchmark, 168 for EDGAR comparison benchmark, 168 for EDGAR add/subtract benchmark and encrypt using HElib [10]. Finally, the encrypted value and the relevant field in composite event was used for HElib's addition/subtraction/comparison operations homomorphically. The result was replaced with the relevant field(s) in composite event and send to Receiver without any decryption.

4.3 Homomorphic Evaluation at Public Stream Processing Engine

In this section we will discuss how the evaluation done at public stream processing engine in a homomorphic manner for our derived benchmarks. Mainly we have mentioned the details on how we evaluate events as a composite event (mini-batch)

with homomorphic support for each benchmark.

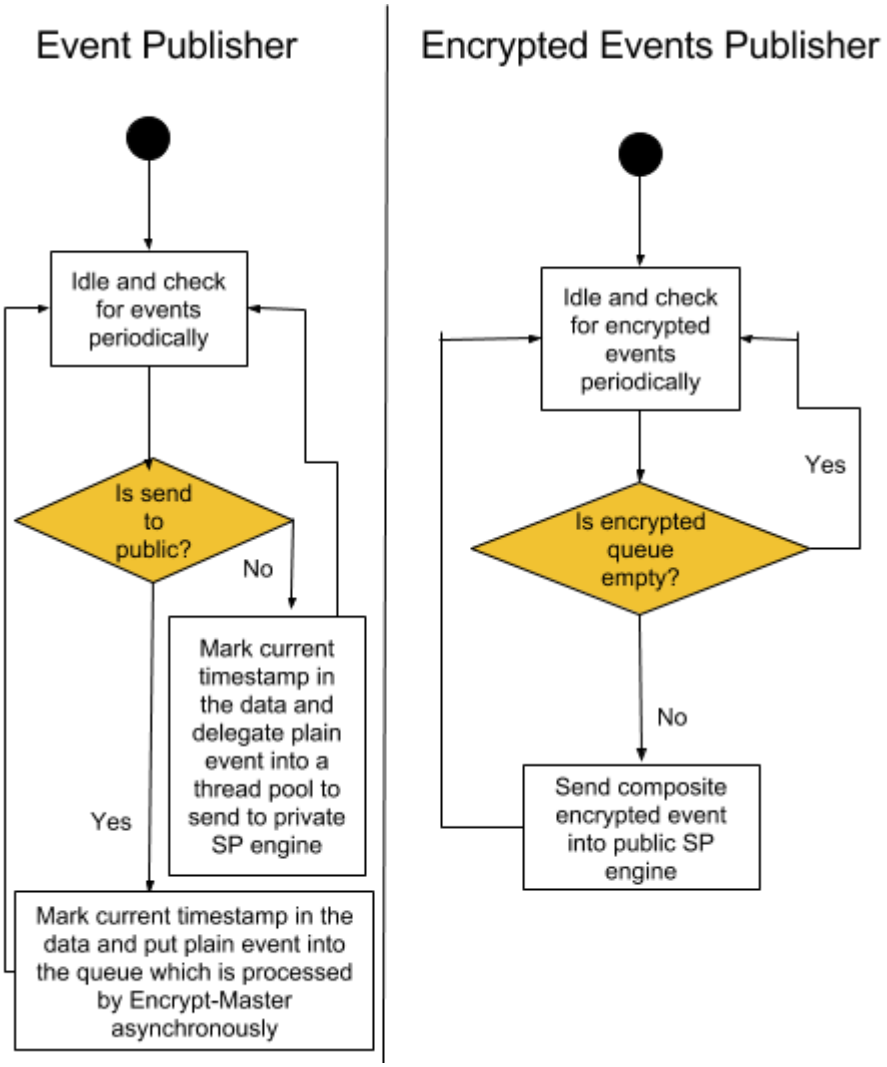

Figure 4.4: Direct event publishing into private SP engine vs Encrypted composite event publishing into public SP engine

4.3.1 Email filter benchmark

At public SP server which supports homomorphic evaluations, we initially convert all three operands as follows to support composite event with size 10. First, we convert each operand which needs to be evaluating homomorphically into an integer array with size 40 using ASCII encoding and necessary 0 padding. After that it will make a 478-length integer array by replicating 10 times 40-length integer array and necessary 0 padding. Then all three 478 length integer arrays will be encrypting using HElib. Then encrypted value and the relevant field in composite event will be used for HElib equal operations homomorphically. The result will be put into the

same field in composite event and send to statistics collector without any decryption. Relevant public VM SP stream and query definitions are listed in Listing 10.

define stream inputHEEmailsStream (iij_timestamp string, fromAddress string, toAddresses string, ccAddresses string, bccAddresses string, subject string, body string);

@info(name = 'query4') from inputHEEmailsStream select iij_timestamp, he:equalStr(fromAddress, 'lynn.blair@enron.com') as fromAddress, he:equalStr(toAddresses, 'richard.hanagriff@enron.com') as toAdds, he:equalStr(ccAddresses, 'richard.hanagriff@enron.com') as ccAdds, bccAddresses as bccAdds, subject as updatedSubject, body as bodyObfuscated insert into outputHEEmailsStream;

Listing 10. Email filter public VM stream processor engine definitions

4.3.2 EDGAR filter benchmark

At public SP server which supports homomorphic evaluations, we initially convert all three operands as follows to support composite event with size 23. We use composite event size as 23 in order to fully utilize 478-length integer array with the maximum operand size 20. First, we convert each operand which needs to be evaluating homomorphically into an integer array with size 20 using ASCII encoding and necessary 0 padding. After that it will make a 478-length integer array by replicating 23 times 20-length integer array and necessary 0 padding. Then all three 478 length integer arrays will be encrypting using HElib. Please note that this 478-number forced from HElib according to its pre-defined settings. Then encrypted value and the relevant field in composite event will be used for HElib equal operations homomorphically. The result will be put into the same field in composite event and send to statistics collector without any decryption. Relevant public VM SP stream and query definitions are listed in Listing 11.

4.3.3 EDGAR comparison benchmark

Since we are doing only bit wise operations, HElib message space should be limited to 2, in order to use only ‘0’s and ‘1’s. Therefore, maximum length for encrypting field when we used message space as 2 is 168, and we used composite event size as

168 when sending to public SP. Here each ‘zone’ and ‘find’ fields should be an Integer which can represent in two-bit numbers.

define stream inputHEEdgarStream (iij_timestamp string, ip string, date string, time string, zone string, cik string, accession string, extension string, code string, size string, idx string, norefer string, noagent string, find string, crawler string, browser string);
@info(name = 'query6') from inputHEEdgarStream select iij_timestamp, ip, he:equal(date, '2016-10-01') as date, time, zone, cik, accession, he:equal(extension, 'v16003sv1.htm') as extension, he:equal(code, '200') as code, size, idx, norefer, noagent, find, crawler, browser insert into outputHEEdgarStream;

Listing 11. EDGAR filter public VM stream processor engine definitions

First bits of zone field in 168 events will be combined and assign to the new field ‘zoneBit1’ after encryption. Similarly, second bit of zone field in 168 events will be combined and assigned to the new field ‘zoneBit2’ after encryption. Same computation is applied to ‘find’ field as well. At public SP server which supports homomorphic evaluations, we initially convert all three operands as follows to support composite event with size 168. First, we convert constant operand with two bit Integer, into two Integer buffers with size 168, representing each bit in each buffer and then encrypt using HElib. Finally, the encrypted value and the relevant field in composite event will be used for HElib comparison operations homomorphically. The result will be put as another new field in composite event for each comparison and send to statistics collector without any decryption. Relevant public VM SP stream and query definitions are listed in Listing 12.

define stream inputHEEdgarStream(iij_timestamp string, ip string, date string, time string, zoneBit1 string, zoneBit2 string, cik string, accession string, extension string, code string, size string, idx string, norefer string, noagent string, findBit1 string, findBit2 string, crawler string, browser string);

@info(name='query6') from inputHEEdgarStream select iij_timestamp, ip, date, time, he:equal(zoneBit1, zoneBit2, '00') as zone, cik, accession, extension, code, size, idx, norefer, noagent, he:lessThan(findBit1, findBit2, '00') as findLessThanSatisfied, he:greaterThan(findBit1, findBit2, '11') as findGreaterThanSatisfied, crawler, browser insert into outputHEEdgarStream;

Listing 12. EDGAR comparison public VM stream processor engine definitions

4.3.4 EDGAR add/subtract benchmark

These addition and subtraction HE operations' supported message space range is from 0 to 1201. This range also decides at HElib initialization according to the settings. According to Shaih[] if operands are small we can go with limited range support. If we need to support full range addition/subtraction operations, we need to come-up with at least 32-bit full adder circuits using HElib. Please note that, here we are not going to address that. Therefore, all operands and results should lie between the range. In EDGAR, selected fields anyway within the range and used operands and results are also within the range. Here also our composite event size would be 478 as equivalent to EDGAR filter benchmark. When creating composite event, each field correspond to a single slot and we combine above mentioned four fields separately and encrypt using HElib and assign to the respective field. Other non-operational fields will be combine using a defined separator.

At public Siddhi server which supports homomorphic evaluations, we initially replicate all four operands into 478 times and create four integer arrays separately as composite event size is 478. Then we encrypt those four integer arrays and keep in memory. In evaluation at public stream processing engine, the encrypted value and the relevant field in composite event will be used for HElib add/subtract operations homomorphically. The result will be put into the same field in composite event for each add/subtract operation and send to statistics collector without any decryption. Relevant public VM stream processing engine stream and query definitions are listed in Listing 13.

```
define stream inputHEEdgarStream (iij_timestamp string, ip string, date string, time string, zone string, cik string, accession string, extension string, code string, size string, idx string, norefer string, noagent string, find string, crawler string, browser string);

@info(name = 'query6') from inputHEEdgarStream select iij_timestamp, ip, date, time, zone, cik, accession, extension, he:subtract(code, 100L) as code, size, he:add(idx, 30L) as idx, he:add(norefer, 20L) as norefer, noagent, he:subtract(find, 10L) as find, crawler, browser insert into outputHEEdgarStream;
```

Listing 13. EDGAR add/subtract public VM stream processor engine definitions

4.4 Decryption at Receiver

'Event Receiver' thread triggers when there is an event which is received from a SP engine. First it checks whether the event is from public SP which uses Homomorphic encryption. If so it delegates composite event into 'Composite Event Decode Worker' to handle decomposition and decryptions. If event is from private SP it will directly put into profiler to read payload data and calculate the latency without any post processing.

After receiving a composite event from 'Event Receiver', 'Composite Event Decode Worker' handles all decomposition and decryptions of the composite event. It first split non- operational fields in the composite event by the pre-defined separator. Secondly it performs decryption on operational fields using HElib API and split decrypted fields into fixed-length strings depends on composite event size and create plain events using split fields. Then it checks for each operational field in the plain event, whether it contains zeros for equality and '0' OR '1' for comparison outputs. After that, it performs filtering logic according to the relevant benchmark (Listing 14, 15, 16). Please note that there is no condition checking on addition/subtraction outputs as those are not fall into filtering. Finally, it calculates latency of all decoded events which satisfy the filtering logic.

```
NOT (
        (fromAddress contains all 0s) AND
        (toAddresses contains all 0s) AND
        (ccAddresses contains all 0s)
)
```

Listing 14. Email equal benchmark's logic at receiver to determine whether the events need to be filtered out.

```
NOT (
        (date contains all 0s) AND
        (extension contains all 0s) AND
        (code contains all 0s)
)
```

Listing 15. EDGAR equal benchmark's logic at receiver to determine whether the events need to be filtered out.

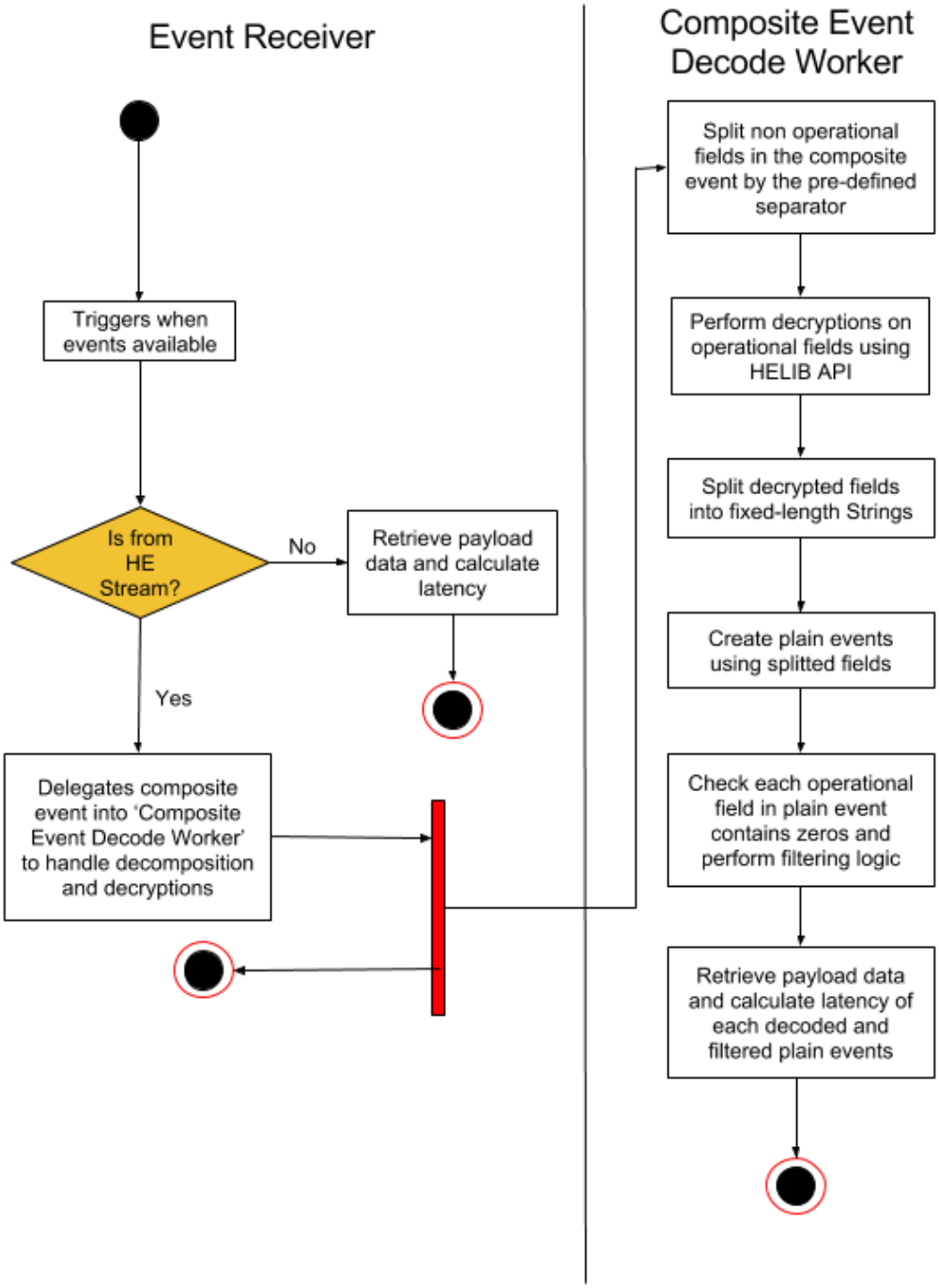

Figure 4.5: Event flow at receiver with necessary decryption. Here red color bar depicts delegating decomposition and decryption to 'Composite Event Decode Worker' from 'Event Receiver'

Note that we implemented 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. show that 4.3 and 4.4 equations will satisfy greater-than and equal operations respectively.

NOT (
(zone equals 1) AND
(findLessThanSatisfied equals 1) AND
(findGreaterThanSatisfied equals 1)
)

Listing 16. EDGAR comparison benchmark's logic at receiver to determine whether the events need to be filtered out.

$$x > y \leftrightarrow xy + x = 1 \quad (4.3)$$

$$x = y \leftrightarrow x + y + 1 = 1 \quad (4.4)$$

Listing 17. 1-bit gate circuit equations for comparison operations

Togan *et al.* has come up with comparison functions for n-bit numbers using divide and conquer methodology. In our case we derived 2-bit number comparisons as follows. x1x0 and y1y0 are the two numbers with 2-bits. Here every '+' operation is for XOR gate operation and every '.' operator is for AND gate operation.

$$x1x0 > y1y0 \leftrightarrow (x1 > y1)(x1 = y1)(x0 > y0) = 1$$

$$\leftrightarrow (x1.y1 + x1) + (x1 + y1 + 1)(x0.y0 + x0) = 1$$

$$\leftrightarrow x1.y1 + x1 + x1.x0.y0 + x1.x0 + y1.x0.y0 + y1.x0 + x0.y0 + x0 = 1$$

$$x1x0 == y1y0 \leftrightarrow (x0 + y0 + 1).(x1 + y1 + 1) = 1$$

$$\leftrightarrow x0.x1 + x0.y1 + x0 + y0.x1 + y0.y1 + y0 + 1 = 1$$

$$x1x0 < y1y0 \leftrightarrow (x1x0 > y1y0) + (x1x0 == y1y0) + 1 = 1$$

$$\leftrightarrow (x1.y1 + x1 + x1.x0.y0 + x1.x0 + y1.x0.y0 + y1.x0 + x0.y0 + x0) + (x0.x1 + x0.y1 + x0 + y0.x1 + y0.y1 + y0 + 1) + 1 = 1$$

Listing 18. Derivation of 2-bit gate circuit equations for comparison operations

Reason that we build up comparison functions for two-bit numbers is just to apply the concept of homomorphic encryption and evaluation into CEP engine. Even for 2-bit number comparisons, there are number of XOR and AND gate evaluations need to be applied as above.

5 EVALUATION

Initially this chapter describes the private/public cloud setup which we use for our experiments. Then it demonstrates our experimental results against above discussed benchmarks. Finally, it will discuss on overall results, limitations and failed attempts within our work.

5.1 Overview of Setup

We conducted the experiments using three VMs in Amazon Elastic Compute Cloud (Amazon EC2) [42]. In order to simulate private/public cloud environment we had to purchase two VMs from two different regions for private and public SP engines separately. In these experiments two VMs hosted in North Virginia, USA were used as private cloud while the VM used as public cloud was located in Ohio, USA. Initially we tried two VMs between US east and west coast for two SP engines, but network speed is around 180Mbits/sec which is very low for our experiments measured by the network speed measurement tool, iPerf [43]. Out of the two VMs in North Virginia, one was a m4.4xlarge instance which had 16 cores, 64GB RAM while the other one was a m4.xlarge instance which had 4 cores, 16GB RAM. Event publisher and Statistics collector deployed in m4.4xlarge VM while private CEP Engine deployed in m4.xlarge VM. Here we have to use high performance VM instance type m4.4xlarge, because composite event composing, and decomposing require more CPU for publisher and statistics collector. 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. Using iPerf [43] 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 5.1 shows the test setup which described above.

5.2 Email Filter Benchmark

In the first round we used Email Filter benchmark. The results of this experiment are shown in Figure 5.2. The curve in the blue color (dashed line) indicates the private only 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 only deployment.

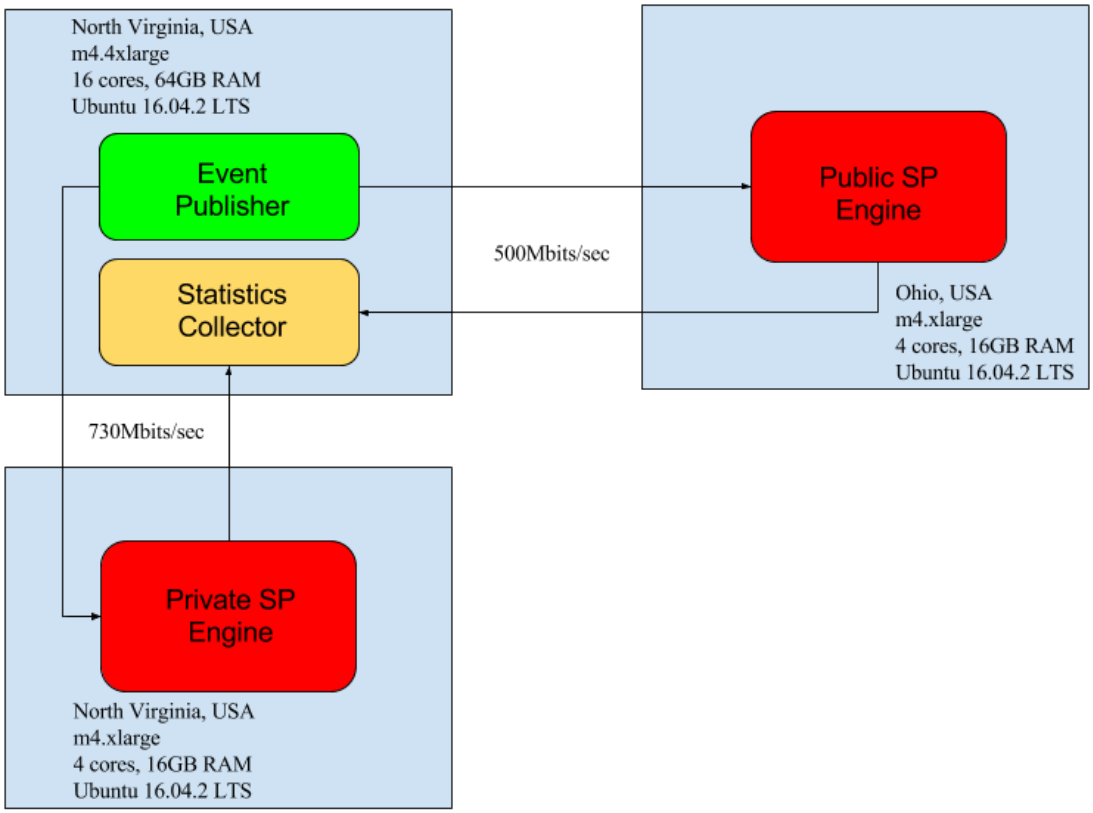

Figure 5.1: Amazon EC2 VM arrangement which used for evaluation

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 we have marked the times where VM start/VM stop operations have been invoked to start/stop the VM in the public cloud.

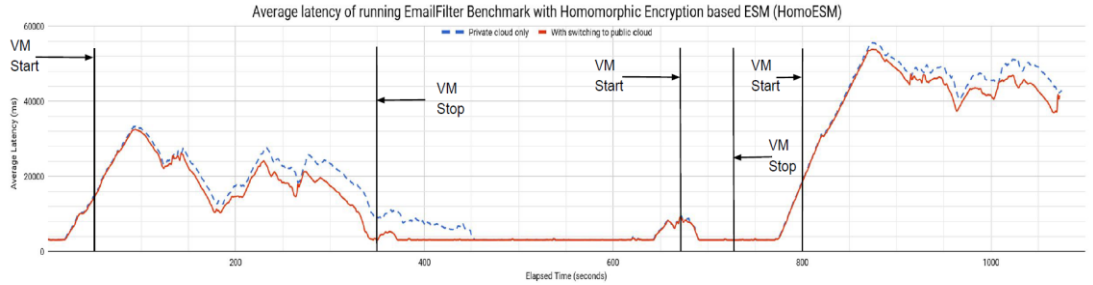

Figure 5.2: Average latency of elastic scaling of the Email Filter benchmark with securing the event stream sent to public cloud via homomorphic encryption

5.3 EDGAR Filter Benchmark

In the second round we used EDGAR Filter benchmark for evaluation of our technique. The results are shown in Figure 5.3. 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%.

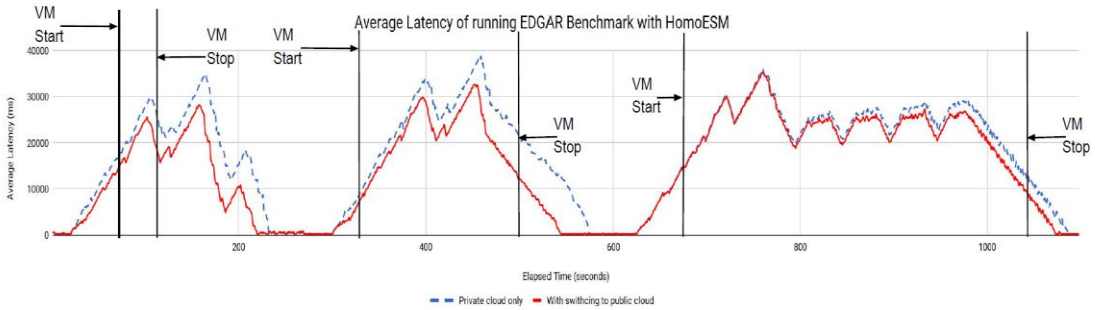

Figure 5.3: Average latency of elastic scaling of the EDGAR Filter benchmark with securing the event stream sent to public cloud via homomorphic encryption

5.4 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. The results are shown in Figure 5.4. We could see only 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 previous EDGAR Filter benchmark test which uses only homomorphic equal operation, less-than and greater-than homomorphic operations consume more XOR and AND gate operations within HElib. Due to that, evaluation complexity in public SP engine is higher and it cannot process the events in much speed when compared to previous case. Due to that, latency wise advantage is not much visible here. Therefore, ultimately the portion of events send to public SP is lesser than other cases. That is why we could not see much advantage (only 3%) on latency curves for both private and public SP case compared to private SP only case.

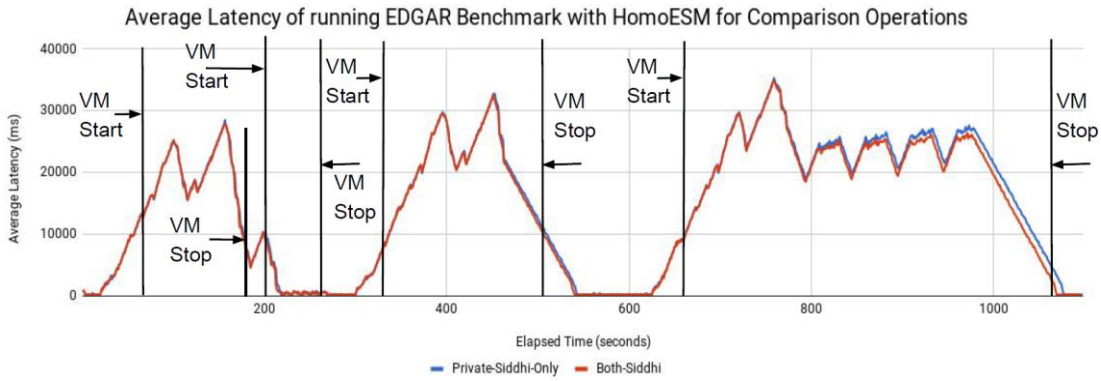

Figure 5.4: Average latency of elastic scaling of the EDGAR Comparison benchmark with securing the event stream sent to public cloud via homomorphic encryption

5.5 EDGAR Add/Subtract Benchmark

Further we evaluated homomorphic addition and subtraction operations. Here we used previously described EDGAR add/subtract benchmark. The results are shown in Figure 5.5. Here also we could see slight improvement in latency with EDGAR add/subtract benchmark similar to EDGAR comparison benchmark. Results show that there is only 3% improvement in latency compared to the private only deployment.

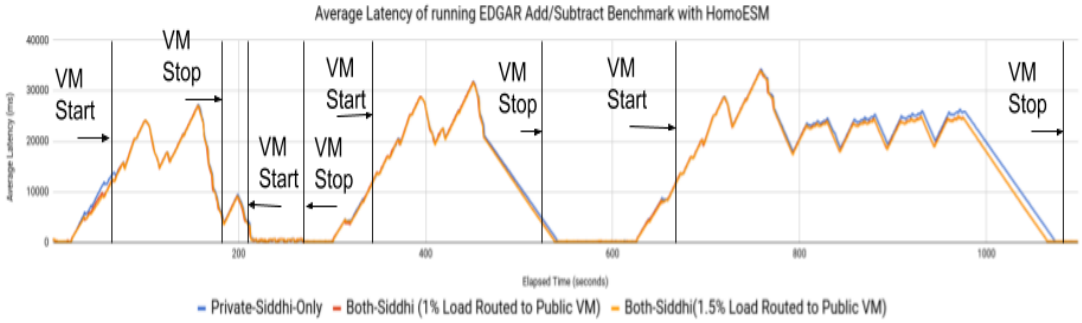

Figure 5.5: Average latency of elastic scaling of the EDGAR Add/Subtract benchmark with securing the event stream sent to public cloud via homomorphic encryption

5.6 Multiple VM Test for Email Filter Benchmark

In order to test the advantage of using multiple VMs, we performed Email Filter benchmark test again with 2 and 4 public VMs. But as we expected there is no improvement with respect to latency. Then we further investigate on the scenario and we have identified this is because when we use single public VM at current routing

load percentage (1.5%), the public VM is still not overloaded. Due to that, even though we increase the number of public VMs, our expected advantage cannot be achieved. From the other hand we cannot increase 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 as shown in Figure 5.6.

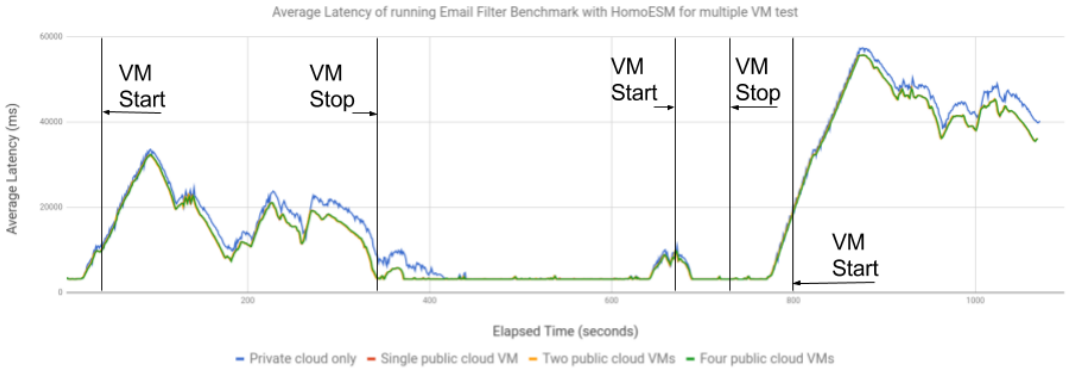

Figure 5.6: Average latency of elastic scaling of the Email Filter benchmark with securing the event stream sent to multiple public clouds via homomorphic encryption

5.7 Discussion

According to the above experiments we can see better results only in Email 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.

Apart from above EDGAR comparison and Add/Subtract benchmark experiments have limitations. EDGAR comparison benchmark experiment performs 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 and that will take longer time.

In multi VM experiment; there is a critical resource limitation at event publisher/statistics collector VM. When we routed 1.5% load into public VM, CPU

utilization is almost reach to 100%. This is due to higher CPU consumption when performing composition and decomposition by event publisher and statistics collector respectively. Figure 5.7 shows a Java Flight Recorder (JFR) output for event publisher when sending data to public SP engine.

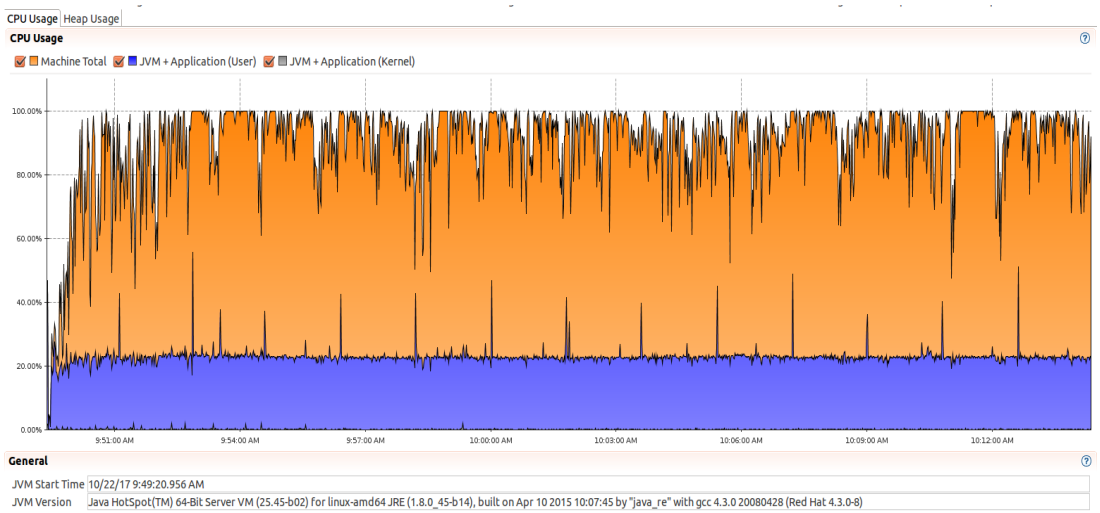

Figure 5.7: CPU utilization at event-publisher/statistics-collector VM when sending data to public SP engine

In order to reduce the average latency, we tried to change the algorithm of switching functions for VM start/stop and Data send/stop functions. Previously it used only static threshold value to take decisions. Here we came up with dynamic threshold as in Eq. (5.1) which can address the changing trend in latency.

$$\text{Dynamic threshold} = \text{Latency threshold} - (W * dy/dx) \quad (5.1)$$

If ‘Current latency’ is greater than ‘Dynamic threshold’ ESM will take decision to start VM or start data send. Even though above algorithm triggers VM start/stop and Data send/stop quickly when there is an increment or decrement in latency, the overall max load which we can send to public SP VM has limitation due to above discussed high CPU consumption. Therefore, this approach did not make any difference in terms of latency.

6 CONCLUSION AND FUTURE WORK

Privacy has become an utmost important barrier which hinders leveraging IaaS for running stream processing applications. In this research we introduce a mechanism called HomoESM which conducts privacy preserving elastic data stream processing. We evaluated our approach by using derived benchmarks based on two data sets called Email Filter and EDGAR. We observed significant improvements on overall latency of 10% and 17% for Email Processor and EDGAR datasets with using HomoESM on equality operation. We also implemented comparison and add/subtract operations in HomoESM which resulted in 3% improvement in average latency. Comparison and add/subtract operations in HomoESM are limited up to 2-bit numbers. The reason for the limitation is the complexity of the computations on higher bit numbers for comparison and add/subtract operations. There is another limitation on CPU when performing composition and decomposition at event publisher and statistics collector due to split and append computations.

In this work we use data batching technique in our HomoESM implementation by creating a composite event using several plain events in order to address SIMD support given by HElib. This approach is the key advancement in our HomoESM which enables to realize the elastic stream processing with HomoESM. To the best of our knowledge this is the first work done on elastic scaling of stream processing using privacy preserving stream data analytics.

In future we plan to extend this work to handle more complicated streaming operations such as length/time windows, event pattern matching and multiplication/division. Moreover, we can enhance HomoESM by evaluating new Homomorphic encryption libraries which leverage latest hardware like GPUs or improving current algorithms in homomorphic supported functions. We also plan to experiment with multiple query-based tuning for privacy preserving elastic scaling.

In addition to above if we can reduce CPU consumption at event publisher and statistics collector when performing composition and decomposition, we will be able to route more load to public SP engine and achieve good latency reduction.

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] R. Cocci, T. Tran, Y. Diao, and P. Shenoy. Efficient data interpretation and compression over rfid streams. In 2008 IEEE 24th International Conference on Data Engineering, pages 1445–1447, April 2008.

[4] A. Cuzzocrea and S. Chakravarthy. Event-based lossy compression for effective and efficient {OLAP} over data streams. Data and Knowledge Engineering, 69(7):678 – 708, 2010. Advanced Knowledge-based Systems.

[5] 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.

[6] DERA. Edgar log file data set. URL: https://www.sec.gov/dera/data/edgar-log-file-data-set.html, 2017.

[7] N. Dindar, . Balkesen, K. Kromwijk, and N. Tatbul. Event processing support for cross-reality environments. IEEE Pervasive Computing, 8(3):34–41, July 2009.

[8] 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.

[9] Google. Cloud dataflow. URL: https://cloud.google.com/dataflow/, 2017.

[10] S. Halevi. An implementation of homomorphic encryption. URL: https: //github.com/shaih/HElib, 2017.

[11] S. Halevi and V. Shoup. Algorithms in HElib, pages 554–571. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[12] J. Hazra, K. Das, D. P. Seetharam, and A. Singhee. Stream computing based synchrophasor application for power grids. In Proceedings of the First International Workshop on High Performance Computing, Networking and Analytics for the Power Grid, HiPCNA-PG '11, pages 43–50, New York, NY, USA, 2011. ACM.

[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] T. Hunter, T. Das, M. Zaharia, P. Abbeel, and A. Bayen. Large-scale estimation in cyberphysical systems using streaming data: A case study with arterial traffic estimation. Automation Science and Engineering, IEEE Transactions on, 10(4):884–898, Oct 2013.

[15] IBM. Streaming analytics. URL: https://www.ibm.com/cloud/streaming-analytics, 2017.

[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] S. R. Jeffery, M. J. Franklin, and M. Garofalakis. An adaptive rfid middleware for supporting metaphysical data independence. The VLDB Journal, 17(2):265–289, 2008.

[18] W. Kleiminger, E. Kalyvianaki, and P. Pietzuch. Balancing load in stream processing with the cloud. In Data Engineering Workshops (ICDEW), 2011 IEEE 27th International Conference on, pages 16–21, April 2011.

[19] 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.

[20] Y. Nie, R. Cocci, Z. Cao, Y. Diao, and P. Shenoy. Spire: Efficient data inference and compression over rfid streams. IEEE Transactions on Knowledge and Data Engineering, 24(1):141–155, Jan 2012.

[21] 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.

[22] Striim. Striim delivers streaming hybrid cloud integration to microsoft azure. URL: http://www.striim.com/press/hybrid-cloud-integration-to-microsoft-azure, 2017.

[23] 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.

[24] M. Thompson, D. Farley, M. Barker, P. Gee, and A. Stewart. High performance alternative to bounded queues for exchanging data between concurrent threads. technical paper, LMAX Exchange, 2011.

[25] M. Togan and C. Plesca. Comparison-based applications for fully homomorphic encrypted data. Proceedings of the Romanian Academy, Series A, Volume 16, Special Issue 2015, pp. 329-338.

[26] WSO2. Wso2 stream processor. URL: https://docs.wso2.com/display/SP400/Stream+Processor+Documentation, 2017.

[27] Z. Nabi, E. Bouillet, A. Bainbridge, and C. Thomas. Of streams and storms. IBM White Paper, 2014.

[28] B. Klimt and Y. Yang. Introducing the enron corpus. In CEAS 2004 - First Conference on Email and Anti-Spam, July 30-31, 2004, Mountain View, California, USA, 2004.

[29] S. Ravindra, , M. Dayarathna, S. Jayasena. Latency Aware Elastic Switching-based Stream Processing Over Compressed Data Streams, ICPE'17, April 22-26, 2017, L'Aquila, Italy.

[30] Email Processor benchmark data reader. URL: https://github.com/miyurud/EmailProcessor_Siddhi

[31] Event Publisher. URL: https://github.com/sajithshn/event-publisher

[32] Statistics Collector. URL: https://github.com/sajithshn/statistics-collector

[33] Approximate Queries on WSO2 Stream Processor. URL: https://www.infoq.com/articles/WSO2-algorithms-applied

[34] Apache Storm. URL: http://storm.apache.org/

[35] Apache Spark. URL: https://spark.apache.org/

[36] Apache Flink. URL: https://flink.apache.org/

[37] Equifax Data Breach. URL: https://www.equifaxsecurity2017.com/consumer-notice/#notice

[38] Distributed Scaling of WSO2 Complex Event Processor. URL: https://wso2.com/library/articles/2015/12/article-distributed-scaling-of-wso2-complex-event-processor/

[39] Homomorphic encryption. URL: https://en.wikipedia.org/wiki/Homomorphic_encryption

[40] Homomorphic encryption implementation - cuHE. URL: https://github.com/vernamlab/cuHE

[41] HE addition support. URL: https://github.com/shaih/HElib/issues/81

[42] Amazon EC2. URL: https://aws.amazon.com/ec2/

[43] iPerf network speed measurement tool. URL: https://iperf.fr/

[44] W. Dai and B. Sunar. cuHE: A Homomorphic Encryption Accelerator Library