STYX: Stream Processing with Trustworthy Cloud-based Execution∗

Julian James Stephenα Savvas Savvidesα Vinaithierthan Sundaramα,β Masoud Saeida Ardekaniγ† Patrick Eugsterα,β,δ

αPurdue University, βSensorHound Inc, γSamsung Research America, δTU Darmstadt

Abstract

With the advent of the Internet of Things (IoT), many billions of devices are expected to continuously collect and compute on sensitive data (e.g., location, personal health). Due to limited computation capacity available on IoT devices, the current de facto model for building IoT applications is to send the gathered data to the cloud for computation. Since building private cloud infrastructures to handle such large amount of data streams is very expensive, low cost public (untrusted) cloud infrastructures such as Amazon EC2 are used. However, using public clouds for processing continuous queries including on sensitive data is leading to increasing concerns over data confidentiality, and is a major factor preventing more widespread adoption of IoT solutions.

This paper presents STYX, a novel programming abstraction and managed runtime system, that ensures confidentiality of IoT applications whilst leveraging the public cloud for continuous query processing. The key idea is to intelligently utilize partially homomorphic encryption to perform as many computationally intensive operations as possible in the untrusted cloud. STYX provides a simple abstraction to the IoT developer to hide the complexities of (1) applying complex cryptographic primitives, (2) reasoning about performance of such primitives, (3) deciding which computations can be executed in an untrusted tier, and (4) optimizing cloud resource usage. An empirical evaluation with benchmarks and case studies shows the feasibility of our approach.

1 Introduction

The ubiquity of computing devices is driving a massive increase in the amount of data generated by humans and machines. With the advent of the Internet of Things (IoT), many more billions of devices are expected to continuously collect sensitive data (e.g., location data, personal health data) and compute on it. Due to limited storage and computation capacity available on IoT devices, the current de facto model for building IoT applications is to send the data gathered from physical devices to the cloud for both computation and storage (e.g., SmartThings [5], Nest [3]). Many IoT applications thusly leverage the cloud to compute on data streams from a large number of devices. For example, to compute variable tolls or to identify highway accidents or traffic conditions, a smart city application may collect vehicle license plate numbers, speed, and location information at the cloud.

Due to the sheer amount of the streaming data, building private cloud infrastructure is very expensive compared to low cost public (untrusted) cloud infrastructure such as Amazon EC2 or Microsoft Azure. Therefore, public clouds are typically used for processing continuous queries including on sensitive data. However, this trend is leading to increasing concerns over data confidentiality, and is becoming one of the major factors preventing more widespread adoptions of IoT solutions. For instance, a recent study, among 2,062 American consumers, shows that the top concern among consumers is “Who is seeing my data” [23].

One way to mitigate these concerns is to encrypt data at the source (i.e., IoT device), and to solely use cloud infrastructures for storage purposes (e.g., Bolt [24]). Thus, as long as encryption keys are maintained securely by consumers, their data remains secured. While this approach addresses the aforementioned confidentiality concerns, all computations need to be performed in trusted environments. This solution hugely impacts the computational capabilities available for IoT solutions.

A promising approach to tackle these issues is to use partially homomorphic encryption (PHE) techniques, and execute certain operations over encrypted data. Yet,
existing solutions use a storage system to this end. For instance, CryptDB [29] was implemented on top of MySQL, while MONOMI [36] and Talos [31] were implemented on top of Postgres. These database-centric solutions are not a good fit to build many IoT applications because IoT applications, like many other sensor network applications, are typically implemented as continuous queries in a stream processing system.

Applying PHE to stream processing applications however leads to a number of challenges which prevents programmers from doing so explicitly. In short: (G1) a number of PHE schemes exist, varying by operations supported, efficiency, etc. Efficiency incurred by individual crypto systems is of particular concern with resource-constrained IoT devices. Application developers do not necessarily have the domain knowledge for judicious choices. In addition, secret keys need to be managed appropriately. (G2) Processing continuous queries typically involves a pipeline of computing tasks and each task may have one or more instances running concurrently. The deployment profile, which maps task instances to VMs in the cloud, should make balanced use of resources to avoid bottlenecks. Yet, encryption shifts bottlenecks thus invalidating known optimization heuristics. Finally, (G3) as hinted to by their name, PHE schemes do not support arbitrary operations. Unsupported operations have to be performed on the trusted client side in plaintext form. Subsequently, either the query processing can continue on the trusted client side or the result of the operation can be re-encrypted in the scheme required by the subsequent operation’s and continued in the cloud. Deployment profiles must be cognizant of such re-encryptions.

This paper presents STYX, a novel programming abstraction and managed runtime system, that leverages PHE to provide confidentiality for IoT applications delegating online streaming jobs to the public cloud. STYX operates on streaming data without revealing any plaintext information to the untrusted cloud. Figure 1 gives a high level overview of STYX. The user designs, implements, and initiates the stream analysis programs that run in the untrusted cloud. IoT sensors are programmed to encrypt generated data before emitting them in the stream for analysis. The user may also include additional streams of encrypted private data that are required for a specific analysis.

To perform analysis in the untrusted cloud over encrypted data (whilst addressing G1-G3), STYX (C1) utilizes PHE techniques and (C2) provides efficient implementation of these techniques so they can run on IoT devices. If a specific sequence of computations cannot be performed because of limitations of PHE schemes, STYX is capable of (C3) executing the remainder of the computation in the trusted tier or (C4) re-encrypting a data stream (or specific fields in a stream) to enable further computation in the public cloud. STYX also provides an analytical modeling module which (C5) deduces the best deployment profile for the application. Finally, (C6) programmers can develop applications using the STYX API without having to know about the details of the underlying crypto system used to implement the operations that preserve confidentiality.

More specifically, in this paper we make the following contributions.

- We introduce a secure stream abstraction that exposes a high level API through which programmers can express programs that can be executed in the public cloud in a way preserving confidentiality without having to know the details of underlying crypto systems.
- Describe how STYX analyzes programs written using the STYX API and identifies the computations that can be executed purely on encrypted data and those that computations that cannot, due to the limitations of PHE. STYX tries to maximize the amount of computation performed in the cloud by splitting computation between the untrusted cloud and a small number of trusted nodes while automatically performing required re-encryptions. Fast serialization techniques and encryption pre-computation are two key techniques used to assure the efficiency of STYX.
- Propose a heuristic that analyzes the resource availabilities and requirements and generates a deployment profile that optimizes cloud usage.
- Evaluate the implementation of STYX on multiple benchmarks and case studies. Our results indicate that STYX can be used to express many real-world IoT applications, including variable toll application for smart cities, by ensuring confidentiality without any knowledge of crypto systems, and keeping low overhead.

The remainder of this paper is organized as follows. Section 2 presents an overview of our solution. Sections 3 and 4 present details of the design of STYX and its managed runtime system. Section 5 presents the implementation of STYX. Section 6 presents empirical evaluation. Section 7 contrasts with related work. Section 8 concludes with final remarks.
2 STYX Overview

STYX provides strong confidentiality guarantees against a powerful adversary with full access to servers in the cloud. We assume that the adversary can have root access to cloud servers, or even have access to the RAM of physical machines. We note that STYX’s goal is to preserve confidentiality, but not integrity or availability and hence the adversarial model we consider does not allow for arbitrary changes in the program, analysis results or data stored within the cloud. We also assume that STYX has access to a set of limited but trusted resources that is with the users. As we shall see in this paper, this environment is leveraged to perform certain computations.

2.1 Data Flow

STYX provides the same core abstractions as many stream processing systems — streams, tuples, and fields. Briefly, a single logical data unit is a field, one or more fields makes up a tuple and an unbounded stream of tuples is simply denoted as a stream. A stream in STYX is defined by the programmer declaring a schema which names the fields of the tuples in the stream. The tuples in the stream are processed in a distributed fashion. Application logic is arranged as a directed graph where vertices of the graph are computation components and edges are streams that represent the data flow between components. Application programmers write application logic for the vertices of the graph. A subset of these vertices are also designated as source vertexes. These source vertexes act as entry points for data into the graph. Source vertexes typically read data from a queue, log file, or external subscriptions. As data is generated in real time and added to the queue, it is picked up by the source vertexes and gets forwarded down the graph for processing. In order to allow secure processing, STYX allows programmers to specify if a field represents sensitive data by marking it as a sec-field (for secure field) when the schema for the field is defined. If at least one field in a tuple is a sec-field, then STYX marks the stream of such tuples as a sec-stream (for secure-stream). Figure 2 shows a graph with four vertexes with v1 designated as the source vertex. The figure also shows sec-streams s1, s2, s3 and s4. Each vertex of the graph may have multiple runtime instantiations called tasks. In Figure 2, vertex v2 has two tasks running in Node1 and v3 has three tasks running in Node2. We refer to this assignment (i.e., a specific number of tasks to each vertex) the deployment profile of the graph. As can be trivially observed, a good deployment profile is critical for scaling and good performance. A grouping clause defines how tuples in a stream are partitioned among the tasks of a vertex that receives the stream. This grouping clause is also part of the definition of the graph and is provided by the programmer. Common groupings used in STYX are as follows: (a) Shuffle grouping: tuples are distributed randomly across tasks in such a way that each task gets an equal number of tuples. (b) Field grouping: tuples are partitioned according to the field specified in the grouping and distributed among tasks. (c) All grouping: the stream is replicated across all the tasks.

2.2 Execution Flow

We now give an overview of the STYX execution flow. Figure 3 outlines the steps followed by STYX from the time a STYX graph is designed. We outline these steps here and describe detailed working of each step in subsequent sections. STYX leverages the idea that often the user has some amount of computing power, though limited, which can be trusted. We refer to these resources as the trusted tier. The compute resources in the cloud, though potentially unlimited for practice purposes, are untrusted. STYX utilizes the trusted tier for application development and compilation and uses the cloud from the deployment phase. If at all runtime computing resources are required from the trusted tier, STYX tries to minimize their usage. Application programmers use the STYX API to design a graph which contains the application logic. Details of this is outlined in Section 3.1. STYX then performs homomorphism analysis on the graph to identify the crypto systems required to execute the graph. The details of this analysis is communicated to the IoT devices which generate key pairs corresponding to the crypto systems. More details of this analysis is given in Section 3.2. Next, STYX analytically identifies the number of tasks required for different vertexes which is detailed in Section 4.1. Finally the graph is scheduled for execution as explained in Section 4.3.
Table 1: STYX crypto system

<table>
<thead>
<tr>
<th>Crypto system</th>
<th>Property</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random AES</td>
<td>Random encryption (RND)</td>
<td>-</td>
</tr>
<tr>
<td>Deterministic AES</td>
<td>Deterministic encryption (DET)</td>
<td>$x = y \iff Enc(x) = Enc(y)$</td>
</tr>
<tr>
<td>Boldyreva et al. [14]</td>
<td>Order-preserving encryption (OPE)</td>
<td>$x &gt; y \iff Enc(x) &gt; Enc(y)$</td>
</tr>
<tr>
<td>Song et al. [33]</td>
<td>Search</td>
<td>$x$ contains $y \iff Enc(x)$ contains $Enc(y)$</td>
</tr>
<tr>
<td>Paillier [28]</td>
<td>Additive homomorphic encryption (AHE)</td>
<td>$x + y = Dec(Enc(x) \times Enc(y))$</td>
</tr>
<tr>
<td>ElGamal [20]</td>
<td>Multiplicative homomorphic encryption (MHE)</td>
<td>$x \times y = Dec(Enc(x) \times Enc(y))$</td>
</tr>
</tbody>
</table>

Listing 1: STYX code for finding sum of each group in a sliding window

```java
/** Code executed when a vertex receives a tuple */
@EncOperations(operations = { "{eq}\", "{sum}\" })
public class StyxSumVertex extends StyxVertex {
    SlotBasedSum<SecField> slidingWindowGroupSums = new SlotBasedSum<SecField>(60);
    public void execute(Tuple tuple) {
        //if timing tuple ...
        emitCurrentWindowCounts(slidingWindowGroupSums);
        //else ...
        SecField group = FieldOper.getField(tuple, 0);
        SecField sum = FieldOper.getField(tuple, 1);
        slidingWindowGroupSums.updateSum(group, getCurTimeSec(), sum);
        //end if..else
    }
}

/** Class that keeps track of sum of values per group in each time slot */
public class SlotBasedSum<T> {
    ...}
```

3 STYX Secure Streams

In this section we describe the programming abstraction that STYX exposes to express secure streams and the homomorphism analysis STYX performs so that IoT devices can encrypt their output streams appropriately.

3.1 Expressing Secure Streams

Application programmers use the abstractions provided by our STYX Java API to describe a graph. We describe the abstractions that are new to STYX over typical stream processing systems. These are also summarized in Table 2.

SecField. Programmers use this class to realize the sec-field abstraction that refers to a confidential input field. For example, programmers can get a reference to the first value in a tuple as shown in Listing 1 Line 9. sec-fields can also be initialized by reading an encrypted value directly from an input stream.

SecOper. Secure operations (sec-opers) are operations provided by STYX that allow programmers to implement standard operations like multiply, add, compare and equal. Programmers use the SecOper class to perform these operations. sec-opers take sec-fields as input and return a sec-field or Boolean value as result. For example, in order to perform an addition, programmers may write code similar to SecField result = SecOper.add(operand1, operand2) in order to add two sec-fields, operand1 and operand2.

StyxVertex. This base class is extended by programmers to express the computation in a vertex of the graph. This class provides the execute() method which is invoked by STYX when a tuple arrives at vertex for execution.
Table 2: STYX Abstractions.

<table>
<thead>
<tr>
<th>STYX API</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>SecField</td>
<td>Confidential field</td>
</tr>
<tr>
<td>SecOper.add(f1, f2)</td>
<td>f1 \times f2 \mod n \quad \quad 1 \quad n represents the public key used to encrypt f1 and f2.</td>
</tr>
<tr>
<td>SecOper.multiply(f1, f2)</td>
<td>f1.a \times f2.a \quad \quad 2 \quad MHE scheme cipher text contains two components denoted here by a and b.</td>
</tr>
<tr>
<td>SecOper.compare(f1, f2)</td>
<td>f1 \times f2.b \quad \quad 3 \quad OPE scheme require comparisons over integers 128 bits and higher.</td>
</tr>
</tbody>
</table>

Listing 1 shows code snippets used in a STYX vertex class. The code is from a graph that keeps track of the sum of values in different groups within a sliding window (last one minute in this example). The input tuple contains two fields: group name and value for that group (Lines 9 and 10). The code shown retrieves the group and value fields from the input tuple (Lines 9, 10) and updates the sum for that group’s current time slot (Line 11) with the value. Every time the vertex receives a timing tuple, that signifies a minute has elapsed, it emits the sum of all groups in the current sliding window (Line 7). The object maintaining the sliding window internally contains a map and updates the group’s sum every time the updateSum() method is called using STYX’s SecureUpdateSum() method (Line 22).

Listing 2 shows just the function updateSum() from Listing 1 written without using STYX abstractions. As can be seen, this requires the programmer to explicitly read the public key (see Line 2), and perform the exact AHE evaluation operation for addition – multiplication followed by modulus using the square of the public key if using Paillier as in Line 7.

In STYX’s Java implementation typically each vertex is designed as a separate Java class. The stream emitted by each vertex is declared explicitly in the vertex itself. Once all vertices of the graph are designed, the graph itself is put together by defining the input stream of each vertex and specifying the grouping clauses.

3.2 Processing Secure Streams

In order process secure streams, STYX first performs a homomorphism analysis on the user program. The primary responsibility of this homomorphism analysis is to enable seamless transition from the program logic written for plaintext to the program logic required for executing on encrypted streams. This consists in the following functions. Once the analyzer determines the crypto systems required for each stream, it may turn out that some operations cannot be performed over the available homomorphic crypto system. To get around this issue, STYX may decide to either perform the operation in the trusted tier or re-encrypt the stream in the trusted tier. For re-encryption, STYX inserts special vertexes into the graph and marks it such that the tasks executing that vertex get scheduled on the trusted tier. After inserting re-encryption vertexes, the analyzer annotates the graph specifying for each vertex if it must be executed in the trusted tier or untrusted cloud.

Identifying encryption schemes. This step identifies the schemes that are required for the various fields based on the operations that the application wishes to perform on those fields. To apply these inferences, STYX first has to identify the different streams and their grouping clauses in the application logic. This can be derived from the graph declaration as explained in Section 3.1. Secondly, we need to identify the sequence of operations performed on each stream. STYX derives this from program annotations in each vertex class in the graph. Line 2 in Listing 1 shows a vertex class which receives a stream with two fields and declares that the first field is used for equality comparison and the second field is used for performing addition. Once STYX derives the distinct streams and operations to be performed on those streams, we can build on prior work [34] to infer the crypto systems required to execute the graph. We briefly summarize how these techniques work. Considering each stream as an operand and using the operations derived from the annotations, an expression tree is constructed similar to a parse tree where the leaf nodes are the fields in the stream and non-leaf nodes are operators. Using a lookup table which identifies the crypto system required for each operation, leaf nodes are assigned crypto systems that match the parent operator. The lookup table also identifies the crypto system of the result (which will always be same as the crypto system of the operands). Non-leaf operator nodes are assigned this resultant crypto system. There can be a mismatch between parent and child operator nodes if they do not belong to the same crypto system. In this case (unlike in [34]) a re-encryption operator is inserted in between, which converts the stream from one crypto system to the other. These re-encryption nodes are also marked to be executed on the trusted tier so that the scheduler can place the tasks correctly.

Encrypting streams. Though the IoT devices handle most of the encryptions, sometimes the programmer may have plaintext data from public streams like stock quotes which are part of application logic. Though these data are public, we may need to encrypt them for two reasons. 1. If these public data needs to be combined or
public class SlotBasedSum<T> {
  BigInteger publicKey = readPubKey();
  public void updateSum(T group, int slot, BigInteger val) {
    String[] sums = objGroupSum.get(group);
    if (sums == null) { sums = new BigInteger[this.numSlots]; init(sums, val);
      objGroupSum.put(group, sums); }
    sums[slot] = sums[slot].multiply(val).mod(publicKey.multiply(publicKey));
  }
}
Listing 2: Code for finding sum of each group in a sliding window without STYX abstractions

compared with private data already encrypted for computing in the cloud, the public data should also be under the same key for such operations to succeed. 2. Despite data being public, if it is seen in plaintext by an attacker, she can deduce more meaning about the analysis being performed. For easily integrating public data into the program, STYX provides special vertex classes that encrypt plaintext streams.

**Initialization and constants.** One of the challenges for enabling programming using encrypted fields is to handle initializations. For example, a standard operation like `sum = sum + val` usually requires `sum` to be initialized to 0 before being executed. When the same operation is performed over encrypted data, `sum` must be initialized to the encrypted value of 0, and further, encrypted using the same crypto system and key as `val`. STYX handles this by allowing the runtime to request constants from the trusted tier. For example, STYX code like `SecField sum = SecureOper.getConstant(0, val)` is used to initialize a variable to 0 (used within the `init(sums)` method in Listing 1, Line 20). STYX uses its internal meta data to identify that the field `val` is derived from the second field in the input stream and requests the trusted tier to return 0 encrypted using the same crypto system and key as the variable `val`. Note that the number of potential encryptions of a constant is bound by the number of unique fields across all streams in a graph. This allows the trusted tier to pre-compute and cache commonly used values and reduce runtime latency.

**Key management.** One of the challenges of using homomorphic encryption for IoT based streams is that the streams need to be encrypted using the same key for long durations. If a continuous query aggregates data over a sliding window, it is not possible to change the encryption key of the stream without disrupting the output. If we decide to switch over to a new key at time \( t \), aggregations in the sliding window that spans both sides of time \( t \) will fail. We work around this issue by emitting tuples encrypted under the old and new key for the time span of the sliding window in the continuous query.

4 **STYX Deployment**

In this section we describe what happens when a graph is deployed into the STYX runtime.

4.1 **Deployment Profile Generation**

Obtaining good resource utilization and scaling well for streaming systems like STYX requires specifying the number of runtime instantiations (tasks) for each vertex. To reason about this, we use a utilization metric. Utilization of a vertex for a time interval is defined as the amount of time the vertex spends processing in that time interval. For example, if a vertex spends 5 minutes in a 10 minute interval actually processing tuples and the rest of the time waiting for tuples to arrive, then it has utilization of 0.5. As utilization of a vertex approaches 1, we can assume the vertex is starting to become a bottleneck. Good scaling is usually achieved by the programmer explicitly specifying the number of tasks for each computation vertex. The programmer is perfectly suited to do this job as she understands, based on application logic, which vertexes handle more data or computation, and can correspondingly allocate more tasks for it. In STYX, when the computation graph is transformed and operations are converted to their cryptographic equivalents (e.g., an addition operation becomes product mod square of public key with Paillier) the utilization of a vertex changes substantially. This means the programmer needs to have a deep understanding of overheads of each crypto system, which is contrary to STYX’s design goals.

4.2 **Heuristic**

In order to relieve the programmer from having to know the details of the transformed graph, we propose a linear programming based heuristic which automatically converts the instance values the programmer specifies for the plaintext graph into instance values for a STYX graph. Figure 4 shows the formal representation of the heuristics
that can be consumed at the source vertex. These constraints, we try to maximize the amount of data less than the capacity of the nodes to process it. Under these constraints, we try to maximize the amount of data that can be consumed at the source vertex.

Given

\[ S: [s_1, s_2, \ldots, s_n] \]  
Available slots

\[ V: [v_1, v_2, \ldots, v_m] \]  
Vertexes

\[ A: \{[a_{11}, a_{1m}], \ldots, [a_{m1}, a_{mn}]\} \] where,

\[ a_{ij}: \text{Output from } v_i \text{ that goes to } v_j \text{ for unit input} \]

\[ C: [c_1, c_2, \ldots, c_m] \] where

\[ c_i: \text{Relative load on any } s_j \text{ processing } v_i \]

**From definitions**

\[ D_1 \leftarrow d_1 \times t_1 \]

\[ \forall 1 < i \leq m, \]

\[ D_i \leftarrow d_1 \times t_1 \times \{A_{11} + A_{12}^2 + \ldots + A_{1n}^m\} \] where,

\[ l \text{ is the length of longest path between } v_1 \text{ and } v_i \]

**Linear program constraints**

\[ \forall 0 < i < m, t_i \geq 1 \]  
All \( v_i \)'s are allocated

\[ \forall 0 < i < m, D_i < c_i \times t_i \]

**Linear program objective function**

\[ \max_t(D_1) \]

Figure 4: STYX heuristics

4.3 STYX Scheduler

The primary responsibility of the STYX scheduler is to decide on which host machines vertexes of the graph will be executed. The STYX scheduler is provided with two lists of hostnames, one that lists hosts in the untrusted cloud, and another that lists hosts in the trusted client environment. The scheduler reads the graph annotation to identify where each vertex must be executed. For components that need execution in a trusted environment, the scheduler sends the appropriate class files to the worker instances running on the trusted side. The trusted side workers have access to private keys required for encryption or cryptosystem transformation. The workers in the untrusted cloud only see the encrypted data and have access only to the public keys required to perform the homomorphic operations. The scheduler itself can run in the untrusted cloud. An attacker can try to manipulate the scheduler in two ways (i) by trying to executed trusted vertexes in the untrusted cloud and (ii) by trying to execute untrusted code in the trusted tier. Item (i) does not compromise confidentiality because the untrusted cloud does not possess the private keys required to reveal the
plain text data. This may result in availability issues, but confidentiality is retained. Item (ii) can compromise confidentiality if the attacker is successful in executing malicious code that retrieves private keys or read data when they are in plaintext while being re-encrypted. To avoid (ii), a hash of the vertexes to be executed in the trusted tier is generated before deployment. When tasks are delivered to the trusted tier for execution, the trusted tier first computes a hash of the task class and compares it with the hash generated before deployment. Execution proceeds only if there is a match.

5 Implementation

In this section we lay some of the implementation details of STYX. STYX’s processes in the cloud are implemented by modifying Apache Storm [1]. Storm is an online, distributed computation system. The core abstraction that Storm provides is called stream. Application logic is packaged into directed graphs called topologies. Vertexes of the topologies are computation components and edges represent data flows between components. There are two types of components in storm: (i) spouts that act as event generators, and (ii) bolts that capture the program logic. In other words, spouts produce the data stream which the bolts operate upon. Modifications to Storm are limited to implementing a new scheduler (Section 4.3) (by overriding the IScheduler interface) and changes to the way a Storm topology is submitted (StormSubmitter and related classes). These changes add an additional 1031 lines of code to Storm. The programming interfaces and runtime cryptographic classes that allow computations over encrypted data are packaged as a separate jar library, implemented in 3633 lines of Java code. The cryptographic classes make use of the GNU multiple precision arithmetic library GMP [6] to perform fast arbitrary precision arithmetic operations invoked using JNI.

Encryption schemes. We implement randomized encryption (RAN) using AES [17] and use CBC mode with a random initialization vector. We construct deterministic encryption (DET) using an AES pseudo-random permutation block cipher following the approach used in CryptDB [29] and use a variant of CMC mode [25] with a zero initialization vector. We use Boldyreva et al. [14] to implement order-preserving encryption (OPE), Paillier [28] crypto system to implement additive homomorphic encryption (AHE), and the ElGamal [20] crypto system to implement multiplicative homomorphic encryption (MHE).

6 Evaluation

In this section we evaluate STYX using standard benchmarks and use cases. Our goal is to understand the overhead and thus feasibility of the system, and to estimate the efficacy of the heuristics presented in Section 4.1.

6.1 Linear Road Benchmark

We use the Linear Road Benchmark (LRB) [9] that models variable toll calculation for a city or county and is quite widespread in the evaluation of stream processing systems. LRB simulates vehicles traveling through an expressway with vehicles generating position reports at fixed time intervals. Position reports contain information like expressway identifier, direction of travel, lane of travel, mile marker, offset within the mile etc. These position reports are processed by a toll levying agency (e.g., city, county) to dynamically calculate (a) the amount of toll to be levied on the vehicle (b) identify accident locations in order to alert vehicles upstream of the accident etc. LRB also specifies latency invariants like time within which a toll must calculated and the time within which an accident has to be identified. The upper limit within which the system needs to report tolls and accidents is 5 seconds. The benchmark rates the system by the highest number of expressways (L) the system can support while maintaining these invariants. The rate at which position reports are emitted for one single expressway is shown in Figure 6. As can be seen from the figure, the experiment runs for three hours and the rate of input steadily increases up to 1811 tuples per second. Figure 5(a) shows the storm topology which implements the standard linear road and Figure 5(b) shows the transformed STYX topology. Note that Figure 5(b) contains two new vertexes v6, v7 which are re-encryption vertexes that are executed within the trusted tier.

LRB baseline and hypothesis validation. We first run a baseline deployment of LRB by assigning each vertex a single task. This allows us to observe each individual vertex to see how they consume resources and verify the
hypothesis made in Section 4.1 that bottlenecks change when running on encrypted data streams. We plot utilization (cf. Section 4.1) against time for the duration of a LRB run (10784 seconds) on Storm with plaintext data (see Figure 7) and on STYX with encrypted data (Figure 8). We can observe that in Figure 7 vertexes v4 and v2 have the highest utilization values until around the 8000s mark, and after that vertex v1 becomes the node with highest load. This increase is because the number of tuples that require a toll notification increases substantially after 8000s. In the transformed STYX graph, which ran on encrypted data streams, v5 and v1 come under high load until 8000s, and after that v1 becomes the primary bottleneck. This validates our hypothesis that primary bottlenecks differ between graphs running on plaintext vs encrypted streams.

Performance of STYX deployment profile. Now we benchmark both the Storm topology graph and the transformed STYX graph using LRB. For this, both graphs are deployed in the best possible configuration so that the maximum number of highways supported can be identified. Table 3 show the results. For plaintext streams Storm supports 20 expressways, while using STYX with encrypted streams we are able to support 15 expressways.

<table>
<thead>
<tr>
<th>System</th>
<th>L</th>
<th>Time (ms)</th>
<th>Profile</th>
</tr>
</thead>
<tbody>
<tr>
<td>Storm</td>
<td>20</td>
<td>2694.44</td>
<td>5,4,1,3,2</td>
</tr>
<tr>
<td>STYX</td>
<td>15</td>
<td>2672.97</td>
<td>5,2,1,2,3,1,1</td>
</tr>
</tbody>
</table>

1 Average response time.
2 deployment profile for vertexes in order $v_1, v_2, v_3, v_4, v_5$ for Storm and $v_1, v_2, v_3, v_4, v_5, v_6, v_7$ for STYX

We also plot the response times for all notification triggering tuples, which is the time taken for the notification to be issued from the time the tuple enters the system.

The response times for STYX are shown in Figure 10 and the response times for Storm are shown in Figure 9. Response times while running STYX peaks faster than Storm, but for 15 expressways STYX is able to maintain the response time below the threshold allowed by the benchmark.

Effectiveness of analytical model. The effectiveness of the model can be evaluated by looking at how well the model converts the deployment profile for the plaintext streams to the deployment profile for the encrypted streams in STYX. Referring back to Figures 7 and 8, these graphs can also be used as a baseline to understand the model presented in Section 4.1. Vertexes with higher utilization value should get more instances to execute them. Table 4 shows the response time of STYX deployment profile of the Storm graph and corresponding STYX graph. As can be seen, the deployment profile generated by STYX results in lowest response time. This profile is also in accordance with Figure 8 which shows vertex v1 and v4 should get the highest number of instances.

<table>
<thead>
<tr>
<th>STYX deployment profile</th>
<th>Response time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5,2,1,2,3,1,1</td>
<td>2672.97</td>
</tr>
<tr>
<td>4,2,1,4,2,1,1</td>
<td>2714.3</td>
</tr>
<tr>
<td>5,4,1,3,2,1,1</td>
<td>2781.4</td>
</tr>
</tbody>
</table>

1 Deployment profile generated by STYX

Encryption overhead of IoT device. We evaluate the overhead of encryption in IoT devices. Raspberry Pi [7] is a widely popular IoT device and has the capabilities similar to devices found in highway cameras and vehicle tracking and telemetry devices. We implemented three different crypto system namely, AHE, MHE, and DET for Raspberry Pi version B. We measured the latency to encrypt 16 bytes, a typical message size and the results
Figure 9: Response time for LRB on Storm

Figure 10: Response time for LRB on STYX

Figure 11: Encryption overhead for an IoT device

Figure 12: Response time of top-10 taxi route query with key change at the start of every month

are shown in Figure 11. The encryption latency is acceptable, as it is less than 150ms even for complex schemes like AHE and MHE.

6.2 Application Case Studies

New York taxi statistics. This application finds the top 10 most frequent routes during the last 30 minutes of taxi servicing. A route is represented by a starting grid cell and an ending grid cell. The data for this application is based on a data set released under FOIL (Freedom of Information Law) and available at [?]. The input data contains the locations (latitude and longitude) and times of passenger pick ups, MD5 digest of the medallion of the taxi that picked up the passenger, trip times and drop off locations (latitude and longitude). We use a simplified version of this data which contains passenger pick up time, drop off time, and route id. The dataset contains records that span over a one year time frame. Whenever the top 10 change, the output is appended. We use this dataset to study the response time of STYX compared to plaintext streams and also to identify how key changes affect the response time. Response time is defined as the time between an input tuple that triggers a change in top 10 enters the system and when the top 10 corresponding to that tuple is printed. In order to evaluate how response time is affected when a key change as outlined in Section 3.2 is in progress, we initiate a key change at the beginning of every month. This means any data that is emitted with a timestamp within the first 30 minutes of every month will be encrypted under the old and new key. Table 5 summarizes the results of these runs. We can see that STYX completes processing the data with only an additional 25% time compared to the Storm running on plaintext stream. Furthermore, the increase in completion time or response time caused by effecting a key change every month is minimal (less than 1%). We also show the response time for the full run with key changes every month in Figure 12. In this graph we can see intermittent spikes (total of 12) in response time for some tuples around the time a key change is in progress, but the majority of the tuples (90th percentile within 31ms and 99th percentile within 818ms) respond with the same response time as when no change was in effect.

Table 5: Top-10 taxi routes

<table>
<thead>
<tr>
<th>System</th>
<th>Completion time (s)</th>
<th>Average response time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Storm</td>
<td>8106</td>
<td>36.05</td>
</tr>
<tr>
<td>STYX(^1)</td>
<td>10039</td>
<td>45.1</td>
</tr>
<tr>
<td>STYX(^2)</td>
<td>10140</td>
<td>46.61</td>
</tr>
</tbody>
</table>

\(^1\) Entire stream emitted under same key. \(^2\) Stream emitted with a new key every month.

Heartbeat analysis. Now we study how STYX can be used for a realistic, online application like a heartbeat monitor. The end user application runs on specialized hardware (the monitoring device) that is capable of fast encryption and decryption. The monitoring device counts the number of heartbeats per minute, encrypts this value and sends it to the cloud for processing and storing. The graph running in the cloud keeps track of daily,
week, monthly and yearly statistics. The end user may request to see these statistics on their device, in which case the data is retrieved from the topology and shown to the end user. The statistics are maintained by two vertexes, a “per user” vertex (v1) and an “all users” vertex (v2). User statistics are distributed across the multiple instances of v1. v1 also emits a summary of its per user statistics every minute which is grouped by week, month, or year by v2 to find the average value across all users. The client’s device emits a message every time the client requests to see a specific data point. For this application, the most critical metric is the response time, that is the time a user has to wait after requesting to see a metric until the metric is actually displayed. The results of this evaluation are presented in Figure 13. As can be seen, the response times for STYX are very close to the plaintext version for up to 1000 client devices after which STYX’s response time degrades compared to the plaintext stream running on Storm. This demonstrates that for applications where the input rate is not very high, we are able to get response times similar to plaintext streams.

7 Related Work

To the best of our knowledge, STYX is the first system to address the issue of data confidentiality in the context of continuous query execution using partially homomorphic encryption. In what follows, we discuss prior work in the areas of stream processing and cloud security.

Large body of works on stream and data flow processing about a decade ago has addressed many challenges related to stream processing such as fault tolerance, continuous query processing and analytics (e.g., Aurora project [16, 26, 11], and Telegraph project [15, 32]). With the advent of cloud computing, the need for highly scalable stream processing systems has resulted in the next generation of stream processing systems such as Storm [1], Heron [27], Spark streaming [37], and Samza [4]. STYX design is based on the stream processing design of Storm, noting that our contributions can also be implemented on top of other stream processing.

Gentry introduced an implementable fully homomorphic encryption scheme [21] which has been becoming more practical ever since [22], but is still not suited for encrypted query processing due to its prohibitive cost. Instead, a lot of research work is focused on using partially homomorphic encryption schemes to perform computations over encrypted data. Most prominently, CryptDB [29] is a database system, focusing on executing SQL queries on encrypted data. Monomi [36] extends CryptDB to handle analytical queries and introduces techniques to improve performance. Both of these systems follow a centralized database design and do not consider streaming workloads as supported by STYX. As such it is unclear whether CryptDB or Monomi can effectively scale up to typical big data workloads. MrCrypt [35] consists in a program analysis for MapReduce jobs that tracks operations and their requirements in terms of PHE. When sequences of operations are applied to a same field, the analysis defaults to FHE, noting that the system does not currently execute such jobs at all due to lack of available FHE crypto systems.

Differential privacy [19, 18, 13] is another approach that aims to improve the accuracy of statistical queries, without revealing information about individual records. The server performing the query in this case is trusted with access to plain text data. This is in contrast to STYX that assumes data and computation reside in an untrusted server.

Another way of offering data confidentiality is through the use of specialized hardware that provide a trusted computing base [12, 10, 8]. These approaches are orthogonal to the approach of STYX and could be used to extend the design of STYX by allowing secure computations to be performed on a trusted hardware in an untrusted cloud. Therefore, reducing the amount of computation that needs to be performed in the trusted tier.

8 Conclusion

In this paper, we presented STYX, a practical distributed system for evaluating continuous queries over encrypted data streams. STYX makes use of a set of partially homomorphic encryption schemes that allow computation on encrypted data. This allows STYX to leverage untrusted public cloud resources as its compute platform without sacrificing confidentiality. STYX exposes an API which allows programmers to develop secure applications with little or no knowledge of the underlying crypto systems and STYX heuristics ensure a deployment that optimizes cloud usage. We evaluated our approach using standard benchmarks and applications, demonstrating its applicability and performance. Our evaluations show that we can meet the latency requirements even when the volume of encrypted traffic is high.
References


