STYX: Stream Processing with Trustworthy Cloud-based Execution *

Julian James Stephen[α] Savvas Savvides[α] Vinaitheerthan Sundaram[α,β]
Masoud Saeida Ardekani[α †] Patrick Eugster[α,β,δ]

[α]Purdue University, [β]SensorHound Inc, [δ]TU Darmstadt

Abstract

With the advent of the Internet of Things (IoT), billions of devices are expected to continuously collect and process sensitive data (e.g., location, personal health). Due to limited computational 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. While private cloud infrastructures for handling large amounts of data streams are expensive to build, using low cost public (untrusted) cloud infrastructures for processing continuous queries including on sensitive data leads to concerns over data confidentiality.

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.

Keywords IoT, Confidentiality, Stream data processing,

Categories and Subject Descriptors C.2.4 [*Distributed Systems*]: Distributed applications; D.4.6 [*Security and Protection*]: Cryptographic controls

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 [6], Nest [1]). Many IoT applications therefore 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, 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 a private cloud infrastructure is very expensive compared to using a 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 adoption of IoT solutions. For instance, a recent study, among 2,062 American consumers, shows that the top concern is *"Who is seeing my data?"* [25].

One way to mitigate these concerns is to encrypt data at the source (i.e., IoT device), and to solely use cloud infrastructure for storage purposes (e.g., Bolt [26]). Thus, as long as encryption keys are maintained securely by consumers, their data remains secured. While this approach addresses

* Financially supported by Northrop Grumman Cybersecurity Research Consortium, Amazon EC2, Cisco Systems grant #585544, DARPA grant #N11AP20014, as well as NSF TC grant #1117065 and NSF TWC grant #1421910. P. Eugster was partly supported by European Research Council under grant FP7-617805 "LiVeSoft – Lightweight Verification of Software" and German Research Foundation under grant SFB-1053 "MAKI – Multi-mechanism Adaptation for Future Internet".

† Now at Samsung Research America.

SoCC '16, October 05 - 07, 2016, Santa Clara, CA, USA.

ACM ISBN 978-1-4503-4525-5/16/10. . . $15.00.
DOI: http://dx.doi.org/10.1145/2987550.2987574

the aforementioned confidentiality concerns, *all computations* need to be performed in trusted environments. This solution strongly limits the computational capabilities available for IoT solutions.

A promising approach to tackle these issues is to use *homomorphic encryption* and execute all operations over encrypted data. However, *fully* homomorphic encryption (FHE) is prohibitive in practice [23], causing slowdowns by an order of $10^9\times$. An alternate, practical approach is to use less expensive *partially* homomorphic encryption (PHE) to execute specific operations over encrypted data. Yet, existing solutions use a storage system to this end. For instance, the seminal CryptDB [31] was implemented on top of MySQL, while MONOMI [36] and Talos [32] were implemented on top of Postgres. These database-centric solutions are not a good fit for many IoT applications because IoT applications are typically implemented as continuous queries in a stream processing system.

A straightforward application of PHE to existing stream processing solutions to support computations over encrypted data is however unlikely to be practical: (G1) Dozens of PHE *schemes* exist, varying by operations supported, efficiency, size of encrypted data etc.; IoT application developers do not necessarily possess sufficient in-depth knowledge of crypto(graphic) schemes to judiciously select among these. For resource-constrained IoT devices, efficiency incurred by individual crypto systems is a major concern. (G2) Moreover, encryption typically increases the size of input data, with different factors for different crypto systems. Specific optimizations are required to reduce this size difference to make the solution practical. (G3) Furthermore, due to the limitations of PHE schemes, special handling is required for variable initializations and constants in a program. (G4) With applications running continuously and potentially infinitely, secret keys used for encryption need to be updated periodically or on demand (e.g., when there is a compromise). Such updates on the IoT devices should be made transparently to the IoT application and should not lead streaming queries to miss results. (G5) 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. While some optimization heuristics are known, they do not consider encryption which shifts bottlenecks. (G6) Finally, as hinted to by their names, PHE schemes do not support arbitrary operations. Unsupported operations have to be performed on the trusted client side in a plaintext form. Consequently, either the query processing can continue on the trusted client side or the intermediate results can be re-encrypted to the schemes required by subsequent operations and continued in the cloud. Deployment profiles must be cognizant of such re-encryptions.

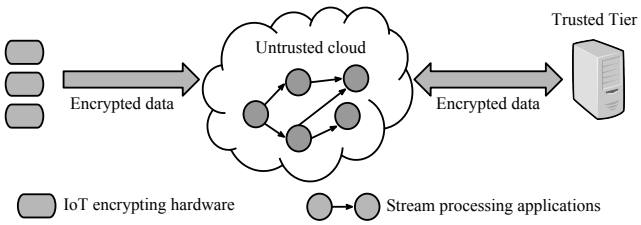

Figure 1: STYX overview

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. A user designs, implements, and initiates the stream analysis program that runs in the untrusted cloud. IoT sensors automatically encrypt generated data before emitting them in the stream for analysis. Additional streams of encrypted private data required for analysis can be sent independently from a trusted tier maintained by the user.

To perform analytics in the untrusted cloud over encrypted data (whilst addressing G1-G6), STYX: 1. enables programmers to develop applications using the STYX API for typical plaintext streams and automatically transforms the application to work with encrypted streams. This means that developers will only focus on application logic and not on the details of the underlying crypto systems (G1); 2. handles variable initializations and constants correctly through transformations (G2 and G3); 3. utilizes PHE-specific optimization techniques (e.g., field masking) to reduce encrypted data size and provides efficient implementation of these techniques so they can run on IoT devices (G3); 4. supports transparent update of secret keys on IoT devices (G4); 5. deduces the best way to deploy an application through an analytical modeling module (G5); and 6. is capable of executing the remainder of the computation in the trusted tier or re-encrypting a data stream (or parts of it) to enable further computation in the public cloud if a given sequence of computations cannot be performed due to PHE limitations (G6).

This paper makes 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. We also introduce STYX, a system that support this API and addresses related challenges.
- Describe how STYX analyzes programs written using the STYX API and identifies the computations that can

Table 1: STYX crypto systems

Crypto system	Operation
Random AES	–
Deterministic AES	$x_1 = x_2 \iff E(x_1) = E(x_2)$
Boldyreva et al. [17]	$x_1 > x_2 \iff E(x_1) > E(x_2)$
Paillier [30]	$x_1 + x_2 = D(E(x_1) \psi E(x_2))$
ElGamal [21]	$x_1 \times x_2 = D(E(x_1) \psi E(x_2))$

be executed purely on encrypted data and the computations that cannot, due to the limitations of PHE. STYX maximizes 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 techniques used to assure the efficiency of STYX.

- Propose a heuristic that analyzes 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 smart city toll applications, while ensuring confidentiality transparently and keeping low overhead.

The remainder of this paper is organized as follows. Section 2 presents background information on PHE and continuous queries. We give an overview of our solution in Section 3. Sections 4 and 5 present the design of STYX and its runtime system respectively. We discuss implementation details of STYX in Section 6. We present our evaluation results in Section 7. Section 8 contrasts with related work, and Section 9 concludes with final remarks.

2. Background

In this section we present background information on partially homomorphic encryption (PHE) and systems that support continuous queries.

2.1 PHE

A crypto system is said to be *homomorphic* (with respect to certain operations) if it allows computations (consisting in such operations) on encrypted data. If $E(x)$ and $D(x)$ denote the encryption and decryption functions for input data x respectively, then a crypto system is said to be homomorphic with respect to ϕ if $\exists \psi$ s.t.

$$D(E(x_1) \psi E(x_2)) = x_1 \phi x_2$$

E.g., a crypto system is said to provide *additive homomorphic encryption* (AHE) when $\phi \leftarrow +$. Other homomorphisms are with respect to operators such as "$\times$" (*multiplicative homomorphic encryption* – MHE), "$\leq$" and "$\geq$" (*order-preserving encryption* – OPE) or equality comparison "=" (*deterministic* encryption – DET).

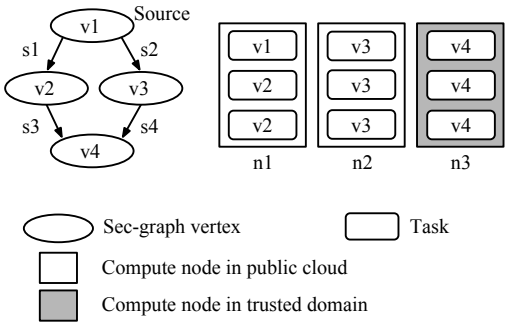

Figure 2: STYX graph and tasks

2.2 Continuous Queries

The core abstractions offered by systems that support continuous queries are *streams*, *tuples*, and *fields*. Briefly, a single logical data unit is a field, one or more fields make up a tuple and an unbounded stream of tuples is denoted as a stream. The tuples in the stream are processed in a distributed fashion. Application logic is arranged as a directed graph where vertexes of the graph are computation components and edges are streams that represent the data flow between components. Application programmers write application logic for the vertexes of the graph. A subset of these vertexes 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 forwarded down the graph for processing according to a specified grouping clause provided by the programmer.

Figure 2 shows a graph with four vertexes with $v1$ designated as the source vertex. The figure also shows 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 Node 1 (a *node* represents a virtual or physical machine) and $v3$ has three tasks running in Node 2. We refer to this assignment (i.e., a specific number of tasks to each vertex) the *deployment profile* of the graph.

The stream emitted by each vertex is declared explicitly in the vertex itself. Once all vertexes of the graph are designed, the graph is assembled by defining the input stream of each vertex and specifying grouping clauses.

3. STYX Overview

In this section, we give an overview of STYX's threat model, program abstractions, and runtime execution flow.

3.1 Threat Model

STYX provides strong confidentiality guarantees against an adversary with full access to servers in the cloud: the ad-

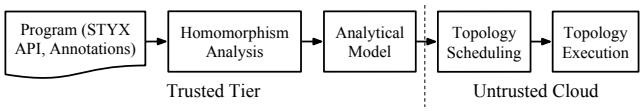

Figure 3: STYX execution flow

versary can have root access to cloud servers. This means that the adversary may inspect all data in the server and even examine RAM or CPU caches of physical machines. STYX's goal is to preserve confidentiality against such an adversary. We note that STYX's goal is to preserve confidentiality and not integrity or availability. We assume that STYX has access to a set of limited but trusted resources outside the cloud. As we shall see shortly, this environment is leveraged to perform certain computations.

3.2 STYX Abstractions

One of the main challenges of computing over encrypted data is that the application developer needs to have a detailed understanding of each crypto system used to encrypt fields of a stream. Adoption of PHE and even FHE for generic application development will depend on the ease with which a programmer can incorporate the properties offered by the crypto system into their regular programming tasks. STYX tackles this problem by offering simple programming abstractions to express and operate on encrypted data streams. E.g., fields representing sensitive data are defined using the *secure field* abstraction, irrespective of the kind of operation that needs to be performed on them or the underlying crypto system used to represent the field. STYX relieves the programmer from the task of identifying the crypto system required for the operation that the programmer needs to perform. To this end, a programmer simply annotates the stream with the desired operation and STYX deduces the crypto system that needs to be used at the source IoT devices.

3.3 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. Application programmers use the STYX API to design a graph which contains the application logic. STYX then performs *homomorphism analysis* on the graph to identify the crypto systems required to execute the graph in a confidential manner. The details of this analysis are communicated to the IoT devices which generate key pairs corresponding to the crypto systems. A detailed explanation of these steps are given in Section 4.

Next, STYX analytically identifies the number of tasks required for different vertexes. It then schedules the graph for execution. STYX leverages the idea that often users have some limited (but trusted) computing resources. We refer to these resources as the *trusted tier*. The compute resources in the cloud, though potentially unlimited for practical purposes, are untrusted. STYX utilizes the trusted tier for application development and compilation and uses the cloud for the deployment phase. In deployments that require resources from the trusted tier, STYX tries to minimize their usage. The detail of deployment steps are presented in Section 5.

4. STYX Secure Streams

In this section, we describe the programming abstractions used in STYX and explain how these abstractions are leveraged for improving performance.

4.1 Program Model

Application programmers use the abstractions provided by our STYX Java API to specify a graph. In what follows, we explain the abstractions that are new to STYX over typical stream processing systems such as Storm. These abstractions are also summarized in Table 2.

STYX API. STYX exposes its programming abstractions using an API described below.

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

`SecOper`. Secure operations 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. Secure operators take secure fields as input and return a secure field or Boolean value as result. E.g., in order to perform an addition, programmers may write code similar to `SecField result = SecOper.add(operand1, operand2)` in order to add two secure 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 a vertex for execution.

Stream annotations. Along with using the STYX API, programmers also annotate each stream with the operations they want to perform on that stream. E.g., `StyxSumVertex` in Listing 1 shows a secure stream with two fields, and the vertex implementing the standard `Group By..Sum` operation. For this, the programmer uses the definition `EncOperations( operations = { "{eq}", "{sum}" })`. This shows that the first field is used in equality comparisons and the second field in summing. These annotations enable our compile-time graph analysis to identify the crypto systems for performing these operations and to apply additional performance improvement techniques introduced in Section 4.2.

```
1  /** Code executed when a vertex receives a tuple */
2  @EncOperations(operations = { "{eq}", "{sum}" })
3  public class StyxSumVertex extends StyxVertex {
4      SlotBasedSum<SecField> slidingWindowGroupSums = new SlotBasedSum<SecField>(60);
5      public void execute(Tuple tuple) {
6          //if timing tuple ...
7          emitCurrentWindowCounts(slidingWindowGroupSums);
8          //else ...
9          SecField group = FieldOper.getField(tuple, 0);
10         SecField value = FieldOper.getField(tuple, 1);
11         slidingWindowGroupSums.updateSum(group, getCurTimeSec(), value);
12     }
13 }
14 /** Class that keeps track of sum of values per group in each time slot */
15  public class SlotBasedSum<T>  {
16     ...
17     public void updateSum(T group, int slot, SecField value) {
18         SecField[] sums = objGroupSum.get(group);
19         if (sums == null) {
20             sums = new SecField[this.numSlots];
21             init(sums, value);
22             objGroupSum.put(obj, sums);
23         }
24         sums[slot] = SecureOper.add(sums[slot], value);
25     }
26 }
```

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

Table 2: STYX abstractions

STYX API	Function
`SecField`	Confidential field
`SecOper.add(f1, f2)`[1]	`f1` × `f2` *mod* `n`[2]
`SecOper.multiply(f1, f2)`[2]	`f1.a` × `f2.a` `f1.b` × `f2.b`
`SecOper.compare(f1, f2)`[3]	`f1` ≤ `f2`

[1] `n` represents the public key used to encrypt `f1` and `f2`.
[2] MHE scheme cipher text contains two components denoted here by `a` and `b`.
[3] OPE scheme requires comparisons over integers 128 bits and higher.

Example. Listing 1 shows code snippets used in a STYX vertex class. The code is part of 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: the group name and the 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, signifying 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 `SecureOper.add()` method (Line 24). Note that Line 2 also shows the stream annotations.

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 the Paillier crypto system as in Line 10.

4.2 Processing Secure Streams

We now give details on how we tackle the challenges introduced in Section 1 when processing continuous queries over encrypted streams.

Field masking. Computing on encrypted data introduces the additional challenge of dealing with operands with increased sizes. For instance, an addition of two `long` operands (each 64 bits) in plaintext may transform into an operation over 1024 bit operands in the encrypted data stream. This means a factor of 16 increase in the operand size. Typically, in stream processing systems, the source vertex receives all the fields in the stream, irrespective of a field being used or not. For plaintext program graphs, this is usually not a substantial overhead and the additional computation required for removing unused fields may not always offset the improvement that is observed. When the computation happens over encrypted data, filtering out fields will have a much more significant impact because of the size of the fields. E.g., consider a stream with two fields similar to the stream used for `Group By..Sum` in Listing 1. If there is a continuous query which finds unique groups, the second field will be unused. Simply removing the unused field re-

```
1  public class SlotBasedSum<T>  {
2     BigInteger publicKey = readPubKey();
3     public void updateSum(T group, int slot, BigInteger value) {
4         BigInteger[] sums = objGroupSum.get(group);
5         if (sums == null) {
6             sums = new BigInteger[this.numSlots];
7             init(sums, "AHE");
8             objGroupSum.put(group, sums);
9         }
10        sums[slot] = sums[slot].multiply(value).mod(publicKey.multiply(publicKey));
11    }
12 }
```

Listing 2: Code for finding the sum of each group in a sliding window without STYX abstractions

duces the size of a tuple from 160 bytes to 32 bytes. STYX performs this unused field removal automatically using field masking. Considering the fact that an unused field may be at any index within a tuple, if we simply drop the field, program logic that accesses the field after it has been dropped may fail. To avoid this problem, we mask unused fields by not removing them, but replacing them with `null`s. To identify unused fields, STYX relies on the stream annotations described in Section 4.1. For each vertex, we identify fields for which no operations are specified. Our masking process itself is very lightweight. Since we have information about fields to be masked at compile time, we update the STYX runtime with this information. STYX then suppresses the emission of the masked fields.

Key management. If a device is compromised, it will need to update its private key along with relevant security patches. There also may be a need to periodically update or revoke keys in an IoT device. If a continuous query aggregates data over a sliding window, it is not possible to perform these operations without disrupting the output. This is because if we decide to switch over to a new key at time t, aggregations in the sliding window that span both sides of time t will contain tuples encrypted with both old and new keys and will fail. STYX supports such key changes without disrupting the output. When a key change is initiated, tuples are emitted under the old and new key for the time span of the sliding window in the application graph. When the `StyxVertex` base class detects a key change in the stream, it creates a new instance of the application vertex class to process the stream. The stream encrypted with the new key is channeled into the new instance of the vertex class, while the stream encrypted with the old key gets processed as before. STYX suppresses emissions from the new vertex instance until the new instance contains tuples spanning the full length of the sliding window. At this point, the old instance of the application vertex class is discarded and the stream from the new instance is emitted.

Initialization and constants. Now we address the challenge of variable initializations. Standard operations like `sum = sum + value` usually require `sum` to be initialized to 0 or another constant value (say `k`) before being executed. When the same operation is performed over encrypted data, `sum` must be initialized to the encrypted value of the `k`, and further, encrypted using the same crypto system and key as `value`. STYX handles this by allowing the runtime to request constants from the trusted tier. E.g., STYX code like `SecField sum = SecureOper.getConstant(k, value)` is used to initialize a variable to `k` (used within the `init(sums, value)` method in Listing 1, Line 21). STYX uses its internal meta data to identify that the field `value` is derived from the second field in the input stream and requests the trusted tier to return `k` encrypted using the same crypto system and key as variable `value`. 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 latency.

Identifying encryption schemes. This step identifies the crypto systems 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 different streams and their grouping clauses in the application logic. This can be derived from the graph declaration as explained in Section 4.1. Secondly, we need to identify the operations performed on each stream. STYX derives this from program annotations in each vertex class in the graph. Once STYX derives the distinct streams and operations to be performed on those streams, we can proceed similarly as in our prior work [34] to infer the crypto systems required to execute the graph.

In what follows, we briefly summarize how these techniques work. We start by constructing an expression tree where fields in tuples form the leaf nodes. Operations performed on those fields form the non-leaf nodes. For STYX graphs, we use field annotations (as specified in Section 4.1) to determine the non-leaf, operator nodes. For each operator, a lookup table identifies the crypto system of the operands and result of the operator. Our goal now is to identify the crypto system in which all the leaf nodes (fields) should be encrypted. This can be done by identifying the parent operator node for each leaf node and using the lookup table to identify the type of crypto system required for operands

for that operator node. There can also 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 marked to be executed on the trusted tier so that the scheduler can place the tasks correctly.

Automatic re-encryption. 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 systems in the cloud. To get around this issue, STYX may decide to either perform those operations in the trusted tier or re-encrypt the stream in the trusted tier. For re-encryption, STYX inserts special *re-encryption vertexes* into the graph and marks them so they get scheduled on the trusted tier only.

Encrypting public 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 an application logic. Though these data are public, we may need to encrypt them because if they are to be combined or compared with private data already encrypted for computing in the cloud, the public data should also be encrypted under the same key for such operations to succeed. For easily integrating public data into a program, STYX provides special vertex classes that encrypt plaintext streams.

5. STYX Deployment

In this section we describe how graphs are deployed into the STYX runtime.

5.1 Deployment Profile Generation

As defined in Section 2, the number of runtime tasks assigned for each vertex in the graph is called the deployment profile of the graph. A good deployment profile is required to avoid bottlenecks and ensure good resource utilization.

Utilization. To reason about the effectiveness of deployment profiles, we first define *utilization*. Utilization of a vertex for a time interval is defined as the amount of time the vertex spends processing in that time interval. E.g., if a vertex spends 5 minutes in a 10 minute interval 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 it is starting to become a bottleneck. Good resource utilization is usually achieved by the programmer explicitly specifying the number of tasks for each computation vertex. Programmers are perfectly suited to do this as they understand, via application logic, which vertexes handle more data or computation, and can correspondingly allocate more tasks for those vertexes. In STYX, when the computation graph is transformed and operations are converted to their cryptographic equivalents, the utilization of a vertex changes substantially. This means that programmers need to thoroughly understand overheads of each crypto system, which goes against STYX's design goals.

Heuristic. We propose a linear programming based heuristic which automatically converts the deployment profile for a plaintext graph into an optimized deployment profile for the corresponding STYX graph. Figure 4 shows the formal representation of the heuristics that we use. S represents the slots available for instances to use and V represents the vertexes in the graph that needs to be allocated. A slot is typically a Java virtual machine (JVM) or an executor thread within a JVM. We assume all slots have the same processing capacity. The matrix A represents how much each vertex amplifies its input. A is derived by executing the plaintext graph on sample data. To compute the amount of data arriving at a vertex, we consider all paths of varying lengths that end up at that vertex from the source. The A matrix gives the amount of data at vertexes which are one edge away from the source (for unit input). To find data arriving at the vertex i (represented by d_i) through paths of length 2 and higher, we compute the power matrix of A represented in Figure 4 as A^2, A^3 etc. The vector C represents the load on each vertex relative to one another. C is derived by first inverting the number of instances for each vertex (from the deployment profile) in the plaintext version of the graph and then scaling it with respect to the crypto operations performed by the vertex.

For example, assume that a programmer specifies the number of instances for each vertex as $v1:1, v2:3, v3:2, v4:6$ for the plaintext graph presented in Figure 2. Thus, we start with the vector $\{1, 1/3, 1/2, 1/6\}$ or simply $\{6, 2, 3, 1\}$. The intuition here is that for vertexes that come under heavy load, the programmer will allocate a higher number of instances in the deployment profile to accommodate the load. At the next step, we scale down the value of each element in the above vector based on a reduction factor. This reduction factor is derived empirically based on our observations. We use a reduction factor of 3 for AHE schemes, 2 for DET and 6 for re-encryption nodes. Consequently, and based on Figure 2, if $v3$ receives a stream ($s2$) with a field encrypted under AHE, we scale down the value corresponding to v_3 to 1, after which we arrive at $C \leftarrow \{6, 2, 1, 1\}$. After scaling down each element we get the C vector which can be used in the linear program.

In Figure 4, T represents the deployment profile, and t_i represents the number of slots allocated to execute vertex v_i. Our target now is to derive each t_i. To this end, we define two sets of constraints. The first set of constraints ensures each vertex is allocated to at least one slot. The second set of constraints ensures that for all vertexes, the load (described by c) is less than the capacity of the nodes to process it. Under these constraints, we 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: $\{[a_{11} \ldots a_{1m}], \ldots, [a_{m1} \ldots a_{mm}]\}$ where,

a_{ij}: Output from v_i that goes to v_j for unit input

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

c_i: Relative load on any s_j processing v_i

Unknowns

d_i: Input consumed by single instance of i

D_i: Input consumed by all instances of i

T: $\{t_1, t_2 \ldots t_m\}$ where

t_i: Number of slots for v_i

From definitions

$D_1 \leftarrow d_1 \times t_1$

$\forall 1 < i \leq m,$

$D_i \leftarrow d_1 \times t_1 \times \{A_{1i} + A_{1i}^2 + \ldots A_{1i}^l\}$ where,

l is the length of longest path between v_1 and v_i

Linear program constraints

$\forall 0 < i < m, t_i \geq 1$ All v_is 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

5.2 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 that are required for encryptions or crypto system transformations. 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.

We note that the scheduler service can also run in the untrusted cloud. An attacker can try to manipulate the scheduler in the following two ways: (i) by trying to execute trusted vertexes in the untrusted cloud and (ii) by trying to execute untrusted code in the trusted tier. The former way does not compromise confidentiality since the untrusted cloud does not possess the private keys required to reveal the plaintext data. However, the latter 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 this, 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.

6. Implementation

In this section we lay out some of the implementation details of STYX. STYX's processes in the cloud are implemented by modifying Apache Storm [7]. Storm is an online, distributed computation system. Application logic in Storm 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 streams upon which the bolts operate. Modifications to Storm are limited to implementing a new scheduler (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 the GNU multiple precision arithmetic library GMP [2] to perform fast arbitrary precision arithmetic operations invoked using the Java native interface (JNI). Randomized encryption (RAN) is implemented using AES [20] with CBC mode with a random initialization vector. Deterministic encryption (DET) is implemented using an AES pseudo-random permutation block cipher following the approach used in CryptDB [31] that uses a variant of CMC mode [27] with a zero initialization vector. The Boldyreva et al. [17] crypto system is used as our OPE scheme implementation. The Paillier [30] crypto system is used as our AHE scheme, and finally ElGamal [21] is used as the MHE scheme.

7. 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 5.

7.1 Smart Meter Analytics

In this evaluation we study the *throughput* of STYX by running a set of analytical queries to analyze electricity usage

Table 3: Description of continuous queries used for smart meter analytics

#	Query	Output
Q1	Number of readings	Total number of readings for the given time window
Q2	Consumption	Sum of total resource consumption for the given time window
Q3	Peak consumption	Sorted list of the aggregate consumption per 10 seconds in the given window
Q4	Top consumers	List of the distinct consumers, sorted by their total (monthly) consumption
Q5	Consumption series	Time-series of aggregate consumption per 10 seconds in the given window
Q6	Billing	Monthly bill for each consumer based on the time of usage

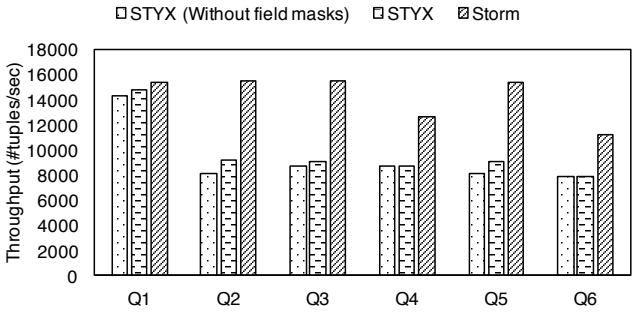

Figure 5: Smart meter query throughput

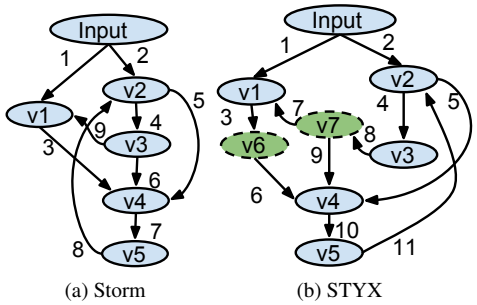

Figure 6: LRB graph

of homes. We use the Smart * dataset [15] available at [5] as our input. This dataset represents electrical meter readings collected over a 24-hour period at the rate of 1 reading per minute from 443 unique homes, totaling 637526 records. Each reading is a tuple of three fields: timestamp, meter id and meter reading. We define throughput as the number of tuples processed by the application graph in unit time. The runtime was configured to not drop tuples using a fixed sized queue. When a queue becomes full, the source vertexes will stop emitting tuples. When slots in the queue free up, the vertex starts emitting tuples again. To measure throughput, we adapted the queries used in IoTBench [12] for streaming systems. Details of queries are given in Table 3. We used a time window of 60 seconds. To run continuous queries for at least 600 seconds, we repeated the same dataset but adjusted the timestamps. We ran these queries on 4 *large* nodes on Amazon EC2. For STYX, one of the 4 nodes was specified as a trusted node. The bandwidth of the trusted node was throttled to 8 Mbit/s to simulate a wide area network link. Results of our evaluation are presented in Figure 5. Q1 simply counts the number of readings and performs at 96% throughput of plaintext stream. Queries Q2 to Q6, all perform Paillier addition and execute between 59% and 70% of plaintext throughput. The results also show the effect of field masking. For Q1, Q2, Q3 and Q5, we are able to mask one field, resulting in an average of 7% increase in throughput. Since queries Q4 and Q6 use all the fields in the stream, no fields could be masked.

7.2 Linear Road Benchmark

We use the Linear Road Benchmark (LRB) [10] 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: (i) calculate the amount of toll to be levied on the vehicle, and (ii) 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 be 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. Figure 6(a) shows the Storm topology which implements the standard linear road and Figure 6(b) shows the transformed STYX topology. Note that Figure 6(b) contains two new vertexes $v6, v7$ which are re-encryption vertexes that are executed within the trusted tier. We ran the experiment for three hours, and the rate at which position reports are emitted for one single expressway is shown in Figure 7. Observe that the rate of input steadily increases up to 1811 tuples per second.

LRB baseline and hypothesis validation. We first ran a baseline deployment of LRB by assigning each vertex a sin-

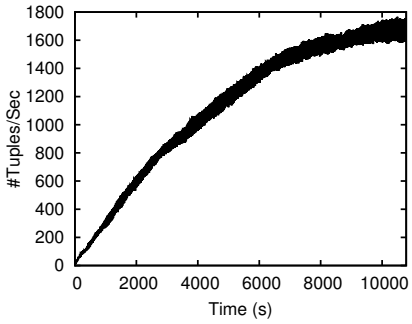

Figure 7: LRB data profile

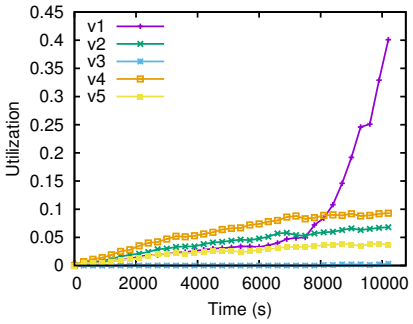

Figure 8: Storm LRB baseline

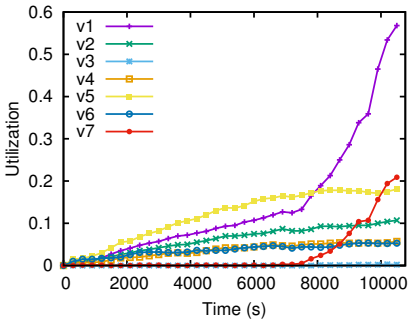

Figure 9: STYX LRB baseline

gle task. This allows us to observe each individual vertex to see how they consume resources and verify the hypothesis made in Section 5.1 that *bottlenecks change when running on encrypted data streams*. We plot utilization (cf. Section 5) against time for the duration of a LRB run (10784 seconds) on Storm with plaintext data (see Figure 8) and on STYX with encrypted data (Figure 9). We can observe that in Figure 8 vertexes $v4$ and $v2$ have the highest utilization values until around the 8000s mark, and after that vertex $v1$ becomes the node with the highest load. This increase is because the number of tuples that requires a toll notification increases substantially after 8000s. In the transformed STYX graph running on encrypted 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.

Table 4: LRB comparison

System	L	Time (ms)[1]	Profile[2]
Storm	20	2694.44	5, 4, 1, 3, 2
STYX	15	2672.97	5, 2, 1, 2, 3, 1, 1

[1] Average response time.
[2] Deployment profile for vertexes in order $v1$, ..., $v5$ for Storm and $v1$, ..., $v7$ for STYX.

Performance of STYX deployment profile. Now we benchmark both the Storm topology graph and the transformed STYX graph using LRB. For this, we deployed both graphs on 15 large nodes in Amazon EC2 in the best possible configuration so that the maximum number of highways supported can be identified. Table 4 show the results. For plaintext streams Storm supports 20 expressways, while using STYX with encrypted streams we can support 15 expressways. We also plot response times for all notification triggering tuples – times taken for notifications to be issued from the time respective tuples enter the system. The response times for STYX are shown in Figure 11 and the response times for Storm are shown in Figure 10. Response times for STYX peak 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 8 and 9, these graphs can also be used as a baseline to understand our model from Section 5.1. Vertexes with higher utilization value should get more instances to execute them. Table 5 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 the lowest response time. This profile is also in accordance with Figure 9 which shows vertexes $v1$ and $v4$ should get the highest numbers of instances.

Table 5: LRB deployment profile response time

STYX deployment profile	Response time (ms)
5, 2, 1, 2, 3, 1, 1[1]	2672.97
4, 2, 1, 4, 2, 1, 1	2714.30
5, 4, 1, 3, 2, 1, 1	2781.40

[1] Deployment profile generated by STYX.

Encryption overhead of IoT device We chose Raspberry Pi [9] to evaluate the overhead of encryption on IoT devices. Raspberry Pi is a widely popular IoT device and has the capabilities similar to devices found in highway cameras, vehicle tracking, and telemetry devices. We implemented AHE, MHE, and DET for Raspberry Pi version B. We measured the latency to encrypt 16 bytes, a typical message size. Our results are shown in Figure 12. The encryption latency is acceptable, as it is less than 150ms even for complex schemes like AHE and MHE.

7.3 Application Case Studies

New York taxi statistics. This application finds the top 10 most frequent routes during the last 30 minutes of taxi ser-

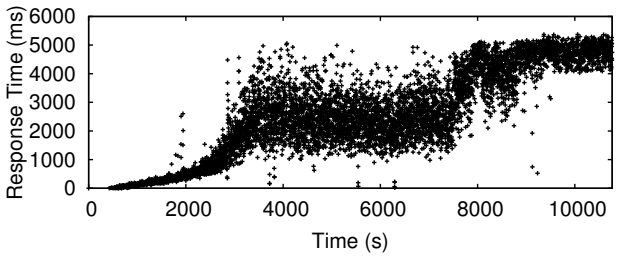

Figure 10: Response time for LRB on Storm

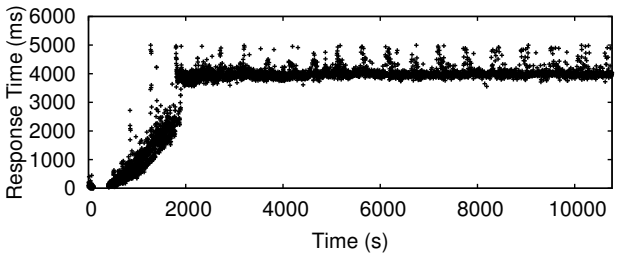

Figure 11: Response time for LRB on STYX

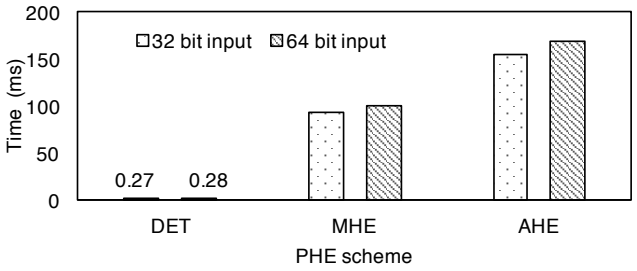

Figure 12: Encryption overhead for an IoT device

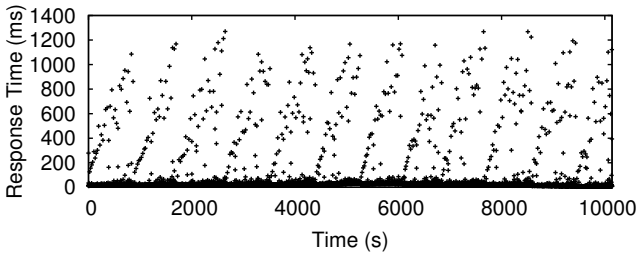

Figure 13: Response time of top-10 taxi route query with key change at the start of every month

vicing. 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 publicly available [8]. 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. In order to evaluate the effect of key changes on response time, we simulate a key change at the beginning of each month. This means all data that are emitted with a time stamp within the first 30 minutes of every month will be encrypted under the old and new key. 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 output. We deployed this application on 10 large nodes on Amazon EC2. Table 6 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 (about 1%). Figure 13 shows the response time for the full run with key changes every month. In this plot, 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 tuples (90th percentile within 31ms and 99th percentile within

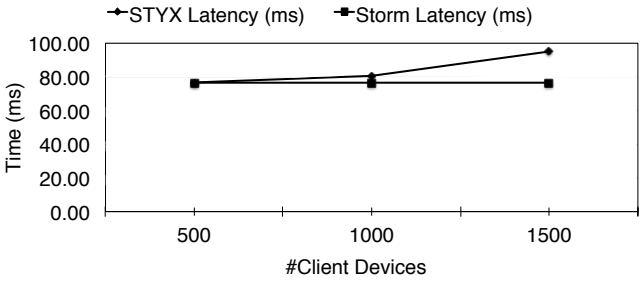

Figure 14: Heartbeat analysis response time

818ms) respond with the same response time as when no change was in effect.

Table 6: Top-10 taxi routes

System	Completion time (s)	Average response time (ms)
Storm	8106	36.05
STYX[1]	10039	45.10
STYX[2]	10140	46.61

[1] Entire stream emitted under same key.
[2] Stream emitted with a new key every month.

Heartbeat analysis. Finally, we study how STYX can be used for a realistic online application like a heart beat 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, weekly, 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 14. 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.

8. 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 by successfully putting PHE to work. In what follows, we discuss prior work in the areas of stream processing and cloud security.

A large body of work 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 [14, 19, 28], and Telegraph project [18, 33]). 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 [7], Heron [29], Spark streaming [37], and Samza [3]. STYXs design is based on the stream processing design of Storm; our concepts can also be implemented on top of other stream processing systems.

Gentry introduced an implementable FHE scheme [22] which has been becoming more practical ever since [24], but is still not suited for encrypted query processing due to its prohibitive cost. Instead, a lot of research work is focused on using PHE schemes to perform computations over encrypted data. Most prominently, CryptDB [31] 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 of 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.

Another way to enforce data confidentiality is through specialized hardware that provides a trusted computing base [11, 13, 16]. An approach that is gaining popularity now uses the Intel SGX [4] processor as a trusted computing base. SGX offers hardware encrypted and integrity protected physical memory, which allows data and code to reside in the untrusted cloud. Though promising, practical performance of the processor for big data workloads is yet to be ascertained and widespread adoption by cloud service providers seems unlikely in the immediate present.

These approaches could also be used to extend the design of STYX by allowing secure computations to be performed on trusted hardware in an untrusted cloud.

9. Conclusion

We presented STYX, a practical distributed system for evaluating continuous queries over encrypted data streams in public clouds. STYX makes PHE practical for stream processing by using a novel API, encryption inference, automatic re-encryption, and a set of other original techniques. The API allow programmers to develop secure applications with little or no knowledge of the underlying crypto systems. We evaluated our approach using standard benchmarks and applications, demonstrating its applicability and performance. Our evaluations show that we can meet latency requirements even with high volumes of encrypted traffic.

References

[1] Nest. `https://nest.com/`.

[2] The GNU Multiple Precision Arithmetic Library. `https://gmplib.org/`.

[3] Samza. `http://samza.apache.org/`.

[4] Intel SGX. `https://software.intel.com/en-us/isa-extensions/intel-sgx#`.

[5] Smart * Data Set for Sustainability, . `http://traces.cs.umass.edu/index.php/Smart/Smart`.

[6] SmartThings, . `http://www.smartthings.com/`.

[7] Apache Storm. `https://storm.apache.org/`.

[8] NYCs Taxi Trip Data. `http://chriswhong.com/open-data/foil_nyc_taxi/`.

[9] Raspberry Pi, 2012. `https://www.raspberrypi.org`.

[10] A. Arasu, M. Cherniack, E. F. Galvez, D. Maier, A. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts. Linear road: A stream data management benchmark. In *Int. Conf. on Very Large Data Bases (VLDB)*, pages 480–491, 2004.

[11] A. Arasu, S. Blanas, K. Eguro, R. Kaushik, D. Kossmann, R. Ramamurthy, and R. Venkatesan. Orthogonal security with

Cipherbase. In *Biennial Conf. on Innovative DataSystems Research (CIDR)*, 2013.

[12] M. F. Arlitt, M. Marwah, G. Bellala, A. Shah, J. Healey, and B. Vandiver. Iotabench: An internet of things analytics benchmark. In *Int. Conf. on Performance Engineering*, pages 133–144, 2015. .

[13] S. Bajaj and R. Sion. TrustedDB: A trusted hardware based database with privacy and data confidentiality. In *Int. Conf. on the Mgt. of Data (SIGMOD)*, pages 205–216, 2011. .

[14] M. Balazinska, H. Balakrishnan, S. R. Madden, and M. Stonebraker. Fault-tolerance in the Borealis distributed stream processing system. In *Int. Conf. on the Mgt. of Data (SIGMOD)*, pages 13–24, 2008.

[15] S. Barker, A. Mishra, D. Irwin, E. Cecchet, P. Shenoy, and J. Albrecht. Smart*: An open dataset and tools for enabling research in sustainable homes. In *Workshop on Data Mining Applications in Sustainability (SustKDD)*, 2012.

[16] A. Baumann, M. Peinado, and G. C. Hunt. Shielding applications from an untrusted cloud with Haven. In *Symp. on Op. Sys. Design and Implementation (OSDI)*, pages 267–283, 2014.

[17] A. Boldyreva, N. Chenette, Y. Lee, and A. O'Neill. Order-preserving symmetric encryption. In *Int. Conf. on The Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, pages 224–241, 2009.

[18] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Biennial Conf. on Innovative DataSystems Research (CIDR)*, volume 20, page 668, 2003.

[19] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. Zdonik. Scalable distributed stream processing. In *Biennial Conf. on Innovative DataSystems Research (CIDR)*, volume 3, pages 257–268, 2003.

[20] J. Daemen and V. Rijmen. *The design of Rijndael: AES - the advanced encryption standard*. Springer Verlag, Berlin, Heidelberg, New York, 2002.

[21] T. ElGamal. A public-key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, 1985.

[22] C. Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford, CA, USA, 2009. AAI3382729.

[23] C. Gentry, S. Halevi, and N. P. Smart. Homomorphic evaluation of the AES circuit. In *Annual Int. Cryptology Conf. (CRYPTO)*, pages 850–867, 2012.

[24] C. Gentry, S. Halevi, and N. P. Smart. Homomorphic evaluation of the AES circuit. *IACR Cryptology ePrint Archive*, 2012. Informal publication.

[25] J. Groopman and S. Etlinger. Consumer perceptions of privacy in the Internet of things. Technical report, 2015.

[26] T. Gupta, R. P. Singh, A. Phanishayee, J. Jung, and R. Mahajan. Bolt: Data management for connected homes. In *Networked Sys. Design and Implem. (NSDI)*, pages 1–14, 2014.

[27] S. Halevi and P. Rogaway. A tweakable enciphering mode. In *Annual Int. Cryptology Conf. (CRYPTO)*, pages 482–499, 2003.

[28] J. H. Hwang, M. Balazinska, A. Rasin, U. Çetintemel, M. Stonebraker, and S. Zdonik. High-availability algorithms for distributed stream processing. In *Int. Conf. on Data Engineering (ICDE)*, pages 779–790, 2005.

[29] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja. Twitter Heron: Stream processing at scale. In *Int. Conf. on the Mgt. of Data (SIGMOD)*, pages 239–250, 2015.

[30] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Int. Conf. on Theory and Application of Cryptographic Techniques (EUROCRYPT)*, pages 223–238, 1999. ISBN 3-540-65889-0.

[31] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: Protecting confidentiality with encrypted query processing. In *Symp. on Op. Sys. Principles (SOSP)*, pages 85–100, 2011.

[32] H. Shafagh, A. Hithnawi, A. Droescher, S. Duquennoy, and W. Hu. Talos: Encrypted query processing for the internet of things. In *Conf. on Embedded Networked Sensor Sys. (SenSys)*, pages 197–210, 2015.

[33] M. A. Shah, J. M. Hellerstein, and E. Brewer. Highly available, fault-tolerant, parallel dataflows. In *Int. Conf. on the Mgt. of Data (SIGMOD)*, pages 827–838, 2004.

[34] J. J. Stephen, S. Savvides, R. Seidel, and P. T. Eugster. Program analysis for secure big data processing. In *Int. Conf. on Automated Software Engineering (ASE)*, pages 277–288, 2014.

[35] S. D. Tetali, M. Lesani, R. Majumdar, and T. D. Millstein. MrCrypt: Static analysis for secure cloud computations. In *Conf. on Object-Oriented Prog. Sys., Lang. and Applications (OOPSLA)*, pages 271–286, 2013.

[36] S. Tu, M. F. Kaashoek, S. Madden, and N. Zeldovich. Processing analytical queries over encrypted data. *Proc. VLDB Endow.*, 6(5):289–300, 2013.

[37] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters. In *W. on Hot Topics in Cloud Computing (HotCloud)*, pages 10–10, 2012.