# Mitigating Network Side Channel Leakage for Stream Processing Systems in Trusted Execution Environments

Muhammad Bilal*
Université catholique de Louvain
muhammad.bilal@uclouvain.be

Hassan Alsibyani
KAUST
hassan.alsibyani@kaust.edu.sa

Marco Canini
KAUST
marco@kaust.edu.sa

## ABSTRACT

A crucial concern regarding cloud computing is the confidentiality of sensitive data being processed in the cloud. Trusted Execution Environments (TEEs), such as Intel Software Guard eXtensions (SGX), allow applications to run securely on an untrusted platform. However, using TEEs alone for stream processing is not enough to ensure privacy as network communication patterns may leak information about the data.

This paper introduces two techniques – anycast and multicast – for mitigating leakage at inter-stage communications in streaming applications according to a user-selected mitigation level. These techniques aim to achieve network data obliviousness, i.e., communication patterns should not depend on the data. We implement these techniques in an SGX-based stream processing system. We evaluate the latency and throughput overheads, and the data obliviousness using three benchmark applications. The results show that anycast scales better with input load and mitigation level, and provides better data obliviousness than multicast.

## CCS CONCEPTS

• **Security and privacy → Distributed systems security**;

## KEYWORDS

Stream processing, Intel SGX, network data obliviousness

**ACM Reference Format:**
Muhammad Bilal, Hassan Alsibyani, and Marco Canini. 2018. Mitigating Network Side Channel Leakage for Stream Processing Systems in Trusted Execution Environments. In *DEBS '18: The 12th ACM International Conference on Distributed and Event-based Systems, June 25–29, 2018, Hamilton, New Zealand.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3210284.3210286

## 1 INTRODUCTION

Cloud computing provides a cost-effective and flexible means of executing large-scale computations to extract timely insights from streaming data. However, running stream computations in the cloud requires that the cloud tenants trust the cloud provider with both their code and their data, which exposes them to a range of privacy concerns. These concerns include the possibility that a malicious administrator or an intruder may gain privileged access to system software (such as the OS or the hypervisor) in order to leak or manipulate confidential data or to tamper with the tenants' computations to produce incorrect results. Therefore, cloud tenants want to protect the confidentiality and integrity of their data by preventing it from being accessed by unauthorized parties.

Fortunately, Trusted Execution Environments (TEEs) allow to harden the security of cloud applications. Moreover, TEEs are poised to become widely available in cloud environments. For example, Intel Software Guard eXtensions (SGX) is a TEE that was released by Intel with the Skylake CPU series and is now being adopted in subsequent generations of Intel processors. SGX introduces the notion of a secure enclave, defined as an execution environment with a processor-protected memory region that shields both the application code and the data from access by other software, including higher-privileged software. Although SGX provides additional security, side-channel attacks can still be used to extract information about the data being processed by the enclaves [12, 13, 16, 18]. Therefore, simply "enclaving" applications is not sufficient to guarantee data security and privacy.

Prior studies have attempted to protect SGX-based big data analysis frameworks from side-channel attacks [12, 18]. While the proposed techniques for protecting against memory side channels are also applicable to stream processing systems, the techniques for protecting against network side channels are not. For example, Ohrimenko et al. [12] suggested avoiding network side-channel leakage by shuffling the batched data offline and using sampling to extract the dataset characteristics. However, neither of these techniques are feasible when processing streaming data in or near real-time, which is the focus of our work.

In distributed stream processing systems, network communication patterns directly reflect the structure of the streaming applications. These applications typically consist of multiple processing stages organized into a Directed Acyclic Graph (DAG) that runs on a collection of networked machines. In general, each stage is partitioned into multiple nodes that are executed in parallel. Each node performs local computations on the input streams from its inedges, and produces output streams to its outedges. By observing network-level communication between the different stages of the DAG, an adversary may be able to extract information about the data being processed by the application. To understand why this happens, we consider how messages are routed among the nodes of a distributed streaming application. Several stateful computations (e.g., aggregation or join operations) require that all the messages that are logically related must be processed at the same node. The common scenario that we consider is when messages are key-value

*Work done while visiting at KAUST.

*DEBS '18, June 25–29, 2018, Hamilton, New Zealand*

ACM ISBN 978-1-4503-5782-1/18/06…$15.00
https://doi.org/10.1145/3210284.3210286

tuples. Then, *key-based grouping* is used to consistently route all tuples with the same key to the same node (typically based on a hash of the key). Thus, key-based grouping implies that the network traffic among nodes (i.e., the routed tuples) is data dependent. And so, in the presence of data skew, attacks based on frequency analysis can reveal information about the data even when those data are encrypted.

A straw man approach to avoiding data-dependent network communication patterns is broadcast, i.e., each node sends every tuple it produces to *all* nodes of the next stage. Broadcast leads to a completely uniform traffic pattern across nodes of the same processing stage. However, broadcast increases the computational and network resource requirements superlinearly with the number of nodes. While computational resources can be easily (albeit expensively) scaled in the cloud, network resources are not provisioned on-demand in typical cloud environments. Additionally, even when sufficient computational and network resources are available, broadcast may be unnecessary when the workload is not highly skewed.

Here, we investigate techniques for mitigating information leakage via network side channels in stream processing systems. Based on the above observations, we aim to provide data obliviousness for network communication patterns without incurring the high overheads of broadcast. We propose two simple yet effective techniques: multicast and anycast. Compared to broadcast, multicast reduces the set of next-stage nodes that are candidates for receiving duplicate tuples. Anycast, on the other hand, outputs just a single tuple to one node chosen from a set of candidate next-stage nodes. This efficiency gain comes at the expense of increased latency, since we must introduce a subsequent aggregation stage to ensure application correctness in stateful computations.

We implement these techniques in a home-built stream processing framework. This framework allows the user to select what mitigation technique to use and the amount of tolerable leakage, as evaluated according to a user-chosen security property (e.g., an entropy metric should be below a threshold). The framework observes the application at runtime and reports whether the security property is satisfied at the current workload. Lastly, our framework adapts dynamically by changing the mitigation level when the skew of the input workload changes, ensuring that the user-defined security property is satisfied.

Our evaluation shows that anycast achieves higher throughput than multicast and broadcast with the same number of resources, and scales well to higher mitigation levels without adding significant overhead. Multicast and broadcast require more resources to provide the same latency and throughput as key-based grouping. The results also show that it is both feasible and effective to adapt the mitigation level for anycast at runtime to ensure that the security property is satisfied despite changes in workload skew.

Our work contributes (*i*) mitigation techniques that can be used to avoid network side-channel leakage, (*ii*) the implementation of a stream processing framework using SGX with integrated side-channel mitigation techniques, (*iii*) an evaluation of each technique in both real and simulated scenarios for different applications, multiple metrics, and workloads, and (*iv*) mechanisms for performing adaptive mitigation-level selection.

We use SGX as the example TEE in our system prototype, but our techniques are equally applicable to frameworks built with other TEEs. The mitigation techniques can also be applied to any non-TEE setting where the user does not trust the network. TEE simply provides additional protection for data against direct attacks on the compute nodes.

## 2 PRELIMINARIES

### 2.1 Background on Stream Computations

Stream processing systems are distributed systems designed to perform computations on a potentially infinite stream of data, in real-time or near real-time. Each data element is called a tuple. Generally, a streaming computation is a pipeline of stages that operates on streaming data arranged into a DAG. We refer to an application DAG as a topology. The nodes represent continuous and often stateful computations in a stage, while the edges represent the downstream flow of tuples from one node to another. Each stage is comprised of multiple nodes that operate in parallel. Nodes execute as threads on a collection of server machines. There are different types of stages in the DAG. Spouts are source stages that ingest data, generate tuples, and introduce the tuples into the streaming application. Bolts are intermediate processing stages that have one or more input streams and one or more output streams. Lastly, Sinks, as their name suggests, are processing stages that serve as exits from the streaming application. Two of the most common communication patterns among stages are shuffle/randomized and key-based grouping. In the shuffle pattern, tuples are routed to a random node of the next stage. In the key-based grouping pattern, tuples are routed based on keys, such that tuples with the same key are always routed to the same node.

### 2.2 Background on Intel SGX

SGX allows systems to execute code and process data in enclaves. Since enclaves are hardware-protected regions, they remain protected even when higher-privileged components, such as the BIOS, hypervisor, OS, or drivers, are compromised. Thus, an adversary can gain control over the machine, but the enclaves will remain protected.

To securely deploy applications, developers can build an enclave and take a measurement, i.e., a secure hash of the initialized enclave. This measurement is used to confirm the integrity of the enclave during a local or remote attestation, which is a procedure that proves to users that they are in communication with the expected enclave.

SGX applications are typically comprised of both trusted and untrusted code. The trusted code includes everything that is executed within the SGX enclaves; the rest of the application constitutes the untrusted code. The untrusted code is unable to access data within the enclaves, and the enclaves cannot issue system calls; they delegate this task to the untrusted code. Enclaves communicate with the untrusted code through function calls named *OCalls*, while the reverse communication uses *ECalls*.

### 2.3 Assumptions and Threat Model

We expect users of a secure stream processing system to trust their own code. This generally assumes that the code is bug-free and does not expose private data or secrets. We trust SGX and its associated components, such as remote attestation. We assume that network communication is encrypted.

We assume that the adversary is not necessarily just curious, but also malicious. The adversary may have control of the software and hardware stack, including the OS, hypervisor, BIOS, device drivers, and the network. If the adversary has full control over the software and hardware stack, then side-channel attacks exploiting network communication patterns are possible. A powerful adversary may have control over many machines in the data center and would be able to launch coordinated attacks against the system. Such an adversary could collect and analyze network traffic.

In addition to the technical ability of the adversary, the adversary might have previous knowledge about the application, the type of workload expected to run, and other public information. Even when the actual workload is not available to the adversary, they may combine general statistics with any other information that can be obtained, particularly network patterns, to get a better idea of the actual workload.

We focus on information leakage due to network communication patterns. Other side-channel, denial of service, and physical attacks on the cloud environment are out of scope.

## 2.4 Stream Processing and Data Obliviousness

When a stream processing system routes tuples based on the shuffle pattern, it is random and data independent. However, traffic patterns between stages that use key-based grouping, which is a data-dependent routing scheme, can leak information. Previous research has described several attacks resulting from this leakage. For instance, Ohrimenko et al. [12] described attacks on MapReduce jobs, and how communication patterns were exploited to infer information about the workload. Unfortunately, the risk-mitigation techniques proposed for MapReduce do not apply to stream processing systems. This is because in MapReduce, data is processed offline, which allows shuffling or randomly sampling of the entire dataset to observe its characteristics. In stream processing, we deal with online data streams.

As we already discussed, the straw man approach is to broadcast tuples to all nodes of a downstream stage. This method is inefficient because of its resource requirements. We illustrate the inefficiency of broadcast with an example. We simulate the number of tuples that broadcast would generate in comparison to key-based grouping. The details of the applications under consideration are given in Section 3.3. Figure 1 shows that broadcast can increase the number of tuples up to a factor of $n$, where $n$ is the number of parallel nodes (threads) of the downstream stage that receive tuples routed as per key-based grouping. In the case of the tumbling count topology, if we change the communication pattern between splitter and count stages from key-based grouping to broadcast for a parallelism level of 64, we experience about 43 times more tuples. This means that for large deployments, broadcast can drastically increase traffic and, thus, overload the network. Additionally, any downstream processing node must process duplicate tuples so that an adversary cannot distinguish between the original and duplicate tuples generated by broadcast just by looking at the traffic going to downstream stages. Generating and processing duplicate tuples also consumes additional computational resources. Scaling the computational resources, in fact, leads to even more duplicate tuples.

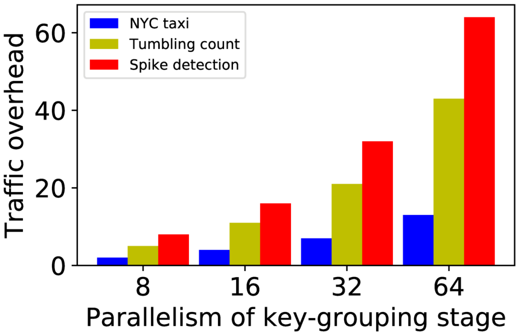


**Figure 1: Increased traffic from broadcast.**

Broadcast is especially inefficient in situations that have low workload skew; in those situations, a milder mitigation technique could provide the same benefits as broadcast with fewer resources. Therefore, there is a need for techniques other than broadcast that can be used in place of key-based grouping to mitigate network side-channel leakage. We now turn our attention to techniques that are more efficient in terms of resources and performance.

# 3 NETWORK DATA OBLIVIOUSNESS

We aim to provide data obliviousness for network communication patterns. Data obliviousness, in this context, means that the network communication patterns are not dependent on the data being processed. Since key-based grouping routes tuples based on their keys, this leads to data-dependent network communication, which may leak information. We introduce mitigation techniques that obfuscate the communication patterns and, thus, decrease the amount of information that an adversary can extract by performing traffic analysis on a distributed streaming application. Based on the idea behind broadcast, we present two techniques that are more resource efficient: multicast and anycast.

While broadcast produces a fully uniform communication pattern (i.e., is data oblivious by definition), other techniques only approximate network data obliviousness from a communication pattern perspective. On the other hand, we posit that different users have different requirements for network data obliviousness. In particular, full uniformity may not be necessary to avoid information leakage. Therefore, we introduce several metrics to quantify network data obliviousness according to information-theory principles. These metrics are by no means exhaustive and should be seen as proof-of-concept. We envision that the user is able to specify, based on domain knowledge, certain security properties about the network traffic that must hold at runtime. Automatically determining these security properties based on attack scenarios is left as future work. Further, the user is free to customize the obliviousness metrics, as they can easily be integrated into our framework.

In the following, we introduce the multicast and anycast mitigation techniques, and present our proof-of-concept metrics. Then, we describe the applications and workloads used throughout the paper. Finally, we evaluate the effectiveness of multicast and anycast according to our metrics.

## 3.1 Mitigation Techniques

We first introduce some notation. Since mitigation techniques apply to inter-stage communication, w.l.o.g., we consider a pair of two consecutive stages: current stage and next stage. We let $C$ and $N$ be the set of nodes of the current and next stages, respectively. The number of nodes in each stage is called the parallelism level. Tuples flow from the nodes in $C$ to the nodes in $N$ according to key-based grouping. In other words, the topology contains an edge from every node in $C$ to every node in $N$. We let $x_k \in N$ denote the node receiving tuples with key $k$, and $\text{succ}(x, i)$ denote the $i^{th}$ successor of node $x$ in a circular space of a stage's nodes.[1] For a given key $k$, we define the candidate set $X_k$ of size $n$ as $X_k = [x_k, \text{succ}(x_k, 1), \ldots, \text{succ}(x_k, n-1)]$. Key-based grouping is also denoted by *kbg*.

We are now ready to introduce the techniques.

**Multicast:** In contrast to broadcast, which sends an output tuple to every node of the next stage, multicast only sends one output tuple for each node contained in the candidate set $X_k \subset N$ of the downstream nodes. Thus, multicast generates $n = |X_k| < |N|$ tuples. The tuples received by all nodes in $X_k \setminus x_k$ are redundant. Only node $x_k$ accounts for the tuple's contribution toward the computation. However, as we will discuss, the nodes that receive extra tuples must also process them to avoid side-channel leakage via timing. Extra tuples are tracked as they propagate throughout the DAG, but their effects toward the computation (e.g., state changes for stateful computations) are ignored. Multicast might expose some data-dependent information through communication patterns, but it consumes fewer resources than broadcast. Intuitively, as $n$ increases, the level of network data obliviousness and the required resources also increase. Thus, the user can select a suitable trade-off by changing the value of $n$.

**Anycast:** Similarly to multicast, anycast uses the candidate set $X_k \subset N$ of the downstream nodes. However, anycast sends a single output tuple to just one of the $n$ nodes of $X_k$.[2] Therefore, unlike broadcast and multicast, no extra tuples are generated; thus, anycast generally requires fewer resources. However, tuples with the same key are partitioned across different nodes. This implies that, in general, any stateful computation will be incorrect because the next-stage nodes only receive a subset of the same-key tuples.

To ensure correctness, anycast requires an *additional stage,* which is placed after the next stage. The role of the additional stage is to "merge" the tuples arriving from nodes of the candidate set. The tuples are routed between the next stage and the additional stage according to key-based grouping. With the additional stage, the computation of the original next stage is effectively split into two stages. In our framework, splitting the computation is responsibility of the user; doing the split automatically is an interesting avenue for future work.

One may wonder how anycast improves data obliviousness, since it sends a single tuple (as key-based grouping) and requires an additional stage, which further increases latency. The answer is two-fold. First, the next stage performs partial aggregation before sending tuples onward to the additional stage. This reduces the load on the additional stage. Second, given that the load is smaller, the additional stage can use a lower parallelism level, which increases obfuscation for the communication pattern of the key-based grouping between the next and additional stages. Anycast provides an obfuscation level between that of broadcast and key-based grouping, while using fewer resources than broadcast and multicast.

Throughout this paper, we will use the mitigation level as a way to indicate the value of $n$ for multicast and anycast. Increasing the mitigation level corresponds to an increased value of $n$ and vice versa.

## 3.2 Obliviousness Metrics

To achieve network data obliviousness, we could require that an adversary observing network traffic would be unable to distinguish between any two different data streams being processed by an application. However, we think that this definition of security is too strict for the many cases where user requirements are more specific and may not need "complete obliviousness." Therefore, we define several metrics for quantifying obliviousness that the user can use according to their requirements. Later, we use all of the proposed metrics to evaluate the feasibility of each mitigation technique under different obliviousness requirements.

It is important to note that here we consider the difficult scenario where we assume that the $n$ most popular keys are mapped to $n$ distinct nodes of a stage. Under this assumption, a frequency analysis attack would yield the most accurate results.

We now review the proposed metrics and then formally define them. The metrics are ($i$) Entropy, ($ii$) Adversarial disadvantage, and ($iii$) n-Hamming distance. Entropy measures the overall uniformity of the distribution of tuples across different nodes of a stage. Adversarial disadvantage provides the accuracy of the attacker's guesses under certain assumptions. The n-Hamming distance is an intuitive metric that reflects the level of obfuscation for the top $n$ keys.

Each metric concentrates on different aspects of obliviousness. We defer to the user of our framework to use domain knowledge when defining the security properties. It is possible to combine different metrics to address different leakage concerns. For example, different mitigation techniques could have the same n-Hamming distance, but different entropies. In that case, the user may elect to combine the entropy metric with the n-Hamming distance by using entropy to break ties or vice versa.

**Entropy:** We let $t_x$ be the number of tuples received by node $x \in N$. Then, we calculate the entropy $S$ as:

$$S = -\sum_x \frac{t_x}{\sum_y t_y} \log\left(\frac{t_x}{\sum_y t_y}\right) \forall x, y \in N$$

If the distribution of tuples across the nodes in $N$ is uniform, then the entropy is maximized and given as:

$$S = -\log\left(\frac{1}{N}\right)$$

[1] A node's successor may be selected using the node with the next index (circularly) or using a different hash function of the tuple's key, without replacement.

[2] In our implementation, the sending node selects the next-stage node that has received the least tuples from that sending node, but other strategies are possible.

Entropy values lower than this maximum would mean less uniformity and data obliviousness.

**Adversarial disadvantage:** We consider an adversary who is trying to guess what tuples are associated with the top $n$ keys. Adversarial disadvantage $AD$ is defined as the likelihood that the adversary's guess will be incorrect under a given mitigation technique relative to the probability of a correct guess when observing key-based grouping communication patterns. We calculate it as a ratio of $G$, the number of correct guesses when mitigation is applied, over the key-based grouping baseline, $G_{kbg}$:

$$AD = 1 - G/G_{kbg}$$

We assume that the adversary performs frequency analysis by monitoring the tuple distribution across nodes, and then guessing which node handles which key(s). For example, we consider three keys, $k_1$, $k_2$, and $k_3$, and three nodes, each dedicated to a key. The adversary might already know that the frequency of these keys is $k_1 > k_2 > k_3$. In this case, the attacker would rank each node $x$ by the number of received tuple $t_x$, and guess that the node with most tuples processed $k_1$, and so on.

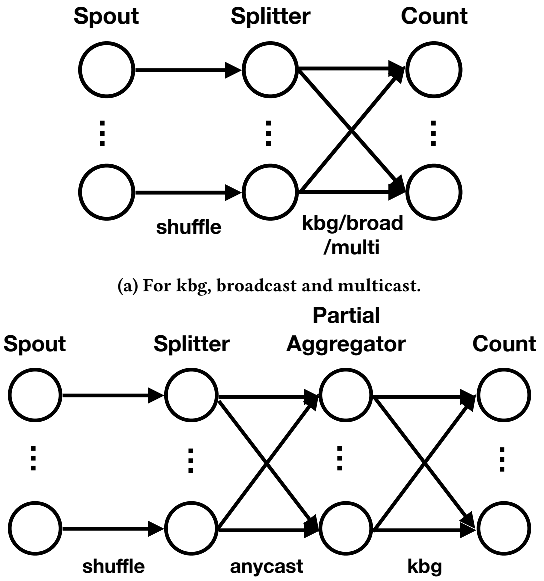
**n-Hamming Distance:** Here, we consider an adversary who is trying to guess which nodes are associated with the top $n$ keys. With a skewed key distribution and key-based grouping, it is relatively easy to detect the most popular keys. The n-Hamming distance $nHD$ reflects "how far off" the tuple distribution is across nodes when mitigation is applied compared to the tuple distribution of key-based grouping. We let $\overline{r}$ be a vector of size $|N|$ consisting of all nodes $x \in N$, which are sorted in decreasing order according to the received tuple count $t_x$; that is, $t_{\overline{r}^1} \geq t_{\overline{r}^2} \geq \ldots \geq t_{\overline{r}^{|N|}}$, where $\overline{r}^i$ is the $i^{th}$ element of $\overline{r}$. We let $\overline{r}_{kbg}$ be a similarly derived vector for $t_x$ that is observed over the key-based grouping baseline. Then, the n-Hamming distance metric is calculated as the number of different elements of the vectors $[\overline{r}^1, \ldots, \overline{r}^n]$ and $[\overline{r}^1_{kbg}, \ldots, \overline{r}^n_{kbg}]$ compared element-wise:

$$nHD = \sum_{i=1}^{n} d_i; \qquad d_i = \begin{cases} 1 & \text{if } \overline{r}^i \neq \overline{r}^i_{kbg} \\ 0 & otherwise \end{cases}$$

We also use a special case of the n-Hamming distance, the 1-Hamming distance, which reflects whether the attacker can correctly guess the node that is handling the most popular key.

## 3.3 Applications

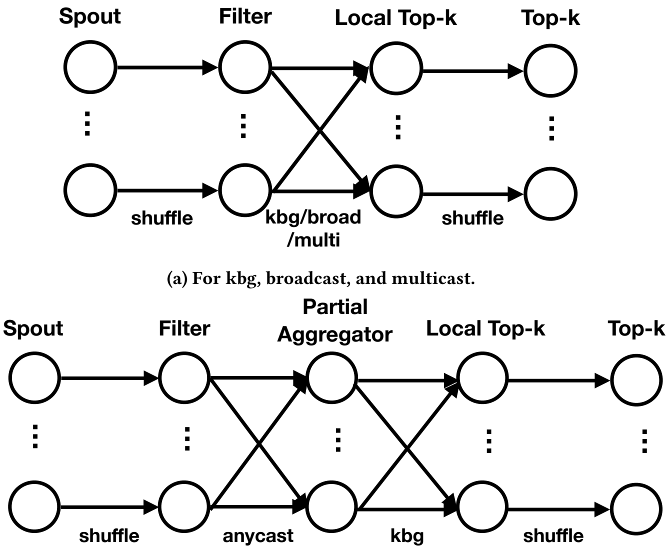
We present three applications for our evaluation, and we emphasize where data obliviousness might be desirable for these applications. The application topologies are shown in Figures 2, 3, and 4. The edges show the type of tuple routing in use. Anycast routing requires an additional stage in the topology to perform computation correctly. However, this is not the case for broadcast or multicast because the extra tuples generated by these techniques are tackled within the same DAG; therefore, the correctness of the computation is not impacted.



(a) For kbg, broadcast and multicast.

(b) For anycast.

**Figure 2: DAG of Tumbling count topology.**



(a) For kbg, broadcast, and multicast.

(b) For anycast.

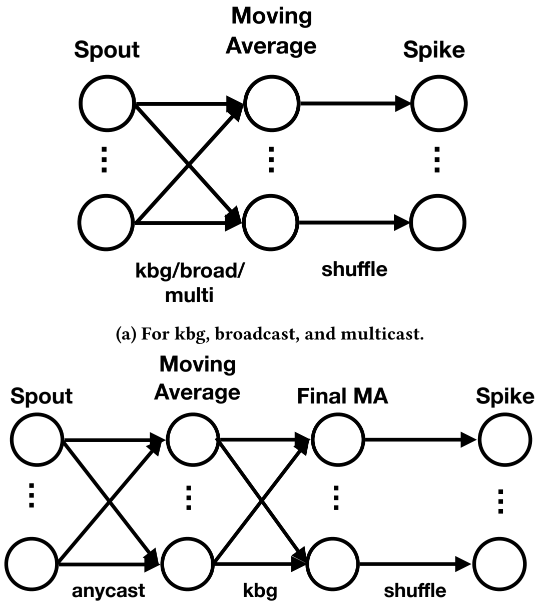
**Figure 3: DAG of NYC taxi topology.**

### 3.3.1 *Tumbling count.*

This application receives entire sentences as input and counts the occurrence of each word of the sentence in a tumbling window with a specific duration. The topology is shown in Figure 2. An adversary can gain information by observing traffic between the splitter stage and the count stage, since the tuple distribution across the different nodes of the count stage can be skewed. A mitigation technique can be applied between the splitter and count stages. In the case of anycast, a partial aggregator stage is introduced, which uses a window to accumulate partial aggregates before sending them to the count stage. In this case, the mitigation is applied between the splitter and the partial aggregator stage.

### 3.3.2 *NYC taxi.*

This application is inspired by taxi data provided by the City of New York and was introduced in [17]. The



(a) For kbg, broadcast, and multicast.

(b) For anycast.

**Figure 4: DAG of Spike detection topology.**

topology is shown in Figure 3. The task of this application is to calculate the $k$ most popular taxi routes in a rolling window. The filter stage extracts the latitude and longitude values for both the origin and destination of a taxi trip from a list of comma-separated values. The filter stage sends this information to the count stage (or the partial aggregator stage, in the case of anycast) to calculate the number of trips that have the same origin and destination locations. Thus, the key for the key-based grouping (after the filter stage) is the tuple's origin-destination pair. The local top-$k$ bolt generates the local $k$ most frequent origin-destination pairs after each one-second window, and the top-$k$ bolt combines the local top-$k$ trips to calculate the global top-$k$ most frequent origin-destination pairs for the taxi trips. Since the key-based grouping happens between the filter and local top-$k$ stages, the mitigation techniques can be applied between these stages. Again, anycast introduces a partial aggregator stage. Similar to the tumbling count application, the partial aggregator stage uses a window to accumulate partial aggregates.

*3.3.3 Spike detection.* This application is based on a benchmark for Apache Storm [2]. The topology is shown in Figure 4. The application performs anomaly detection on a data stream coming from sensors. It maintains a moving average of data from each sensor. The application detects and reports any moving average that increases beyond a specific threshold. In this application, key-based grouping happens between the spout and the moving average bolt. Therefore, this is also where the mitigation techniques are applied. In the case of anycast, the additional stage is the final Moving Average (MA) stage, which corrects the results by computing a single moving average based on the partial moving averages reported by nodes during the moving average stage.

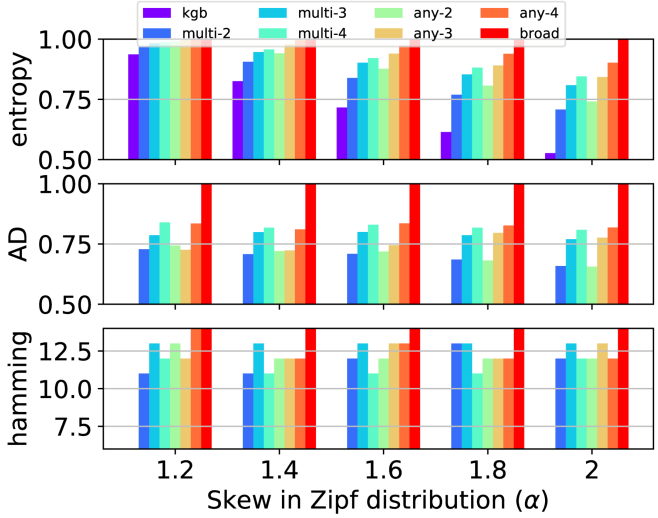


**Figure 5: Data leakage results.**

This application differs from the previous two applications in that there is no windowing and each stage has a one-to-one mapping between the input tuple and the output tuple. Thus, while the application uses key-based grouping between the spout and the moving average stage, each input tuple generates the current moving average as an output tuple. This implies that anycast does not make use of partial aggregates and its obfuscation depends on a lower parallelism level during the final MA stage.

**Workloads:** The tumbling count application uses text from an English book as the default workload. The NYC taxi application uses taxi data [1] as the input workload. Lastly, the spike detection application uses a sensor dataset from the original benchmark [2]. We also generate synthetic workloads with Zipf distributions so that we can present results using a dataset with a configurable skew.

## 3.4 Results

We evaluate how effective the mitigation techniques are at providing data obliviousness according to the metrics defined in Section 3.2. For this evaluation, we perform simulations using the same application logic and the same workloads as in our performance evaluation since tuple routing decisions can be faithfully simulated without errors.

We report results using the tumbling count application and a workload generated using a Zipf distribution parameterized with a skew value $\alpha$ ranging from 1.2 to 2. The parallelism level for the next stage (where mitigation is applied) is 14. Therefore, the maximum n-Hamming distance is 14. We consider multicast and anycast with mitigation level $n \in [2, 3, 4]$, labeled multi-$n$ and any-$n$, respectively.

Figure 5 shows the results as we vary $\alpha$. We use both key-based grouping and broadcast as baselines. We normalize entropy and adversarial disadvantage to the broadcast values. We observe interesting consequences of using different obliviousness metrics.

Broadcast provides the maximum entropy, as expected. Anycast provides better entropy than multicast for corresponding values of $n$. Naturally, increasing $n$ in both anycast and multicast improves the entropy. As the workload becomes more skewed, the entropy values for anycast and multicast drop because the mitigation techniques provide increasingly less uniform distributions of the tuples across different nodes. Therefore, we note that the mitigation level necessary to achieve a particular entropy goal depends on the skew in the workload. For example, with $\alpha = 1.4$, any-4 provides an entropy comparable to that of broadcast, but for a higher skew, a higher value of $n$ is required to provide comparable entropy.

We observe that a technique suitable for achieving one obliviousness metric may not be appropriate for another metric. For example, for $\alpha = 1.2$, any-3 provides comparable entropy to broadcast, but the adversarial disadvantage is significantly smaller for any-3 than for broadcast. Thus, the suitable mitigation technique depends on not only the skew, but also the chosen metric.

Overall, anycast provides better data obliviousness for entropy and adversarial disadvantage than multicast with the same value of $n$. For the n-Hamming distance, all techniques perform fairly well compared to key-based grouping, achieving a distance value of 10 or more (out of 14).

Lastly, we also evaluate the 1-Hamming distance, which indicates whether the adversary can correctly guess the node handling the most popular key. The results show that only key-based grouping and any-2 expose this information.

## 4 STREAM PROCESSING WITH SGX

We designed and implemented an SGX-based distributed stream processing system. It integrates the network obliviousness techniques discussed in the previous sections and offers a framework for the user to select the appropriate mitigation level for a given obliviousness metric.

### 4.1 Design

The main question when designing an SGX-based system is what functionality should execute inside the enclave, and what functionality should not be a part of the Trusted Computing Base (TCB) and instead run as untrusted code. We execute the streaming application logic inside the enclave. To transparently handle the encryption/decryption, key management, and attestation procedures, we place these functions inside the enclave. As we elaborate below, the system must produce routing decisions within the enclave; otherwise, important information about the data could be leaked [12]. Other services and components, such as networking, serialization/deserialization, and fault tolerance, do not execute in the enclave. All network communication is encrypted.

**Usage:** Our system provides an API to create three different types of components (Spouts, Bolts and Sinks) and an API to abstractly define the application topology. The user uses the API to write their own implementation of the application logic. The system instantiates each component inside the enclave.

Therefore, to use the system, the user needs to $(i)$ write application functions that operate on each tuple and, if required, maintain state; $(ii)$ create a driver that specifies functions to be instantiated within the enclave by each component; and $(iii)$ specify the application topology, i.e., a DAG of the components, the parallelism level of each component, the type of tuple routing between the stages, and what mitigation technique and level are to be used.

**Tuple routing:** To correctly realize key-based grouping and the mitigation techniques, tuples are routed based on their keys. However, only the code running within the enclave can access the tuples in clear text. There are two approaches to supporting tuple routing. The first approach, also discussed in prior work [12, 15], is to encrypt the key and the value parts of each tuple separately. This allows routing decisions to be made based on the encrypted keys outside of the enclave. However, this method leaks information about the key distribution unless other obliviousness techniques are introduced [12]. The second approach is to make routing decisions inside the enclave so that the key distribution is not exposed to adversaries. This decreases the scenarios in which an attacker will be able to execute a frequency analysis attack. We adopt the second approach, as it provides better security and leaks no information regarding the key distribution from the encrypted data.

### 4.2 Implementation

Our system is implemented as a distributed stream processing library written in C/C++ using the Intel SGX SDK. We implement the applications discussed in Section 3.3 using the library. The library uses ZeroMQ for distributed messaging, and MsgPack for the message serialization/deserialization protocol. The scheduling algorithm in our system is similar to Apache Storm's default scheduler, which assigns the nodes so that they are spread uniformly across machines. For example, a stage with a parallelism level of 8 deployed on four machines will run two nodes per machine. The mitigation techniques are implemented as routing mechanisms that execute inside the enclave. We highlight a few implementation details.

**Multicast:** The candidate set includes a node $x_k$, selected by key-based grouping, and $n - 1$ additional nodes that succeed $x_k$ in the circular space of the stage's nodes. Multicast sends $n$ output tuples. A valid tuple is sent to node $x_k$ and other nodes receive duplicate tuples. The application logic can check the validity of a tuple using the API. The network I/O is randomized to avoid easily determining the valid tuple (e.g., guessing the first). A flag in the tuple denotes whether it duplicate. We use AES GCM encryption with Initialization Vector (IV) so that all the tuples have different encrypted representations and, thus, the adversary cannot distinguish which tuples are duplicates just by looking at the encrypted data.

One challenge that arises with duplicate tuples is that if the nodes receiving them do not process these tuples at all, then an adversary can use a timing attack to identify the duplicate tuples by observing when a tuple enters an enclave and when it leaves. Therefore, each duplicate tuple must go through the same computation as a valid tuple; however, no state is updated as a result of the computation for duplicate tuples. Thus, the timing differences between valid and duplicate tuples are kept to a minimum. In addition, duplicate tuples must also generate output tuples to avoid being exposed by the downstream traffic.

**Anycast:** The candidate set is the same as above. However, instead of sending tuples to all of the $n$ nodes, anycast only sends one tuple to a node chosen from the candidate set. Specifically, the sending node tracks how many tuples it sends to each downstream node and selects the downstream node that has received the least. This form of load balancing is purely local to the sending node and applies to all the downstream nodes (not just those in the candidate set for a given key $k$). This results in a more uniform tuple distribution.

## 5 PERFORMANCE EVALUATION

We focus on evaluating the median latency and throughput that the applications exhibit when using different mitigation techniques. Throughput is measured based on the number of tuples that have been processed by the last stage in both the spike detection and tumbling count applications. In the NYC taxi application, the throughput is measured at the local top-$k$ stage. This is to uniformly report the throughput of the input tuples. The latency measurements are end-to-end median latencies calculated as the sum of the median latencies for each stage. The stage latency is calculated as the difference between the time at which a tuple is emitted from the upstream stage and the time at which a corresponding output tuple is emitted from the downstream stage.

To enable a fair comparison, the latency and throughput experiments maintain the same amount of resources for all the techniques. Resources are assigned to different stages so as to ensure that the topology does not become saturated in the key-based grouping baseline. The parallelism level that we use for different stages in each of the topologies is shown in Table 1. The same number of total threads are also used for each mitigation technique. For anycast, the number of threads originally assigned to the next stage with key-based grouping is split between the next stage and the additional stage. For example, the 11 threads of the count stage in the tumbling count application are split between the partial aggregator and count stages, as shown in Table 1. We vary the load of the system by increasing the number of spouts. Increasing the number of spouts by two times roughly doubles the input load.

We evaluate each mitigation technique in terms of its effects on latency and throughput compared to the key-based grouping baseline. The combined overheads in terms of performance and resource consumption allow a user to decide the best technique for their application. Performance results are presented for key-based grouping, broadcast, multi-2, multi-3, any-2, and any-3. Multiple versions of anycast and multicast (with mitigation level $n \in [2, 3]$) are used to illustrate how the mitigation level influences performance.

### 5.1 Setup

We use a four-machine setup, each with an Intel Xeon E3-1240v5 processor (4C/8T) and 32GB of DDR4 memory. We run Precision Time Protocol (PTP) to synchronize the clocks of the machines for latency measurements. All of the experiments are executed for 100s, and the per-tuple latency from the first 50s is discarded to allow the system to reach a steady state.

### 5.2 Latency Overheads

Figure 6 shows the median per-tuple latency for each application. The plots for tumbling count (Figure 6a) and spike detection(Figure 6c) use a log-based y-axis. The end-to-end latency for key-based grouping serves as the baseline.

**Tumbling count:** As soon as the number of spouts is increased beyond one, the broadcast latency drastically increases. This is because the system only has the capacity to process all the duplicate tuples generated with broadcast when one spout is used. Once the number of spouts increases, the system is already overloaded. For the used mitigation levels, multicast performs similarly to key-based grouping until the number of spouts increases beyond four. After that point, the system becomes overloaded by the duplicate tuples, and the end-to-end latency increases drastically. Throughout the experiments, anycast maintains the same latency (roughly 100 ms). This is due to the partial aggregation stage, which uses a tumbling aggregation window of 100 ms. After each window, the partial aggregates are sent to a count stage where the final counts are calculated. Any-2 and any-3 perform very similarly. In summary, anycast is the only feasible mitigation technique for more than four spouts given the available resources.
**NYC taxi:** Multicast outperforms anycast in this experiment. This is because the system is able to handle the additional load due to duplicate tuples at the selected parallelism level. Moreover, multi-3 provides a latency comparable to that of any-2. Broadcast is infeasible beyond eight spouts.
**Spike detection:** Anycast outperforms both multicast variants beyond eight spouts. Unlike the tumbling count application, there is no need for a window to calculate partial aggregates in this application and, therefore, there is no extra latency overhead.

Throughout the experiments, the latency appears to saturate while using a specific mitigation technique when the system is overloaded. However, the plots only present the end-to-end latency measurements, which include tuples that have been processed at the sinks. Due to the short duration of the experiments, this hides the in-flight tuples that experience increased queuing times. So, in practice, the median latency grows exponentially after the system is overloaded, as expected.

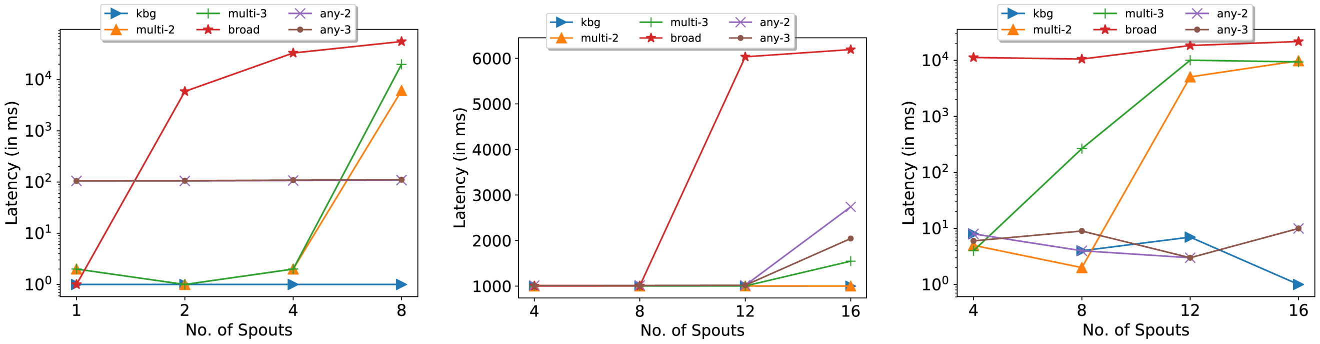
There are two main causes of latency overheads with broadcast and multicast techniques: (*i*) the extra latency in the current stage due to the time it takes to generate duplicate tuples and to perform the networking I/O sending them to the next stage, and (*ii*) the time required to process duplicate tuples in the downstream stages. As the number of duplicate tuples grows linearly in $n$, the latency grows quickly, and the system quickly approaches saturation. On the other hand, the main source of latency overheads in anycast is the delay added by the partial aggregation stage, which contributes a static increase in the latency. However, as we show later, there are a few more causes of latency overheads.

### 5.3 Throughput Overheads

Figure 7 shows the throughput for each application, corresponding to the experiments shown in Figure 6.
**Tumbling count:** All the mitigation techniques except broadcast achieve comparable throughput. For broadcast, the throughput saturates beyond two spouts. As the load increases, the throughput of anycast drops about 25% below that of key-based grouping. Multi-2 has a higher throughput than the anycast variants. At the highest load, any-2 has a slightly higher throughput than multi-3.

**Table 1: Parallelism configurations for different applications.**

| Application | Parallelism | | | |
|---|---|---|---|---|
| | Splitter | Partial Aggregator | Count | |
| Tumbling count | 8 | - | 11 | |
| Tumbling count (anycast) | 8 | 8 | 3 | |
| | Filter | Partial Aggregator | Local Top-k | Top-k |
| NYC taxi | 12 | - | 6 | 1 |
| NYC taxi (anycast) | 12 | 4 | 2 | 1 |
| | Moving Average | Final MA | Spike | |
| Spike detection | 16 | - | 4 | |
| Spike detection (anycast) | 8 | 8 | 4 | |

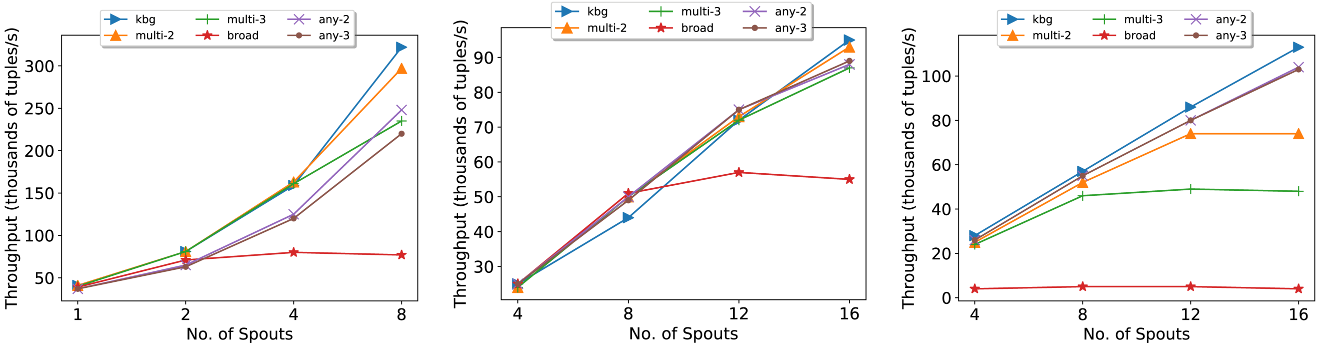


**(a) Tumbling count.**

**(b) NYC taxi.**

**(c) Spike detection.**

**Figure 6: Median per-tuple latency across applications.**



**(a) Tumbling count.**

**(b) NYC taxi.**

**(c) Spike detection.**

**Figure 7: Throughput across applications.**

**NYC taxi:** All the mitigation techniques except broadcast perform similarly to key-based grouping. For broadcast, the throughput saturates beyond eight spouts.

**Spike detection:** Only any-2 and any-3 maintain a throughput sufficiently close to key-based grouping. The throughputs of multi-2, multi-3, and broadcast saturate at four, eight, and twelve spouts, respectively. The results for multi-2 and multi-3 show that both the throughput and the latency increase sharply (Figure 6c). This is counterintuitive, as the latency should spike once the system saturates the throughput. For example, we consider multi-2 with eight spouts. We find that the system is able to accommodate the extra load going from four to eight spouts, but that the load increase already puts the system beyond capacity at the spike bolt. Thus, the latency increases significantly. At twelve spouts, the load increases by 50%, clearly overloading the system.

In summary, while the aggregation stage for anycast adds extra latency, anycast’s throughput is comparable to key-based grouping. Anycast also scales well as the input load changes compared to multicast and broadcast. We observe that anycast also appears to scale well as *n* increases from 2 to 3. Next, we investigate whether this behavior holds for higher values of *n*.

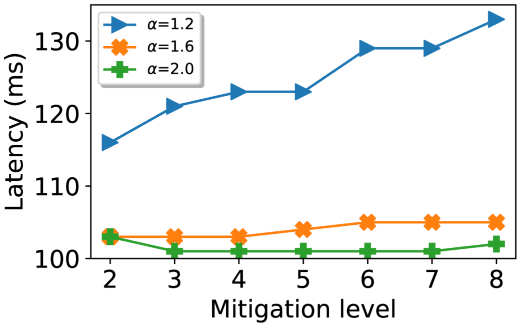


**Figure 8: Latency of anycast when varying mitigation level and workload skew.**

## 5.4 Impact of Mitigation Level

With multicast, the overall load on the system clearly increases linearly with the mitigation level $n$. However, this is not necessarily true for anycast, since it does not introduce duplicate tuples. We now show how increasing the mitigation level with anycast affects the latency by varying $n$ from 2 to $|N| = 8$ (the parallelism level of the next stage).

**Effects of workload skew:** Figure 8 shows the median per-tuple latency in the tumbling count application when executing three different Zipfian workloads with skews of 1.2, 1.6, and 2.0. For highly skewed workloads, we observe that the latency remains stable as we increase $n$. However, with a low skew (1.2), the latency increases by 17% when $n$ increases from 2 to 8. This is due to the behavior of the partial aggregation stage. With lower skew, in any given window of partial aggregates, there are many more distinct keys; therefore, the number of tuples sent from the partial aggregation stage to the count stage at the end of each window is significantly higher than when the workload is more skewed. For example, with a skew of 1.2, there are a total of roughly 19,000 distinct keys in a 100,000-tuple window. With a skew of 2.0, there are only about 500 distinct keys. Thus, with high skew, the keys with lower frequencies become more rare and partial aggregation is more efficient, which reduces the latency overhead.

**Effects of mitigation level across applications:** Figure 9 shows the impact of changing the mitigation level on the median latency for anycast, while using the default workloads for each application. We see that, unlike multicast, anycast accommodates changes in $n$ without incurring a drastic increase in the end-to-end latency. The decrease in the latency as the mitigation level increases can be attributed to the improved load balancing as a result of the increasingly uniform tuple distribution.

In addition to the latency overheads caused by the partial aggregation stage, there are two other factors that contribute to the latency when we vary $n$. First, decreasing skew leads to more unique keys in the partial aggregation window and, thus, more tuples per-window. Second, the value of $n$ affects how partial aggregates associated with each key are distributed. A larger $n$ means that more nodes will have more tuples to send at the end of each partial aggregation window, increasing the overall processing time. Additionally, the downstream nodes will have to process more partial aggregates per tuple and per window as $n$ increases.

# 6 ADAPTIVE MITIGATION

Since anycast scales well with the mitigation level, it is a good candidate for adaptive mitigation at runtime. Figure 5 shows that the mitigation level needed to achieve a particular obliviousness depends on the data skew. Because the skew of the streaming data is likely to change over time, an adaptive mechanism ensures that the system dynamically changes the mitigation level to maintain the security property of interest (e.g., a certain obliviousness metric) above a chosen threshold. In this section, we discuss one such mechanism that we implement in our system and evaluate its behavior as the data skew varies over time.

## 6.1 Mechanism

We design a simple adaptive mechanism that changes the mitigation level as the data skew in the workload changes. The mitigation level is adapted one step at a time.

Each of the current stage nodes locally records two tuple count distributions. First, each node tracks the number of tuples that it has sent to each of the next stage nodes. Second, each node also records a virtual tuple-count distribution to the next stage nodes as if the mitigation level were the current level minus one. Each node shares these distributions every $\delta$ time units with a single controller; $\delta = 100ms$ in our implementation. Then, the controller aggregates these local distributions into a global view.

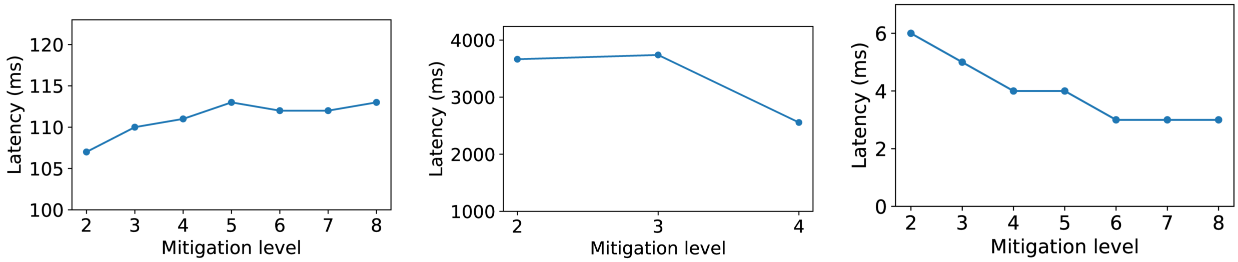
The controller evaluates the security properties from the global view. Our framework provides simple built-in properties and accepts user-defined properties. If the current mitigation level is insufficient to satisfy the security property, it increases the mitigation level by one step. If the current mitigation level minus one would be sufficient, it steps the level down by one. Otherwise, the current mitigation level is maintained. To avoid oscillating, the controller updates the mitigation level only after the above procedure has provided the same decision $x$ consecutive times. This delays the decision by $x \cdot \delta$; we use $x = 5$. If the mitigation level has changed, the controller applies it to all the current stage nodes.

This mechanism has a low overhead. Each node must maintain state in $O(|N|)$ and communicate it periodically to the controller. Assuming a tuple count is 8 bytes, and the parallelism level for both the current and next stage is 8, the resulting traffic overhead to the controller is $2 \cdot 8 \cdot 8 \cdot 8 = 1$KB per $\delta$ time units.

## 6.2 Evaluation

To observe how the system adapts the mitigation level in response to variations in the workload skew, we run an experiment using the tumbling count application. We set the parallelism level of the partial aggregator stage to 16. This implies a maximum entropy of 4. We define a security property that requires the entropy to be ≥ 3.85. The initial workload skew is $\alpha = 1.2$. The initial user-defined value of $n$ is 4.

During the experiment, we first increase and then decrease the skew. We expect to see a corresponding increase and subsequent decrease in the mitigation level. Figure 10 shows the change of



**(a) Tumbling count.** **(b) NYC taxi.** **(c) Spike detection.**

**Figure 9: Latency across applications when varying mitigation level.**

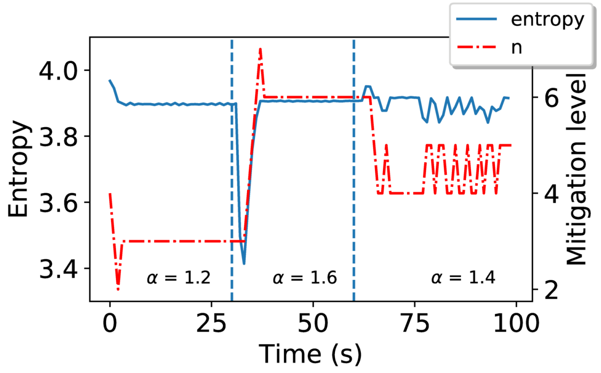


**Figure 10: Dynamic mitigation.**

mitigation level $n$ over time. The workload skew changes at time $t = 30$ and $t = 60$; the events are marked with dashed vertical lines. The right y-axis plots $n$; the left y-axis plots the measured entropy. In the beginning, the system quickly detects that the initial value of $n$ is higher than needed and decreases $n$ to 3. At time $t = 30$, the skew changes to 1.6. As soon as the entropy drops below 3.85, the system adapts $n$ one step at a time until the security property is satisfied. At $t = 60$, the skew drops to 1.4 and the entropy rises. At $t = 65$, the entropy is high enough that the system can safely decrease the value of $n$ to 4, the minimum possible value that still satisfies the security property. Lastly, there are slight changes of entropy between $t = 80$ and $t = 95$. The system responds by adjusting the value of $n$ between 4 and 5, to ensure that the security property is satisfied.

## 7 DISCUSSION

We acknowledge that our framework still has some limitations.

We described how duplicate tuples generated by either multicast or broadcast are tagged as duplicates, but must be processed like valid tuples except that no state changes occur to ensure correctness. While this makes it harder for an adversary to distinguish between the valid and duplicate tuples, it does not guarantee that it is impossible. For example, the adversary may succeed in observing whether state updates happen by performing a memory side-channel attack. Even though other works [5, 6, 10, 16] already address several of these attacks, a possible remedy to this issue would be to process both the valid and the duplicate tuples identically and correct the computation results in the end. This would worsen the performance of broadcast and multicast, thereby making anycast even better in comparison.

Combining leaked information across time or applications (executed on the same input stream) could lead to more information leakage. Further research on the impact of partial leakage is a promising direction for future work.

## 8 RELATED WORK

The work by Ohrimenko et al. [12] is closely related to our work. Their paper presents techniques for mitigating network side-channel leakage in a MapReduce-based system such as VC3 [15]. Their approach mainly relies on techniques such as oblivious shuffling between mappers and reducers, as well as evenly distributing the shuffle traffic between the mappers and reducers. Their techniques also requires that the data is available prior to executing the MapReduce job. While this assumption is reasonable for batch processing applications, it is not reasonable for stream processing systems, since streaming data is often being generated in real time and future data is not available.

Data-oblivious primitives and cache-access techniques have been presented in contexts different from stream processing in prior work [13]. We think that these techniques might be applicable to stream processing as well. Other works, such as T-SGX [16], Déjá Vu [5], Sanctum [6], and GhostRider [10], have proposed techniques for thwarting memory side-channel attacks. However, an evaluation of their performance overheads and applicability to stream processing remains necessary. In addition, these techniques are useful when dealing with data leakage through cache- and memory-access patterns only.

Other recent works have also used SGX. Haven [4] allows the secure execution of unmodified Windows applications using a library OS in an enclave. Similarly, SCONE [3] presents an SGX-based, secure container for running unmodified applications. These generic mechanisms support unmodified applications; however, their TCB is large and they have higher overhead than a specialized system. VC3 [15] realizes a MapReduce framework that minimizes the TCB, where just the map and reduce functions run within the enclave. However, this approach does not apply to the cases such as stream processing where the computations are continuous and latency is critical. Ryoan [9] is a framework for running untrusted code inside the enclave that still ensures that there are no privacy leaks. However, Ryoan relies on a request-based processing model with stateless operators, which makes it unsuitable for stateful stream processing. SecureStreams [8] shows a LuaVM port that runs in enclaves, and RxLua is used to create streaming applications. In contrast to this study, we are concerned with the different design choices that exist when only part of the streaming application is running within enclaves, and how to tame access-pattern leakage. Eleos [14] mitigates memory overheads in SGX using a Secure User-managed Virtual Memory (SUVM), which may provide a solution for mitigating memory overheads in an SGX-based stream processing system as well.

STYX [17] is a streaming system that uses homomorphic encryption primitives to do basic computations over encrypted data. STYX provides a limited choice of operations, and the user needs to be aware of which cryptosystem, such as order-preserving encryption, additive homomorphic encryption, or search-enabling encryption, they should use based on their unique case. $M^2R$ [7] provides a solution to leakage channels that is analogous to distributed Oblivious RAM (ORAM), but incurs far less overhead. The solution has been implemented as an enhanced version of Hadoop to provide privacy in MapReduce computations.

Lastly, our anycast technique is inspired by an algorithm known as the power-of-n choice [11] that achieves better load balancing in stream processing systems. However, our goal and approach are different.

## 9 CONCLUSION

In this paper, we introduce the multicast and anycast mitigation techniques for decreasing network side-channel leakage due to key-based grouping, which is a data-dependent routing scheme, in stream processing systems. We implement these techniques in the prototype of an SGX-based stream processing system. Our performance evaluation shows that anycast supports higher throughput than multicast and broadcast with the same number of resources, and scales well to higher mitigation levels without adding significant overhead. Multicast and broadcast also require more resources to provide the same latency and throughput as key-based grouping. We introduce a mechanism that performs adaptive mitigation-level selection for anycast, which is crucial since the appropriate mitigation level depends on the security property under consideration and the skew in the workload, which may vary over time.

We conclude that broadcast is infeasible unless the input load is very low. Multicast is a feasible option when the load is low enough that each processing node of the application topology can handle approximately twice the load of the most popular keys. We believe that anycast is the best option when it is feasible to accept a static latency overhead. Anycast can sustain high input loads with a few more resources than key-based grouping.

**Acknowledgments.** We thank the anonymous reviewers for their useful feedback. We would like to thank Taesoo Kim for helpful comments and discussions. Muhammad Bilal was supported by a fellowship from the Erasmus Mundus Joint Doctorate in Distributed Computing (EMJD-DC) program funded by the European Commission (EACEA) (FPA 2012-0030).

## REFERENCES

[1] NYC Taxi dataset. https://www.dropbox.com/s/k1qo0j9pehd3vfv/nyc-taxi.csv?dl=0.
[2] Storm applications. https://github.com/mayconbordin/storm-applications.
[3] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer. SCONE: Secure Linux Containers with Intel SGX. In *OSDI*, 2016.
[4] A. Baumann, M. Peinado, and G. Hunt. Shielding Applications from an Untrusted Cloud with Haven. In *OSDI*, 2014.
[5] S. Chen, X. Zhang, M. K. Reiter, and Y. Zhang. Detecting Privileged Side-Channel Attacks in Shielded Execution with Déjá Vu. In *Asia CCS*, 2017.
[6] V. Costan, I. Lebedev, and S. Devadas. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *USENIX Security*, 2016.
[7] T. T. A. Dinh, P. Saxena, E.-C. Chang, B. C. Ooi, and C. Zhang. M2R: Enabling Stronger Privacy in MapReduce Computation. In *USENIX Security*, 2015.
[8] A. Havet, R. Pires, P. Felber, M. Pasin, R. Rouvoy, and V. Schiavoni. SecureStreams: A Reactive Middleware Framework for Secure Data Stream Processing. In *DEBS*, 2017.
[9] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel. Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data. In *OSDI*, 2016.
[10] C. Liu, A. Harris, M. Maas, M. Hicks, M. Tiwari, and E. Shi. Ghostrider: A Hardware-Software System for Memory Trace Oblivious Computation. *CAN*, 2015.
[11] M. A. U. Nasir, G. D. F. Morales, D. García-Soriano, N. Kourtellis, and M. Serafini. The Power of Both Choices: Practical Load Balancing for Distributed Stream Processing Engines. In *ICDE*, 2015.
[12] O. Ohrimenko, M. Costa, C. Fournet, C. Gkantsidis, M. Kohlweiss, and D. Sharma. Observing and Preventing Leakage in MapReduce. In *CCS*, 2015.
[13] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa. Oblivious Multi-Party Machine Learning on Trusted Processors. In *USENIX Security*, 2016.
[14] M. Orenbach, P. Lifshits, M. Minkin, and M. Silberstein. Eleos: ExitLess OS Services for SGX Enclaves. In *EuroSys*, 2017.
[15] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. VC3: Trustworthy Data Analytics in the Cloud using SGX. In *S&P*, 2015.
[16] M.-W. Shih, S. Lee, T. Kim, and M. Peinado. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In *NDSS*, 2017.
[17] J. J. Stephen, S. Savvides, V. Sundaram, M. S. Ardekani, and P. Eugster. STYX: Stream Processing with Trustworthy Cloud-based Execution. In *SoCC*, 2016.
[18] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica. Opaque: An Oblivious and Encrypted Distributed Analytics Platform. In *NSDI*, 2017.