Behind Enemy Lines: Exploring Trusted Data Stream Processing on Untrusted Systems

Cory Thoma
University of Pittsburgh, PA
corythoma@cs.pitt.edu

Adam J. Lee
University of Pittsburgh, PA
adamlee@cs.pitt.edu

Alexandros Labrinidis
University of Pittsburgh, PA
labrinid@cs.pitt.edu

ABSTRACT

Data Stream Processing Systems (DSPSs) execute long-running, continuous queries over transient streaming data, often making use of outsourced, third-party computational platforms. However, third-party outsourcing can lead to unwanted violations of data providers' access controls or privacy policies, as data potentially flows through untrusted infrastructure. To address these types of violations, data providers can elect to use stream processing techniques based upon computation-enabling encryption. Unfortunately, this class of solutions can leak information about underlying plaintext values, reduce the possible set of queries that can be executed, and come with detrimental performance overheads.

To alleviate the concerns with cryptographically-enforced access controls in DSPSs, we have developed Sanctuary, a DSPS that makes use of Intel's Software Guard Extensions (SGX) to protect data being processed on untrusted infrastructure. We show that Sanctuary can execute *arbitrary* queries while leaking no more information than an idealized Trusted Infrastructure system. At the same time, an extensive evaluation shows that the overheads associated with stream processing in Sanctuary are comparable to its computation-enabling encryption counterparts for many queries.

ACM Reference Format:
Cory Thoma, Adam J. Lee, and Alexandros Labrinidis. 2019. Behind Enemy Lines: Exploring Trusted Data Stream Processing on Untrusted Systems. In *Proceedings of Ninth ACM Conference on Data and Application Security and Privacy (CODASPY '19).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3292006.3300021

1 INTRODUCTION

Data Stream Processing Systems (DSPS) have been proposed to execute long-running continuous queries (CQs) over large volumes of fast moving, transient data. DSPSs have applications in a variety of domains such as medical device monitoring, social media, and wearable/mobile devices. Often, DSPSs make use of outsourced third-party computational platforms, such as Microsoft Azure or Amazon EC2, to reduce the overall monetary cost of maintaining the DSPS and to allow for hardware flexibility and service scalability. Such use of a third-party system, however, may violate the confidentiality constraints and access controls of data providers by permitting an unauthorized third-party to view their data.

To address this issue, data providers often specify access controls to limit the disclosure of their sensitive data. These access controls can be enforced by either trusting the DSPS (and its underlying infrastructure) to abide by the specified policies, or through cryptographic mechanisms. The use of a DSPS for enforcement requires that the data provider have some level of established trust with not only the DSPS, but also with the platform on which the DSPS is executing (e.g., Amazon EC2), which may often not be the case. Naive use of cryptography requires the decryption of information prior to query processing, and thus eliminates the possibility of in-network processing on untrusted infrastructure. Fortunately, recent work such as PolyStream [35] and Streamforce [3] makes use of computation-enabling cryptographic techniques. Computation-enabling cryptographic schemes allow some level of query execution to be processed directly on encrypted data (e.g., an order-preserving cryptographic scheme will allow a user to execute an arbitrary range query on a protected data stream). Such solutions can allow for third-party systems to be utilized for processing since data is again protected, and the underlying DSPS framework can remain unchanged.

Unfortunately, these systems are limited in the scope of what *types* of query operators can be supported. For instance, without prior collaboration between data providers, data consumers cannot execute an arbitrary join query over streams emitted by two or more data providers, as each provider will be encrypting their streams using a different key. Moreover, the operations available to consumers are dictated by *how* data providers encrypt their data. For example, if a data provider does not encrypt using an order-preserving encryption scheme, consumers will not be able to execute range queries over encrypted data. As a result, supporting a rich—albeit still limited—collection of operations requires a data provider to encrypt its data in multiple ways (once for each type of operations to be enabled), resulting in increased data transmission overheads.

The use of computation-enabling encryption further has the side effect of leaking peripheral information about a data stream. For instance, a deterministic encryption scheme retains the equality property of the underlying data, which permits a querier to test the equality of two values. This, however, also allows outside observers to learn a distribution of the underlying values. Similarly, the use order-preserving encryption allows outside observers to learn the distribution and relative ordering of underlying values. A data provider can opt out of any given computation-enabling encryption scheme if they deem that the information leaked is too excessive, but this limits the scope of queries that can be executed.

To help alleviate these types of concerns, we explore the use of Intel's Software Guard Extensions (SGX) for securely executing arbitrary data stream queries on third-party systems while minimizing peripheral leakage of information. Our prototype system, Sanctuary, handles only encrypted data and makes use of SGX enclaves to process arbitrary operations over this data in a manner that prevents the exposure of underlying plaintext characteristics. In developing Sanctuary, we make the following contributions:

CODASPY '19, March 25–27, 2019, Richardson, TX, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6099-9/19/03...$15.00
https://doi.org/10.1145/3292006.3300021

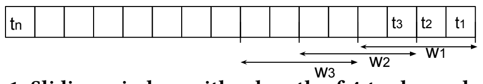


Figure 1: Sliding window with a length of 4 tuples and a slide of 2 tuples.

- We develop Sanctuary, a Data Stream Processing System that utilizes SGX enclaves to support the execution of arbitrary streaming relational operations over sensitive data on untrusted third-party infrastructure.
- SGX enclaves have access to a limited memory (128 MB). Often, stateful operators will require more memory than what an enclave can provide. To this end, we present algorithms for stateful relational operators used by Sanctuary that are designed for the memory-limited enclave environment.
- We provide a detailed analysis of the information that can be gathered by an adversary when using Sanctuary. We show that Sanctuary can achieve a greater level of data protection when compared to state-of-the-art cryptographically-enforced access controls, and further show Sanctuary to be near ideal in terms of information leakage when compared to a baseline system.
- Finally, we carry out an in-depth evaluation of each relational streaming operation in Sanctuary and compare it to similar relational streaming operations for both unprotected (i.e., plaintext) data, and different computation-enabling encryption techniques [3, 35]. We further include enclave-enabled operators as part of larger query networks and evaluate the overheads associated with their use.

The remainder of this paper is organized as follows. We overview SGX and related work in Section 2. We describe the Sanctuary architecture and threat model in Section 3. We detail the challenges in enclave-enabled relational streaming operators and the overheads associated with them in Section 4. We present enclave-enabled limited memory streaming operators used by Sanctuary in Section 5. We provide a detailed discussion on information leakage in Section 6 and evaluate operations and queries that use SGX enclaves in Section 7. Finally, we conclude in Section 8.

2 BACKGROUND AND RELATED WORK

Here, we overview related work and describe Intel's SGX at a level that is sufficient to fully understand the remainder of the paper.

2.1 Data Streaming Systems

Data Stream Processing Systems (DSPSs) either operate on a single machine [1, 4], or are distributed over a cluster or wide-area-network of machines [2, 10, 21, 34]. In a distributed environment (DDSPSs), continuous queries can place individual streaming operations on different, sometimes geographically distant, compute nodes to reduce the network overheads [7, 15, 27, 30], total monetary cost [27], or computational cost of the query [19, 33]. Given the nature of a DSPS, outsourcing computation is desirable to help allocate or re-allocate appropriate resources for each streaming query. However, this may lead to a potential violation of a data provider's access controls.

In this paper, we assume a common streaming model (as assumed in [23, 24, 35]) where data providers distribute data through third-party cloud computing systems. Data consumers place streaming operators onto the cloud system for data processing.

Data streaming operators can broadly be classified into two types: stateless and stateful. *Stateless* operators execute on one *tuple* of a data stream at a time, without any knowledge of prior tuples, to produce a result (e.g., filter out all values over a certain threshold). *Stateful* operators require some knowledge of prior or concurrent tuples in order to produce a result. Typically, these operators keep this information in a *window* that defines a length of time or a number of tuples that are used to compute a result. For instance, a querier may wish to know the average stock price over the last *five minutes*, which requires the streaming operator to keep a window for the last five minutes worth of tuples. In addition to this window, a querier can elect to use a *slide* to form a *sliding window*. A slide simply defines how often the querier desires to receive a window's result. For instance, consider Figure 1. A querier requests the average stock price for the last four tuples but wants the latest average reported *every two tuples* (i.e., a window of 4 with a slide of 2 tuples) (the moving brackets labeled w1, w2, and w3) to yield a finer granularity for their application.

2.2 Access Controls in DDSPSs

To address the privacy and access control concerns of a data provider, several systems and algorithms have been proposed. These systems can be broadly characterized into two groups: *trusted third-party* and *untrusted third-party* access control enforcement. In a trusted third-party access control enforcement environment, data providers specify access controls and allow an outsourced third-party to enforce these access controls. Systems such as FENCE [23, 24] enforce access controls by adding special streaming operators that enforce access controls by filtering tuples that are not permitted to be accessed by a querier. Other systems rewrite queries or alter streaming operators [11–13, 25], while others focus on protecting a single system, such as Borealis [22]. This class of solutions exposes provider data to the infrastructure itself, and must trust the infrastructure to correctly enforce provider access control policies.

Systems that do not trust third-party access control enforcement will rely on cryptographically-enforced access controls. Rather than forcing a querier to process data only after it has been decrypted, systems like PolyStream [35] and Streamforce [3] allow the data provider to use specialized computation-enabling encryption techniques to enable third-party computation for a querier *directly on encrypted data*. These systems, however, limit the expressiveness and accessibility of a queriers' potential query. In Streamforce, a querier may only access *integer* data via a *view-like* format, (i.e., only allowing filtering and aggregations on numeric data). PolyStream supports a richer set of query operations than Streamforce, but cannot support join or complex user-defined functions over streams from multiple providers. Furthermore, these systems also leak information about the underlying plaintext values, such as equality, relative partial ordering, or relationships between groups of tuples (i.e., the encrypted aggregate of some encrypted data).

To help overcome these limitations and provide an alternate avenue for untrusted third-party computation, in this paper, we enable a querier to employ remote SGX enclaves to ensure private computation and to restore expressiveness by allowing for *any* streaming operation to execute on the third-party system.

2.3 SGX

*Overview:*Intel's Software Guard Extensions (SGX) [16] are a set of architectural enhancements to recent Intel processors that provide developers with the ability to create a trusted environment within an untrusted machine. An enclave is given exclusive use of a core of the CPU while it is executing, meaning that no other processes can access on-chip storage, as enforced by the CPU. Further, when the enclave is either finished processing or the CPU has an interrupt, all data is encrypted and written to memory so that no other process can access the plaintext data. An enclave, therefore, offers a developer the opportunity to trust the computation of a third party device in terms of confidentiality (sensitive data values cannot be observed outside of the enclave) and integrity (attempted modifications to protected data or instructions will be detected). SGX also supports the use of remote attestation processes to ensure the integrity of an enclave being staged onto a remote machine.

There are certain aspects of SGX enclave use that a developer must consider. Each interrupt to the SGX-enabled core causes the CPU to encrypt and write data out to unprotected memory, and each further startup causes that data to be brought back into the CPU and decrypted. These *context switches* can cause the overall enclave execution to slow down and can negatively impact performance. Developers need to be cognizant of these types of overheads when making use of enclave technology. Further, enclave memory is limited to at most 128 MB at any given time. Any large-scale, in-memory processes will be severely hindered by this cap and will require the developer to manage swapping to (encrypted) non-enclave memory on-the-fly. The remainder of this paper is devoted to exploring the use SGX enclaves for streaming application, with a focus on exploring algorithms that work with the above issues with SGX enclaves to allow data consumers to make use of third-party computation platforms that provide SGX-enabled CPUs.

SGX for Data Protection: Current work in SGX-enabled computation has focused on many different areas, from securing ZooKeeper data [9]; to managing transactions, or enterprise rights management privately [18]; to segregating linux containers [6]. SecureStream [17] is a system that explores the use of Intel's SGX as a way to execute Map-Reduce streaming applications. Further, VC3 [31] builds a Map-Reduce engine on top of SGX hardware that allows for attestation of code and data on a powerful adversary. In Sanctuary we focus on streaming relational data operators and the challenges associated with windowed operators and memory limitations.

Opaque [37] augments SQL operators so that their memory accesses are hidden and operate within an SGX enclave. Similarly, EnclaveDB [29] provides SQL operators that execute with an SGX enclave. Finally, SGX-BigMatrix [32] provides a high level language that can be used to provide secure, enclave-enabled computations. Sanctuary is complementary to these previous works, as it is designed to enable enclave-protected, real-time relational stream processing with the goal of overcoming the limitations of state-of-the-art cryptographic stream processing systems while also providing enhanced security (cf. Section 6).

3 ARCHITECTURE AND THREAT MODEL

This section overviews the architecture, system model, and threat model assumed by Sanctuary.

3.1 Architecture and System Model

The architecture and system model assumed by Sanctuary are presented in Figure 2. We assume queries are specified in a declarative language like the Continuous Query Language (CQL) [5]. The Sanctuary optimizer (which is/can be run locally on the querier's trusted machine) transforms the query into a set of streaming operators that will need to be ordered and placed in the query network. The

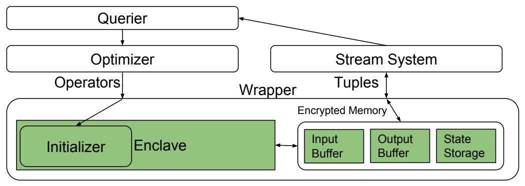


Figure 2: Architecture of an SGX-enabled stream processing system (with a single SGX-enabled core on a single node).

optimizer in Sanctuary may decide to place an operation in an SGX enclave and will use the initializer (which can either operate as part of the streaming operator's code base or even within an enclave) to do so. The work of the initializer is detailed in Section 3.3. We assume that all encryption keys between the data provider and the consumer are transmitted off-line (as described in Section 3.3).

Once the initializer and optimizer have finished executing, all operations are placed and the query may execute. As encrypted data arrives, it is placed into a non-enclave memory Input Buffer. Once the enclave occupies the CPU-core, data will be read from this buffer, decrypted (with the key provided to the initializer during the enclave provisioning step), processed, and encrypted results are stored in the non-enclave memory Output Buffer to be further propagated through the query network. If state must be kept when the enclave does not occupy the CPU, it is encrypted and stored in the State Storage non-enclave memory buffer.

In this paper, we assume that all relevant data arrives encrypted. Individual data packets are represented as *tuples* with *fields* for each piece of information within a data tuple. Tuples can be represented by a *schema* that describes each field (e.g., {ID(int), name(string), heartrate(int), date(dateTime)} or something similar). A schema can also be described using a *key-value* approach.

3.2 Threat Model

Sanctuary is designed to execute operators on an *untrusted* computational platform. We assume that the untrusted party is *honest-but-curious* (an assumption also made in [35], [3], and [28]). An honest-but-curious adversary is one that will not maliciously alter, drop, or add data but will rather try to learn information about the victim by reading and understanding their data. This follows from our system model: it is expected that third-party service providers are attempting to earn money and be successful by providing a cloud computing service. Maliciously altering, dropping, or adding data will result in lost customers and a negative reputation, ultimately causing them to lose money. We further assume that the SGX hardware itself remains uncompromised; i.e., it is patched against side-channel attacks such as Spectre [20] and Foreshadow [36]. As we later detail, Sanctuary is data oblivious with respect to its use of cryptographic keys, which are unlikely to be leaked via side-channel attacks. We further detail how Sanctuary is *not* data oblivious in terms of the data being processed in Section 6.

Sanctuary aims to prevent third-party service providers and adversaries observing the network from being able to obtain or infer the underlying plaintext data produced and transmitted by a data provider. Moreover, Sanctuary allows for data to be fully encrypted during transit to prevent inference of the underlying plaintext values. Finally, Sanctuary aims at limiting the leakage of ancillary data (e.g., tuple value distributions, orderings, etc.) to the service provider and third-parties observing the system.

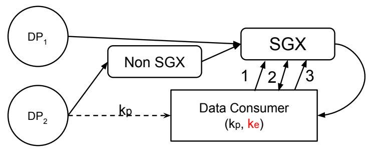


Figure 3: An example scenario of Sanctuary query deployment.

3.3 Deployment Model

Processing a query in Sanctuary involves three main steps: ensuring that the data consumer has the necessary cryptographic keys to decrypt all relevant streams; generating and deploying standard or enclave-based data stream operators; and executing the running query. We will describe each of these phases using Figure 3 as a simple example. This query involves two streams: an unencrypted stream originated by *DP*1, and an encrypted stream originated by *DP*2. In this query, a standard selection operator is first applied to the unencrypted data stream. The resulting stream is then joined to the encrypted stream using SGX-enabled operator.

Key acquisition. In order for a data consumer to access an encrypted data stream, they must have access to the cryptographic key (or keys) used to encrypt the stream. For simplicity, in Figure 3, we assume that a single key, k_p, is used to encrypt the stream originated by *DP*2. In Sanctuary, key management is handled either in an offline manner, or online using a mechanism such as Fence [24] or Polystream [35]. Once a data consumer is able to decrypt streaming data, they are in a position to leverage the ability of Sanctuary to deploy SGX-enabled query operators.

Query deployment. Sanctuary is developed on top of the Apache Storm [34] infrastructure (cf. Section 7 for details). Plaintext relational operators are deployed as Storm bolts in the typical manner. The deployment of SGX-enabled operators is a multi-step process, as shown in Figure 3. First, Sanctuary will create an SGX enclave capable of executing the desired streaming operator (cf. Sections 4 and 5 for details). Next, this enclave is deployed to the Storm infrastructure (arrow 1). SGX remote attestation is then used to ensure the integrity of this operator as it is instantiated within Storm (bidirectional arrow 2). This process results in the derivation of a session key k_s that can be used to communicate securely between the data consumer and the enclave. Finally, k_s is used to encrypt and transmit the data stream key k_p to the operator enclave (arrow 3). At this point, the query network is ready to receive input tuples.

Query execution. In steady state, unencrypted tuples from *DP*1 and processed by the selection operator as in a standard DSMS. Encrypted tuples flowing from *DP*2 into the enclave-based join operator are decrypted using k_p and joined with the output of the selection operator. All result tuples are encrypted with k_s and forwarded to the data consumer.

4 STATELESS OPERATORS

In this section we briefly describe common DSPS stateless operators supported by Sanctuary and discuss how these operators must be altered to execute within an SGX enclave.

4.1 Stateless Operator Overview

Stateless operators interact with the enclave in a manner depicted in Figure 4. Operators are sent to the untrusted third-party by the data consumer. As data arrives from the data providers, it is decrypted, processed, re-encrypted, and finally sent to the data consumer. This decryption key is provided at the initialization of the enclave as part of the operation itself. There are three common stateless operators used in DSPSs: Filter, Projection, and Map. Given the straightforward nature of these operators, the only changes needed for execution within an enclave are the decryption of input tuples and the (potential) re-encryption of output tuples with a constant overhead (explored further in Section 7).

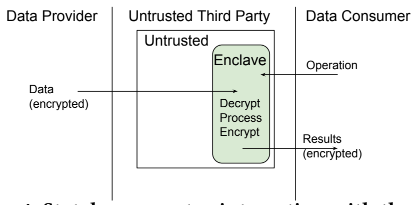


Figure 4: Stateless operator interaction with the enclave.

Filter: The filter operation simply verifies that a particular field matches a desired expression for range or equality (e.g., $x = y$, $x > y$, etc.). A filter must store *the predicate* (i.e., the constant comparator as described by the data consumer's query), the field to compare the predicate too (i.e., the "name" field, or a field identified by its placement in the tuple), and the operator code itself within the enclave, which reduces the overall enclave capacity. Processing a tuple will likely require a decryption (of at least the required field) and may require a re-encryption, depending upon whether the result must be transmitted in ciphertext or plaintext.

Projection: The projection operation reduces the size of a tuple by filtering out a specific set of fields to be passed along. The only information required to be saved in the enclave for a filter operation is the set of field identifiers (e.g., field name or placement within the tuple) to be preserved in the resulting tuple. It is likely that the tuple will not need to be decrypted, as no value is being checked. However, if fields are identified by a key-value type of system (i.e., fields may appear in any order in a given tuple), then a decryption of the entire tuple is required.

Map: Similar to the projection operation, a mapping operation reduces the size of a tuple by performing a function on several fields. For instance, a mapping operation may take fields for *revenue* and *expenses* and produce one field called $profit$. A mapping operator will have to decrypt the desired fields required to preform its function, and may need to encrypt the resulting field if the result needs to passed further down the operator network.

4.2 Enclave vs. Non-Enclave CPU Contention

Recall from Section 2 that when a program requests enclave operations, the CPU will halt all other processes and load the enclave-enabled program, as well as any data that is associated with the program itself. Similarly, whenever the enclave-enabled program completes its task within the enclave, it must write back any instance data, its code, CPU memory, and its metadata back to encrypted memory so that the CPU can preform other tasks and maintain the proper separation between enclave and non-enclave processes. This context-switching adds a processing overhead to the overall operator execution for every switch that is required. Specifically, entering an enclave will have a cost c_{enter} and exiting has a cost of c_{exit}. Each cost is dependent on the machine, workload, and other processes on the CPU and can vary with every entry and exit. The best way to mitigate this context-switching cost is to reduce the number of entries into and exits from the enclave.

Some context switching is unavoidable, but a streaming operation can mitigate the negative impact of context switching by simply reducing the number of calls into the enclave by batching data tuples. Rather than calling the enclave and paying c_{enter} and c_{exit} for *every* data tuple received, data tuples can be batched so that a single c_{enter} and c_{exit} is incurred and amortized over all n tuples in a batch. In a data stream, batching may add considerable latency to a query since results are delayed until a batch has been filled. In Section 7 we explore the benefits of batching.

5 STATEFUL OPERATIONS

When an operation is required to consider multiple tuples in order to execute a query, it is considered to be stateful. In this section, we overview the common stateful operators used by Sanctuary, and detail the challenges associated with implementing this class of operations within an enclave. We further propose three algorithms for executing stateful operations inside an SGX enclave.

5.1 Operators

There are two main types of stateful operations in a DSPS: joins an aggregations. This section overviews those operations and classifies the different types of each.

Joins: In a streaming system, a join operation compares two or more different streams in a given window and returns a set of data tuples comprised of data from each stream based upon some join condition. The specified period for comparing each stream is called the *window* which can be expressed either in time or in number of data tuples. A join must keep state on all data tuples that are within the current window for all streams in the join. For example, if a data consumer requests "all tuples where `streamA.id = streamB.id` for the last 10 minutes", all tuples in `streamA` and `streamB` that were timestamped within the last 10 minutes must be stored to compare with new tuples within the window.

Streaming join algorithms can be designed to either 1.) consider all possible pairs of join tuples, or 2.) consider a smaller set of tuples in each stream by using some auxiliary data structure. The *nested loop join* (NLJ) is a join algorithm that must consider all of the tuples in each stream's state by attempting to join every data tuple in one stream with every data tuple in the other. In practice, such a join algorithm is undesirable because of the overheads incurred. However, looping over all tuples avoids the leakage of positional information regarding the specific tuples being joined. *Hash joins* use an auxiliary hashing structure to reduce the number of tuples that need to be compared, reducing the overhead for the join algorithm, but potentially leak information about underlying data.

Distributive and Algebraic Aggregation: An aggregation is *distributive* if the input can be distributed to many partitions where a partial aggregation is processed, followed by a final aggregation of the partials (e.g., a sum can be broken down into smaller sums, with a final sum of the partials generating the overall result). *Algebraic* aggregations are those that can be represented as an algebraic function of two or more distributive aggregations (e.g., average can be calculated by a summation and a count). Distributive and algebraic aggregates have a constant memory overhead.

Holistic Aggregation: An aggregation is holistic if there is no constant bound on the memory required for partial or final aggregation. For instance, the *median* operation is holistic because there is no way to determine the size of the resulting set of median values. A window in a holistic aggregate is treated in the same manner as in the distributive or algebraic aggregations.

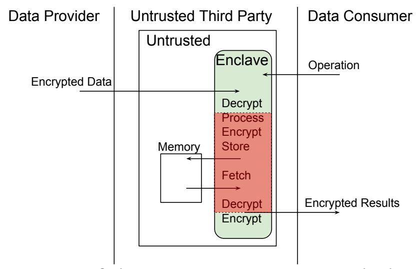


Figure 5: Stateful operator interaction with the enclave.

5.2 Issues with Stateful operators and Enclaves

Stateful operators cannot be directly implemented in an enclave environment without alteration. Specifically, there are two main concerns when implementing a stateful operation on an enclave: *memory limitations* of the enclave and *update costs* associated with auxiliary data structures that may be required.

Memory Limitations: Recall from Section 2 that an enclave is limited to just 128 MB of memory, which is further reduced by the need to store operator code o and meta-data m within the enclave. Stateful operations must make use of this limited memory to store each operation's windows of tuples. Windows can be of a nondeterministic size w (e.g., the last 10 minutes saw 10k tuples, but the next 10 minutes may see 13k tuples) and may not fit into enclave memory. Any stateful algorithm will therefore have to consider swapping between (encrypted) non-enclave memory and enclave memory.

In addition to the window itself, some operations (e.g., hash joins) maintain auxiliary structures. Such structures will vary in size s across operators and will likely need to be kept (at least in part) in enclave memory, further reducing the available memory. Therefore, the total capacity for storing data tuples in an enclave with $enclaveSize = 128MB$ and n operators each with w_s windows (that can vary in size, 0 for stateless operations) is fixed at:

$$capacity = enclaveSize - (m + (\Sigma_0^n (o + s + \Sigma_0^{w_s} w))). \qquad (1)$$

Update Cost Overhead: In a traditional DSPS, removing tuples from or adding tuples to a window in relatively straightforward: one simply checks the timestamp for each tuple as new tuples are added, and removes or dereferences those that have expired. The use of an SGX enclave presents a new challenge with regards to updating the state of a stateful operator. Specifically, when the state of an operator is encrypted in non-enclave memory, the timestamp and most recent tuple information may not be available without either trusting the third-party service to remove expired tuples or leaking some temporal information about the tuples. We can calculate the cost c_{up} of an update by simply multiplying the number of tuples being updated n_{up} by the time it takes to execute that update l_{up}. This cost is added to the overall latency for stateful operators.

5.3 Enclave-Enabled Stateful Operators

We now introduce three algorithms for stateful streaming operations that can execute within an SGX enclave: Nested Loop Join (NLJ), Hash Join (HJ), and generic aggregation (AGG). All algorithms follow the structure depicted in Figure 5. Data consumers use SGX's remote attestation capabilities to ensure that their stateful operator

Algorithm 1 *enclaveNestedLoopJoin(Array{tuple} batch)*

```
1:  Array[tuple] batchSide       ▷ Stored tuples from the same stream as the tuples in batch
2:  Array[tuple] otherBuffer                          ▷ Stored tuples from the other stream
3:  int maxJoinSetSize                                   ▷ Available in-enclave memory.
4:  Object metadata                                  ▷ Storage describing the operator.
5:  for $t \in batch$ do
6:      $batchSide.add(t)$
7:      $enclave.decrypt(t)$
8:  for $i = 0; i \leq \lceil otherBuffer/maxJoinSetSize \rceil; i++$ do
9:      if $i != \lceil otherBuffer[i]/maxJoinSetSize \rceil$ then
10:         $segment_{tuple} = memGet(i * maxJoinSetSize, (i * maxJoinSetSize) + maxJoinSetSize)$
11:     else
12:         $segment = memGet(i * maxJoinSetSize, otherBuffer.size)$
13:     for $l \in segment$ do
14:         if $l.timeStamp < currentTime - window$ then
15:             $evict(otherBuffer.getiAllMatchingValues(l.value))$
16:         $enclave.decrypt(l)$
17:         for $s \in batch$ do
18:             if $l[metadata.joinField] \bowtie s[metadata.joinField]$ then
19:                 $emit(enclave.encrypt(join(l, s)))$
```

enclaves have been appropriately provisioned to the remote infrastructure. As (encrypted) data is received by the enclave from a data provider, it is decrypted and processed. Once processed, it is either discarded (aggregation) or stored in untrusted, encrypted memory. Tuples can be brought back into enclave memory as needed (e.g., to be joined with new tuples) or the partial aggregate to which it contributed can be brought into memory (e.g., the sum for the slides affected by the tuple is brought in to sum new tuples to). This process of fetching and storing continues until all new data is processed. Note that these alrorithms are deisgned to work in *any* memory limited environment, but are more well suited for the SGX use-case as they aim at avoiding costs associated with CPU context switching.

5.3.1 *Join Algorithms.*

Nested Loop Join: We first overview our Nested Loop Join (NLJ) in Algorithm 1. The enclave sets up two spaces in non-enclave memory to represent the windows for each stream. When new data for a stream enters the enclave, the window for the other stream is loaded into enclave memory. If the entire window does not fit, it is segmented (lines 8–12). The *memGet* function simply takes two indices, maps them to registers in memory, and fetches the values. Each segment is then compared and joined to the new tuples being processed (lines 13–19), any joined results are emitted to the next operation or data consumer. In addition to being compared to new tuples, a tuple being brought in from non-enclave memory is also evicted if its timestamp no longer fits within the window (line 14). Finally, all new tuples are added to the end of their window without bringing that state into memory. To implement such an operator, Sanctuary need only the field names from each stream, as well as the slide and window. Sanctuary simply submits the operator with this metadata to a remote system as a function and then verifies it via remote attestation. Whenever the query is ready to be executed, Sanctuary simply executes the function.

A Nested Loop Join is not generally desirable, given that it must compare every tuple in one window with every tuple in the other (or at the very least compare new tuples from each window with the other one). This does, however, offer a nice confidentiality guarantee in an enclave setting, as it does not reveal the relationship between any *specific* tuples in non-enclave memory with new tuples being processed (discussed further in the next section), since every tuple is compared against all buffered tuples.

Algorithm 2 *hashJoin(Array{tuple} batch)*

```
1:  Map[String, Array[tuple]] batchSideHash
2:  Map[String, Array[tuple]] otherHash
3:  Array[tuple] batchSideBuffer          ▷ Stored tuples from batch's corresponding stream.
4:  Array[tuple] otherBuffer                            ▷ Stored tuples from the other stream.
5:  int maxJoinSetSize                                     ▷ Available in-enclave memory.
6:  Object metadata                                    ▷ Storage describing the operator.
7:  for $t \in batch_{tuple}$ do
8:      $enclave.decrypt(t)$
9:      if $otherHash.get(t.value) != null$ then
10:         int $jSize = otherBuffer(otherHash.get(t.value)).size$
11:         for $i = 0; i \leq \lceil jSize/maxJoinSetSize \rceil; i++$ do
12:             if $i != \lceil jSize/maxJoinSetSize \rceil$ then
13:                 $matchSet = memGet(otherBuffer, i * maxJoinSetSize, (i * maxJoinSetSize) + maxJoinSetSize)$
14:             else
15:                 $matchSet = memGet(otherBuffer, i * maxJoinSetSize, otherBuffer.size)$
16:             for $r \in matchSet$ do
17:                 $enclave.decrypt(r)$
18:                 if $r.timeStamp < currentTime - window$ then
19:                     $evict(otherHash.get(r.value).get(r))$
20:                 else if $t[metadata.joinField] \bowtie r[metadata.joinField]$ then
21:                     $emit(enclave.encrypt(join(t, r)))$
22:     if $t.value \in batchSideHash$ then
23:         $batchSideBuffer[batchSideHash.get(t.value)].add(t)$
24:     else
25:         $batchSideHash.put(t)$
26:         $batchSideBuffer.extendByOne()$
27:         $batchSideHash.get(t) = batchSideBuffer.size - 1$
28:         $batchSideBuffer[batchSideBuffer.size - 1] = newArray()$
29:         $batchSideBuffer[batchSideBuffer.size - 1].add(t)$
```

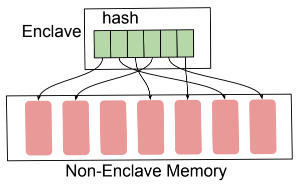


Figure 6: Use hashing to split a window in non-enclave memory.

Hash Join: Algorithm 2 details our Hash Join (HJ) algorithm. Within the enclave memory, a hash structure it maintained for each stream in the join. For every unique key in the join predicate (e.g., each name in a "name" field that has been processed), an entry is made into the hash structure where the key is the predicate, and the value is a space in non-enclave memory where the actual tuples are held, as pictured in Figure 6.

The process of operating on a new tuple is simple. It is first decrypted (Line 8) and then the tuple's join predicate value is hashed to see if there exists at least one match in the other stream (Line 9). If so, the entire set of tuples in the hash entry, or the *matching set* (i.e., the segment of all tuples with the same hash key) is brought into the enclave by enclave-memory sized segments (Lines 10–15). Once all matching tuples are joined to the new tuple, the new tuple is added to its originating stream's hash with its corresponding entry. If the tuple's predicate did not exist in the stream's hash, a new entry is created and the tuple is added to the endpoint for the pointer stored in the hash (Lines 22–29). Note that if the internal hash structures(s) run out of memory, a secondary hash is created in non-enclave memory that will absorb some hash values (i.e., all non-matching predicates are sent to a second hash to be checked). This adds one extra memory operation and one extra hash operation to fetch the desired hash-value. This is not included in Algorithm 2 for simplicity. Similar to the requirements for Nested Loop Join, Sanctuary only requires the fields needed to preform the join, as well as the window and slide. The user may also specify some memory specifications for how they prefer to handle allocation of enclave memory.

Updates in the HJ algorithm are a bit more complicated. A tuple is only removed from a set referred to by a hash entry if that entry is brought into enclave memory. The benefit of this approach is rather straightforward in that updates are handled in an ad-hoc, on-the-fly manner without requiring any extra loads from memory. The obvious drawback is that some data may linger around for a while if its predicate is not matched by the other stream. We leave garbage collecting expired tuples to future work.

5.3.2 Generic Aggregation. Our generic aggregation algorithm (AGG) can handle distributive, algebraic, and holistic aggregations. There are two main memory structures for an aggregation: storage for internal partial aggregations, and the final aggregation step. For distributive and algebraic operations, the state needed for storing the partial results during a window is deterministic and can be provisioned accordingly. To accommodate for holistic operations, we assume that the size of the returned result is non-deterministic for all aggregations. In a memory limited environment, this means that results for each window of the aggregation may need to be stored in non-enclave memory. Further, in a holistic aggregation, these results may vary in size depending on the data and the window.

To accommodate for all three types of aggregation operations, we adopt an approach similar to the hash join approach. Each slide (or window in the case where there is no slide) gets an entry in an in enclave memory array. This entry (potentially) points to a hash table kept in non-enclave memory. For distributive and algebraic aggregation, this hash table may contain only one entry and may fit in enclave memory. For holistic aggregations, this hash table may contain many entries that need updating (e.g., the total sales for each company in a given stream for a given window).

Algorithm 3 details our aggregation algorithm. Each slide is given an index in an array that is stored in enclave memory. Every time a new tuple is received (or a batch of tuples), the algorithm loops through this array. If an entry has expired (based on checking the time inside a designated hash entry (Line 7) it is brought into memory (Lines 8–18) where each entry is emitted (Line 15) and cleared (Line 16) so that the hash may be reused.

For non-expired slides, they are similarly brought into memory, but instead of being emitted, they are aggregated with each tuple in the batch (Line 24). If the entry already exists in the hash for the batched value, it is aggregated to the matching value (Lines 25- 26). The function *genAgMemHelp* takes the array that the hash refers to, the current index, and the maximum size of the buffer, and fills that buffer with values from the encrypted memory by converting the index into a starting and ending register. Once all of the hash has been brought into enclave memory, if any new tuples remain (e.g., a new company has entered the stream that was not yet encountered during this window), they are simply added to the hash as the first entry for that value (Line 28- 30).

Again, to make use of this operator, Sanctuary simply needs the field to aggregate, the type of aggregation, and the window and slide information. The user may also specify the allocation of memory here as well, but Sanctuary handles the bulk of the load by allowing a user to just specify the most basic of information and doing the submission, attestation, and execution for them.

State updates in AGG are simple in that they only require that a slide expire for state to be reset. An adversary can only gain information on how many entries are in a hash table and how many slides are in a window based on what is stored in non-enclave memory. This is no different than in distributive or algebraic operations directly on encrypted data, as the length of the slide can be determined by the rate at which results are produced, and the size of the result set is equivalent to the size of the hash table in AGG.

Algorithm 3 *aggregate*(*Array*{*tuple*}*newBatch*, *int window*, *intslide*, *int maxBufferSize*)

```
 1: Takes: aggregate(Array[tuple]newBatch, int window, int maxBufferSize)
 2: Array[Array[tuple]] slideArray                ▹ Stores current slides in the window
 3: Map[String, Array[tuple]] hash
 4: int maxBufferSize                             ▹ Available in-enclave memory.
 5: for hash ∈ slideArray do
 6:     ▹ If the hashed aggregates are now greater than the window length, emit them as results.
 7:     if earliestTime(newBatch) ≤ (hash.get(startTime)) + slide then
 8:         for i = 0; i < ⌈hash.size/maxBufferSize⌉; i + + do
 9:             Map[String, Array[tuple]] currentHash
10:             if i < ⌈hash.size/maxBufferSize⌉ then
11:                 currentHash = genAgMemHelp(slideArray.indexOf(hash), i, maxBufferSize)
12:             else
13:                 currentHash = genAgMemHelp(slideArray.indexOf(hash), i, hash.size)
14:             for j = 0; j < currentHash.size; j + + do
15:                 emit(currentHash.get(j))
16:                 currentHash.get(j).clear
17:                                               ▹ Otherwise aggregate the new batch into each slide.
18:     else
19:         for i = 0; i < ⌈hash.size/maxBufferSize⌉; i + + do
20:             if i < ⌈hash.size/maxBufferSize⌉ then
21:                 currentHash = genAgMemHelp(slideArray.indexOf(hash), i, maxBufferSize)
22:             else
23:                 currentHash = genAgMemHelp(slideArray.indexOf(hash), i, hash.size)
24:             for t ∈ newBatch do
25:                 if t.group ∈ keys(currentHash) then
26:                     aggregate(t.value, currentHash.get(t.group))
27:                     newBatch.remove(t)
28:             if i = ⌈hash.size/maxBufferSize⌉ && newBatch.size > 0 then
29:                 for t ∈ newBatch do
30:                     currentHash.put(t.group) = t.value
31:             memOut(hashArray.indexOf(hash), i, currentHash)
```

6 SECURITY ANALYSIS

We now detail the information that an adversary can learn by observing the execution of queries within Sanctuary. To contextualize this analysis, we compare directly to two alternative DSMS approaches.

6.1 Comparison Framework

Below are the three system models (including Sanctuary) within which we will compare information leakage:

- **Sanctuary**: In this system model (cf. Section 3) we assume that our adversary is the third-party computational infrastructure hosting a query comprised of the SGX-enabled streaming operators described in this paper. As such, the adversary can observe all (encrypted) traffic flowing between operators, as well the encrypted traffic flowing between the enclave and non-enclave portions of an individual operator. To upper-bound information leakage, we assume one operator per enclave.
- **Cryptographic**: In this system model, we assume that our adversary is a third-party computational platform hosting a query comprised of cryptographic streaming operators. I.e., data streams are encrypted using computation-enabling encryption as in [3, 28, 35]. As such, the adversary can observe all (encrypted) traffic flowing in and between operators.
- **Trusted Infrastructure**: As a baseline for comparison, we consider a trusted third-party computational platform capable of processing standard streaming operators over plaintext tuples (e.g., [23, 24]). This is effectively the *optimal* approach in terms

Table 1: Level of leakage for the various approaches (S = Selectivity, TM = tuple matching, VO = tuple ordering, VD = value distribution, SM = segment matching, W = Window)

Operator		Sanctuary	Cryptographic Data	Trusted Infrastructure
Filter	Equality	S	S, TM, VD	S
	Range	S	S, TM, VO, VD	S
Project		∅	∅	∅
Join		S, SM, W	*Not Supported*	S
Aggregation		W	W	W

of minimizing leakages with our current threat model. We assume that all streaming tuples are encrypted (e.g., using TLS) while in the network. Our adversary is *not* the computational infrastructure, but rather an entity capable of monitoring all communications between nodes in the system. To upper-bound information leakage, we assume one operator per node.

The Sanctuary and cryptographic models are meant to provide a level playing field for comparing the approach presented in this paper with the current state-of-the-art by considering streaming computations that execute on an untrusted infrastructure. The latter Trusted Infrastructure model serves as a basis of comparison for considering what information can be learned by an outside observer who is watching data being processed on a trusted platform. We now examine the types of leakages exhibited by each type of operator considered in this paper, within each of the above system models.

.

6.2 Properties

To understand the leakage of information in various DDSMS deployment models, we first identify types of leakage. In this section, we use the notation $\mathsf{E}_{\mathsf{DET}}(k, v_i)$ (resp. $\mathsf{E}_{\mathsf{OPE}}(k, v_i)$ or $\mathsf{E}_{\mathsf{CCA}}(k, v_i)$) to denote the deterministic (resp. order-preserving or CCA-secure) encryption of a tuple v_i using the key k. We use the notation $\mathsf{E}(k, v_i)$ in situations where the specific type of encryption used is immaterial.

- **Tuple Matching (TM)**: Given a set of input tuples $S^{in} = \{t_1^{in}, \ldots, t_i^{in}\}$ and a set of output tuples $S^{out} = \{t_1^{out}, \ldots, t_j^{out}\}$, compute the matching $M = \{\forall t^{out} \in S^{out} : (t^{in}, t^{out}) \,|\, t^{in} = \mathsf{E}(k, v^{in}) \wedge t^{out} = \mathsf{E}(k, v^{out}) \wedge v^{in} = v^{out}\}$.
- **Value Ordering (VO)**: Given a set of tuples $S = \{t_1 = \mathsf{E}(k, v_1), t_2 = \mathsf{E}(k, v_2) \ldots, t_i = \mathsf{E}(k, v_i)\}$, compute an ordering $t_1', t_2', \ldots, t_i'$ such that $v_1' \leq v_2' \leq \ldots \leq v_i'$.
- **Value Distribution (VD)**: Given a set of tuples $S = \{t_1 = \mathsf{E}(k, v_1), t_2 = \mathsf{E}(k, v_2) \ldots, t_i = \mathsf{E}(k, v_i)\}$, compute the frequency distribution $\hat{v_1} = \text{count}(v_1, S), \hat{v_2} = \text{count}(v_2, S), \ldots, \hat{v_i} = \text{count}(v_i, S)$. Note $\hat{v_i}$ does not necessarily reveal the value v_i.
- **Segment Match (SM)**: Given an input tuple t and a segmented window $w = \{ms_1, \ldots, ms_k\}$, identify the matching segments ms_i within which t can complete a join.

In addition, we will explore whether the selectivity (S) of a given predicate or the window size (W) of an operation can be inferred.

6.3 Leakage Comparison

We now examine the information that can be inferred by an adversary when observing the execution of each of the above systems. We consider the streaming relational operators described in this paper, and summarize our results in Table 1.

Select/Filter. Each selection operator takes as input a stream $S^{in} = t_1^{in}, \ldots, t_i^{in}$, applies a filter f, and produces as output a stream $S^{out} = t_1^{out}, \ldots, t_j^{out}$. In each system considered, the adversary can compute the selectivity of f by comparing the cardinality of S^{in} and S^{out} over some window. In both the Sanctuary and Trusted Infrastructure models, all tuples are encrypted using CCA-secure cryptography (i.e., $t_i = \mathsf{E}_{\mathsf{CCA}}(k, v_i)$). As such, the values v_i comprising the input and output streams are protected.

In the Cryptographic model, equality filtering is enabled by the use of deterministic encryption (i.e., $t_i = \mathsf{E}_{\mathsf{DET}}(k, v_i)$). As a result, $\mathsf{E}_{\mathsf{DET}}(k, v_i) = \mathsf{E}_{\mathsf{DET}}(k, v_j)$ if and only if $v_i = v_j$. This enables the adversary to infer a distribution of values over the encrypted tuples in S^{in} irrespective of the filter f. Further, the adversary can infer exactly which tuples t_i^{in} match the predicate f, as these tuples appear unmodified in S^{out}. For range filtering, order-preserving encryption must be employed (i.e., $t_i = \mathsf{E}_{\mathsf{OPE}}(k, v_i)$) so that $\mathsf{E}_{\mathsf{OPE}}(k, v_i) \leq \mathsf{E}_{\mathsf{OPE}}(k, v_j)$ if and only $v_i \leq v_j$. This enables the adversary to determine an ordering over tuples appearing in S^{in} and S^{out}.

Projection. For all three frameworks, a projection simply removes fields from *every* tuple and therefore always has a 100% selectivity. Since input and output values are encrypted, tuple values, distributions, and orderings remain hidden in all cases.

Join. A join operator takes as input two streams $S_1^{in} = t_{11}^{in}, \ldots, t_{i1}^{in}$ and $S_1^{in} = t_{12}^{in}, \ldots, t_{j2}^{in}$, applies a join predicate p, and produces an output stream $S^{out} = t_1^{out}, \ldots, t_k^{out}$ that joins S_1^{in} and S_2^{in} using p over some (time- or tuple-based) window W. Given that S_1^{in} and S_2^{in} are typically encrypted with different keys, no existing cryptographic DDSMS supports join operations over streaming data.

In both the Sanctuary and Trusted Infrastructure models, input tuples are encrypted using randomized encryption (i.e., $t_{ij}^{in} = \mathsf{E}_{\mathsf{CCA}}(k_j, v_i)$), thereby preventing the inference of tuple values, distributions, and orderings. In both cases, the adversary can easily compute the selectivity of p by comparing the cardinality of S_1^{in}, S_2^{in}, and S^{out} over some time window. In Sanctuary, the adversary has the ability to monitor the enclave's accesses to non-enclave memory. Recall that given enclave memory limitations, a join window $w_i^1 \in S_1^{in}$ is managed as a set of matching segments $w_i^1 = \{ms_{i1}^1, \ldots, ms_{ik}^1\}$ each of which fits into enclave memory. By monitoring the eviction rate from these segments, the adversary can infer the window size used by the join. Further, as new tuples arrive, auxiliary data structures are used to retrieve only the segments that will join with the incoming tuples, thereby leaking the segments that incoming tuples match to.

Aggregation. During aggregation, a sliding window of tuples is combined to produce a single output tuple. In the Cryptographic and Trusted Infrastructure models, the adversary can infer the window size, W, used for the aggregation operation by counting the number of tuples $\mathsf{E}(k, v_i)$ consumed by the operator prior to emitting each output tuple. Likewise, in Sanctuary, the adversary can infer the W by watching the transition of encrypted tuples between enclave and non-enclave memory. In all cases, the use of randomized encryption prevents the inference of tuple values, ordering, and distribution.

Summary: Sanctuary leaks the same minimal information as the Trusted Infrastructure system for all operations excepting the join, which can be mitigated if the join state fits in enclave memory. Further, Sanctuary not only leaks the same or less information as cryptographic DDSMS systems, but also enables arbitrary join operations, the lack of which is a severe limitation of existing cryptographic systems.

7 EVALUATION

In this section, we explore the efficiency of Sanctuary operations in a real streaming system. Specifically, we benchmark each operation in comparison to Trusted Infrastructure approaches under different experimental conditions. We then use SGX-enabled operations within the context of deployed streaming queries to evaluate the impact of enclave-based operations on query performance.

7.1 Configuration

Our evaluation framework builds upon the Apache Storm infrastructure [34] to manage the network topology and deliver tuples. Our work can be trivially deployed over any DSPS. Storm uses two main computational components called *spouts* (provide data) and *bolts* (execute on it). For this work, we use bolts to emulate a node. This implies that a single bolt has access to one SGX-enabled CPU. We use Storm to emulate the temporal aspect of real-time data stream processing since it can be configured to guarantee in-order tuple delivery, and spouts can emit tuples at a given timestamp (to better control input rate). All experiments were executed on a machine using the Windows 10 operating system with a dual core Intel i5-6200 CPU (2.30 GHz) and 4 GB of ram.

Datasets: To better explore the tradeoffs in size, speed, and selectivity of data, we will use two different types of data-sources. The first type is synthetically-generated data that is purposefully created to test the boundaries for each SGX-enabled operation as well as its alternatives. Each evaluation that alters this data will describe, in detail, how it is generated and used. The second set of data is the 2015 DEBS Grand Challenge dataset [14] consisting of tuples that describe an instance of a taxi ride (i.e., the start time, taxi driver ID, cab ID, end time, fare, distance, etc.).

Comparison Approaches: For each operator or algorithm we test, we will compare it to the same operation executed over *plaintext* and also in a DSPS using *computation-enabled encryption*. We will use three different encryption techniques for comparison: 1.) Deterministic Encryption (DET, which uses AES in CBC mode with padding, see [35] and [28]) for equality operations, 2.) Order Preserving Encryption (OPE, which uses the Boldyreva et. al [8] technique) for range operations, and 3.) Homomorphic Encryption (HOM, using the Paillier technique [26]) for aggregate operations.

7.2 Micro Benchmark: Stateless Operations

For stateless operators, there is no state to keep track of, tuples are simply fed directly to the enclave and processed in *batches*, with batched results being returned. All results are based on the average processing time for 10,000 tuples. For batch execution, the results are based on the average of 10 runs for each batch size.

7.2.1 Filter. **Configuration:** Given that the time to decrypt a field depends on the size of the field, we evaluate the processing time for different sized fields. We further evaluate enclave batching by including five different batch sizes (1, 1K, 10K, 100K, 1M) and their overheads. We compare Sanctuary to a plaintext system as well as one that uses order-preserving encryption.

Results: Sanctuary filters will incur roughly 2x-4x overall execution time versus plaintext approaches (but leak less information), and 1.5x-3x overall execution time versus computation-enabling encryption. Note that most of the overhead is due to context switching between trusted and untrusted code in the CPU. To this end, using larger batch sizes reduces this overhead, and most execution times are under 10ms overall. Moreover, Sanctuary operations are similarly impacted by the size of the underlying ciphertext.

7.2.2 In-Memory Aggregation. Here, we evaluate Sanctuary summation operations against a HOM computation-enabling cryptographic operation. Since a HOM summation is done through multiplication, processing on encrypted data requires greater overhead.

Configuration: We use a summation operation to evaluate an in-memory aggregation within the enclave. For simplicity, we assume no window semantics (meaning the final aggregation is simply inclusive of all tuples) to better understand the underlying operation. We will use a windowed query in Section 7.4 to evaluate performance on an actual streaming query. Again, we include five batching sizes and compare to a plaintext non-enclave summation, and we use a HOM computation-enabled encryption scheme.

Results (Figure 8): Batching for an in-memory aggregation algorithm has a similar impact as the filter operation. However, when using the Paillier homomorphic encryption scheme, computing a sum requires multiplication which comes at a greater cost. This is evident from the HOM line in Figure 8 being the most costly line. Even in the case where there is a batch size of one, the enclave-enabled in-memory aggregation operation can outperform its encrypted counterpart. Since both the HOM-encrypted processing and the Sanctuary enabled operation provide an adversary with the same level of information, Sanctuary becomes a desirable choice when considered in conjunction with the performance advantages.

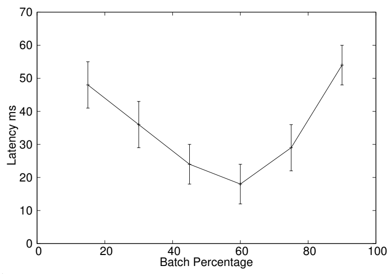
7.3 Micro Benchmark: Stateful Operations

This section explores the overheads required for the security advantages of the memory-limited algorithms presented in Section 5. Recall from Section 5 that there are five factors that can influence the overhead of a memory-limited operation; 1.) Batch Size, 2.) Enclave Memory Structure Size 3.) Operator State Size, 4.) Window Size, and 5.) Update Cost. We reduce all arbitrary units of size measurement (e.g., tuple, enclave memory) to megabytes for simplicity when making comparisons. We evaluate each of the four algorithms presented in Section 5 based on the tradeoffs below:

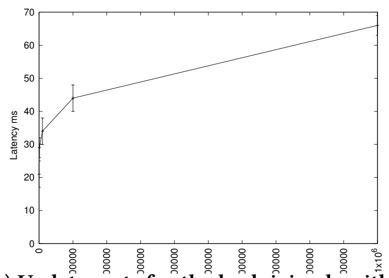
- ***Batch Size vs. Enclave Memory Size***: Given the limited memory of an enclave, there exists an inherent tradeoff between the batch size of incoming tuples and the enclave memory available for bringing operator state into the enclave for processing.
- ***Operator State Size vs. Enclave Memory Size***: Stateful operations require a comparison or computation over some amount of internal state. Given the limitation on the internal enclave memory available for operator state, there is a tradeoff that can affect the execution time (or the latency) of the operation.
- ***Window Size vs. Update Cost***: When a tuple expires or is introduced to the state of the operation, there is an associated update cost. For larger windows (either by definition or through high-bandwidth streams), these updates can come with a greater overhead, affecting the overall performance of an operation.

Each experiment depicts the average of seven runs with the lowest and highest removed to ensure that background tasks are weighted similarly as the CPU is shared by other processes.

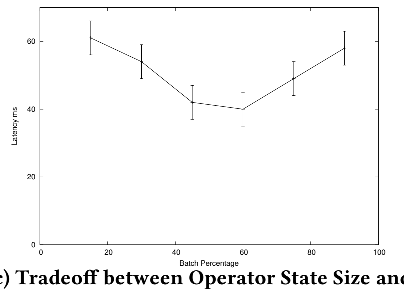
7.3.1 Symmetric Hash Join. **Configuration:** Each tuple contains two fields: a *comparator* (8 Bytes) and a *payload* (92 Bytes). Joined



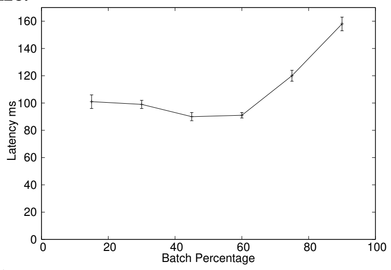
(a) Hash Join batch size vs enclave memory size.



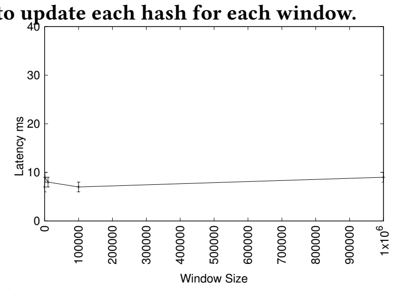
(b) Update costs for the hash join algorithm to update each hash for each window.



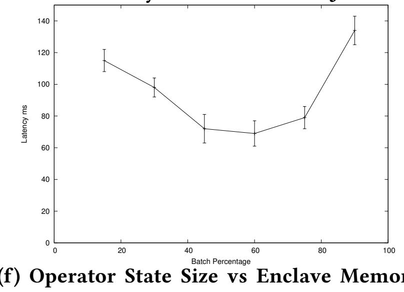
(c) Tradeoff between Operator State Size and Enclave Memory Size for the Hash Join.



(d) Aggregation batch size vs enclave memory size sows the impact of batching on aggregation operations.



(e) Aggregation operations require very little for updates by nature, as shown by this window size versus update cost evaluation.



(f) Operator State Size vs Enclave Memory Size for the General Aggregation.

Figure 7: Results for the the common evaluation for the three stateful operations from Section 5

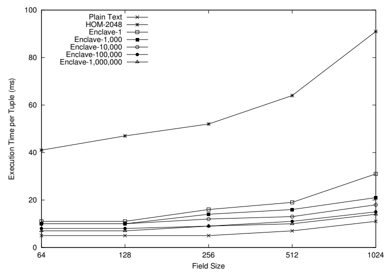


Figure 8: Execution time for in memory aggregation operation for non-enclave and enclave-enabled operations.

tuples are combined into 184 Byte payloads. Comparator fields are uniformly selected from a range of integers from 1-256. After metadata, the internal enclave memory is roughly 122 MB, of which all is available for buffers and state comparisons.

Results (Figure 7a) Batch Size vs Enclave Memory Size: For this experiment, we evaluate the batch size versus the available enclave memory. Specifically, we hold the window size constant at 100 MB with an input rate of 10 MB/s. We reserve 30% of the enclave memory to be given to the internal hash for tracking non-enclave memory. We see the tradeoff between batching and freed enclave memory maximize at roughly 60%.

Results (Figure 7b) Window Size vs. Update Cost: For this experiment, we fix the batching to enclave memory ratio at 50% each. We adjust the window size from 10MB to 100 MB and hold an input of 10 MB/s. We again allocate 30% of enclave memory to the internal hash. Recall that an update to the external structure to expire a tuple will only occur when that value is joined (Lines 18- 19 in Algorithm 2). Note here that each data tuple must be hashed, brought into memory, compared, then evicted, which all adds to a higher latency compared to a plaintext system.

Results (Figure 7c) Operator State Size vs. Enclave Memory Size: Here, we fix the batch size at 30% of enclave memory. We then tradeoff internal state size with free memory size. We hold the input rate at 10 MB/s as usual with a window size of 100 MB. This tradeoff exhibits similar behavior to all of the others. Giving more memory to the hash for each stream yields the best results with a 60% ratio. It is important to note, however, that for a join that has a very low selectivity (i.e., one in which the two streams rarely join), the enclave memory will not be filled since there is insufficient data with which it can be filled. This means an enclave will only employ external memory when the join is highly selective.

7.3.2 General Windowed Aggregation. We evaluate the general windowed aggregate operation using the four costs from the previous section and the update cost. We also evaluate the impact on an operation's overhead based on the size and number of slides.

Configuration: We use the same data as in Section 7.3.1, but focus on a portion of the payload for aggregation operations (16 B integers) that is symmetrically encrypted. The 16 B integers are uniformly generated from 1-30,000 and used in a summation aggregation operation across 250 different groups. We use a window that can be divided into 10 slides for all experiments.

Results (Figure 7d) Batch Size vs Enclave Memory Size: This experiment is set up exactly like the SHJ Batch Size vs Enclave Memory Size experiment, with 30% allocated to a possible internal hash structured. Note that we no longer see a profound benefit from greater batching (gaining only 10ms), and we see a far greater increase in latency after about 40% to 60% batch allocation due to the consistent size of the hash structures storing each slide. No matter what we batch, every tuple must be aggregated to the same segmentation size of non-enclave memory, meaning that the relative effects of batching are reduced, since the cost of bringing non-enclave memory into the enclave is normalized.

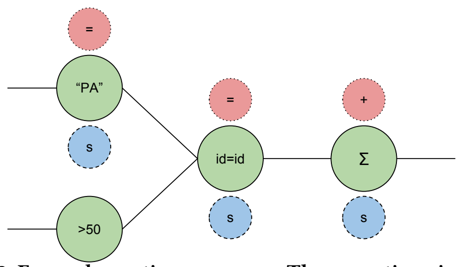


Figure 9: Example continuous query. The operations in the red dotted circles execute on computation-enabling encrypted tuples ("=" for deterministic, "+" for homomorphic), the blue dashed circles represent non memory-limited enclave-enabled operations, and the green solid circles represent plaintext operations.

Results (Figure 7e) Window Size vs. Update Cost: This experiment is similarly set up the same as SHJ Window Size vs. Update Cost, but with slides dividing the window size by 1 t Larger windows do not cause significant increases in update cost as one might expect with larger slides due to the consistent slide size for each hash brought into memory. More specifically, this consistency can be attributed to the uniform group size of 250. If we remove this uniformity and increase the number of groups (not included as a figure), we see that the greater the number of groups, the larger the update cost, especially with a greatly segmented hash table.

Results (Figure 7f) Operator State Size vs. Enclave Memory Size: This experiment is configured exactly like the SHJ Operator State Size vs. Enclave Memory Size experiment with the internal operator structure representing a cache of some hash tables. Similar to what we saw in the Batch Size vs Enclave Memory vs. Operator State Size evaluation for this aggregation, giving more memory to caching is advantageous up to about 60%.

7.4 Macro Benchmark

Here, we chose a query to execute on streaming data with varying conditions. We break the query evaluation into two different environments. The first environment illustrates the effectiveness of an enclave-enabled operation when all of the state fits into enclave memory. The second uses operations where state does not fit in memory. For each query, we compare to 1.) a plaintext version (i.e., no enclave) and 2.) a version where operators are compared with a computation-enabling encryption version. We further test by altering the input rate, selectivity, and size of the data tuples.

7.4.1 Non Memory-Limited Query. The query we use to evaluate the operations that fit entirely in enclave memory is intended to test each of the operation types (i.e., join, aggregation, and a stateless operation) on a real system. The query (below) aims to get the total profit for all companies whose profit margin is more than $500. The query is presented below in SQL and graphically in Figure 9:

```
SELECT SUM(t.sales) FROM t JOIN o
WHERE t.id = o.id AND t.state = "PA"
  AND o.profitMargin > 500
GROUP BY t.company EVERY 30s UPDATE 5s
```

There are two filter operations (the "=" in Figure 9), a join (the "id=id" in Figure 9), and an aggregate-group by operation (the "+" in Figure 9). The encrypted versions of the operations are presented in dotted red circles in Figure 9.

Results: In this experiment, we wish to evaluate the overhead of introducing Sanctuary and SGX-enabled operators into the normal query processing pipeline. When evaluating a streaming query, one must consider the effects of input rate, selectivity, and tuple size on the responsiveness of the system. In this section, we explore changes in input rate and selectivity since tuple size was explored through micro-benchmarking. For each result, we evaluate a completely plaintext set of operations, a set of operations using only computation-enabling encryption, and then queries where each of the operator types (the summation, the join, and a filter) are all placed on an enclave using one of the algorithms from Section 5.

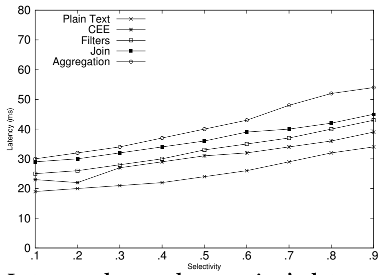
Selectivity (Figure 10a): Here, we change the overall selectivity and inspect the result. To change the selectivity, we simply increase the selectivities of each of the filters and the join incrementally to reach the desired selectivity (.1-.9) while maintaining an input rate of 1,000 tuples/s. As you can see from Figure 10a, a decrease in selectivity generally causes an increase in individual tuple latency, regardless of the operator type. Note that latency here includes the window time for each tuple. An increase in latency signifies that operators have difficulty processing data within a given window. Given that all of the operator state fits into memory, the effects of increasing selectivity was equally beneficial for all operators.

Input Rate (Figure 10b): We evaluate the input rate by increasing from 500 to 2,500 tuples/s (kept sufficiently slow so that we can evaluate scenarios wherein the input rate does not cause the memory capacity to be exceeded) and inspecting the latency. Similarly to the selectivity result, we notice from Figure 10b no significant difference between each operator type with increased throughput. We also notice that latency was within 15% of plaintext, and 7% of computation-enabling cryptosystems for SGX-enabled operators.

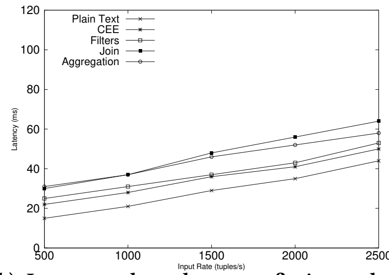
7.4.2 Memory-Limited Query. For our memory-limited query, we simply re-use the query above, but with a drastically increased workload to force the utilization of non-enclave memory.

Selectivity (Figure 10c): For this experiment, we increase the input rate to 20,000 tuples/s (where only roughly 10,000 tuples can fit into memory. We otherwise keep the same configuration as the non-memory-limited version above. Notice from Figure 10c that selectivity has a greater impact on latency in this scenario. Specifically, when more results are filtered, less state is kept, and therefore SGX operators benefit (especially the join operation) since less state is needed to compute the final result, with fewer iterations to non-enclave memory. Specifically, a join performs up to 2.5x faster on data with very low selectivity versus very high, and an aggregation can perform up to 2x faster.

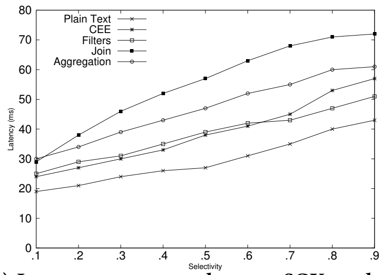
Input Rate (Figure 11): We generate input rates from 10,000 to 50,000 tuples/s to explore the impact on throughput of SGX-enabled operations. We can immediately see from Figure 11 that higher input rates negatively affect latency across the board, but more noticeably for SGX operations. This is expected since it will increase the state being stored in non-enclave memory, and therefore increase the overall processing time per tuple. In some instances we see the join operation increasing latency as much as 78% for large input rates, and as much as 31% for aggregations versus plaintext, and 58% for joins and 2% for aggregations versus a computation-enabling encryption operation. Note that this increase in throughput allows a user to have near-minimal leakage when compared to the encrypted version, and also allows for third-party joins to be executed.



(a) Latency when each operation's data can fit in enclave memory; changing selectivity.



(b) Latency when data can fit in enclave memory; with changing input rate.



(c) Latency as executed on an SGX enclave with changing selectivity.

Figure 10: Changes in latency as changing input selectivity and input rate.

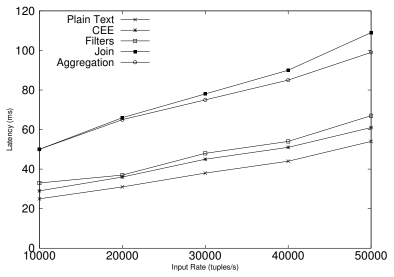


Figure 11: Latency of each operation type as executed on an SGX enclave with changing input rate.

8 CONCLUSION

The current state-of-the-art access control enforcement systems for Data Stream Management Systems either rely on trusted third-party systems to enforce controls, or use some form of computation-enabling encryption that limits query expressiveness, increases data transmission overheads, and can leak information about underlying plaintext values. In this paper, we introduce Sanctuary to implement and evaluate and a method for using Intel's SGX as a trusted computing base for executing streaming operations on untrusted cloud providers. In doing so, we are able to overcome the limitations of the state-of-the-art computation-enabling systems by allowing for *any* query to be executed on an untrusted machine while maintaining *near-Trusted Infrastructure* level information leakage. Moreover, we discuss and resolve issues related to enclave memory size limitations by introducing memory-aware, stateful streaming operators. Finally, we demonstrate that the use of enclave-based processing in a streaming environment incurs only modest overheads when compared to the state-of-the-art systems.

Acknowledgements. This work was supported in part by the National Science Foundation (CNS−1253204 and CNS−1704139).

REFERENCES

[1] Daniel Abadi et al. 2003. Aurora: a new model and architecture for data stream management. *VLDB* 12, 2 (2003), 120–139.
[2] D.J. Abadi et al. 2005. The design of the borealis stream processing engine. In *CIDR*.
[3] Dinh Tien Tuan Anh and Anwitaman Datta. 2014. Streamforce: outsourcing access control enforcement for stream data to the clouds. In *ACM CODASPY*.
[4] Arvind Arasu et al. 2004. Stream: The stanford data stream management system. *Book chapter* (2004).
[5] Arvind Arasu et al. 2006. The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal* 15, 2 (2006), 121–142.
[6] Sergei Arnautov et al. 2016. SCONE: Secure linux containers with Intel SGX. In *12th USENIX OSDI.*
[7] Nathan Backman, Rodrigo Fonseca, and Uğur Çetintemel. 2012. Managing parallelism for stream processing in the cloud. In *HOTCDP*. ACM, 1–5.
[8] Alexandra Boldyreva et al. 2009. Order-preserving symmetric encryption. In *Eurocrypt.* Springer, 224–241.
[9] Stefan Brenner et al. 2016. SecureKeeper: Confidential ZooKeeper using Intel SGX. In *Middleware.*
[10] Paris Carbone et al. 2015. Apache flink: Stream and batch processing in a single engine. *Data Engineering* (2015), 28.
[11] Barbara Carminati et al. 2007. Enforcing access control over data streams. In *ACM SACMAT*. 21–30.
[12] Barbara Carminati et al. 2007. Specifying access control policies on data streams. In *DASFAA*. Springer, 410–421.
[13] Barbara Carminati et al. 2010. A framework to enforce access control over data streams. *ACM TISSEC* 13, 3 (2010), 28.
[14] Debs Grand Challenge. 2014. DEBS Grand Challenge. http://dl.acm.org/citation.cfm?id=2772598. (2014).
[15] Andreas Chatzistergiou and Stratis D Viglas. 2014. Fast heuristics for near-optimal task allocation in data stream processing over clusters. In *CIKM*. ACM.
[16] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *IACR Cryptology ePrint Archive* 2016 (2016), 86.
[17] Aurélien Havet et al. 2017. SecureStreams: A Reactive Middleware Framework for Secure Data Stream Processing. In *DEBS*. ACM, 124–133.
[18] Matthew Hoekstra et al. 2013. Using innovative instructions to create trustworthy software solutions.. In *HASP@ ISCA*. 11.
[19] Yuanqiang Huang et al. 2011. Operator placement with QoS constraints for distributed stream processing. In *CNSM*. IEEE, 1–7.
[20] Paul Kocher, Daniel Genkin, et al. 2018. Spectre attacks: Exploiting speculative execution. *arXiv preprint arXiv:1801.01203* (2018).
[21] Sanjeev Kulkarni et al. 2015. Twitter Heron: Stream Processing at Scale. In *SIGMOD*. ACM, 239–250.
[22] Wolfgang Lindner and Jörg Meier. 2006. Securing the borealis data stream engine. In *IEEE IDEAS*. 137–147.
[23] Rima Nehme et al. 2008. A security punctuation framework for enforcing access control on streaming data. In *ICDE*. 406–415.
[24] Rimma V Nehme et al. 2013. FENCE: Continuous access control enforcement in dynamic data stream environments. In *ACM CODASPY*. 243–254.
[25] Wee Siong Ng et al. 2012. Privacy preservation in streaming data collection. In *ICPADS*. 810–815.
[26] Pascal Paillier. 1999. Public Key Cryptosystems Based on Composite Degree Residuosity Classes. *Advances in Cryptography - EURPCRYPT'99* 1562 (1999).
[27] Peter Pietzuch et al. 2006. Network-aware operator placement for stream-processing systems. In *ICDE*. IEEE, 49–49.
[28] Raluca Popa et al. 2011. Cryptdb: protecting confidentiality with encrypted query processing. In *ACM SOSP*. 85–100.
[29] Christian Priebe, Kapil Vaswani, and Manuel Costa. 2018. EnclaveDB: A Secure Database using SGX. In *EnclaveDB: A Secure Database using SGX*. IEEE, 0.
[30] Stamatia Rizou et al. 2010. Solving the multi-operator placement problem in large-scale operator networks. In *ICCCN*. IEEE, 1–6.
[31] Felix Schuster, Manuel Costa, et al. 2015. VC3: Trustworthy data analytics in the cloud using SGX. In *SP*. IEEE, 38–54.
[32] Fahad Shaon, Murat Kantarcioglu, et al. 2017. SGX-BigMatrix: A Practical Encrypted Data Analytic Framework With Trusted Processors. In *SIGSAC*. ACM, 1211–1228.
[33] Utkarsh Srivastava, Kamesh Munagala, and Jennifer Widom. 2005. Operator placement for in-network stream query processing. In *SIGMOD*. ACM, 250–258.
[34] StormProject. 2014. Storm: Distributed and Fault-Tolerant Realtime Computation. http://storm.incubator.apache.org/documentation/Home.html. (2014).
[35] Cory Thoma et al. 2016. PolyStream: Cryptographically Enforced Access Controls for Outsourced Data Stream Processing. In *SACMAT*, Vol. 21. 12.
[36] Jo Van Bulck, Marina Minkin, et al. 2018. Foreshadow: Extracting the Keys to the Intel {SGX} Kingdom with Transient Out-of-Order Execution. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 991–1008.
[37] Wenting Zheng, Ankur Dave, Jethro G Beekman, et al. 2017. Opaque: An Oblivious and Encrypted Distributed Analytics Platform.. In *NSDI*. 283–298.