UCLouvain

epl

École polytechnique de Louvain

Secure multi-tenant stream processing

Author: **Martin DEVOLDER**
Supervisor: **Etienne RIVIÈRE**
Readers: **Matthieu BETTINGER, Ramin SADRE**
Academic year 2025–2026
Master [120] in Cybersecurity

Acknowledgements

A special thank you to Professor Etienne Rivière for his excellent supervision since the beginning of the year. Despite his demanding schedule, he has always made himself available to guide me and offer advice, which has been incredibly helpful in carrying out this thesis.

I would also like to thank my two reviewers, Matthieu Bettinger and Ramin Sadre, for taking the time to review my work.

Martin Devolder

LLMs usage

This thesis utilized a Large Language Model (LLM), namely ChatGPT, for the following purposes: improving text quality, inventing a fictional scenario with imaginary tenants, writing documentation for code artifacts, creating examples of Intel SGX enclave creation in C and in C++, and generating Python scripts to create graphs from result files.

However, LLMs were not used for the following purposes: searching for sources, creating schemas, getting ideas, or interpreting results.

It should be noted that two additional tools were used: DeepL for translation from French to English, and Grammarly for grammar and spelling checks.

The content and interpretations remain the responsibility of the author.

Abstract

Modern stream processing engines, such as Apache Flink, enable the processing of large volumes of data in real time within distributed environments. However, in many collaborative scenarios involving multiple organizations, these streams contain sensitive data, and processing them on untrusted cloud infrastructure raises major privacy concerns. Traditional approaches that rely solely on encryption protect data at rest and in transit but do not preserve its confidentiality whilst processing is underway.

This thesis addresses the problem of collaborative processing of sensitive streams belonging to multiple tenants in an untrusted distributed environment. To address this issue, we propose a hybrid architecture combining a distributed stream processing engine with Intel Software Guard Extensions (SGX) secure enclaves. The aim of this approach is to retain the advantages of modern stream processing engines for the distributed orchestration of streams, whilst delegating operations handling sensitive data to secure components running within enclaves.

Our architecture is based on a clear separation between the distributed infrastructure and confidential processing. Apache Flink continues to handle the orchestration, routing, and parallelization of data streams, whilst sensitive operations, e.g., joins between streams from distinct tenants, are executed exclusively within SGX enclaves after the data has been decrypted in it. We also offer an architecture that separates the responsibilities for processing orchestration from those for processing sensitive data, as well as a configurable operator that enables the dynamic integration of secure collaborative stream processing.

A working prototype based on Apache Flink and Intel SGX has been developed to experimentally evaluate this architecture. The results show that it is possible to integrate confidential collaborative processing into a distributed stream processing pipeline without running the entire engine within an enclave. However, the evaluation highlights the costs of secure execution, particularly latency, sustained throughput, and state management in complex collaborative processing. Despite these limitations, the results demonstrate the practical feasibility of a hybrid architecture enabling the confidential processing of multi-tenant streams whilst maintaining compatibility with existing distributed infrastructures.

Contents

Chapter 1

Introduction

The explosion of event-driven distributed systems and the widespread adoption of cloud infrastructure have profoundly transformed the way organizations collect and utilize their data. In many fields, data is now generated as continuous streams that must be processed in real time to enable event detection, the monitoring of critical infrastructure, and rapid operational decision-making. Modern stream processing engines thus enable the construction of distributed pipelines that continuously ingest and process large volumes of data from multiple sources.

In many scenarios, the true value of this processing becomes apparent when multiple organizations can collaborate and cross-reference their respective data. This collaboration becomes particularly valuable in fields such as healthcare, transportation, energy, or critical infrastructure, where information held by different entities can generate significant value when correlated in real time.

However, these collaborations also raise major privacy and security concerns. The data being processed may contain sensitive information, such as medical, location, behavioral data, or any *Personally Identifiable Information (PII)*[1]. In the European context, regulations such as the *General Data Protection Regulation (GDPR)* [1] impose strict requirements regarding the processing, sharing, and protection of such data. More generally, recent regulations on critical infrastructure, such as the *Network and Information Security Directive (NIS2)* [2], also strengthen security requirements for systems that handle sensitive or critical data.

These constraints make it difficult to use shared cloud infrastructure directly for the collaborative processing of sensitive data. Traditional stream processing engines generally require full access to the data during processing, making it visible to the execution infrastructure, the cloud operator, or the various distributed components of the pipeline. In a multi-tenant context involving several independent organizations, this trust model often becomes unacceptable.

[1]Data that directly or indirectly identifies an individual.

Consider, for example, a collaboration between a hospital network and a public transit operator. The hospitals collect real-time data on medical admissions, observed symptoms, and the geographic areas involved. Meanwhile, the transit operator has information on the stations with the highest ridership, the routes used, and passenger density across the urban network.

The Table 1.1 shows the main passenger flow data provided by the transport operator. Each event describes the status of a station at a given time, including the numbers of arriving and departing passengers and an estimated occupancy rate. This flow thus provides a real-time overview of station usage and local mobility density across the network.

Table 1.1: Passenger occupancy stream

Attribute	Description
event_id	Unique event identifier
timestamp	Event timestamp
source_ts_ms	Source timestamp (ms)
station_id	Station identifier
count_in	Number of entering passengers
count_out	Number of exiting passengers
occupancy_pct	Station occupancy percentage
ingestion_ts	Ingestion timestamp

The Table 1.2 shows the main flow of admissions generated by the hospital network. Each event corresponds to a pseudonymized hospital admission and contains temporal, geographical, and medical information, such as the hospital concerned, the nearest station, the symptom category, the severity level, and the diagnostic outcome.

Table 1.2: Admissions stream

Attribute	Description
event_id	Admission event identifier
patient_id	Pseudonymized patient identifier
timestamp	Admission timestamp
source_ts_ms	Source timestamp (ms)
hospital_id	Hospital identifier
nearest_station_id	Nearest station
zip_code	Patient area code
symptom_code	Symptom category
severity	Severity level
test_result	Diagnostic result
ingestion_ts	Ingestion timestamp

Correlating these two data streams could enable rapid identification of local epidemiological trends or areas with high concentrations of respiratory cases. Such a collaboration could, in particular, rely on a join query between the data streams produced by the two organizations:

```
SELECT
     admissions.symptom_code AS symptom_code,
     admissions.severity AS severity,
     admissions.source_ts_ms AS admissions_source_ts_ms,
     passengers.source_ts_ms AS passengers_source_ts_ms,
     passengers.station_id AS station_id,
     passengers.occupancy_pct AS occupancy_pct
  FROM admissions
  JOIN passengers
  ON admissions.nearest_station_id = passengers.station_id
```

This collaborative query enables correlation between two sensitive data streams from two separate organizations: hospital admissions and transport network occupancy data. The hospital tenant provides, for each admission, a symptom code, a severity level, and the ID of the nearest station. The transport tenant provides, for each station, a measure of network occupancy. The join thus enables a medical event to be linked to a local mobility context.

The value of this query lies in generating information that neither the hospital nor the transport operator can obtain from their own data. It makes it possible, for example, to identify admissions associated with certain symptoms in areas with high transport density, or to contextualize more serious cases by examining the occupancy level at the relevant station. The value of the analysis, therefore, lies in

the inter-organizational correlation itself.

This thesis focuses on the challenge of enabling collaborative processing of sensitive data streams across multiple tenants within an unreliable distributed infrastructure, as represented in Figure 1.1, while limiting data exposure during processing.

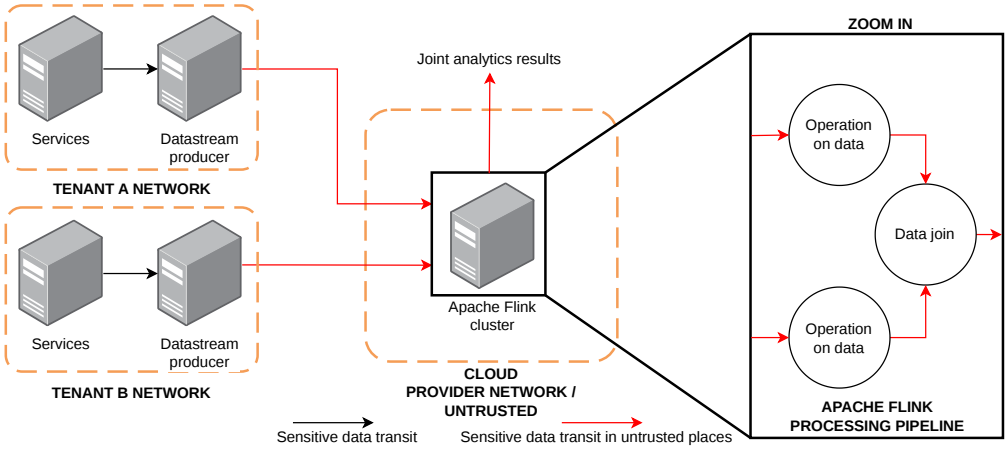

Figure 1.1: Classical architecture for stream processing

To address this problem, we propose a hybrid architecture combining a distributed stream processing engine with *Intel Software Guard Extensions (SGX)* enclaves. The main idea behind our approach is to retain the advantages of modern stream processing engines for distributed stream orchestration, while delegating operations that manipulate sensitive data to secure components running within enclaves.

In our architecture, data is encrypted before being fed into the distributed pipeline. The stream processing engine acts primarily as a communication relay and orchestration component, without access to the content of the business data until the data is no longer confidential. Sensitive collaborative operations, particularly joins between streams belonging to different tenants, are executed exclusively within `Intel SGX` enclaves after the events have been decrypted in it.

Our main contribution is therefore the design and implementation of a confidential multi-tenant stream-processing architecture that clearly separates distributed infrastructure from secure processing, as shown in Figure 1.2. Specifically, we propose:

- A hybrid architecture that separates the stream processing engine from the embedded components.

- A configurable operator that enables the execution of secure collaborative processing.
- An `Intel SGX` enclave responsible for processing sensitive streams and performing join operations.
- A separation between the control plane and the data plane.
- a working prototype based on *Apache Flink*, *Message Queuing Telemetry Transport (MQTT)*, and `Intel SGX`.

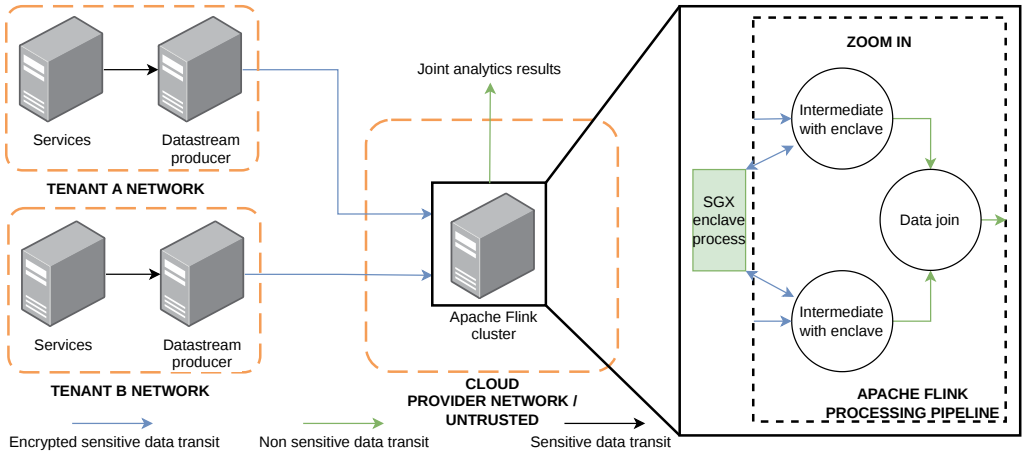

Figure 1.2: Secure multi-tenant stream processing [2]

We then experimentally evaluate this architecture across several processing workflows and join operations between streams from different tenants. The results show that it is possible to integrate confidential collaborative processing into a distributed stream processing pipeline while maintaining an architecture compatible with existing engines. The evaluation also highlights the costs of cryptographic operations and the constraints associated with execution within `Intel SGX` enclaves. The evaluation notes in particular that `Intel SGX`'s impact is most pronounced for stateful collaborative workloads involving joins and stream correlation, where sustained throughput declines more rapidly under load.

However, the prototype presented in this work remains limited to a restricted subset of processing operations and does not constitute a complete *Structured*

[2]The diagram shows an Apache Flink workflow in which two tenants process their data individually using Intel SGX before performing a join, but it could just as well be a join using Intel SGX.

Query Language (SQL) engine, capable of applying any `SQL` operation to given data. Furthermore, certain advanced communication security mechanisms, such as the full integration of *Remote Attestation over Transport Layer Security (RA-TLS)* between tenants and enclaves, have not yet been implemented in our experimental environment. Our primary objective in this thesis is to demonstrate the feasibility of a hybrid architecture enabling the confidential collaborative processing of multi-tenant streams in a distributed environment.

The remainder of this thesis is organized as follows.

Chapter 2 introduces the background and state of the art. It presents the fundamental concepts of stream processing engines, with a particular focus on `Apache Flink`, as well as existing approaches to privacy-preserving collaborative data processing, including cryptographic techniques and *Trusted Execution Environments (TEE)* based on `Intel SGX`.

Chapter 3 defines the problem addressed in this work. It introduces the system model, trust assumptions, threat model, and objectives considered throughout the thesis.

Chapter 4 presents our contributions. It describes the proposed architecture, discusses the main design choices, details the prototype implementation, and illustrates the execution of a secure collaborative workflow.

Chapter 5 evaluates the proposed approach through a series of experiments. It analyzes the impact of `Intel SGX` on latency and throughput across a complete workflow, as well as the practical feasibility of our secure multi-tenant stream processing architecture.

Finally, Chapter 6 concludes the thesis and discusses several directions for future work.

All our development artifacts, including the prototype, benchmark scripts, and other auxiliary tools used for evaluation, are available on a dedicated GitHub repository.[3]

[3]https://github.com/mdevolde/multi_tenant_stream_processing

Chapter 2

Background and state of the art

This chapter presents the key concepts necessary for understanding our work, as well as a review of the state of the art in existing approaches.

We will begin by introducing modern stream processing engines, and more specifically `Apache Flink`, to understand the benefits these distributed architectures offer for real-time data stream processing, as well as the privacy and security issues they raise in shared cloud environments.

Next, we present the main approaches proposed to protect the collaborative processing of sensitive data. We first discuss cryptographic approaches, then approaches based on `Trusted Execution Environments (TEE)`, specifically `Intel SGX`.

2.1 Stream processing engines

To understand the advantages and disadvantages of the solutions discussed below, it is important to know how stream processing engines work. To do this, we will use `Apache Flink` as our example, since it is the stream processing engine our solution integrates with.

2.1.1 Apache Flink

`Apache Flink` is a stream processing engine based on the *Dataflow Programming Model* [3]. In this model, an application is represented as a graph composed of operators, where each node corresponds to a transformation applied to a data stream, and each edge represents the flow of data between these transformations.

Data is processed continuously, as it is generated, unlike traditional batch approaches, where data is collected and then processed. This model is particularly

well-suited to systems that generate data in real time, such as *Internet of Things (IoT)* sensors.

`Apache Flink` provides a set of operations for transforming data streams, including basic transformations such as map and filter, aggregations, joins between streams, and time-window operations. This model enables the creation of complex pipelines by combining multiple operators, each applying a specific transformation to the data it receives, as represented in Figure 2.1.

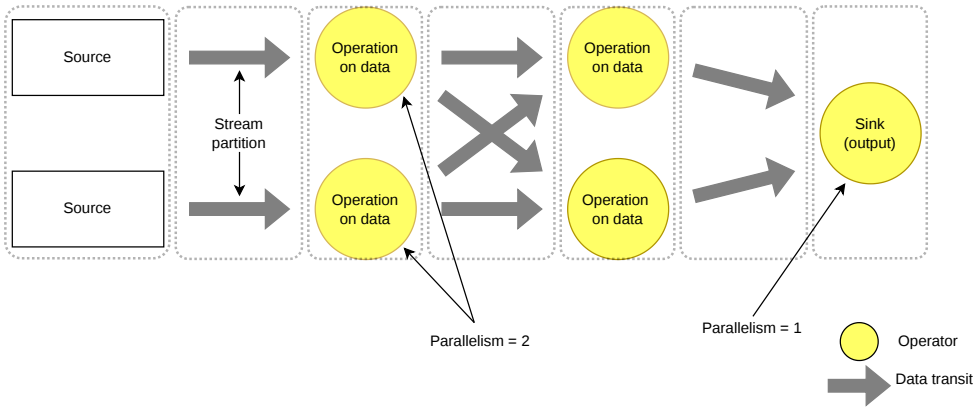

Figure 2.1: Parallel dataflow, operators graph[1]

System architecture

`Apache Flink` consists of two main components. The *JobManager* is responsible for coordinating processes, planning tasks, and managing resources, whilst the *TaskManagers* are responsible for ensuring that operators process data correctly, as represented in Figure 2.2.

When a job is submitted, it is converted into a distributed execution graph. The operators are distributed across different `TaskManagers`, enabling parallel and scalable processing.

Parallelism is one of `Apache Flink`'s key advantages. Each operator can have multiple instances, with each instance processing a portion of the data stream. This makes it easy to utilize resources effectively across a cluster.

This architecture, which involves transferring data between multiple machines, has security implications. When data travels between multiple nodes in the cluster, several privacy risks arise. Sensitive information may be transmitted over untrusted

[1]Inspired by Apache Flink documentation [4]

networks, temporarily stored in memory or on disk on remote machines, or be accessible to administrators who control the infrastructure. Distributing operators across multiple machines also increases the attack surface and makes it more difficult to implement strong privacy guarantees during processing.

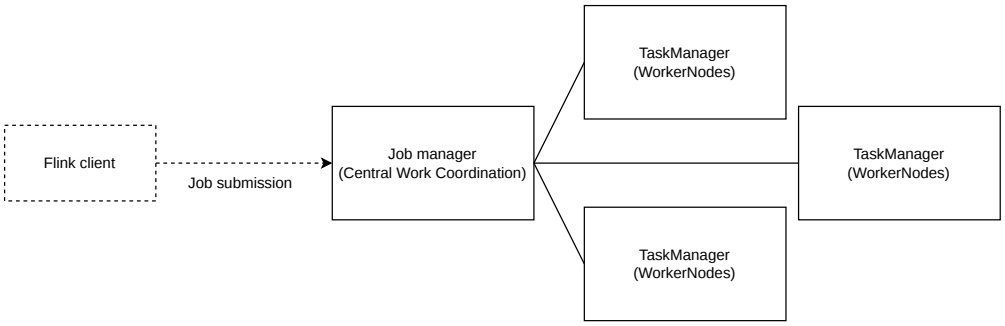

Figure 2.2: Job and Task managers[2]

Time and event management

`Apache Flink` offers several time management models, which are essential for stream processing. Processing time corresponds to the local time of the machine processing the data, whereas event time corresponds to the time at which the event was generated at the source.

This concept is important for event processing. If, for example, we want to process all events that occur within a 1-minute window, this distinction is important.

The event-driven model is the preferred choice, as it allows data streams to be processed correctly even when they arrive out of order or with delays.

`Apache Flink` handles these situations using watermarks[3], which enable it to estimate the elapsed time in the stream and trigger computations for time windows.

Windows (tumbling, sliding, session) allow data to be grouped by time intervals, enabling processing on these groups.

[2]Inspired by Apache Flink documentation [4]

[3]In Apache Flink, a watermark is a timestamp embedded in the data stream, indicating that no events with a timestamp older than this watermark should arrive.

Fault tolerance

Some `Apache Flink` operators are stateful, meaning they can maintain internal state between events, thereby influencing processing.

`Apache Flink` supports checkpointing, which means it periodically saves its internal state and, in the event of a failure, restores the operators' state to the last checkpoint.

This state can be distributed and stored across the cluster's nodes, or even offloaded to an external storage system. Once again, this has security implications, as the data is stored in memory, persisted to disk, and transferred between nodes. Because operator states may contain sensitive data, their persistence or transfer between different nodes in the cluster introduces additional privacy risks. An attacker controlling the infrastructure could potentially access the checkpoints, observe state transfers, or attempt to modify the persisted data.

Programming interfaces

`Apache Flink` provides several programming interfaces for defining processing pipelines. The *DataStream API* is a low-level API for manipulating data streams, whilst the *Table API* provides declarative interfaces for expressing transformations as queries. These interfaces enable users to manipulate data streams and apply complex transformations in a simple and expressive way. This is particularly true of `SQL` queries, which allow joins and aggregations to be expressed naturally.

2.2 Security and privacy-preserving approaches for collaborative stream-processing

Currently, various solutions are available for processing sensitive data in an untrusted environment. These solutions enable data exploitation while ensuring its confidentiality.

In our context, it is also important to have solutions that allow collaboration on sensitive data without revealing its content to the other party.

There are two main approaches:

- Cryptographic approaches can enable calculations on encrypted data.
- Approaches based on `Trusted Execution Environments (TEE)`, which protect the execution context

2.2.1 Cryptographic approaches

The first approach to handling sensitive data in an outsourced environment is cryptographic.

Secure Multi-Party Computation (SMPC)

In our context, several parties need to perform joint calculations and analyses on their data whilst keeping it strictly confidential. The data cannot, therefore, be shared in plain text, and it is difficult to identify a suitable location for analyzing it. Sending the data to one of the parties does not meet confidentiality requirements, nor does analyzing it in the cloud without taking any security measures. Yet we need a result.

Zhao et al. [5] present a survey about *Secure Multi-Party Computation (SMPC)*, a theoretical solution to this problem. The idea of `SMPC` is that each participant keeps their data private. This data is then transformed (e.g., via secret sharing or garbled circuits), and a distributed protocol is established to compute a common function that produces a result without ever revealing any information beyond what can be deduced from the output.

Despite strong technical guarantees, `SMPC` has significant limitations. The protocols are often costly, particularly for complex queries, and impose significant efficiency overhead. These two factors make `SMPC` difficult to scale in real-world applications. This paper clearly illustrates the dilemma between privacy and efficiency.

Privacy-preserving inter-database operations

Another cryptography-based solution, presented by Liang et al. [6], involves performing common operations (joins, intersections) whilst preserving the confidentiality of each party's data.

In this context, sensitive data must not be shared, but it must still be possible to compare or combine it to extract common information.

The paper proposes distributed cryptographic protocols that enable intersection and join operations between two databases, whilst limiting the information disclosed. Each participant learns nothing about the other's data that does not contribute to the result of the intersection or join, except for the sizes of certain data sets.

These protocols rely on hash functions and digital signatures, which reduce computational costs compared to more general approaches such as `SMPC`, and are designed for asymmetric use.

Although experimental evaluation suggests they are performance-feasible, these protocols assume participants are semi-honest. That is, they may try to deduce information but must adhere to the protocol.

Privacy-preserving joins

Li et al. [7] focuses specifically on the problem of secure joins between mutually untrusting entities, in which neither party wishes to reveal its sensitive data. The aim is to enable joins on confidential data so that no information is revealed beyond what can be deduced from the inputs and the final result, assuming other tenants are honest but curious.

The authors highlight the limitations of existing solutions, particularly in certain security definitions used for privacy-preserving joins. They show that these definitions may allow unwanted information leaks, for example, via implicit assumptions or intermediate results. To address this issue, they propose a new, stricter definition in which confidentiality is formalized as the independence of memory access patterns from the data being processed.

The proposed approach relies on a secure coprocessor, which is the only trusted component in the system. This model allows computation to be delegated to an untrusted infrastructure whilst ensuring the confidentiality of data processed within the coprocessor. This solution, therefore, represents a trade-off between approaches that rely on a fully trusted third party and those based on secure multiparty computation, which are often more computationally-intensive.

Three algorithms are proposed, offering different trade-offs between privacy and performance. The first two guarantee complete privacy, making memory accesses independent of the data, whilst the third compromises on privacy to achieve performance gains, accepting a certain probability of data leaks.

Experimental results show that this approach delivers significant performance gains over `SMPC`-based solutions whilst maintaining strong privacy guarantees.

Yannakakis algorithm

Wang et al. [8] examine a scenario in which several entities (in this case, two parties) hold sensitive data and wish to execute complex queries (joins, aggregations) together without compromising their data confidentiality.

One possible solution, as discussed above, is to use secure computation techniques that enable calculations to be performed on data without revealing the data. However, such approaches are often very costly and inefficient for complex queries.

To address this, the authors draw on a classic database algorithm, the *Yannakakis algorithm*, which enables certain join queries to be evaluated efficiently in an unsecured environment. The idea is to adapt this algorithm for use in a secure environment.

To achieve this, it is necessary to protect not only the final data but also the intermediate results and the data-manipulation process to prevent information from being disclosed during the calculation. To achieve this, they used cryptographic

methods (such as secret sharing and garbled circuits) that allow encrypted data to be manipulated and distributed among participants, consequently preventing any single participant from gaining access to the complete data.

The resulting protocol enables a well-defined class of queries to be executed efficiently. Experimental results show excellent performance, since the cost is linear in data size and polynomial in query size.

Discussion

Cryptographic approaches deliver particularly strong confidentiality guarantees. By distributing computations across multiple participants or by processing encrypted data directly, they prevent any single entity from gaining access to the plaintext data. In many models, they also reduce, or even eliminate, the need to rely on a central infrastructure or a trusted third party.

These properties make these approaches particularly attractive in collaborative contexts involving multiple organizations that do not wish to disclose their respective data. They also enable the construction of security models based primarily on mathematical guarantees rather than on trust in a hardware or software infrastructure.

However, these solutions also have several practical limitations. Advanced cryptographic mechanisms, such as secure multiparty computation or certain forms of homomorphic encryption, often introduce significant complexity in the design and implementation of applications. This complexity increases further when processing becomes dynamic or distributed, or when it requires complex operations on the data.

In the context of distributed stream processing, these constraints become particularly apparent. Modern stream processing engines rely on continuous processing, frequent network exchanges, and significant latency constraints. However, certain cryptographic approaches introduce a significant computational overhead that can severely reduce the system's sustainable throughput or complicate integration with existing engines such as `Apache Flink`.

As we have seen in the papers below, solutions with a more reasonable overhead must either be less generic, compromise on the security model, or partially change the trust model by introducing trusted third parties such as coprocessors. This clearly illustrates the dilemma between performance and confidentiality.

Furthermore, integrating these mechanisms into an existing distributed architecture often requires a significant redesign of the processing chain. Operators, data formats, and communication mechanisms sometimes need to be adapted to the cryptographic primitives used, which limits the transparency of the integration.

Thus, cryptographic approaches constitute a family of solutions offering particularly strong security guarantees, but at the cost of integration complexity and

performance issues that can become problematic in certain large-scale or low-latency stream processing scenarios.

2.2.2 Trusted Execution Environments approaches

To overcome certain limitations inherent in cryptographic approaches, another family of solutions relies on secure execution environments, known as `Trusted Execution Environments (TEE)`. Unlike cryptographic approaches, the aim is not to perform computations on encrypted data, but to protect the environment in which sensitive data is analyzed.

The emergence of these solutions stems from the need to address the complexity of cryptography-based solutions, which only increases as the volume and complexity of processed data grow. As we shall see, these `TEE`-based solutions involve a trade-off: placing one's trust in the processor manufacturer.

Intel SGX

`Intel SGX (Software Guard Extensions)` is a hardware extension available on selected Intel Core and Intel Xeon processors that enables the creation of secure execution environments known as enclaves [9]. These enclaves are isolated execution contexts in which code can run apart from the rest of the system.

Unlike traditional security models, `Intel SGX` assumes that the operating system, the hypervisor, and system administrators cannot be trusted. Isolation is guaranteed by the processor itself, so even a compromised system cannot access the contents of an enclave, as represented in Figure 2.3.

The `Intel SGX` model is based on reducing the trusted zone. Only the enclave itself and the hardware are considered trustworthy if they have been verified (and thus, indirectly, the enclave manufacturer), whilst the rest of the chain is regarded as potentially malicious.

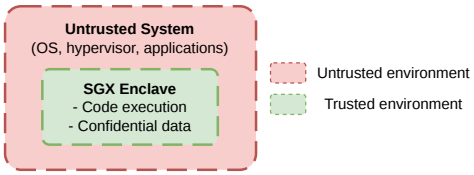

Figure 2.3: Intel SGX principle

Memory model and isolation `Intel SGX` enclaves store their code and data in a dedicated memory area known as the *Enclave Page Cache (EPC)*. This memory is protected by a hardware encryption mechanism known as the *Memory Encryption Engine (MEE)*, which encrypts enclave pages as they leave the processor package and verifies their integrity upon restoration.

`EPC` is a protected region of the *Processor Reserved Memory (PRM)*, a memory region reserved by `Intel SGX` inside the *Dynamic Random Access Memory (DRAM)*. Data in it is decrypted only within the *Central Processing Unit (CPU)* while in use. This ensures that even an attacker with access to the physical memory cannot read or edit the data within the enclave. These processes are represented in the Figure 2.4.

Access to the enclave's memory is subject to strict controls. Only code executed by the enclave can access its data, and any attempt to access enclave memory from outside the enclave is blocked by the hardware, thanks to the *Enclave Page Cache Map (EPCM)*, where metadata about `EPC` is stored, including the enclave that owns `entries`. Although the operating system remains responsible for memory management, including allocation and paging, it cannot read the contents of enclave memory.

If the `EPC` memory becomes full, `Intel SGX` uses a secure paging mechanism. Pages from the enclave can be moved to unprotected memory, where they remain encrypted, and their integrity is maintained. When they are restored to the `EPC`, their integrity and freshness are verified by the hardware, ensuring they have not been altered.

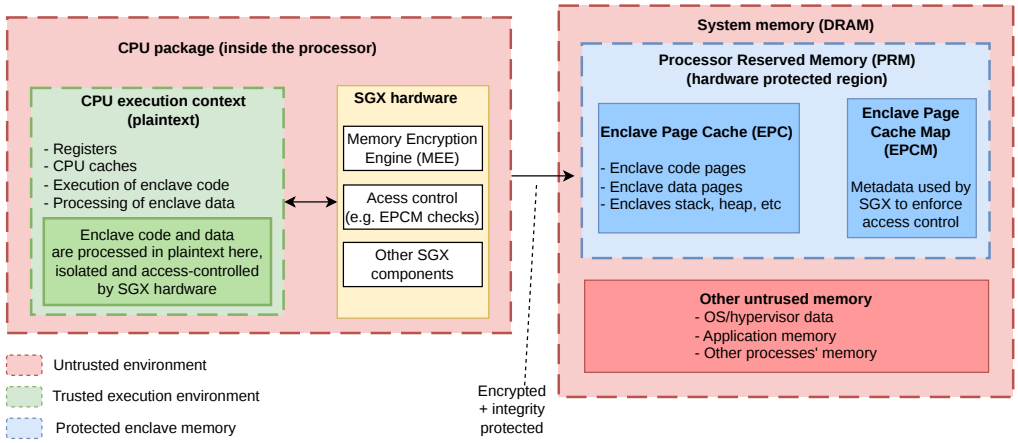

Figure 2.4: Intel SGX Architecture

Enclave lifecycle The lifecycle of an enclave [10] is strictly controlled by the processor via a set of specific instructions. *ECREATE* initializes an empty enclave, *EADD* adds memory pages containing code and data, *EEXTEND* updates the enclave's cryptographic measure, and *EINIT* finalizes the enclave and makes it executable. Once initialized, the enclave can be entered through *EENTER* and exited through *EEXIT* to return to the untrusted environment. This entire process ensures that the contents of the enclave are fully defined and verified prior to execution, consequently enabling subsequent verification of its authenticity.

Enclave transitions Unlike a conventional process, an `Intel SGX` enclave cannot interact directly with the operating system or with external hardware resources. Interactions between the untrusted world and the enclave, therefore, take place via explicit transition mechanisms known as *Enclave Call (ECALL)* and *Outside Call (OCALL)*.

An `ECALL` corresponds to a call made from the untrusted application to a function executed within the enclave. This mechanism is used when an application wishes to transmit data to the enclave for confidential processing.

Conversely, an `OCALL` corresponds to a call made from the enclave to the untrusted world. As `Intel SGX` enclaves cannot directly execute certain system operations, such as network access or standard system calls, these operations must be delegated to the host process via `OCALL`s.

These transitions introduce an additional cost associated with context switches between the secure and untrusted worlds, as well as with protected memory copy mechanisms. In applications that rely heavily on `Intel SGX`, these transitions can thus contribute significantly to the overall execution cost.

Enclave identity Each enclave has a cryptographic identity called *MRENCLAVE*, computed by `EEXTEND`, which is a cryptographic hash of the enclave's initial contents (code + data).

Another important property of the enclave is *MRSIGNER*, which identifies the enclave's signer (it is a cryptographic hash of the signer's public key).

These credentials provide several security guarantees. They ensure that the authenticated code is executed, namely the code that was initially loaded. They form the basis of the certification mechanisms discussed in the following sections and enable remote tenants to make informed decisions.

The key point to understand is that `Intel SGX` not only allows you to create a secure enclave, but also to measure the initial code and then verify its authenticity.

Local attestation Local attestation is a mechanism that allows two enclaves running on the same machine to mutually verify each other's identity.

This mechanism relies on generating a report that contains, in particular, the enclave's identity (`MRENCLAVE`).

This report is created by Enclave A and is intended for Enclave B. The processor will derive a REPORT KEY specific to enclave B, which will be used to calculate a *Message Authenticated Code (MAC)* on the REPORT. Enclave B can derive the same key and verify the `MAC`.

The receiving enclave can then verify this report and confirm the identity of the sending enclave. This mechanism is effective for local multi-enclave architectures, but does not apply to remote communications.

Remote attestation Unlike local attestation, remote attestation allows an external entity to verify that an authentic enclave is executing code defined within a legitimate `Intel SGX` enclave.

To perform remote attestation, an enclave generates an object called a quote, which contains its measure (`MRENCLAVE`), information about the platform, and optionally application data. This quote is signed by an Intel-certified hardware key, guaranteeing its authenticity.

The legacy mechanism is based on *Enhanced Privacy ID (EPID)*, a group signature that preserves the platform's anonymity. The token is sent to a centralized service, the *Intel Attestation Service (IAS)*, which verifies the signature's validity and returns the result.

This model has several limitations, including its reliance on an external Intel service and its limited suitability for modern distributed environments. Furthermore, this system has been out of use since `IAS` was discontinued in 2025.

The current mechanism is based on **ECDSA** signatures and a certificate infrastructure, and is called *Data Center Attestation Primitives (DCAP)*. In this model, the quote is signed with a key linked to the platform, a certificate chain is used to verify the signature, and validation can be performed locally without contacting Intel, provided you have the necessary certificates.

This mechanism is better suited to cloud environments, as it enables scalable and autonomous attestation.

Remote Attestation over Transport Layer Security (RA-TLS) Remote attestation can be used in conjunction with *Transport Layer Security (TLS)* to form a mechanism known as *Remote Attestation over Transport Layer Security (RA-TLS)* [11]. This approach intends to establish a standard secure channel whilst ensuring that the remote end corresponds to a legitimate `Intel SGX` enclave.

In the context of our thesis, this mechanism plays a particularly important role, as it allows the various tenants to verify that they are communicating with an authentic `Intel SGX` enclave executing the expected code. This property is

essential for establishing a secure channel between a tenant and a secure enclave.

In this model, the server runs within an enclave and automatically generates a TLS key pair at startup (unique for each startup). To bind this key to the enclave, a hash of the public key is included in the Intel SGX report. This report is then converted into a quote, which provides cryptographic proof that the TLS key belongs to a given Intel SGX enclave.

The attestation elements (quote, attestation report, certificates, etc.) are then incorporated into the server's X.509 certificate as custom extensions. This certificate is self-signed, with Intel SGX serving as the trust anchor in place of a traditional *Public Key Infrastructure (PKI)*.

When establishing the TLS connection, the certificate is transmitted to the client, as in a standard handshake. The client then performs the following verifications:

- Validation of the TLS certificate.
- Validation of the Intel SGX attestation contained in the certificate extensions.
- Validation of the correspondence between the TLS public key and the key from the quote.

If all checks pass, the client can be confident that it is communicating with an authentic Intel SGX enclave, that the code executed by the enclave is as expected, and that the TLS key used indeed belongs to this enclave. Thanks to these mechanisms, a secure channel is established that protects both the communication and the identity of the remote endpoint. The full procedure is detailed in the Figure 2.5.

Vulnerability to side-channel attacks Intel SGX enclaves do not protect processes from side-channel attacks.

Nilsson et al. [12] present a survey of known attacks against Intel SGX, with a particular focus on side-channel attacks. Unlike traditional attacks that seek direct access to protected memory, side-channel attacks exploit indirect effects of CPU execution, such as cache hits, page faults, execution times, branch predictions, and speculative execution. The paper shows that, even though Intel SGX effectively protects enclaves from a malicious operating system or hypervisor, it remains vulnerable to these microarchitectural leaks.

One of the most significant attacks against Intel SGX relies on side channels, particularly cache attacks, demonstrated by Götzfried et al. [13].

Intel SGX enclaves use the processor's caches to access data and instructions efficiently. Although these caches are shared with the rest of the system, Intel SGX does not provide mechanisms to prevent information leaks through these channels.

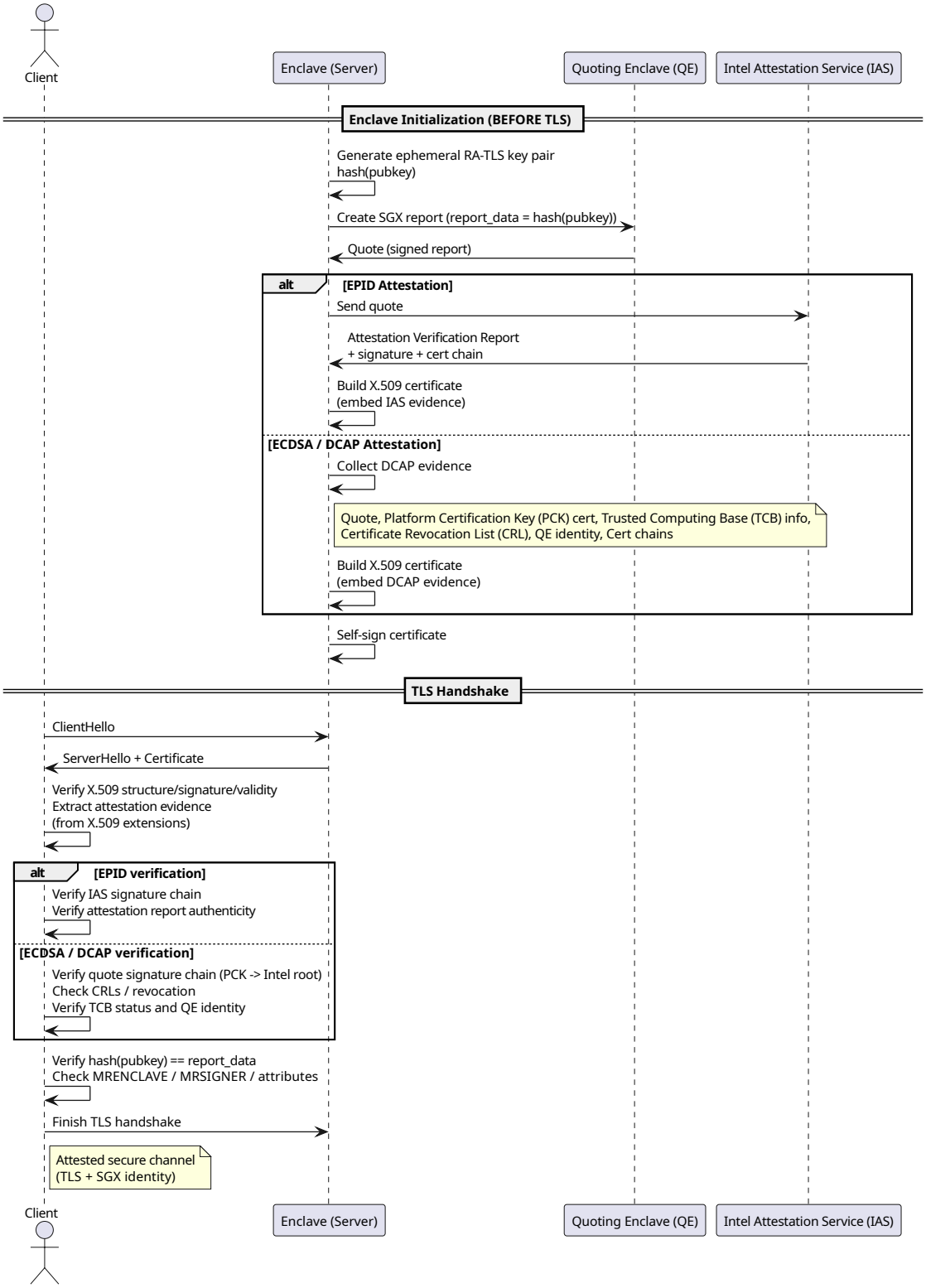

Figure 2.5: RA-TLS workflow

The authors of the paper demonstrate a `cache timing` attack. In this model, an attacker with elevated privileges (root level) deliberately fills certain cache lines with their own data, then allows the enclave to execute. After execution, they measure which cache sets have been modified, using hardware counters among other methods.

These observations reveal which regions of memory were accessed during execution. For cryptographic algorithms such as *Advanced Encryption Standard (AES)* [14], these access patterns depend directly on the secret key in certain vulnerable implementations.

By exploiting this information and applying a method of progressively eliminating candidates, the authors manage to fully reconstruct the `AES` key used within the enclave.

This attack highlights a fundamental limitation of `Intel SGX`: it remains vulnerable to side channels.

Usage of Intel SGX in stream processing

`Intel SGX` has already been implemented in a real-world application for analyzing medical data streams by Segarra et al. [15].

In the medical sector, a series of sensors continuously collect inherently sensitive medical data (e.g., cardiac data). However, they lack the processing power to analyze it. This analysis must therefore be delegated to cloud services. But cloud services are external third parties that may be malicious or compromised with respect to our data's privacy. We therefore need to outsource due to limited resources, but we have concerns about confidentiality given the nature of the data.

The idea was to use `TEE`s (in this case, `Intel SGX`) to process the data within secure enclaves. A complete prototype based on *SGX-Spark* (a modified version of *Apache Spark* to run sensitive code with `Intel SGX`) was therefore developed, specifically adapted for stream processing (by integrating `Intel SGX` into an existing engine), demonstrating that `Intel SGX` provided strong guarantees of confidentiality and integrity whilst maintaining a reasonable overhead.

Secure pipelines with Intel SGX

In the context of stream processing, current engines (`Apache Flink`, `Apache Spark`, etc.) offer few security guarantees, and integrating security mechanisms is difficult because it complicates application development, deployment, and data management. Havet et al. [16] present a solution to enable secure, scalable, and easy-to-program stream processing in untrusted environments.

This paper proposes combining several approaches. It relies on a `Dataflow Programming Model`, which allows applications to be structured as data transfor-

mation pipelines, and uses `Intel SGX` to ensure the confidentiality and integrity of the processed data.

The idea is to enable sensitive data to be encapsulated within `Intel SGX` enclaves (encrypted in transit, sent to the enclaves, and decrypted and processed within them), whilst providing developers with a high-level abstraction. This allows the writing of processing pipelines without explicitly managing cryptographic mechanisms and low-level security (note that attestation and key management are assumed to have already been handled in the paper).

The developed framework, *SecureStreams*, enables the construction and deployment of these pipelines using a Lua-based scripting language and has demonstrated high throughput with manageable overhead, showing that the overhead associated with using `Intel SGX` is real but does not compromise usability in practice.

Data aggregation with Intel SGX

One way to anonymize sensitive data so that individuals can no longer be linked to it is to aggregate the data before analysis. However, in many contexts, this aggregation must be outsourced to an untrusted environment.

Silva et al. [17] present a solution to address this problem. A good example is the processing of data generated by smart metering systems, which continuously collect detailed data on users' energy consumption. This data collection enables optimized energy management, but raises a major concern: a household's consumption data can be used to infer lifestyle habits (presence at home, activity at home, etc.). However, the data is sent to an infrastructure that may not be trustworthy.

This paper, therefore, proposes using `Intel SGX` to securely aggregate this data. The idea is to send this data to the cloud (via a secure channel after authentication), process it within enclaves, and output only aggregated results, without exposing individual data. The solution is generic and simple.

This paper, therefore, presents a comprehensive architecture for secure data aggregation in the cloud using `Intel SGX` and demonstrates that this architecture provides the confidentiality and integrity guarantees associated with `Intel SGX` but incurs considerable overhead, making the solution suitable for scenarios where strict latency requirements are not essential.

Discussion

Approaches based on `Trusted Execution Environments` offer a different strategy from purely cryptographic solutions. Rather than performing calculations on encrypted data, they seek to protect the execution environment itself in order to guarantee data confidentiality during processing.

Technologies such as Intel SGX thus enable sensitive code to be executed in enclaves isolated from the rest of the system. This hardware-based isolation significantly limits data exposure, even in an untrusted environment where the operating system, hypervisor, or infrastructure administrator may not be trustworthy.

One of the main advantages of these approaches lies in their performance. Compared with many advanced cryptographic solutions, TEEs generally incur lower overhead and enable relatively complex processing to be performed at performance levels that may even approach normal, depending on how you use the enclave's resources. They also integrate more easily with existing systems, as processing can often be carried out with minimal changes to the application code.

The attestation mechanisms offered by these environments are also a key factor. They enable remote verification of the code's identity executed within the enclave before any secrets or sensitive data are transmitted to it, thereby strengthening the system's security guarantees.

Nevertheless, these approaches also have several limitations. Unlike cryptographic approaches, they rely on an assumption of trust in the hardware manufacturer and in the TEE's implementation. The trust domain, therefore, includes the processor, the microcode, and part of the associated hardware ecosystem.

Intel SGX enclaves also present significant technical constraints, notably limited protected memory. As memory requirements increase, secure paging mechanisms can significantly degrade performance. These limitations become particularly apparent in complex or highly stateful distributed processing.

Approaches based on Trusted Execution Environments thus strike a balance between security, performance, and ease of integration. They make it possible to build confidential distributed systems that are more practical to use, whilst introducing new trust assumptions and certain hardware limitations.

Chapter 3

Problem statement

As part of this work, it is essential to precisely define the actors involved in the system, their trust relationships, and the security assumptions underlying our approach.

Our goal is to study a model for the collaborative processing of sensitive data streams in an untrusted cloud environment, while maintaining strict control over the data actually disclosed to the various parties.

3.1 System model

Our system is a distributed data stream processing system that enables multiple independent organizations to collaborate on their data in an untrusted cloud environment.

Each organization generates its own data streams, which may be sensitive, and wishes to use this data jointly with other parties without disclosing all the raw information it possesses.

In this context, the system must enable operations on data, such as aggregations, data cleaning, or joins between streams, while limiting the exposure of sensitive data to what is strictly necessary.

3.1.1 Actors and components

The system comprises several categories of participants and components.

Tenants are independent organizations that generate continuous data streams. This data may contain sensitive information, such as medical data, personal identifiers, or location information. Generally speaking, the data contains `Personally Identifiable Information (PII)`. Tenants wish to collaborate on certain joint

analyses while retaining control over the data they agree to disclose through the system.

Data processing is performed using `Apache Flink`, a distributed stream processing engine in which processing is represented as graphs of operators. This engine is deployed on a distributed cloud infrastructure that provides the resources necessary to run the processing pipelines.

To protect sensitive data during processing, certain operations are delegated to `Intel SGX` enclaves. These enclaves execute code that can be remotely attested and are the only components authorized to handle sensitive data in plaintext.

The system may also include several separate enclaves. Some enclaves can be deployed individually by a tenant to perform data transformation or cleaning operations before sharing or joining, or by multiple tenants to join multiple streams.

3.2 Trust model

The participants in the system do not fully trust one another, but they are nevertheless willing to collaborate through the platform to produce certain shared outcomes.

Each tenant wishes to retain control over the data disclosed to other parties. Tenants, therefore, agree that certain derived information may be produced by the system, such as aggregated, cleaned, or joined data, but do not wish to expose their raw data in plain text to other participants or to the cloud infrastructure.

The trust model also depends on the type of operation being performed.

In the case of operations involving a single tenant, such as cleaning or transformation operations prior to sharing, the tenant can deploy its own enclave and trust only the code it deployed by using an attestation mechanism.

In collaborative operations, such as joins between multiple data streams from different organizations, tenants must share a common trust in the code executed within the enclave responsible for the operation. This trust is also based on `Intel SGX` attestation mechanisms, which allow each participant to verify the enclave's identity and the executed code before transmitting data or encryption keys.

The cloud provider is considered untrusted from a confidentiality standpoint. Since it controls the physical machines, the operating system, and the runtime environment, it can monitor communications and stored data, or even attempt to alter the system's behavior.

Since `Apache Flink` runs on this cloud infrastructure, it inherits the same trust model and is also considered untrusted in the absence of additional safeguards.

`Intel SGX` enclaves, on the other hand, constitute the system's primary trust zone. Sensitive data can be decrypted and processed there, and its memory contents

are protected against external access. The enclaves are expected to execute exactly the code attested to and validated by the relevant tenants.

Finally, communication channels are also considered untrusted. The system, therefore, assumes that data may be observed, intercepted, or modified during transmission. Therefore, no intermediary involved in communications is considered trustworthy.

The trust relations are represented in the Figure 3.1.

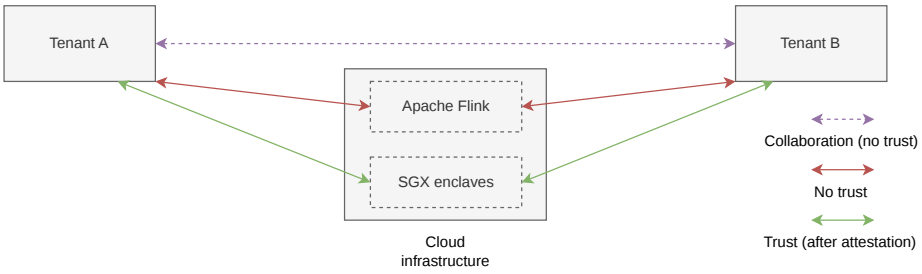

Figure 3.1: Trust relations between system actors

3.3 Actors capabilities

This section describes the capabilities of the various actors in the system and the assumptions made about their behavior.

Tenants are considered cooperative but mutually distrustful participants. They voluntarily use the platform to perform certain joint analyses, but may seek to infer additional information from the results produced, observable metadata, or data they already possess.

They may also decide to share only a portion of their data, or to preprocess it in their own enclaves before participating in a collaborative workflow.

The cloud provider is considered a powerful adversary. It can observe data at rest, in transit, and in storage. It can also modify communications, inject malicious code, or attempt to alter the execution of the processing pipeline. Since it controls the operating system and the hardware infrastructure outside of `Intel SGX`, it has privileged access to the execution environment.

Since `Apache Flink` relies on cloud infrastructure, it is subject to the same trust assumptions and must therefore be considered untrusted for processing sensitive data in plaintext.

`Intel SGX` enclaves are designed to protect the confidentiality and integrity of the code and data they execute. An attacker cannot directly read the enclave's

internal memory or modify the attested code running within it. However, certain external interactions with the enclave remain observable, including network communications.

3.4 Goals

Our solution must meet both functional and security objectives to be usable in a real-world collaborative stream-processing environment.

From a functional standpoint, the system must support stream-processing operations on sensitive data within an existing distributed engine. The various tenants must be able to define transformations, aggregations, or joins without making significant changes to the underlying infrastructure.

The system must also maintain performance levels compatible with the constraints of stream processing. The security mechanisms introduced must not hinder continuous stream processing or impose prohibitive overhead on latency or throughput.

From a security perspective, the system must ensure that sensitive data is never exposed in plain text to the cloud infrastructure or to other tenants, except for information explicitly authorized by the collaborative workflow.

The system must also allow tenants to verify the identity of the enclaves executing sensitive processing, as well as the code actually being executed, before any data or cryptographic secrets are shared.

Finally, the system's objective is not to completely eliminate any possibility of information leakage. Certain information derived from the results produced is intentionally disclosed as part of collaborative operations, particularly during joins or aggregations. The goal is therefore to enable controlled sharing of the information necessary for joint analysis, while minimizing the exposure of raw sensitive data.

In the illustrative scenario used in this study, hospitals and the transportation operator pursue complementary objectives through their collaboration. Hospitals aim to detect local epidemiological trends more quickly by correlating medical admissions with urban mobility data. For its part, the transit operator seeks to identify areas or routes likely to experience an unusual concentration of cases to adapt its operations or real-time monitoring.

However, this collaboration must not lead to uncontrolled disclosure of the data held by each organization. Hospitals do not wish to expose patients' detailed medical information or allow direct access to their internal data flows. Similarly, the transit operator does not wish to reveal all of its ridership or mobility data to external entities. The system must therefore enable collaborative processing that produces only the information explicitly necessary for the joint analysis,

while limiting exposure of raw data to the distributed infrastructure and to other participating tenants. Of course, in addition to these guarantees, the system must deliver acceptable performance to ensure that data is processed correctly in real time.

3.5 Out of scope

For the purposes of this work, vulnerabilities related to side-channel attacks affecting `Intel SGX` are considered to be outside the scope of the study. We therefore assume that the confidentiality and integrity guarantees provided by enclaves are fulfilled.

Chapter 4

Contributions

The main objective of this thesis is to propose an architecture that enables collaborative stream processing operations on sensitive data within an untrusted cloud infrastructure while significantly limiting exposure of plaintext data.

Our approach seeks to address several concurrent constraints. On the one hand, tenants must be able to collaborate to perform certain common operations, such as stream joins or transformations before stream joins. On the other hand, they wish to retain control over the data actually revealed through these processing operations. The system must therefore allow for the execution of collaborative operations without exposing the entirety of the raw data to other participants or to the cloud infrastructure.

A second important objective is to preserve, as much as possible, the properties and execution model of modern stream processing engines. Systems like `Apache Flink` already provide advanced mechanisms for distribution, parallelism, stream management, and fault tolerance. Fully reimplementing these features in a closed environment would be an extremely complex task and would introduce a particularly large `Trusted Computing Base`.

Our goal, therefore, is not to build a new secure stream processing engine from scratch, but rather to propose a hybrid architecture that integrates secure processing into an existing engine with minimal impact on its operation.

This approach also aims for realistic deployment. Solutions that require deep modifications to the distributed engine's runtime, the operating system, or the virtual machine are often difficult to maintain and integrate with existing infrastructure. We therefore seek to minimize the modifications required to the processing engine in order to maintain an architecture that is easily deployable in standard cloud environments.

Another important objective is reducing the attack surface. The more code executed within the enclave, the larger the potential attack surface becomes, and the more difficult the constraints associated with `Trusted Execution Environments`

are to manage. Our approach, therefore, seeks to limit the role of enclaves to processing truly sensitive operations, while leaving orchestration, routing, and flow management mechanisms to the traditional stream processing infrastructure.

Finally, our architecture must remain sufficiently generic to support different types of processing. The system must not be limited to a single workflow but must allow dynamic definition of single-stream operations as well as collaborative operations between multiple streams belonging to different tenants.

These objectives lead to a hybrid architecture in which the stream processing engine retains its role as a distributed orchestrator, while operations that handle sensitive data are delegated to `Intel SGX` enclaves that execute verifiable, isolated code.

4.1 System overview

This section provides an overview of the proposed architecture and the key security properties it aims to ensure. Specific implementation details will be discussed in the following sections.

The proposed architecture is based on a clear separation between the components that orchestrate data streams and those that process sensitive data. This separation is the central principle of our approach.

The system connects multiple tenants that generate continuous, potentially sensitive data streams. These streams are routed to a distributed stream processing engine that coordinates their flow and execution within the application pipeline. However, unlike a traditional pipeline, the processing engine is not responsible for executing operations that manipulate sensitive unencrypted data.

Sensitive operations are delegated to secure components running in `Intel SGX` enclaves. These enclaves constitute the system's primary trusted zone. They are responsible for decrypting data, performing confidential transformations, and producing results authorized by the collaborative workflow.

The architecture also distinguishes between two categories of logical flows: a control plane[1] and a data plane[2].

The control plane is used to initialize secure processing. In particular, it enables enclave configuration, the exchange of information necessary to establish cryptographic sessions, and the distribution of keys used to protect data flows.

The data plane, on the other hand, carries the business events themselves. This data remains encrypted during its transit through the distributed infrastructure

[1]The control plane encompasses the mechanisms responsible for configuring, orchestrating, and coordinating the system.

[2]The data plane corresponds to the data flows and the processing applied to that data during execution.

and is decrypted only when processed within the enclave.

This separation offers several advantages. It isolates sensitive cryptographic management mechanisms from traditional transport mechanisms while significantly limiting exposure of business data within the distributed infrastructure. The stream processing engine thus retains its orchestration role without becoming a trusted component itself.

The architecture supports two main categories of processing.

The first category involves single-stream processing, in which a transformation is applied independently to each event in a given stream, as represented in Figure 4.1. These processes can be used, for example, to clean, filter, or project data before sharing.

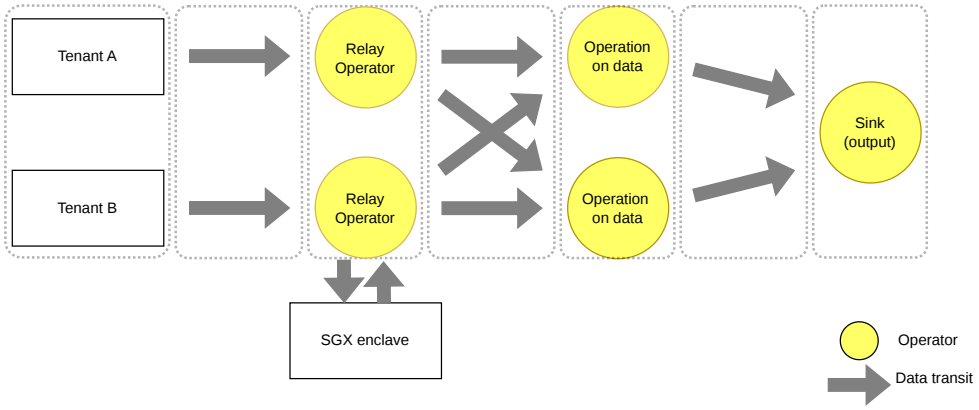

Figure 4.1: Architecture used to securely apply transformations

The second category involves collaborative processing involving multiple streams belonging to different tenants. In this case, multiple streams are jointly processed within a shared enclave to produce derived results, as represented in Figure 4.2, such as joins between events from different organizations.

In this model, tenants do not directly share their data in plaintext with one another. They share only encrypted data with an enclave executing code explicitly validated by the relevant participants. Trust, therefore, rests neither on the cloud provider nor on the other tenants, but on the verified identity of the enclave and the code it executes.

This approach retains the benefits of modern stream processing engines, allowing, for example, the full `Apache Flink` ecosystem to be utilized in the rest of the pipeline. This means that, following an enclaved processing step, the processing pipeline can continue to benefit from `Apache Flink`'s mechanisms and ecosystem

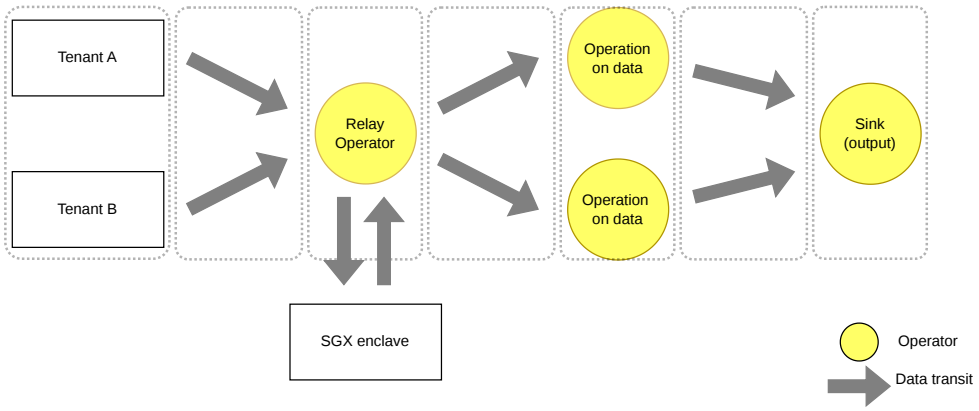

Figure 4.2: Architecture used to securely join two data streams

without complex integration, whilst limiting the number of components that must be trusted to process sensitive data.

In the scenario described in the introduction, the data streams generated by the hospitals and the transportation provider are fed separately into the distributed pipeline. The events remain encrypted while transiting through the `Apache Flink` infrastructure and are decrypted only within the enclave responsible for collaborative processing. A join query can then be executed in the enclave to associate medical events with the mobility information for the relevant station.

4.2 Architectural choices

The design of this architecture is the result of a series of trade-offs between security, performance, integration complexity, and the constraints imposed by `Trusted Execution Environments`. Indeed, several approaches can be considered for integrating `Intel SGX` enclaves into a distributed stream processing engine. Some involve running a significant portion of the engine directly within enclaves, while others, conversely, seek to limit enclaves' role to strictly sensitive operations.

This section, therefore, presents the main architectural choices in our approach, as well as the alternatives considered. In particular, we explain why we chose a hybrid architecture in which the stream processing engine acts primarily as an orchestration and transport component, while processing involving sensitive data is delegated to specialized enclaves.

We also discuss the separation of the control plane and the data plane, a key component of the proposed architecture that limits exposure of sensitive information

and simplifies the management of cryptographic mechanisms.

4.2.1 Why not run Apache Flink inside Intel SGX?

One possible initial approach to securing a stream processing engine is to run all or part of the engine directly within `Intel SGX` enclaves. This approach may seem natural at first glance, since it theoretically allows for the direct protection of both operator execution and the data being processed.

However, several significant limitations make this approach difficult to apply in our context.

First, `Intel SGX` enclaves are not designed to directly execute complex and dynamic runtime environments such as a modern Java virtual machine. As discussed in the previous chapter, an `Intel SGX` enclave must be initialized from a binary that is loaded and verified before execution. This cryptographic verification, used in attestation mechanisms, ensures that the executed code exactly matches the expected code. This property works particularly well for relatively stable, small-sized native components, but becomes much more difficult to maintain in the case of a full-fledged runtime such as the *Java Virtual Machine (JVM)*.

`Apache Flink` is built on a particularly complex Java runtime environment, and directly integrating such a runtime into an enclave would require either enclosing a very large amount of code or making significant changes to the engine and its runtime to ensure compatibility with `Intel SGX` constraints.

This challenge is evident in several state-of-the-art works. For example, `Secure-Streams` [16] proposes executing entire stream processing operators within `Intel SGX` enclaves to protect data during distributed processing. This is a major challenge of this paper, which is addressed by choosing Lua for its light runtime.

Another significant challenge concerns the hardware limitations of `Intel SGX` enclaves themselves. Enclaves have limited protected memory, known as the `Enclave Page Cache (EPC)`. Although paging mechanisms exist, memory accesses outside the `EPC` incur a significant performance cost. This constraint becomes particularly problematic in the context of distributed engines handling large volumes of state, especially for join, windowing, or aggregation operations.

However, modern stream processing engines such as `Apache Flink` rely heavily on potentially large distributed state, network buffers, checkpointing mechanisms, and dynamic memory management, which are difficult to reconcile with the constraints imposed by `Intel SGX`[3].

Furthermore, running a complete engine directly within an enclave would effectively introduce an extremely large `Trusted Computing Base`. The more code

[3]Alternative confidential computing technologies such as Intel TDX [18] or AMD SEV [19] could be considered to embed Apache Flink as they protect a complete virtual machine.

executed within the trusted zone, the larger the potential attack surface becomes and the more difficult it is to accurately assess the security guarantees the system actually provides.

In our context, this approach therefore did not seem suitable for our objective, which is to offer an architecture that can be integrated relatively easily into an existing stream processing environment without significantly altering its runtime.

We therefore chose a different approach: limiting the role of enclaves to operations that handle sensitive data, while keeping the stream processing engine outside the trusted zone.

In this architecture, `Apache Flink` continues to handle the responsibilities for which it was designed, including distributed orchestration, data flow, parallelism, scheduling, and integration into the stream processing ecosystem. The enclaves, meanwhile, are used solely as specialized components to execute confidential processing tasks.

This approach offers several significant advantages.

First, it significantly reduces the `Trusted Computing Base`, since only sensitive business logic is executed within the enclave. The full distributed engine, the Java virtual machine, and the orchestration infrastructure remain outside the trusted zone.

Second, this architecture maintains compatibility with `Apache Flink`'s existing mechanisms without requiring major changes to its runtime. The system, therefore, remains easier to maintain and deploy in standard cloud environments.

Finally, this separation makes it easier to adapt the logic executed in enclaves to the specific constraints of `Trusted Execution Environments`. Sensitive operations can be implemented as relatively simple native components, with bounded state and precise control over the data manipulated in protected memory.

Our goal was therefore not to completely transform `Apache Flink` into an enclave-based engine, but rather to propose a hybrid architecture that integrates secure processing into an existing engine while limiting the modifications required for its operation.

4.2.2 Hybrid architectures

The approach chosen in this work is therefore a hybrid architecture. The central idea is not to move the entire stream processing engine into the trusted zone, but to separate responsibilities between two types of components.

The stream processing engine retains tasks related to pipeline orchestration. It remains responsible for reading streams, routing messages, distributing processing, and integrating with the existing ecosystem. In contrast, operations requiring access to sensitive data in plaintext are delegated to a secure component running within an enclave.

This separation preserves the benefits of a mature distributed engine while limiting the amount of code that must be trusted. The engine does not require extensive modification and does not directly handle sensitive data in plaintext. It becomes an orchestration component surrounding a confidential processing core.

This architecture is therefore based on a compromise: the stream processing engine remains outside the trusted zone, but its role is limited to operations that do not require the confidentiality of business content. The enclave, for its part, receives only the data necessary for sensitive processing, decrypts it, applies the configured operation, and then returns the authorized results to the pipeline.

This model is particularly useful in a multi-tenant environment. Each tenant can encrypt its data before it enters the cloud infrastructure. The data then flows through the pipeline in encrypted form, and only the enclave component can decrypt it until the modified data is no longer sensitive and can be processed in the normal way. The other components of the infrastructure, including the stream processing engine, see only encrypted packets concerning sensitive data.

The hybrid architecture also enables the management of multiple types of processing. For processing involving only a tenant, such as a cleaning or filtering operation prior to sharing, an enclave can be used to transform the data before it is fed back into the collaborative pipeline. For a collaborative operation, such as a join between two streams, the enclave acts as a controlled meeting point between the data of the relevant tenants.

In this second case, the enclave should not be viewed as a traditional trusted third party, but as a verifiable component. Tenants agree to transmit data or secrets to it solely because they can verify the code executed by the enclave using attestation mechanisms. Trust, therefore, rests on a code identity rather than on the cloud infrastructure that hosts the computation.

This architectural choice also allows for a phased adoption. An existing application can keep most of its processing pipeline within the distributed engine and delegate only those operators that require special protection to enclaves. It is therefore unnecessary to convert the entire pipeline into a secure application, thereby reducing integration complexity.

Finally, this architecture makes the system's limitations more explicit. Confidentiality guarantees apply to processing actually performed within the enclave. Operations performed outside the enclave, on the other hand, must only handle data that is already encrypted, aggregated, sanitized, or considered non-sensitive by the workflow. This separation makes the security model more transparent and allows for precise analysis of where sensitive data may appear in plaintext.

4.2.3 Separation of control and data plane

The proposed architecture explicitly distinguishes between two categories of communications: the control plane and the data plane. This separation is a key component of the system, as these two categories of traffic have neither the same objectives nor the same security constraints.

The control plane encompasses all operations required to initialize and configure secure processing. It includes, in particular:

- Retrieving the information needed to identify the enclave.
- Establishing cryptographic sessions.
- Provisioning encryption keys and configuring the processes executed within the enclave.

The data plane, on the other hand, corresponds to the business events transmitted through the stream processing pipeline. This data represents the sensitive streams generated by tenants and constitutes information that must be protected in transit and during processing.

This separation offers several architectural and security benefits.

First, it significantly limits the exposure of sensitive data within the distributed infrastructure. Operations in the control plane primarily handle configuration metadata, cryptographic information, or processing parameters, while the business events themselves remain confined to the data plane.

Second, this separation isolates cryptographic management mechanisms from the traditional processing pipeline. The exchanges required for key establishment or enclave configuration are performed independently of the main data flow. This approach simplifies the system's overall architecture and reduces direct interactions between the stream processing engine and the cryptographic mechanisms internal to the enclaves.

This separation becomes particularly important. The data plane carries only encrypted payloads containing business events. The stream processing engine, therefore, never directly manipulates plaintext data (while they are still sensitive) and essentially acts as a transport and orchestration component.

The control plane also plays a key role in the system's trust model. Before a tenant agrees to send encrypted data to an enclave, it must be able to obtain the information necessary to establish a trust relationship with that enclave. The control plane's exchanges thus enable this relationship to be established before any sensitive data is transferred.

This separation also improves the architecture's modularity. The control, provisioning, and key management mechanisms can evolve independently of the data pipeline itself. This notably facilitates the future integration of more advanced

attestation mechanisms, such as complete workflows based on RA-TLS or DCAP, without significantly altering how data streams are processed.

Finally, this approach makes it easier to assess the system's security guarantees. Operations that handle sensitive data and coordination operations are not mixed within the same logical flow. This makes it easier to precisely identify which components have access to business data, which handle only control metadata, and which information remains visible to the untrusted cloud infrastructure.

4.3 Prototype implementation

The previous sections presented the proposed architecture and the key design choices that guided its development. This section now describes the prototype that was implemented to experimentally validate this approach.

Our goal was not to develop a complete secure stream-processing engine, but to build a functional prototype to assess the feasibility of a hybrid architecture that combines an existing distributed engine with enclave-based processing components.

The prototype thus implements the main mechanisms presented earlier:

- A distributed stream processing pipeline.
- A control plane enabling the configuration of secure processing and key provisioning.
- A data plane carrying encrypted events.
- An Intel SGX enclave responsible for executing sensitive operations.

The system currently supports two main categories of processing: transformations applied to a single stream and explicit joins between two distinct streams.

This section describes the main components of the prototype and their role in the system's overall architecture.

4.3.1 Message Queuing Telemetry Transport (MQTT) communication layer

The prototype relies on an event-driven, MQTT-based [20] communication system to facilitate communication among tenants, the stream processing pipeline, and secure components.

The choice of MQTT addresses several objectives related to the context of distributed stream processing. MQTT is a lightweight publish/subscribe protocol widely used in IoT environments and event-driven architectures. Its topic-based model

naturally represents continuous data streams from multiple independent producers, which aligns well with the multi-tenant context studied in this work.

This approach also allows for the decoupling of data producers from processing components. Tenants publish their events to different topics without needing to know which components are consuming the streams directly. This separation simplifies the system's integration into distributed architectures and facilitates the addition of new producers or new processing tasks.

In our architecture, `MQTT` is used for both the control and data planes.

The control plane is used during the initialization phases of secure processing. Dedicated topics enable, in particular, the retrieval of public information associated with the enclave, the establishment of cryptographic sessions, the provisioning of encryption keys, and certain configuration operations required to initialize secure processing.

The data plan, meanwhile, carries the business events generated by the tenants. Before publication, events are encrypted on the tenant side using a symmetric key provisioned to the enclave. Messages circulating within the `MQTT` infrastructure, therefore, never contain business data in plaintext. Each message carries only a cryptographic payload containing, in particular, the tenant ID, the logical stream ID, a nonce, and the encrypted payload.

The `MQTT` broker itself is not considered trustworthy in our security model. It acts solely as a transport component and never possesses the keys required to decrypt business data. This property is important because it allows us to maintain confidentiality guarantees even when the communication infrastructure is controlled by an untrusted cloud provider.

The use of a publish/subscribe model is also particularly valuable in the context of collaborative processing. Multiple tenants can simultaneously publish distinct streams, while the processing pipeline can consume them asynchronously and in a distributed manner. This approach notably facilitates scenarios involving the joining of multiple streams belonging to different organizations.

Finally, the choice of `MQTT` helps minimize coupling between the various components of the architecture. Producers, the stream processing engine, and the enclave components interact solely through standardized message streams. This allows internal encryption mechanisms and enclave implementation details to evolve independently of the communication system itself.

4.3.2 Generic configurable operator

The central component of the processing pipeline is a configurable generic operator integrated into the stream processing engine. This operator was designed to enable the execution of different types of workflows without having to develop a new, specific operator for each use case.

The main objective of this approach is to separate the configuration logic of the processing steps from their execution mechanism. The operator's behavior is entirely controlled by parameters describing the input streams, the schemas associated with each stream, the mappings used to extract *JavaScript Object Notation (JSON)* fields, and the operations to be applied to the data. This approach allows dynamic construction of different workflows without modifying the operator's code.

The prototype currently supports two main categories of processing. The first corresponds to single-stream processing, in which a transformation is applied independently to each event in a given stream. These transformations can be used, in particular, to perform cleaning, filtering, or projection operations before data sharing. The second category corresponds to collaborative join-type processing between two distinct streams. In this case, the operator receives multiple logical streams and coordinates their transmission to the secure backend, which performs matching operations and produces results.

The operator also plays an important role in abstracting the relationship between the stream processing engine and the embedded backend. The `Apache Flink` pipeline is unaware of the exact business logic executed within the enclave. It primarily handles `MQTT` streams, data payloads, and configuration and processing requests. This separation allows for a relatively generic and loosely coupled architecture.

The configurable component also plays a key role in supporting multi-stream workflows. The various streams handled by the system are associated with logical identifiers and independent schemas. This abstraction enables processing multiple heterogeneous streams within a single pipeline without imposing a fixed data structure on the stream processing engine.

Messages received from `MQTT` consist solely of encrypted payloads containing the information necessary for routing and identifying the relevant stream. Sensitive operations are then delegated to the `Intel SGX` backend via a dedicated interface.

This architectural choice allows for a relatively lightweight processing pipeline from the distributed engine's perspective. The `Apache Flink` operator acts primarily as an orchestration and transport component, while the confidential logic remains confined within the enclave.

Finally, this configurable architecture facilitates the gradual evolution of the prototype. New processing types or new secure operators can be added without disrupting the overall organization of the distributed pipeline.

4.3.3 Intel SGX backend

In the remainder of this thesis, the term `Intel SGX` backend refers to the complete component composed of an untrusted host process and a trusted `Intel SGX` enclave. Only the enclave itself is considered part of the `Trusted Computing Base`.

The Intel SGX backend is the component responsible for executing sensitive processing tasks within the proposed architecture. Unlike the stream processing engine, which primarily handles orchestration and data transport, this component is responsible for manipulating plaintext business data within an Intel SGX enclave.

The backend architecture is based on a separation between an untrusted host process and the enclave itself. The host process exposes a minimal network interface that allows it to receive requests from the stream processing pipeline and forward them to the enclave. This component essentially acts as a gateway between the untrusted world and the Intel SGX-protected trusted zone.

All sensitive logic is executed within the enclave. In particular, the enclave maintains configuration information associated with tenants, provisioned encryption keys, and the data structures required for stream processing.

Each tenant has an independent state within the enclave. This state contains the information necessary to execute the configured processing tasks, including the processing mode, the schemas of the processed streams, and the encryption keys associated with the tenant. This logical separation allows for the isolation of processing tasks belonging to different organizations within the same enclave.

The backend currently supports two main processing modes. The first involves single-stream transformations, in which an operation is applied to each received event individually. The second involves join-based collaborative processing between two distinct streams.

When a message is received from the pipeline, the enclave first identifies the relevant tenant and then attempts to decrypt the payload using the keys provisioned for that tenant. Once the data is decrypted, the JSON content is transformed into a typed internal representation to enable the execution of the configured processing steps.

The enclave also maintains a memory state associated with each processed stream. This state is currently implemented as bounded buffers that retain a limited set of recent records for each stream. This structure notably enables join operations across multiple streams while maintaining explicit control over the amount of memory used within the enclave.

The Intel SGX backend also plays a central role in the system's cryptographic management. The enclave generates its own asymmetric key pair and uses shared-secret derivation mechanisms to securely provision symmetric keys for encrypting business streams.

Once provisioned, these keys are used to decrypt events received from the pipeline. Data processed by the stream processing engine thus remains encrypted as it travels through the distributed infrastructure and becomes accessible in plaintext only when processed within the enclave.

However, the Intel SGX backend is not a complete distributed processing

engine. Its role is intentionally limited to operations requiring access to sensitive data. The mechanisms for orchestration, parallelism, and data flow are still handled by the external stream processing engine.

This separation allows us to limit the amount of code executed within the trusted zone while preserving the distributed nature of the overall pipeline.

4.3.4 Apache Flink relay implementation

In the prototype, the stream processing engine plays a deliberately limited role regarding sensitive data. `Apache Flink` does not directly execute confidential logic or decrypt business events. Its role is to connect the various components of the system, consume incoming streams, forward requests to the `Intel SGX` backend, and republish the enclave's results, in order to continue the processing workflow, where the data can once again be processed normally by standard `Apache Flink` operators.

This implementation puts the architectural choice presented earlier into practice: `Apache Flink` remains responsible for orchestrating the pipeline, while operations that handle plaintext data are delegated to an external enclave component for sensitive data.

The `Apache Flink` relay first comes into play in the control plane. When a tenant requests the public information needed to establish a cryptographic session, the request is received via `MQTT`, forwarded to the `Intel SGX` backend, and then the response is republished to the corresponding `MQTT` topic. The same principle is used for stream key provisioning: `Apache Flink` relays the request to the backend, then sends the acknowledgment back to the tenant.

`Apache Flink` then processes the data. The events received from `MQTT` are encrypted payloads containing the information needed to identify the tenant and the relevant stream. For each message, the operator forwards the payloads to the `Intel SGX` backend via the interface exposed by the host process. The enclave then performs decryption, applies the configured processing, and returns one or more `JSON` outputs.

The results produced by the enclave are then retrieved by the `Apache Flink` operator and published to the configured output topic. In the case of joins, a single incoming event can produce zero, one, or multiple outputs depending on the matches found in the state maintained by the enclave.

This architecture allows us to maintain a pipeline compatible with `Apache Flink`'s processing model while preventing the distributed engine from becoming a trusted component. `Apache Flink` can continue to handle source reading, message routing, and integration with the stream processing ecosystem, but it only processes encrypted wrappers or results explicitly produced by the enclave.

The relay's implementation is therefore based on a strict separation of responsibilities. The distributed engine transports and orchestrates streams, while the `Intel SGX` enclave interprets sensitive data and executes confidential operations. This separation is at the heart of the prototype, as it enables the integration of enclave-based processing into an existing pipeline without significantly altering the stream processing engine's internal workings.

4.4 Example of workflow

To provide a concrete illustration of how the proposed architecture works, this section describes the execution of a collaborative workflow between a hospital network and an urban transit operator. This scenario revisits the join query presented in the introduction and demonstrates how data flows are processed by the various system components while maintaining the confidentiality of business data.

In this scenario, hospitals publish medical events that include information on observed symptoms and the transit station closest to the patient. For its part, the transit operator publishes information related to passenger density and the lines associated with the various stations in the network.

The collaborative workflow executed in the enclave then correlates these two streams through a join operation:

```
SELECT
     admissions.symptom_code AS symptom_code,
     admissions.severity AS severity,
     admissions.source_ts_ms AS admissions_source_ts_ms,
     passengers.source_ts_ms AS passengers_source_ts_ms,
     passengers.station_id AS station_id,
     passengers.occupancy_pct AS occupancy_pct
 FROM admissions
 JOIN passengers
 ON admissions.nearest_station_id = passengers.station_id
```

This query links medical events to their corresponding mobility contexts to generate information useful to both organizations, without exposing their raw data to the distributed infrastructure.

With the help of Figure 4.3, we will detail the execution of this query through our system. Each subsequent paragraph details a section of the Figure, numbered 1-4.

(1) The first step is to configure the `Intel SGX` enclave. As explained above, one of the enclave's objectives is to be configurable, allowing it to be used regardless of the data format. The `Apache Flink` operator will therefore send a request to

the Intel SGX backend to configure it using environment variables defined during deployment. Note that Apache Flink makes the request directly in our prototype, as this has no impact on our evaluation. However, in a real-world scenario, the tenants would make the request, and Apache Flink would simply relay it.

(2) Each tenant establishes a trust relationship with the Intel SGX enclave. This phase allows participants to retrieve public information about the enclave and perform the operations necessary to provision the encryption keys that protect business data streams.

(3) Once this phase is complete, events generated by the various organizations are encrypted before being published to the distributed infrastructure. Medical and mobility data are therefore transmitted as encrypted packets within the MQTT broker and the stream processing engine. The Apache Flink relay then consumes these messages and forwards them to the Intel SGX backend without accessing their business content. The distributed engine thus retains a role in orchestrating and routing data flows, but does not directly participate in the confidential processing of the data. Upon receiving the events, the Intel SGX backend forwards the encrypted payloads to the enclave.

(4) The enclave decrypts both streams, maintains the states required for the workflow, and then performs the join operation between the hospital events and the transport events corresponding to the same station. When the join condition is met, the enclave constructs a new object containing only the fields defined in the output projection. The result is then sent back to the Apache Flink relay, which republishes it in the distributed infrastructure.

This architecture ensures that business data remains protected throughout its entire transit through the cloud infrastructure while enabling collaborative processing across multiple independent tenants.

4.5 Limitations and discussion

The proposed architecture significantly limits the exposure of sensitive data within a distributed stream-processing infrastructure considered untrusted. The business data generated by tenants is encrypted before being fed into the pipeline and remains protected while in transit through the cloud infrastructure. The stream processing engine, the MQTT broker, and the various intermediate components never directly handle business data in plain text. They only see encrypted payloads containing the information necessary to route the streams.

Sensitive data is accessible in plain text only within the Intel SGX enclaves responsible for confidential processing. Collaborative operations, particularly joins between streams belonging to different tenants, are also executed exclusively within this trusted zone. The results produced by these processes correspond only to

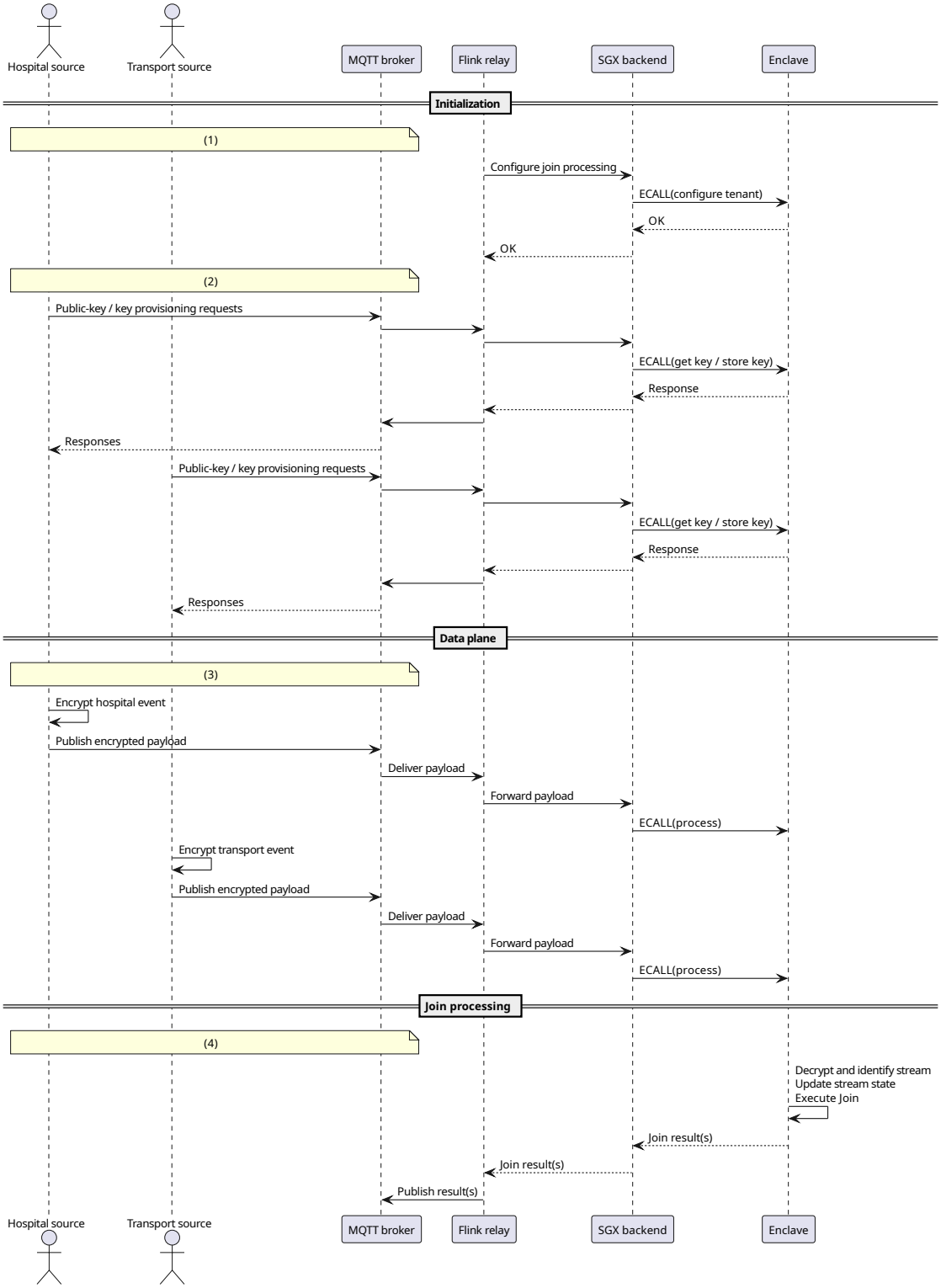

Figure 4.3: Example workflow

data explicitly authorized by the projections configured in the workflow. Tenants, therefore, do not directly share their raw data with one another, but only derived information controlled by the processing executed within the enclave.

The security model relies primarily on the isolation provided by `Intel SGX` and on attestation mechanisms that verify the identity of the code executed in the enclave before any cryptographic secrets are provisioned. However, the current prototype does not yet implement a full `RA-TLS` mechanism between tenants and the enclave.

This limitation stems primarily from the experimental environment used in this work. The available `Intel SGX` platform supports only legacy `EPID` mechanisms and does not allow modern workflows based on `DCAP` and `RA-TLS`. In the current prototype, communications between tenants and the `Intel SGX` enclave therefore rely on a simplified key provisioning model.

In a real-world production architecture, however, integrating `RA-TLS` would be essential. Such a mechanism would allow for the direct establishment of a trusted `TLS` channel between the tenant and the remote enclave, ensuring not only the confidentiality of communications but also the identity of the code executed in the enclave before any cryptographic secrets are shared. This property is particularly important in our architecture, as the stream processing engine acts as a relay between data producers and the `Intel SGX` backend. Without a comprehensive, attested channel mechanism, the intermediate infrastructure can interfere with data streams, including suppressing, delaying, or reordering messages. The integration of `RA-TLS` would therefore significantly strengthen the system's guarantees.

Another major limitation of the prototype concerns the operations that can be performed in the `Intel SGX` enclave. Currently, it supports only a relatively limited subset of processing operations, primarily simple single-stream transformations and explicit joins between two streams.

This limitation stems directly from the architectural choice made in this work. The `Intel SGX` enclave is intentionally isolated from the stream processing engine and therefore does not directly benefit from `Apache Flink`'s abstractions, particularly `Table API` and its associated `SQL` mechanisms. The processing operations executed within the `Intel SGX` enclave must therefore be implemented manually.

For this prototype, we chose to implement only the subset of operations necessary to experimentally validate the proposed architecture. This choice allowed us to limit the complexity of the enclave code base while demonstrating the feasibility of secure collaborative processing in a distributed pipeline.

However, in a real-world use case, broader support for relational operations would be necessary to make the system more flexible and easier to integrate into complex stream processing workflows. Expanding the capabilities of the `Intel SGX` enclave is therefore an important avenue for the system's future development.

To build on our example from the introduction, here is a more advanced collaborative query that combines joins, aggregation, and time windows. This query aims to evaluate, over the last 15 minutes, the degree of correlation between the average number of correlated admissions and the average transport occupancy rate at a station. It enables the production of an aggregate indicator combining local hospital pressure and mobility density, in order to identify potentially vulnerable urban areas more quickly and provide both organizations with actionable information to adapt their operational response:

```
SELECT
     station_id,
     window_start,
     window_end,
     AVG(CAST(severity AS DOUBLE)) AS avg_severity,
     AVG(occupancy_pct) AS avg_occupancy_pct
  FROM TABLE(
     TUMBLE(
         TABLE (
             SELECT
                 admissions.severity AS severity,
                 passengers.station_id AS station_id,
                 passengers.occupancy_pct AS occupancy_pct,
                 admissions.event_time AS event_time -- Non-existing
                     column in our described streams, used for the
                     example
             FROM admissions
             JOIN passengers
             ON admissions.nearest_station_id = passengers.station_id
         ),
         DESCRIPTOR(event_time),
         INTERVAL '15' MINUTES
     )
  )
  GROUP BY
     station_id,
     window_start,
     window_end
```

However, implementing it would require more advanced mechanisms for state management, time windows, and aggregation within the `Intel SGX` enclave.

Chapter 5

Evaluation

The goal of this section is to evaluate the practical feasibility of the operator presented in the previous section and, more specifically, the cost of integrating an `Intel SGX` environment into a distributed stream processing pipeline.

Unlike some methods that measure only the isolated cost of `Intel SGX` primitives, such as `ECALL/OCALL` transitions, our goal here is to evaluate the actual cost of a complete architecture that integrates `Apache Flink`, network communication, encryption mechanisms, and a secure enclave-based backend. This approach yields results that are more representative of a real-world deployment.

5.1 Evaluation objective

Here is the question we aim to answer in this evaluation:

What is the overall performance cost of our hybrid architecture compared to a standard, non-secure pipeline? The goal is not to isolate the raw overhead of `Intel SGX` primitives, but to evaluate the practical feasibility and performance envelope of the complete secure workflow, including encryption, enclave transitions, network communication, and stream processing orchestration, in a realistic deployment scenario.

To this end, we compare two execution modes of the same operator (the same operations applied to the data):

- A mode where the operator is a standard `Apache Flink` operator, written entirely in Java, without any use of enclaves.
- An enclave mode where the data is processed within an `Intel SGX` enclave, written in C++, with `Apache Flink` operator acting solely as a relay (our solution).

The evaluation primarily focuses on collaborative workflows, as these best reflect the multi-tenant, confidential processing scenarios addressed by this work.

Simpler workloads are used as micro-benchmarks to isolate the specific costs associated with our architecture.

5.2 Experimental configuration

The experiments were conducted in a distributed environment that closely resembles a real-world deployment. The platform includes an `Apache Flink JobManager`, six `TaskManagers`, an `MQTT` broker, a data producer, and a generic operator deployed in a cluster. In `Intel SGX` mode, an additional component hosting the enclave runs separately.

The experiments were conducted on an Intel NUC8i5BEH machine equipped with an Intel Core i5-8259U processor (4 cores, 8 threads) that supports `Intel SGX` extensions. The machine has 16 GB of `RAM` and 4 GB of swap space. The host system runs on a Linux kernel 5.10.0-051000-generic, with *Docker Engine* 29.1.3 and *Docker Compose* 2.40.3. This device is dedicated to these experiments.

The platform uses `Intel SGX` in legacy mode, with a stack based on the *isgx* driver [21]. The `Intel SGX` environment is based on `Intel SGX` *Software Development Kit (SDK)* 2.11.100.2 and the corresponding Platform Software. The available `EPC` memory is 128 MB.

The project's software stack is based on `Apache Flink` 2.2.0 and runs on *OpenJDK* 17.0.18. The `MQTT` broker used is *Mosquitto* 2.1.2. Components are deployed via `Docker`, using images based on *Temurin* 17 for Java and Python 3.14 for the data producer.

5.3 Measurement methodology

There are three main observations: latency, fixed-window throughput, and scaling experiments.

Latency is measured end-to-end, from when the producer sends the message to when it is received after processing. Each message includes a timestamp that enables precise calculation of the elapsed time across the entire pipeline.

The first throughput is measured by counting the number of messages processed during a fixed time window.

In addition to fixed-load tests, we also conduct scaling experiments in which the theoretical input load is gradually increased. These experiments are designed to evaluate the system's sustained operational capacity under stress.

This methodology allows us to distinguish between the maximum observed throughput and the sustainable throughput in practice in a real-time data processing context.

In our measurements, two payload sizes are considered.

- Messages without padding
- Messages padded with 8192 bytes

Without the padding, the messages are between 250 and 280 bytes in plain text, and between 468 and 505 bytes when encrypted.

This variation allows us to assess the impact of data size on the secure pipeline's costs.

Our benchmarks are run following a warm-up phase designed to allow the pipeline to stabilize before measurement. For latency measurements, we ignore the first 30 messages and measure latency over 200 sampled messages. For fixed-load throughput measurements, a 10-second warm-up is applied, followed by a throughput measurement over a 20-second window.

Full campaigns are repeated 5 times per scenario. The raw results of each run are then aggregated. For latency, each run produces, in particular, the mean, the median (p50), the p95 and p99 percentiles, and the standard deviation. When aggregating repetitions, the summary retains only the mean, standard deviation, minimum, and maximum for each metric. For throughput, the summary aggregates the input throughput, output throughput, and their ratio in the same way.

For capacity benchmarks, the load is increased gradually in steps. The tick intervals tested are 0.1, 0.05, 0.02, 0.01, and 0.005 seconds. Each step applies a 5-second warm-up for throughput, followed by a 15-second measurement, and a latency measurement over 120 messages after 20 warm-up messages. A step is considered unstable if the p95 latency exceeds 200 ms, or if the input throughput continues to increase whilst the output throughput is no longer growing sufficiently.

Although the generators do not rely on a fixed seed, randomness remains controlled from one run to the next because events are always generated according to the same distribution. Different runs, therefore, do not observe exactly the same sequence, but they sample the same statistical process. The comparability of the results thus depends on the stability of these distributions and on the repetition of the measurements, rather than on the exact reproduction of an identical scenario.

Note that the plain and `Intel SGX` execution modes do not share the same implementation stack. The plain mode uses `Apache Flink`'s native `Table API` and Java-based operators, while the `Intel SGX` mode relies on a custom C++ engine running within the enclave. This difference is intentional and inherent to the architecture: sensitive operations must be implemented natively within the enclave, and cannot directly reuse `Apache Flink`'s runtime. As a consequence, the

comparison between the two modes reflects the overall cost of adopting the proposed secure architecture, rather than the isolated overhead of `Intel SGX` execution. This is consistent with our evaluation objective, which prioritizes practical feasibility over a controlled micro-benchmark of `Intel SGX` primitives.

5.4 Micro-benchmarks

This section presents simple single-stream workloads used as micro-benchmarks to more accurately isolate certain costs associated with secure processing. It should be noted that evaluating our system on simpler queries is not without interest, as our secure operator could well be used in a single-stream context, for example, to sanitize sensitive data prior to a join.

The following analyses examine this query:

```
SELECT
  'passenger' AS type,
  station_id,
  count_in,
  count_out,
  occupancy_pct,
  raw_json AS data
FROM typed_input -- typed_input is the typed view derived from JSON
WHERE occupancy_pct >= 0.25
```

The purpose of this query is to filter the transport network traffic data so as to retain only those events corresponding to stations where passenger numbers exceed a specified threshold.

5.4.1 Latency

In simple workloads, the impact of `Intel SGX` is noticeable. For small messages, the average latency increases from 60.32 ms to 67.99 ms, in the graph on the left in Figure 5.1. For 8192-byte messages, the increase is from 72.93 ms to 89.31 ms, in the graph on the left in Figure 5.2.

The quantiles also show a relatively consistent increase across the entire latency distribution, in the graph on the right in Figure 5.1 and Figure 5.2.

The costs associated with enclaves constitute a significant proportion of the total processing time in this case, given the simplicity of the query processing pipeline.

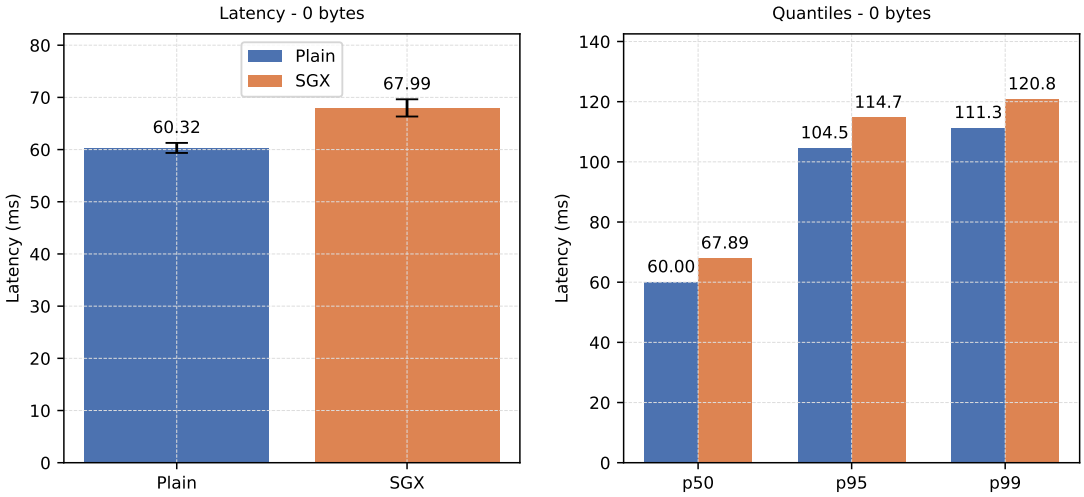

Figure 5.1: Latency and quantiles for simple requests (0-byte padding)

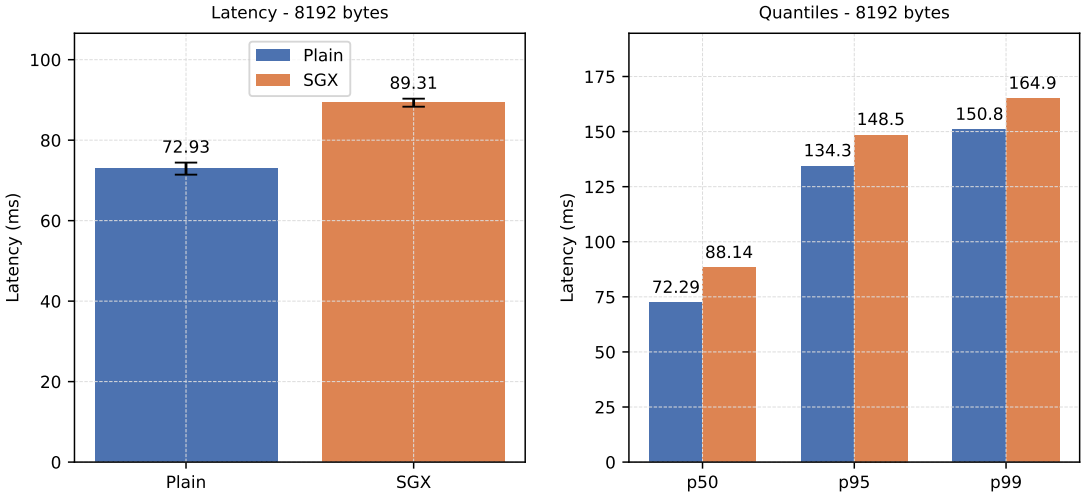

Figure 5.2: Latency and quantiles for simple requests (8192-byte padding)

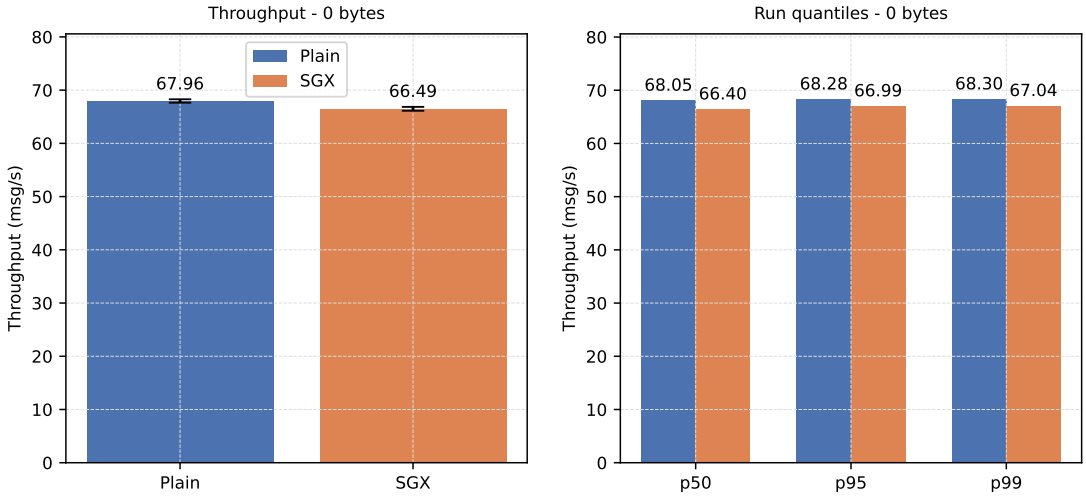

Figure 5.3: Throughput and quantiles for simple requests (0-byte padding)

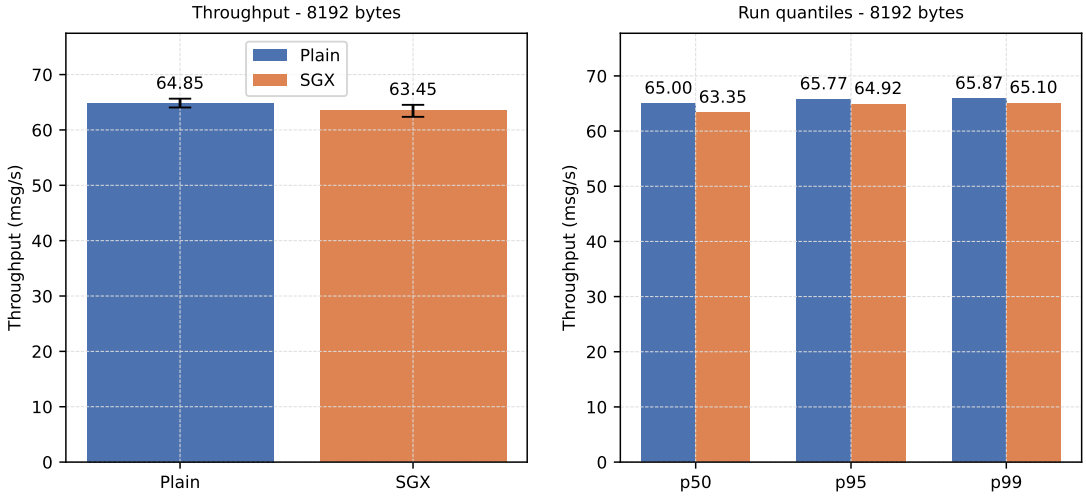

Figure 5.4: Throughput and quantiles for simple requests (8192-byte padding)

5.4.2 Throughput

The throughput results show a relatively small decrease in performance under `Intel SGX`.

For small messages, throughput drops from 67.96 msg/s to 66.49 msg/s, in the graph on the left in Figure 5.3. For larger messages, it drops from 64.85 msg/s to 63.45 msg/s, in the graph on the left in Figure 5.4.

The quantiles show stable results across runs for both non-padded and padded messages, as shown in the graph on the right in figures 5.3 and 5.4.

Although a decrease is observable, it remains limited. This indicates that the additional overhead introduced by the enclave affects individual message latency more than the system's overall ability to maintain a stable throughput.

To supplement the tests conducted under fixed load, we also evaluate the sustainable capacity of the simple workload.

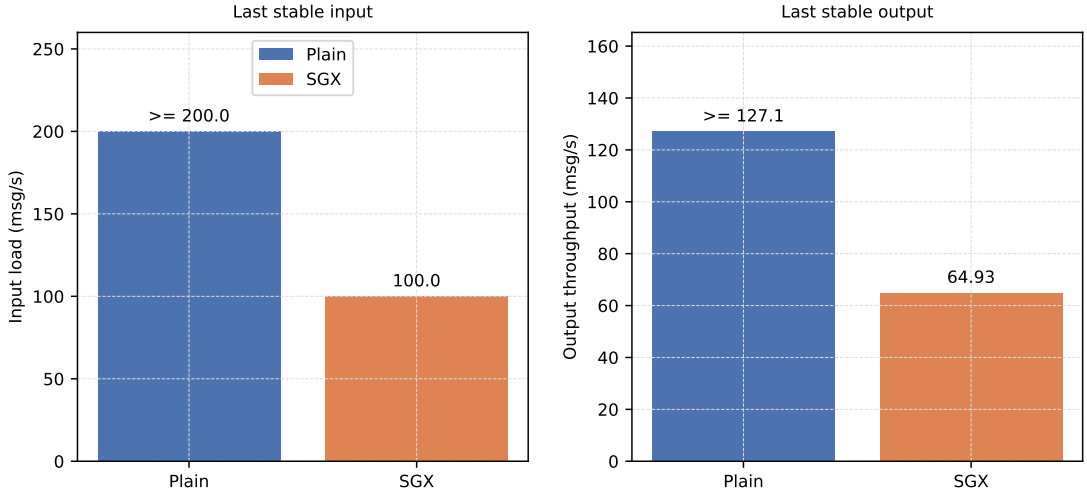

Figure 5.5: Sustainable capacity for simple workloads

The sustainable operating capacity of the simple workload is summarized in Figure 5.5. The figure shows both the highest stable input rate and the corresponding output throughput before the p95 latency exceeds the 200 ms threshold.

Figure 5.6 illustrates how throughput and p95 latency evolve as the theoretical input load increases. It provides a more detailed view of the system behavior under progressively increasing pressure.

Load testing shows that the simple workload remains relatively robust as the load increases gradually.

In plain mode, no saturation is observed within the tested range, and the system remains stable up to the last evaluated level. In `Intel SGX` mode, the system

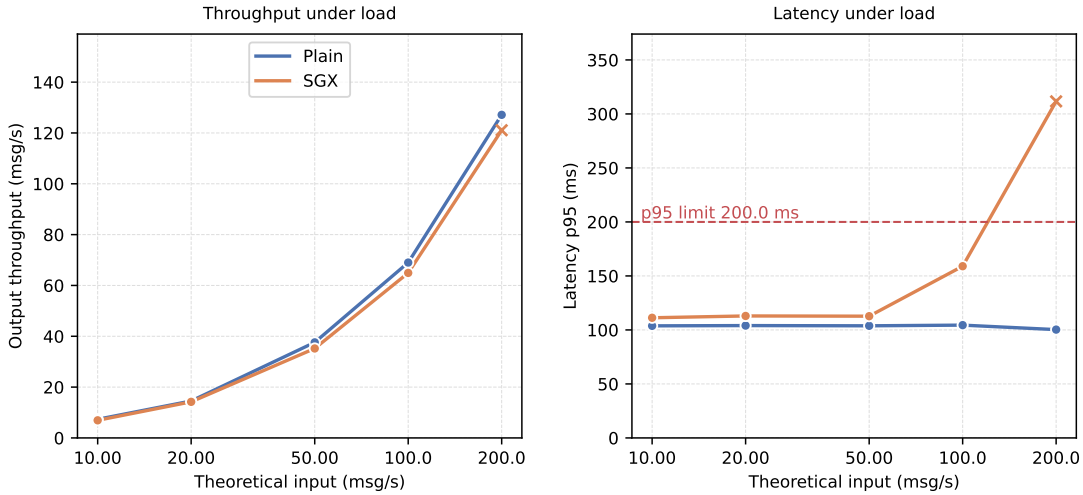

Figure 5.6: Load scaling behavior for simple workloads

remains stable up to 100 messages per second, while the next level exceeds the p95 latency threshold set at 200 ms.

Although `Intel SGX` mode continues to deliver high throughput under high load, these operating points are not sustainable due to observed latency degradation.

5.4.3 Payload impact

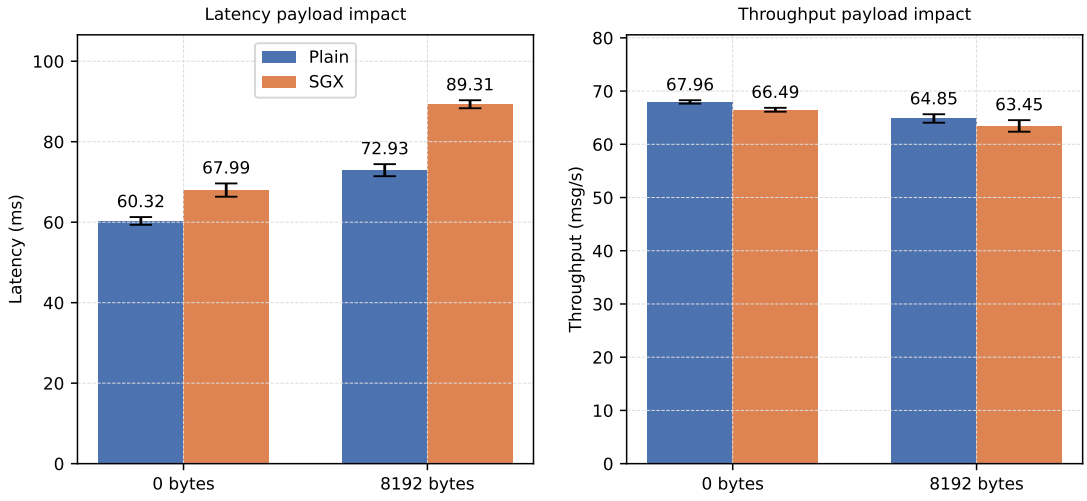

Figure 5.7: Impact of payload size on simple requests

This Figure 5.7 shows that an increase in payload size affects both latency and throughput.

The impact is more noticeable in `Intel SGX` mode, suggesting that the costs associated with encryption, memory copies, and communications with the enclave increase as the size of the data being processed increases.

5.4.4 Discussion

Micro-benchmarks allow us to more precisely isolate the overhead introduced by execution in `Intel SGX` enclaves on a simple workflow that involves neither joins nor complex stateful logic.

The results show that `Intel SGX` mode introduces a measurable overhead in both latency and throughput. This overhead stems primarily from encryption and decryption operations, enclave-to-non-enclave transitions, and constraints on `Intel SGX`-protected memory. Despite this degradation, the system maintains relatively stable performance across all tested moderate workloads.

Scaling experiments also show that the simple workload remains relatively robust to a gradual increase in load. In plain mode, no saturation is observed within the explored range. In `Intel SGX` mode, the system maintains acceptable latency up to 100 messages per second, after which latency degrades more sharply.

These results show that the impact of `Intel SGX` remains relatively moderate when enclave processing is simple and largely stateless, since single-stream processing does not require complex correlation mechanisms or significant state maintenance. The observed costs, therefore, stem primarily from the secure environment itself rather than from the application logic.

The micro-benchmarks thus confirm that the proposed architecture can support simple processing at relatively high throughput while maintaining the privacy guarantees provided by `Intel SGX`.

5.5 Workflow involving join queries

This section presents the main analysis of our evaluation. Unlike the micro-benchmarks presented in the previous section, these experiments focus on a workflow that joins multiple data streams, which is more closely related to the collaborative context examined in this thesis.

To fully understand the following query, we introduce the Table 5.1, which represents a stream of pseudonymized individual trips generated by the transport operator. Each record describes a trip on the network, specifying an entry station, an exit station, the associated timestamps, and a fare category.

Table 5.1: Trip stream

Attribute	Description
card_id	Pseudonymized card identifier
ts_in	Entry timestamp
source_ts_ms	Source timestamp (ms)
station_in	Entry station
ts_out	Exit timestamp
station_out	Exit station
fare_class	Fare category
created_at	Record creation timestamp

The following analyses are based on the query below:

```
SELECT
  passengers.station_id AS station_id,
  passengers.count_in AS count_in,
  passengers.count_out AS count_out,
  passengers.occupancy_pct AS occupancy_pct,
  passengers.source_ts_ms AS passengers_source_ts_ms,
  trips.source_ts_ms AS trips_source_ts_ms,
  trips.fare_class AS fare_class,
  trips.station_out AS station_out
FROM passengers
JOIN trips
ON passengers.station_id = trips.station_in
```

We therefore received data from the transport operator in two separate streams, which allows us to accurately replicate a multi-tenant scenario. The purpose of this query is to correlate the two data streams, linking station passenger volume figures to information on individual trips associated with the same entry point into the network.

5.5.1 Latency

The results show that workload involving join queries naturally incurs higher latency than simple processing, as expected given the additional complexity of stream synchronization and matching operations.

However, contrary to expectations, the use of `Intel SGX` does not significantly increase average latency. For unpadded messages, the average latency drops from 105.2 ms in plain mode to 101.8 ms in `Intel SGX` mode, in the graph on the left

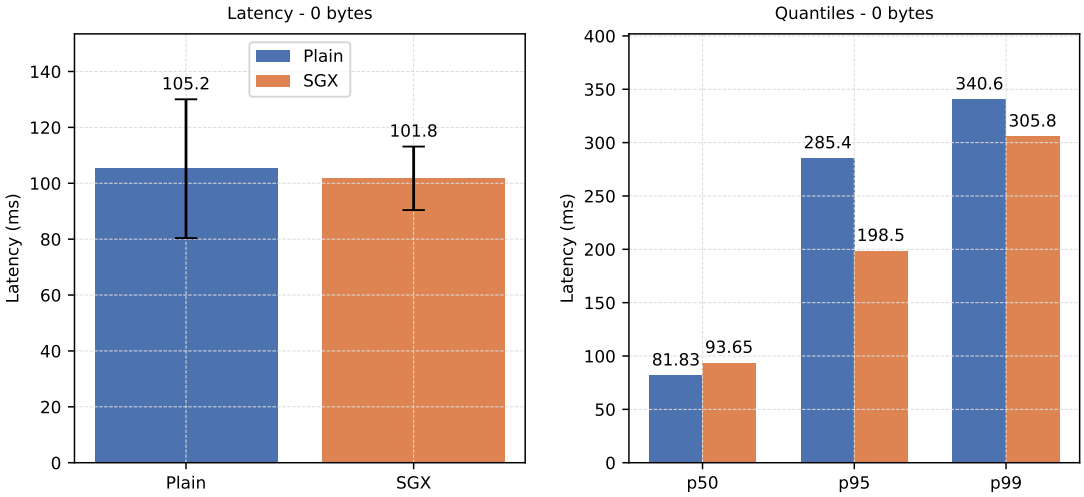

Figure 5.8: Latency and quantiles for join requests (0-byte padding)

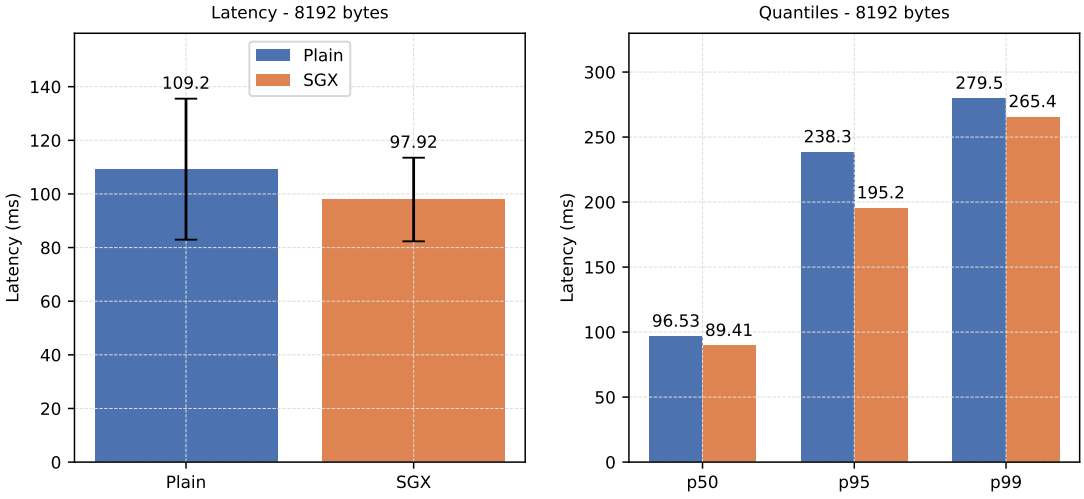

Figure 5.9: Latency and quantiles for join requests (8192-byte padding)

in Figure 5.8. With 8192-byte messages, the average latency drops from 109.2 ms to 97.92 ms, in the graph on the left in Figure 5.9.

The quantiles confirm this trend. The p95 and p99 values remain of the same order of magnitude between the two modes, and are even slightly lower in several scenarios in `Intel SGX` mode, in the graph on the right in Figure 5.8 and Figure 5.9. This suggests that the overall cost of the distributed pipeline now largely outweighs the cost introduced by the enclave.

This result should be interpreted with caution. The two implementations differ not only in the presence of an `Intel SGX` enclave but also in their underlying technology stacks. The plain mode relies on `Apache Flink`'s `Table API` and a Java-based execution engine, while the `Intel SGX` mode uses a custom C++ implementation. The observed difference may therefore reflect implementation-level discrepancies rather than a genuine advantage of enclave-based processing. A more accurate interpretation is that latency in both modes is comparable, and that the cost introduced by `Intel SGX` is absorbed by the overall pipeline complexity in this workload.

Another important observation is that the increase in message size has a relatively limited impact on the overall latency of the workflow involving join queries. The cost associated with application processing and stream synchronization appears to be more significant here than the cost of transferring the data itself.

5.5.2 Throughput

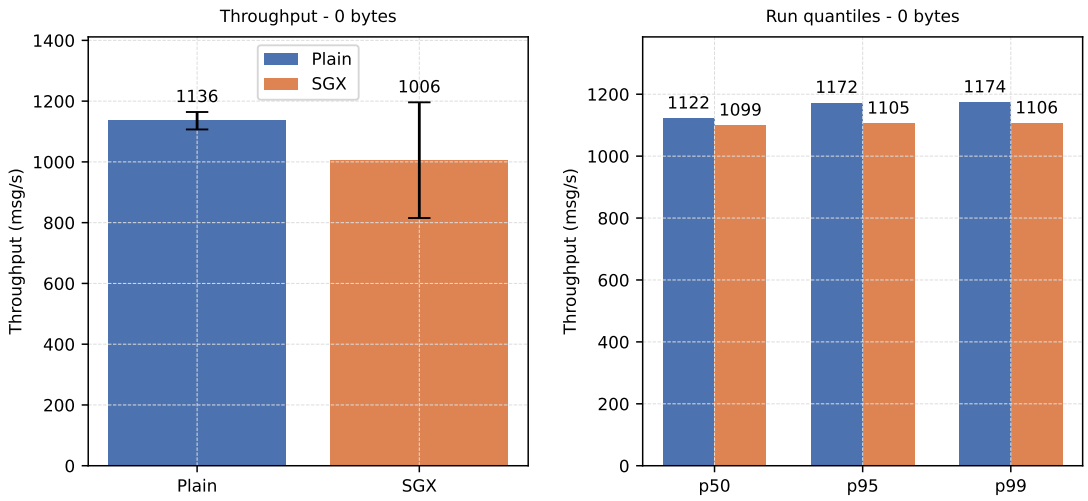

Figure 5.10: Throughput and quantiles for join requests (0-byte padding)

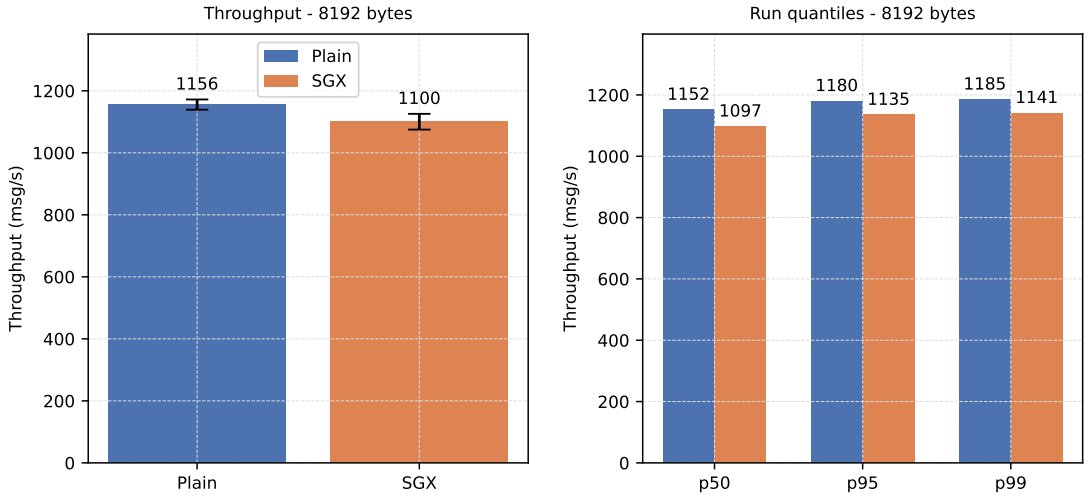

Figure 5.11: Throughput and quantiles for join requests (8192-byte padding)

The throughput results also show an interesting pattern. For unpadded messages, throughput drops from 1136 msg/s in plain mode to 1006 msg/s in `Intel SGX` mode. For 8192-byte messages, it drops from 1156 msg/s to 1100 msg/s.

Although a decrease is observable, the baseline remains high in both execution modes. More importantly, the relative gap between plain and `Intel SGX` remains limited given the complexity of the workflow.

The throughput quantiles also show good stability across different runs, as shown in the graph on the right in figures 5.10 and 5.11. The measured variations remain small, indicating, at first glance, that the integration of `Intel SGX` does not cause significant instability in the pipeline.

The previous figures evaluate the observed throughput for fixed workloads, but they do not characterize the system's behavior as pressure gradually increases. To assess the operational limits of the workflow involving join queries, we also analyze the system's sustainable capacity under load.

In this experiment, a run is considered stable as long as the p95 latency remains below 200 ms. Above this threshold, the system is considered unstable, even if the output throughput continues to increase.

Figure 5.12 summarizes the sustainable capacity of the workflow involving join queries. It reports the last stable input rate and the corresponding output throughput for both execution modes.

Figure 5.13 presents the evolution of throughput and p95 latency as the offered load increases. This allows us to identify the point at which the workflow enters an unstable operating region.

Scaling experiments more clearly highlight the difference between plain and

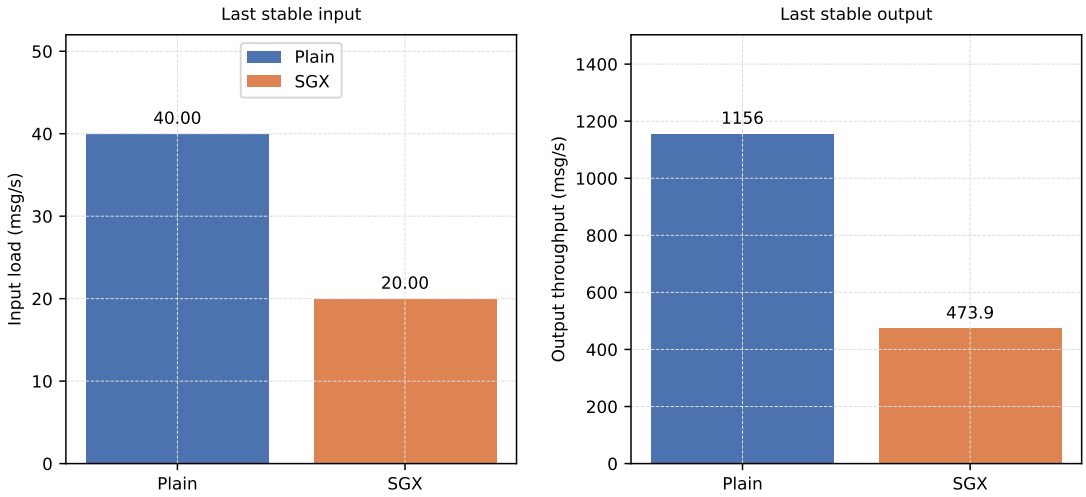

Figure 5.12: Sustainable capacity for workloads involving join queries

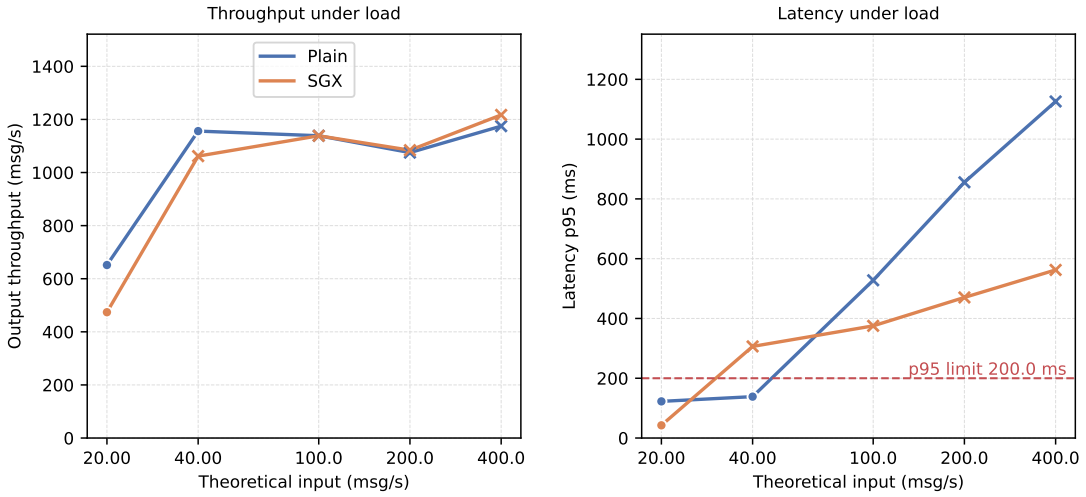

Figure 5.13: Load scaling behavior for workloads involving join queries

`Intel SGX` modes.

For the workload involving join queries, the system remains stable up to 40 input messages per second in plain mode, while `Intel SGX` mode becomes unstable beyond 20 messages per second, depending on the chosen p95 latency threshold.

The throughput curves also show that an increase in input load does not necessarily result in a proportional increase in output throughput. In plain mode, throughput plateaus at around 1100-1200 messages per second, indicating the system is approaching its maximum usable capacity. In `Intel SGX` mode, this saturation occurs earlier, with a rapid degradation in latency.

These results highlight an important distinction between observed maximum throughput and sustainable capacity. Certain unstable plateaus can still produce high throughput, but at the cost of significant latency degradation. In a stream processing context, these operating points cannot therefore be considered operationally viable.

The experiments thus show that execution within an `Intel SGX` enclave reduces the system's sustainable capacity under heavy load, particularly for stateful workloads involving join operations.

5.5.3 Payload Impact

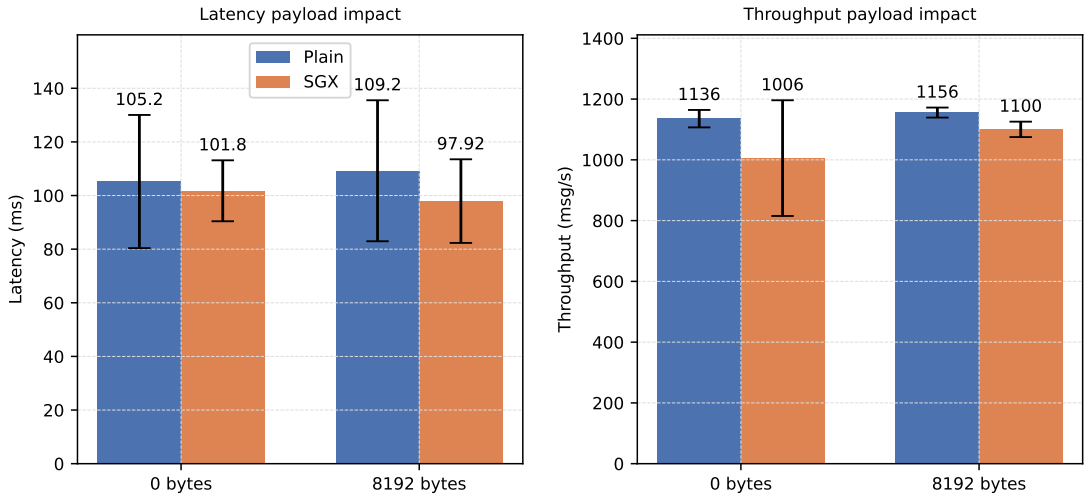

Figure 5.14: Impact of payload size on join requests

This Figure 5.14 summarizes the impact of payload size on the workflow involving join queries.

Experiments show that increasing the payload size has only a limited impact on the workflow involving join queries. The variations remain relatively small in both

execution modes, suggesting that the primary costs stem more from synchronization, coordination, and state management than from the raw size of the payload being transferred.

5.5.4 Input/output analysis

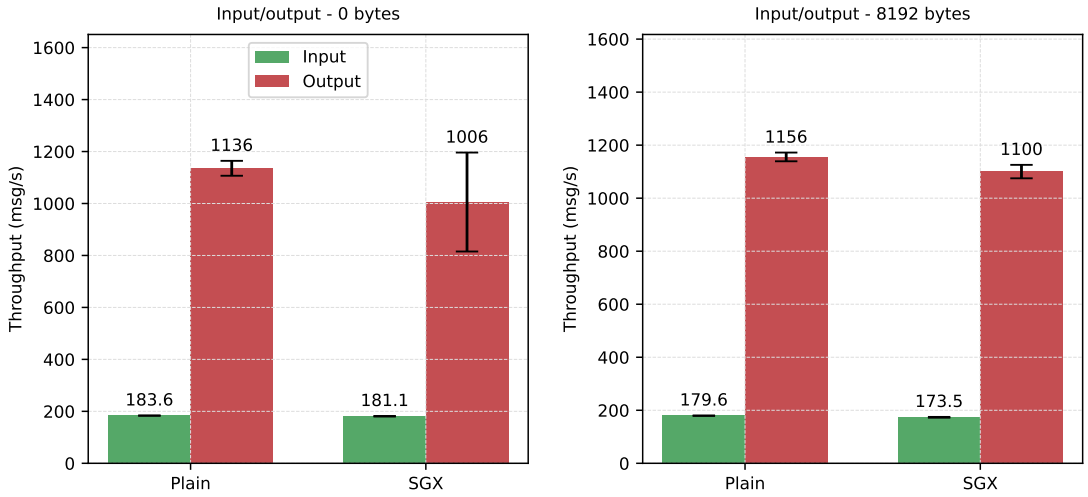

Figure 5.15: Input and output throughput for join requests

This Figure 5.15 compares the throughput of messages injected into the system with the throughput of messages produced after join operations.

Unlike a simple pipeline, a join can produce multiple results from the input messages. It is therefore normal for the output throughput to exceed the input throughput.

The results show that the input and output throughputs remain similar in both plain and `Intel SGX` modes. This confirms that the `Intel SGX` backend remains capable of handling high throughput during joins. However, as the scaling experiments show, maintaining such throughput under increasing load may come at the cost of higher latency and reduced sustained capacity.

5.5.5 Discussion

The results for the workflow involving join queries highlight the overhead of secure execution in `Intel SGX` enclaves, but also demonstrate that confidential collaborative processing remains feasible in a distributed stream processing architecture.

The latency results show that enclave-based execution does not introduce a systematic overhead compared to plain mode for this workload. However, this

comparison must be qualified: the two modes use different implementation stacks, and the slight latency difference observed is unlikely to be attributable solely to `Intel SGX`. The more meaningful takeaway is that latency remains on the same order of magnitude across both modes, confirming that enclave integration does not fundamentally alter the pipeline's responsiveness for join workloads.

The throughput results also show that secure execution reduces the system's processing capacity. This reduction is, however, gradual: the system continues to produce a high throughput even when join operations are executed within the enclave. The measurements also show that the overall cost of the pipeline does not stem solely from `Intel SGX` itself, but also from the distributed mechanisms used by the architecture, notably message passing, serializations, and network exchanges.

Scalability experiments, however, provide important additional insights. Fixed-load measurements allow us to observe the system's behavior under controlled conditions, but they are not sufficient to characterize the architecture's operational limits. Sustainable capacity experiments show that `Intel SGX` mode reaches its saturation point more quickly than plain mode.

As we will see, this difference becomes particularly evident in the workload involving join queries. Stateful operations and stream correlation mechanisms place a greater load on the enclave, reducing the available margin before latency degradation occurs. The results thus show that the system can continue to produce high throughput even in unstable situations, but that such operation comes at the cost of a significant increase in p95 latency. In a stream processing context, these operating points cannot be considered sustainable.

The results also highlight the fact that the complexity of the processing significantly influences the relative impact of `Intel SGX`. While the simple workload maintains a relatively high sustained throughput, the workload involving join queries reaches its operational limits more quickly, particularly in secure mode. This shows that the cost of enclave execution becomes more apparent when the processing requires more state, matching, and memory manipulation.

Overall, these experiments show that our approach effectively enables confidential collaborative processing in a distributed infrastructure, but it introduces a clear trade-off between confidentiality and sustained capacity under heavy load. The results obtained are sufficient to demonstrate the practical feasibility of the proposed architecture in real-time, multi-tenant collaboration scenarios under moderate load.

5.6 Global discussion

Our results allow us to draw conclusions about the usability of our system in practice.

5.6.1 Cost of the complete secure workflow

Experimental results show that integrating `Intel SGX` enclaves into a distributed stream processing pipeline incurs measurable costs in latency, throughput, and sustained capacity. This cost is already apparent in simple workloads, but becomes more noticeable when processing requires stateful collaborative operations such as joins between multiple streams.

Experiments show, however, that this cost does not stem solely from execution within the enclave itself. A significant portion of processing time is also associated with the various components of the distributed pipeline, notably message serialization, network exchanges, `MQTT` communications, and the internal mechanisms of the stream processing engine. The observed cost, therefore, reflects the behavior of the entire secure workflow and not merely the raw overhead of `Intel SGX`.

Scaling tests also show that secure execution reduces the system's sustainable capacity as load gradually increases. This reduction remains relatively moderate for simple workloads but becomes more significant for workloads involving join queries. Correlation operations between streams and state management within the enclave reduce the available margin before saturation, leading to a faster increase in p95 latency.

The results also highlight the importance of distinguishing between observed maximum throughput and sustainable capacity. Some scenarios continue to produce high throughput despite a significant increase in latency. In a stream processing context, however, these operating points are not operationally viable. Scaling experiments, therefore, provide important additional information compared to fixed-load benchmarks, as they allow us to identify the system's stability limits under pressure.

Overall, the results show that the cost of secure execution remains highly dependent on workload complexity. The more state management and coordination between streams the processing requires, the more visible the relative impact of `Intel SGX` becomes.

5.6.2 Practical viability

Despite the observed overhead, the experimental results show that the proposed architecture remains capable of supporting confidential collaborative processing in a distributed stream processing environment.

Simple workloads maintain a relatively high sustainable capacity even in `Intel SGX` mode, while workloads involving join queries remain capable of producing significant throughput while preserving confidentiality guarantees on the data being processed. The results thus demonstrate that it is possible to integrate `Intel SGX` enclaves into a distributed pipeline without running the entire stream processing

engine within an enclave.

The evaluation also validates the main architectural choice of this work, based on a separation between distributed infrastructure and enclave-based confidential processing. The stream processing engine retains its role in orchestrating and routing streams, while sensitive operations remain confined within the enclave. This approach allows compatibility with existing infrastructure while limiting exposure of sensitive data.

Experiments, however, show that this approach involves a clear trade-off between privacy and operational capacity under heavy load. Secure workloads reach their saturation point more quickly than their unsecured counterparts, particularly when processing becomes stateful or requires complex collaborative operations.

Finally, the results presented in this work should be interpreted in light of the prototype's current limitations. The `Intel SGX` enclave currently supports only a limited subset of processing operations, and certain advanced mechanisms, such as full `RA-TLS` integration, have not yet been implemented in our experimental environment. This evaluation should therefore be viewed primarily as a feasibility validation of the proposed architecture rather than as an evaluation of a complete secure engine intended for industrial use.

5.6.3 Limitations

The prototype evaluated deliberately relies on a limited subset of processing operations. The workloads used focus primarily on simple transformations and join operations between streams, as these processes already allow us to assess the main costs introduced by integrating `Intel SGX` enclaves into a distributed pipeline. The results obtained, therefore, remain representative of the fundamental mechanisms studied in this work, notably transitions between the untrusted world and the enclave, the cost of encryption, state management in `Intel SGX`, and the impact of stateful collaborative processing.

Similarly, certain limitations of the prototype, such as the lack of full support for complex aggregations or advanced time windows, do not prevent the analysis of the architecture's general properties. The experiments already clearly highlight the trade-offs introduced by using a secure execution environment, particularly regarding latency, sustainable throughput, and memory pressure in collaborative workloads.

The absence of comprehensive remote attestation mechanisms integrated into the prototype also has only a limited impact on the interpretation of the experimental results. The evaluations carried out primarily aim to measure the cost of confidential processing and the integration of enclaves into a distributed architecture. The observed performance remains relevant for analyzing the system's general behavior, regardless of future integration of mechanisms such as `RA-TLS`.

Finally, the experiments were conducted on limited hardware infrastructure, particularly given the memory constraints of `Intel SGX` enclaves. This choice, however, highlights the effects of hardware limitations on stateful workloads and complex collaborative operations more clearly. The trends observed in the experimental results remain consistent with the expected properties of `Intel SGX`-based hybrid architectures.

Chapter 6

Conclusion

This work addressed the challenge of collaboratively processing sensitive data streams in an unreliable distributed cloud environment. In many contexts, multiple organizations wish to correlate their data to perform joint real-time analyses, while maintaining strict control over the information actually disclosed to other participants and the execution infrastructure.

Modern stream processing engines, such as `Apache Flink`, offer powerful mechanisms for distribution, parallelism, and stream management, but they rely on a trust model in which data is generally accessible in plaintext to the execution infrastructure. This model becomes problematic in a multi-tenant context, handling sensitive data.

To address this issue, we have proposed a hybrid architecture combining a distributed stream processing engine with `Intel SGX` enclaves. The central idea of our approach is to retain the advantages of existing distributed processing engines for stream orchestration while delegating operations involving sensitive data to secure components running within enclaves.

Our main contribution is the design and implementation of an architecture that enables the execution of confidential collaborative computations without running the entire distributed engine in an `Intel SGX` environment. The `Apache Flink` engine serves primarily as an orchestration and transport component, while sensitive operations, notably joins between streams from multiple organizations, are executed exclusively within enclaves.

The experimental evaluation conducted shows that this approach is technically viable in a distributed real-time processing context. The results show that the overhead introduced by `Intel SGX` remains moderate for simple workloads but becomes more noticeable when processing involves stateful operations and joins across multiple streams.

Scaling experiments also reveal a significant distinction between observed maximum throughput and actual sustainable capacity. Although the system can continue

to produce high throughput under heavy load, workloads executed in enclaves reach a latency degradation zone more quickly, particularly for complex collaborative processing. The results thus underscore the trade-off between confidentiality and operational capacity under heavy load.

Despite these limitations, the results obtained demonstrate that it is possible to integrate Intel SGX enclaves into a distributed stream processing architecture without disrupting the overall organization of the processing engine. The proposed approach thus significantly reduces the exposure of sensitive data within the cloud infrastructure while maintaining an architecture compatible with existing processing tools and models.

Finally, this work should be viewed primarily as a feasibility validation. The implemented prototype is intentionally limited to a small subset of operations and does not constitute a complete secure SQL engine. Nevertheless, it demonstrates that a hybrid architecture separating distributed orchestration from confidential processing is a realistic approach to the secure, collaborative processing of multi-tenant streams.

6.1 Future work

Several areas for improvement and expansion can be considered following this work.

A first major development would involve the full integration of modern DCAP-based remote attestation mechanisms and the use of RA-TLS. Such an approach would enable the establishment of attested communication channels directly between tenants and enclaves, guaranteeing both the authenticity of the executed code and the confidentiality of exchanges. This integration would significantly strengthen the trust model of the proposed architecture.

A second avenue would involve extending the Intel SGX enclave's capabilities to support a broader set of relational operations and stream-processing tasks. The current prototype supports only a limited subset of operations, primarily simple transformations and explicit joins between two streams. Better integration with Apache Flink's high-level abstractions, notably Table API and SQL, would make the system more flexible and bring it closer to a full-fledged processing environment.

Improving state management is also an important area of focus. Stateful workloads are currently the main limiting factor for sustainable capacity under Intel SGX. More advanced mechanisms for memory management, state partitioning, or EPC access optimization could help reduce the impact of Intel SGX-imposed memory constraints.

Another avenue of exploration would be the study of the distributed deployment of multiple cooperative enclaves within the pipeline. The current prototype relies primarily on a centralized Intel SGX backend. A distributed enclave architec-

ture could improve scalability and enable more efficient distribution of sensitive processing in large-scale environments.

Bibliography

[1] European Union. Regulation (EU) 2016/679 of the European Parliament and of the Council. `https://eur-lex.europa.eu/eli/reg/2016/679/oj`, 2016. General Data Protection Regulation (GDPR), Official Journal of the European Union, L119. Last accessed on 2026-05-31.

[2] European Union. Directive (EU) 2022/2555 of the European Parliament and of the Council. `https://eur-lex.europa.eu/eli/dir/2022/2555/oj`, 2022. NIS2 Directive on measures for a high common level of cybersecurity across the Union. Last accessed on 2026-05-31.

[3] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, et al. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment*, 8(12):1792–1803, 2015.

[4] Apache Software Foundation. Apache flink documentation. `https://nightlies.apache.org/flink/flink-docs-stable/`, 2026. Last accessed on 2026-05-31.

[5] Chuan Zhao, Shengnan Zhao, Minghao Zhao, Zhenxiang Chen, Chong-Zhi Gao, Hongwei Li, and Yu an Tan. Secure multi-party computation: Theory, practice and applications. *Information Sciences*, 476:357–372, 2019.

[6] Gang Liang and Sudarshan S. Chawathe. Privacy-preserving inter-database operations. In Hsinchun Chen, Reagan Moore, Daniel D. Zeng, and John Leavitt, editors, *Intelligence and Security Informatics*, pages 66–82, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[7] Yaping Li and Minghua Chen. Privacy preserving joins. In *2008 IEEE 24th International Conference on Data Engineering*, pages 1352–1354, 2008.

[8] Yilei Wang and Ke Yi. Secure yannakakis: Join-aggregate queries over private data. In *Proceedings of the 2021 International Conference on Management of*

Data, SIGMOD '21, page 1969–1981, New York, NY, USA, 2021. Association for Computing Machinery.

[9] Victor Costan and Srinivas Devadas. Intel SGX explained. Cryptology ePrint Archive, Paper 2016/086, 2016.

[10] Intel Corporation. Life cycle of an sgx enclave. `https://www.intel.com/content/dam/develop/external/us/en/documents/intelsgxenclavelifecycle.pdf`, 2016. Last accessed on 2026-05-31.

[11] Thomas Knauth, Michael Steiner, Somnath Chakrabarti, Li Lei, Cedric Xing, and Mona Vij. Integrating remote attestation with transport layer security. *arXiv preprint arXiv:1801.05863*, 2018.

[12] Alexander Nilsson, Pegah Nikbakht Bideh, and Joakim Brorsson. A survey of published attacks on intel SGX. *CoRR*, abs/2006.13598, 2020.

[13] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache attacks on intel sgx. In *Proceedings of the 10th European Workshop on Systems Security*, EuroSec'17, New York, NY, USA, 2017. Association for Computing Machinery.

[14] Morris J Dworkin, Elaine Barker, James R Nechvatal, James Foti, Lawrence E Bassham, E Roback, James F Dray Jr, et al. Advanced encryption standard (aes). 2001.

[15] Carlos Segarra, Ricard Delgado-Gonzalo, Mathieu Lemay, Pierre-Louis Aublin, Peter Pietzuch, and Valerio Schiavoni. Using trusted execution environments for secure stream processing of medical data. In José Pereira and Laura Ricci, editors, *Distributed Applications and Interoperable Systems*, pages 91–107, Cham, 2019. Springer International Publishing.

[16] Aurélien Havet, Rafael Pires, Pascal Felber, Marcelo Pasin, Romain Rouvoy, and Valerio Schiavoni. Securestreams: A reactive middleware framework for secure data stream processing. In *Proceedings of the 11th ACM International Conference on Distributed and Event-Based Systems*, DEBS '17, page 124–133, New York, NY, USA, 2017. Association for Computing Machinery.

[17] Leandro Ventura Silva, Rodolfo Marinho, Jose Luis Vivas, and Andrey Brito. Security and privacy preserving data aggregation in cloud computing. In *Proceedings of the Symposium on Applied Computing*, SAC '17, page 1732–1738, New York, NY, USA, 2017. Association for Computing Machinery.

[18] Pau-Chen Cheng, Wojciech Ozga, Enriquillo Valdez, Salman Ahmed, Zhongshu Gu, Hani Jamjoom, Hubertus Franke, and James Bottomley. Intel tdx demystified: A top-down approach. *ACM Computing Surveys*, 56(9):1–33, 2024.

[19] Advanced Micro Devices, Inc. Amd secure encrypted virtualization (sev). `https://www.amd.com/en/developer/sev.html`, 2026. Last accessed on 2026-05-31.

[20] MQTT.org. Mqtt specification. `https://mqtt.org/mqtt-specification/`, 2026. Last accessed on 2026-05-31.

[21] Intel Corporation. Intel sgx linux driver. `https://github.com/intel/linux-sgx-driver`, 2025. Last accessed on 2026-05-31.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**