KafkaProxy: data-at-rest encryption and confidentiality support for Kafka clusters

Fábio Silva, Matteus Silva, Andrey Brito

[1]Laboratório de Sistemas Distribuídos – Departamento de Sistemas e Computação
Universidade Federal de Campina Grande (UFCG)
58.429-900 – Campina Grande – PB – Brazil

{fabiosilva,silvamatteus}@lsd.ufcg.edu.br, andrey@computacao.ufcg.edu.br

Abstract. *Apache Kafka has become a popular tool for building distributed systems. It supports a diversity of use cases that benefit from decoupled N-to-M communication such as publishing IoT data, decoupling and load-balancing microservices, and serve as a central hub for data in a distributed application. Nevertheless, Kafka's security is restricted to encrypted communications and authentication, leaving data unprotected in memory and on the disks. In this work, we design and implement a transparent, drop-in component that provides encryption to incoming and outgoing data in a Kafka cluster. Our component leverages confidential computing techniques not only to ensure data-at-rest encryption, but also to protect data and encryption keys from the operators of the Kafka Cluster. Our evaluation shows that the KafkaProxy can handle message streams with latency overhead of around* 10%*. Finally, in cases where throughput is impacted, simple replication of the KafkaProxy can mitigate the issue.*

1. Introduction

The publish-subscribe (pub-sub) communication paradigm is suitable for distributed applications of all sizes due to how it favors decoupling [Eugster et al. 2003]. In most implementations of such systems, data sources, named publishers, generate data in the form of events submitted to a set of brokers that intermediate the communication with the entities interested in those pieces of data. The consumers are named subscribers and register themselves with the brokers to signal their interest in certain data streams. The brokers' job is then to efficiently manage the storage and delivery of data, no matter if there is none or many interested subscribers.

In this work, we consider topic-based pub-sub systems [Eugster et al. 2003]. In these systems, the publications are organized into topics. Thus, publishers send messages to a topic, and subscribers register to one (or more topics). The most popular open-source system for pub-sub is Apache Kafka[1]. Apache Kafka is topic-based and has become very popular, being used as ingress and temporary storage for a wide range of applications, from IoT to large-scale microsservice ecosystems. Nevertheless, security in Kafka is restricted to encrypted communications and authentication, leaving data unprotected in memory and in the disks.

As more applications are built over richer and more sensitive data and use Apache Kafka as a hub, the lack of encryption creates a considerable surface area for attacks that

[1]https://kafka.apache.org/

steal or expose data. Although this problem may seem manageable at first sight, it is far from trivial. First, some companies provide proprietary solutions that transparently encrypt Kafka data before they are written to disk. Cloudera Navigator Encrypt[2] and Vormetric Transparent Encryption[3] are examples. Nevertheless, the data will be unencrypted in memory, where encryption keys will also be available. Thus, such approaches reduce risks such as data being recovered from decommissioned disks or data being easily (or acidently) exported from the machines, but still leaves the system vulnerable to attackers with access to the Kafka cluster machines, who can steal data from the memory.

Second, applications may want to encrypt data themselves, before publishing to the brokers. Following this approach requires data-producing and data-consuming applications to manage encryption keys, which is also not trivial as keys need to be distributed, rotated and revoked. Finally, some less-popular pub-sub alternatives (such as Apache Pulsar[4]) may facilitate the encryption by having client libraries to transparent encrypt messages. This simplifies key exchange protocols, but still requires complex coding and key-management protocols.

In this work, we propose KafkaProxy (KP), a component that sits between the Kafka cluster and its publishers and subscribers. It has the following features: (i) To enable transparency, it implements the Kafka protocol and, thus, sits transparently between the cluster and the clients; (ii) To protect data at rest, an instance encrypts the data that is published by the consumers, before it is handed to Kafka, and decrypts it before handing to subscribers; (iii) To protect encryption keys and enable simple and efficient revocation, it relies on Intel Software Guard eXtensions (SGX) [Mukhtar et al. 2019] to protect the integrity and confidentiality of the encryption and decryption processes.

The rest of the paper is organized as follows. In Section 2, we provide some contextualization of the related technologies and of our use case. In Section 3, we detail the construction of the KP. Section 4 evaluates the KP and Section 5 discusses related work. Section 6 summarizes the contributions and suggest some future works.

2. Background

In this section, we briefly present the publish-subscribe communication paradigm, Intel Software Guard eXtensions (SGX), confidential computing, and our use case.

2.1. Publish/Subscribe

Also known as distributed event-based systems, publish-subscribe systems enable decoupled communication, which fits the requirements of large distributed applications. Publishers, subscribers, and brokers compose this kind of system[5]. Publishers send structured messages (events), and the broker routes the messages to subscribers interested in that kind of message. The publish-subscribe paradigm offers decoupling in three dimensions:

- Space: there is no direct communication between the publishers and the subscribers; explicit addresses do not need to be known;

[2] `https://docs.cloudera.com/documentation/enterprise/latest/topics/navigator_encryption.html`

[3] `https://cpl.thalesgroup.com/encryption/vormetric-transparent-encryption`

[4] `https://pulsar.apache.org/`

[5] Although some implementations refrain from using a broker, this considerably impacts the decoupling needed for many microservice and IoT applications, with dynamic participants or intermittent connectivity.

- Time: subscribers will receive the produced messages eventually, without needing to be simultaneously online;
- Synchronization: the notification mechanism for the subscribers is asynchronous; that is, they are notified concurrently with other activities.

2.2. Cloud computing and data leaks

The adoption of cloud computing has grown in the last decade. Scalability, reduced costs, and flexibility are the main reasons for that growth and a considerable portion of enterprise workloads now executes in the cloud. Moreover, most companies choose public clouds because of the cost. As the usage of cloud computing increases, the risk of data breaches also increase, breaches which often happen as consequence of attacks to the infrastructure.

Adebayo defines a data breach as an incident in which sensitive, protected, or confidential data has potentially been viewed, stolen, or used by an unauthorized individual [Adebayo 2012]. Various strategies, alone or in combination, have been applied to prevent data breaches, such as secure communication channels and encrypting data at rest. Although these strategies make it difficult for an attacker to leak sensitive data, someone with privileged permissions on the infrastructure where the applications run can still access the data. Eavesdrop the data in processes or leaking the encryption keys used are examples of what someone with high privileges can do.

2.3. Intel SGX

Intel Software Guard eXtensions (SGX) is a set of instructions and changes in memory access mechanisms added to the x86 architecture. It is a hardware-assisted Trusted Execution Environment (TEE) that allows an application to create a protected area in the application address space referred to as an enclave [Mukhtar et al. 2019]. These protected memory regions have access control enforced by hardware. The SGX-enabled processor checks the operating system's memory mapping decisions, ensuring that non-enclave code cannot access enclave memory pages. Moreover, the processor encrypts all the memory pages in the Processor Reserved Memory (PRM) [Maene et al. 2018].

Unfortunately, the PRM is extremely scarce currently. The most common maximum size is 128 MB and only a few very recent processors have 256 MB. Moreover, the memory space available for enclave's memory pages, the Enclave Page Cache (EPC) is about 93 MiB [Weichbrodt et al. 2018]. If data or code loaded in all enclaves in a machine exceed this size, pages are swapped between the protected memory and the regular memory, which increases latency in orders of magnitude [Arnautov et al. 2016].

SGX's greatest advantages are the encrypted memory and the remote attestation. The remote attestation process allows an enclave to prove itself not modified and running in an SGX-enabled machine. Intel provides, along with the Application Enclave Service Manager (AESM), a distinct enclave, the Quoting Enclave (QE), capable of measuring another enclave and generating a *quote*. The *quote* is a verifiable structure that contains the enclave identity, also named as MRENCLAVE. The MRENCLAVE results from an SHA-256 hash function applied in all operations while the enclave is built. In this way, another software can check this identity against a reference value. To prevent falsifications, Intel also provides the Intel Attestation Service (IAS) to verify *quote*'s authenticity.

The basic approach to develop code for running inside Intel SGX enclaves is to use Intel SGX Software Development Kit (SDK). However, this is a difficult task as applications need to be rewritten to be split into secure and insecure parts, where the secure part cannot execute system calls.

One increasingly popular alternative to facilitate the execution of applications with SGX enclaves is SCONE (Secure CONtainer Environment) [Arnautov et al. 2016]. Based on a modified version of the `musl libc`, SCONE enables applications to run inside SGX enclaves without changes in the source code. While the SGX-SDK provided by Intel allows only the development using the C and C++ programming languages, SCONE delivers an environment for running applications written in other programming languages such as Python, Go, Fortran, Rust, and R, among others.

SCONE is compatible with the Docker platform and with other container orchestrators that use Docker, such as Compose and Kubernetes. Thus, running an application inside an enclave requires only a Linux machine, equipped with an SGX-enabled processor and BIOS, and a Docker environment. Moreover, SCONE also has support for remote attestation through the Configuration and Attestation Service (CAS). With the CAS, the provisioning of secrets to an application may be conditioned to the verification of the enclave's identity (MRENCLAVE).

2.4. An example use case

Our use case considers a typical monitoring architecture as depicted in Figure 1. Sensors (in our case, power meters) publish data to the pub-sub system. The sensors often make use of a gateway that offers a simpler interface and convert events to the Kafka protocol. If more powerful sensors are used, the gateway can be omitted. Once the power consumption events are in the message bus, several applications may subscribe to the event streams. In our example, an anomaly processor will detect relevant situations (potentially based on a window of events) and push high-level anomaly events back into the pub-sub. Meanwhile, a *Data Logger* component archives measurements for offline or long-term analysis and a *Load Disaggregation* component analyzes the measurements to detect patterns that indicate which appliances are in use by the consumer and how they are being used [Figueiredo et al. 2012]. Note that the data can reveal sensitive information not only for residential users, but also for businesses. Finally, a dashboard displays high level information, such as anomalies, bill predictions, or energy-efficiency recommendations.

2.5. Threat model

Our threat model assumes that an attacker has the goal of leaking as much data as possible. Our model include attacks and capabilities such as the following: (i) We assume that an attacker can gain administrative access to the cloud infrastructure (either the physical or virtual machines); (ii) We assume that if one gateway is compromised, the attacker will want to leak the encryption keys, so that he can also compromise another data sources. Nevertheless, handling Intel SGX's limitations is out of scope. Thus, we assume that proper firmware mitigation (latest Intel patches) are applied and that adequate measurements are applied to ensure some robustness to side-channel attacks (e.g., runtime checks for disabled hyper-threading and TSX support).

Our main goal is then to ensure that data is encrypted before published to Kafka

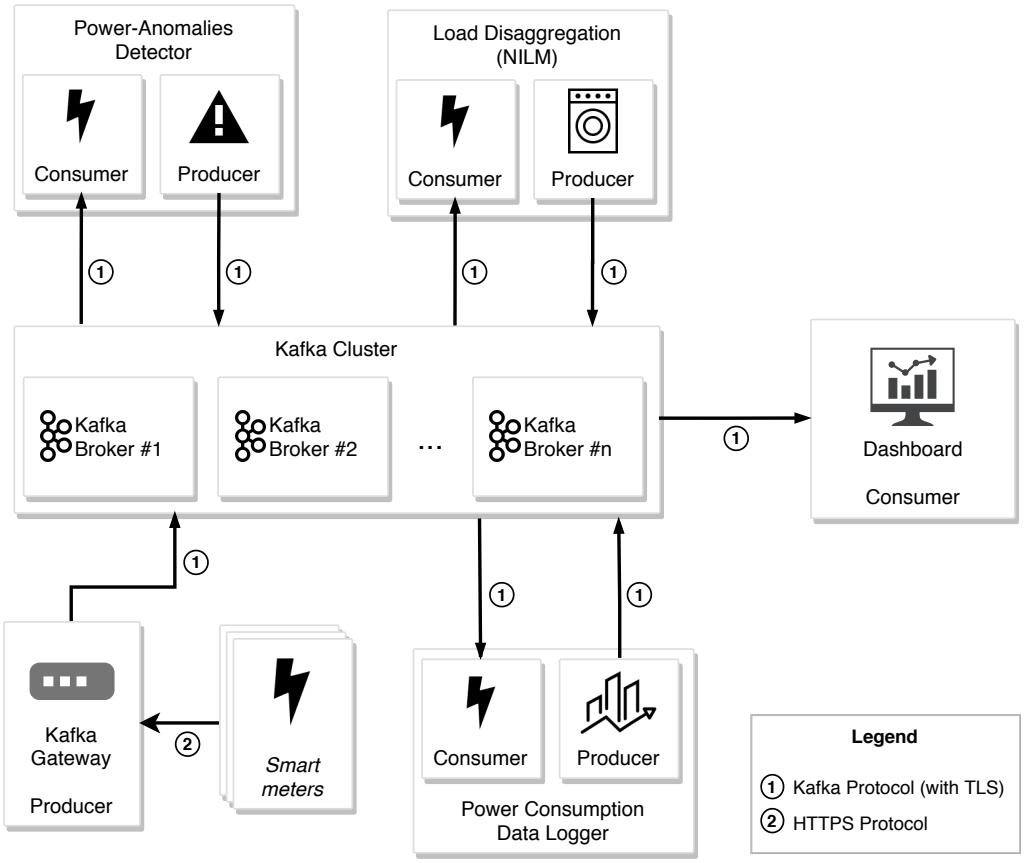

Figure 1. Architecture for the Use Case Application

and that even if a sensor, gateway, or consumer is compromised, encryption keys that would compromise other nodes will not be leaked.

3. The KafkaProxy

To secure Kafka client applications against the threats defined in Section 2.5, we designed the KP: a component that sits transparently between the Kafka cluster and the clients (or their gateways), and enables data-at-rest and memory encryption for the Kafka infrastructure. Figure 2 gives an overview of KP components: the *Catalog* and the *Message Authenticator*, which together with a database compose the *Key Management System* (KMS), and the *Message Interceptor*. These components will be detailed in the next sections.

Figure 2 also depicts two users. In a summary, the system works as follows: (i) users that controls the data source, i.e., the data owner (illustrated simply as *User*), registers information on the *Catalog*, which restricts potential consumer applications; (ii) the controller of an application needs to ask the data owner for authorization; (iii) the data owner updates the *Catalog* with the new allowed user; (iv) using a token, the new allowed user instantiates a KP, which will connect to the KMS and retrieve credentials and keys.

3.1. Key Management System (KMS)

The KMS maintains and serves the sensitive configurations related to Kafka client applications, including the Kafka credentials and encryption keys. It consists of the *Catalog* and *Authenticator* subsystems. The database implementation is customizable and is not

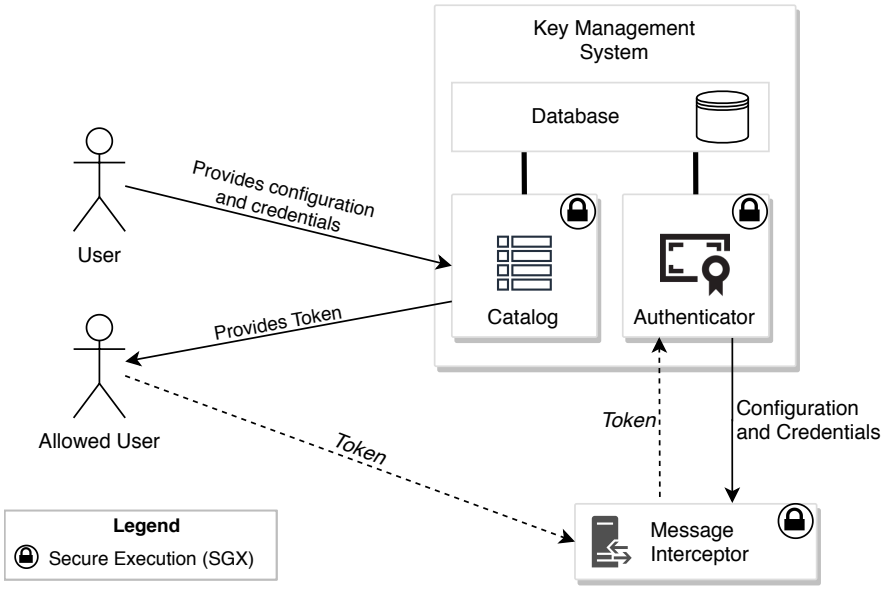

Figure 2. KP Overview

in the scope of this work. The security of the database can be through a third-party trusted service or a database that is run within SCONE[6].

Since the KMS is low-demanded and used only to provision settings and secrets, it was implemented using the Python programming language, hence benefits from the features of such a high-level language. Its execution is done through the SCONE platform to benefit from the hardware security guarantees provided by Intel SGX.

3.1.1. Catalog

The *Catalog* is the subsystem used for configuration and credentials provisioning. It is also responsible for issuing and managing authorization tokens to allow access to the provisioned secrets. The users must use X.509 public-key certificate to identify themselves through a TLS connection to the Catalog and certificates must be signed by a certification authority (CA) recognized by the KMS. Similar to how it is used originally in Apache Kafka, the user and application ids are based on the subject field of the certificate. For simplicity, users and application ids are the content of the Common Name (CN) field.

To store data on the *Catalog*, the user should provide a *catalog entry*. The *catalog entry* is a JSON structure, as seen in Listing 1, and contains configurations, metadata, and credentials, and is associated with a topic and a randomly-generated cryptographic key, allowing isolation of different applications or data domains. After provisioning, the randomly-generated key is returned to the user and should be stored in a safe place as it is required if the user needs to modify the *catalog entry* or recover their data (e.g. in case of a system crash). When configuring a new KP instance, an access token should be requested by an `allowed-user`, and provided to the *Message Interceptor*, so it can retrieve the proper *catalog entry*.

[6] `https://sconedocs.github.io/helm_mariadb/`

```
{
  "data-description": "Smartmeter data", // a brief data description
  "topic-name": "sm.data", // topic referred by this entry
  "allowed-users": [ // list of users allowed to request access tokens
    "sample-user", "another-user"
  ],
  "allowed-applications": [ // MRENCLAVEs of Interceptor instances
    "aa25d6e1863819fca72f4f3315131ba4a438d1e1643c030190ca665215912465",
    "9c56db536e046a5fb84a5c482ce86e6592071dff75dc0e3eb27d701cf2c40508"
  ],
  "expiration-time": 1440, // expiration time (in minutes)
  "app-pubkeys": [ // public keys of Interceptor instances
    {
      "name": "kafka-proxy-instance1",
      "pubkey": "<base64 encoded PEM public key>",
      "name": "kafka-proxy-instance2",
      "pubkey": "<base64 encoded PEM public key>"
    }
  ],
  "broker-credentials": { // credentials to access the broker
    "cert-pem": "<base64 encoded PEM X.509 certificate>",
    "cert-pkey": "<base64 encoded certificate private key>",
    "cert-pkey-secret": "<private key secret - optional>"
  }
}
```

Listing 1. Catalog entry sample

3.1.2. Authenticator

The *Authenticator* authenticates the applications and delivers the secrets if the authentication succeeds. The authentication process has a particular communication protocol, which is implemented by the *Message Interceptor*. The flowchart in Figure 3 describes the authentication process.

Using secure communication and authentication over TLS, to retrieve secrets and configurations, the client application must present (i) a public key certificate signed by a CA recognized by the *Key Management System*, (ii) a valid token, and (iii) its public key should be listed on *catalog entry*. If it complies with (i), (ii), and (iii), the *Authenticator* does a remote attestation process to ensure that the remote application has an expected SGX enclave (MRENCLAVE), running on a trusted platform. Passing all the checks, the *Authenticator* delivers the configurations and secrets.

3.2. Message Interceptor

The *Message Interceptor* is responsible for filtering publications and subscriptions made by Apache Kafka clients, as well as transparently ensuring confidentiality and integrity of the messages. This component should be interposed between the broker and the Apache Kafka clients, which can be producers, consumers, or gateways associated with them.

The credentials and cryptographic keys used by the *Interceptor* must be obtained through the authentication process with the KMS. The *Message Interceptor* authenticates

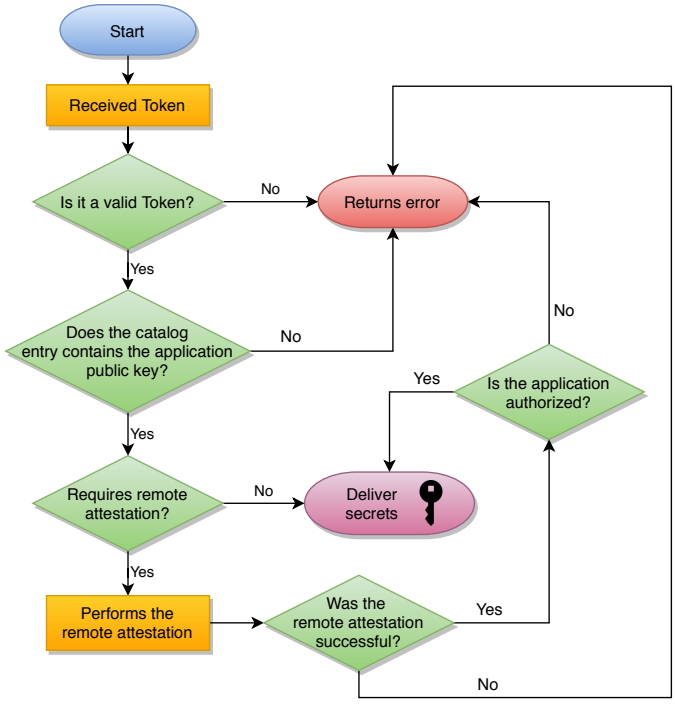

Figure 3. Authentication protocol flow

to the Apache Kafka using X.509 certificates received from KMS. In this way, the connection between the *Message Interceptor* and the message bus has integrity and confidentiality guarantees ensured by TLS.

The filtering process of publications and subscriptions restricts publishers and subscribers access to the topic defined in the *catalog entry* in the "topic-name" field. Published messages and subscriptions made for topics other than what was defined in the *catalog entry* are discarded. In this way, we can restrict the access of publishers and subscribers to their message domains.

The use of symmetric AES-GCM[7] (Galois / Counter Mode) encryption adds both integrity and confidentiality characteristics to the messages, supporting message authentication using a MAC (Message Authentication Code). All the messages that pass through the *Message Interceptor*, depending on the direction, are encrypted or decrypted transparently. Hence, the messages that circulate through the message bus are encrypted and subject to integrity verification.

The internal architecture of the *Message Interceptor*, as seen in Figure 4, is necessary especially when compiling with Intel SGX SDK, which requires having a trusted portion that cannot perform communication, and an untrusted portion that is the bridge with the I/O systems. In this case, the trusted side handles the security of the TLS communication, as well as the steps for authentication, and for encryption or decryption, so that no sensitive data or credentials leave the SGX enclave.

[7] `https://csrc.nist.gov/publications/detail/sp/800-38d/final`

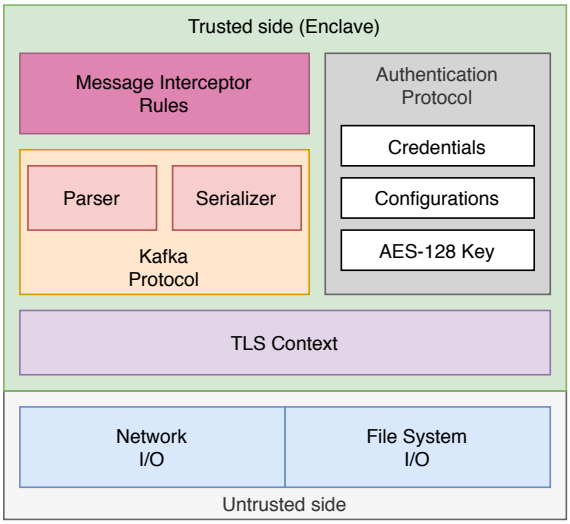

Figure 4. Message Interceptor architecture

4. Evaluation

We designed experiments to analyze the overhead introduced by the KP. More specifically, we address the following questions: (i) What is the overhead in throughput and latency? (ii) What is the added CPU usage?

We choose a complete factorial design to guide our experiments, considering the security configuration and the message size as independent variables. The message sizes were chosen considering the power monitoring application: 128, 256, 512, 1024, 2048, 4096, and 8192 bytes. For the security configuration we have the following four variants.

- **No proxy:** This is the native configuration (as depicted in Figure 1, without a security proxy).
- **Insecure:** In this configuration, we compiled the KP linking it against the standard GNU C Library, without any support for Intel SGX.
- **SGX SDK:** We compiled and ran KP using the Intel SGX SDK.
- **SCONE:** We compiled KP with the `scone-g++` compiler, linking the application against the SCONE `musl libc`. This configuration is executed with the support from the SCONE runtime.

4.1. Tools and Environment

To evaluate the KP, we took advantage of the tools distributed along with Apache Kafka: `kafka-producer-perf-test`, `kafka-consumer-perf-test` and `EndToEndLatency`. The first two were used to simulate a producer and a consumer, respectively. They were used to measure the throughput of message production and consumption. The `EndToEndLatency` simulates a Kafka producer and a Kafka consumer at the same time to measure the latency. In addition, we used the `pidstat` tool to collect data on CPU usage. The configuration of the virtual machines is shown in Table 1.

We run our experiments in an OpenStack cloud with support for creating virtual machines with SGX capabilities. We used the SGX-enabled machine to host the benchmarking tools and the KP. The other three machines were used to host the Kafka cluster, where a topic was created with a replication factor of three, and with nine partitions.

Protected memory size	Number of VMs	Configuration
Not needed	3	4 vCPUs, 8GB RAM
32 MB	1	2 vCPUs, 4GB RAM, SGX

Table 1. VM Flavors used in the experiments

4.2. Experiments

In the treatments where the configuration factor was set to *no proxy*, the performance tools were directly connected to the Kafka cluster. In the other cases, the performance tools send and receive messages through the KP. After the setup of the Kafka cluster we saved the cluster state. Thus, after each replica of the experiments the cluster was restored to its initial state to avoid interference from previous runs.

As depicted in Figure 5, in any configuration, the KP adds an overhead to message producers and message consumers. This is also detailed in Table 2.

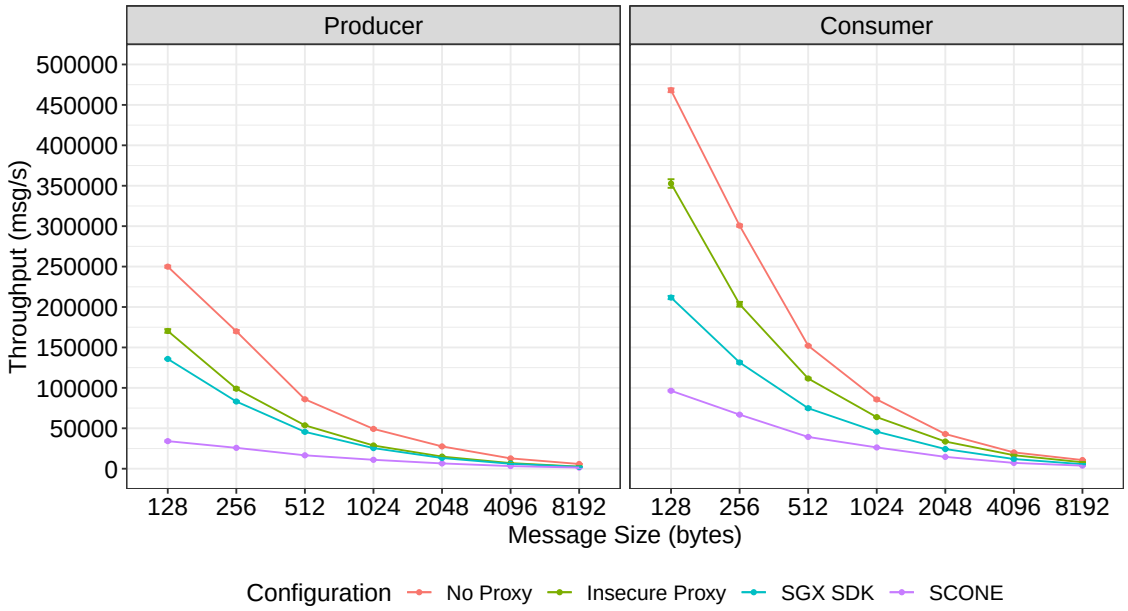

Figure 5. Throughput (msg/s) per message size

Message Size	Producer			Consumer		
	Insecure	SGX SDK	SCONE	Insecure	SGX SDK	SCONE
128	31,82%	45,67%	86,35%	24,65%	54,77%	79,40%
256	41,76%	51,08%	84,81%	32,35%	56,31%	77,76%
512	37,51%	46,89%	80,71%	26,68%	50,83%	74,19%
1024	41,66%	48,22%	77,63%	25,53%	46,52%	69,30%
2048	45,73%	52,07%	76,32%	21,71%	43,33%	65,95%
4096	45,74%	52,34%	74,65%	17,31%	40,67%	64,41%
8192	55,98%	60,41%	76,46%	25,59%	47,90%	64,79%

Table 2. Percentage reduction of throughput by using the KP

As the number of messages per second is clearly affected by the size of these

messages, we also evaluate the network throughput, as shown in Figure 6. The best network utilization is achieved for 2048-byte messages.

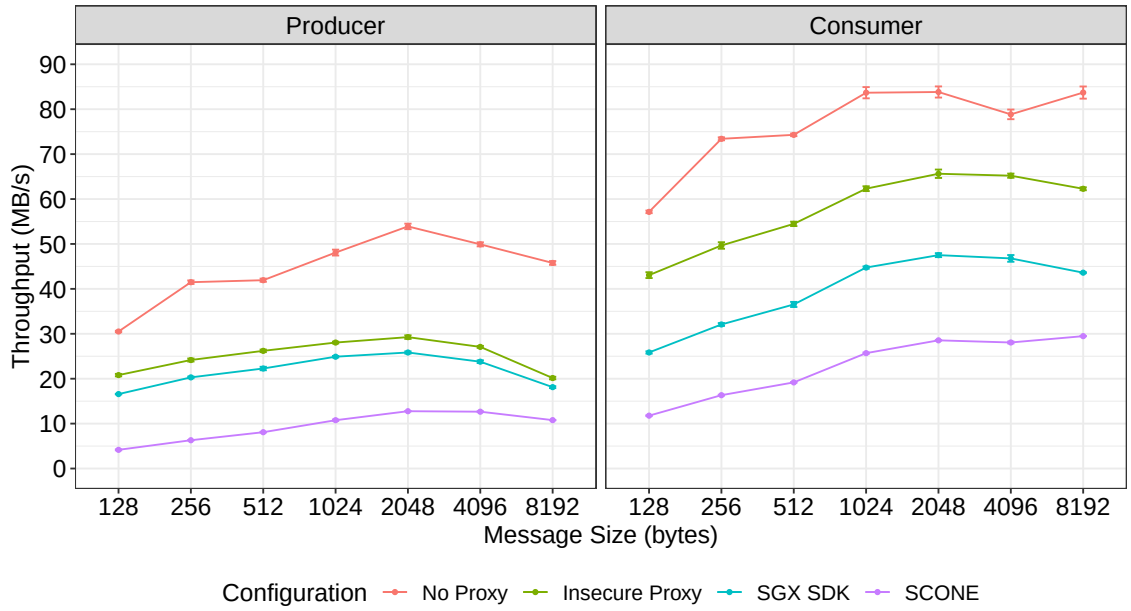

Figure 6. Network throughput (MB/s) per message size (bytes)

Also relevant is the CPU usage, which is depicted in Figure 7 in an experiment for the message size set to 2048 bytes, the one with the best throughput. The time axis is cut at the 25th second as the CPU usage stabilizes. Not surprisingly, the SGX configurations have higher CPU usage.

Next, Figure 8 depicts the end-to-end latency verified in our experiments. We can see that only the configuration using SCONE presents latency values considerably different to the other configurations. The overhead added by KP compiled with SGX SDK is about 2 *ms*, while with the KP running in the SCONE environment, this overhead is approximately 16.55 *ms*. Table 3 details the mean latency for each treatment.

Message Size	**Configuration**			
	No Proxy	**Insecure**	**SGX SDK**	**SCONE**
128	13,53 ms	14,54 ms	15,95 ms	27,33 ms
256	11,35 ms	12,01 ms	12,76 ms	26,24 ms
512	11,38 ms	11,79 ms	13,11 ms	29,17 ms
1024	10,53 ms	11,73 ms	12,77 ms	28,80 ms
2048	10,48 ms	11,45 ms	12,68 ms	26,84 ms
4096	10,27 ms	11,62 ms	12,41 ms	30,74 ms
8192	11,83 ms	13,32 ms	13,66 ms	26,11 ms

Table 3. Mean latency in different configurations

5. Related Work

There is a wide range of papers that address privacy in applications such as ours. Some works (for example, [Borges 2017, Barbosa et al. 2016]) focus on techniques such as

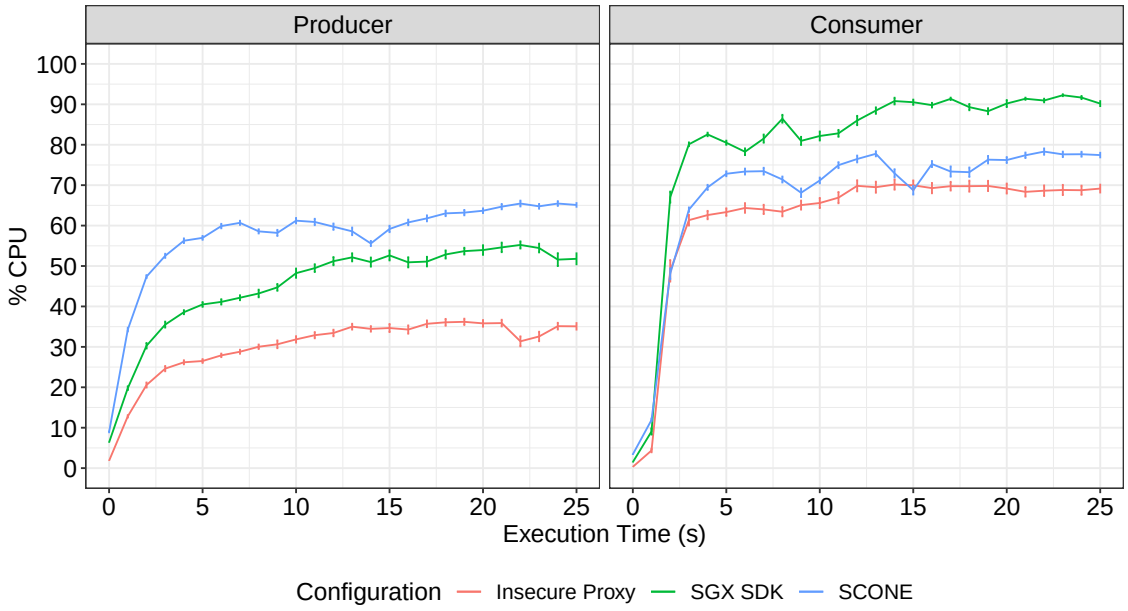

Figure 7. CPU Usage (%) over time (s)

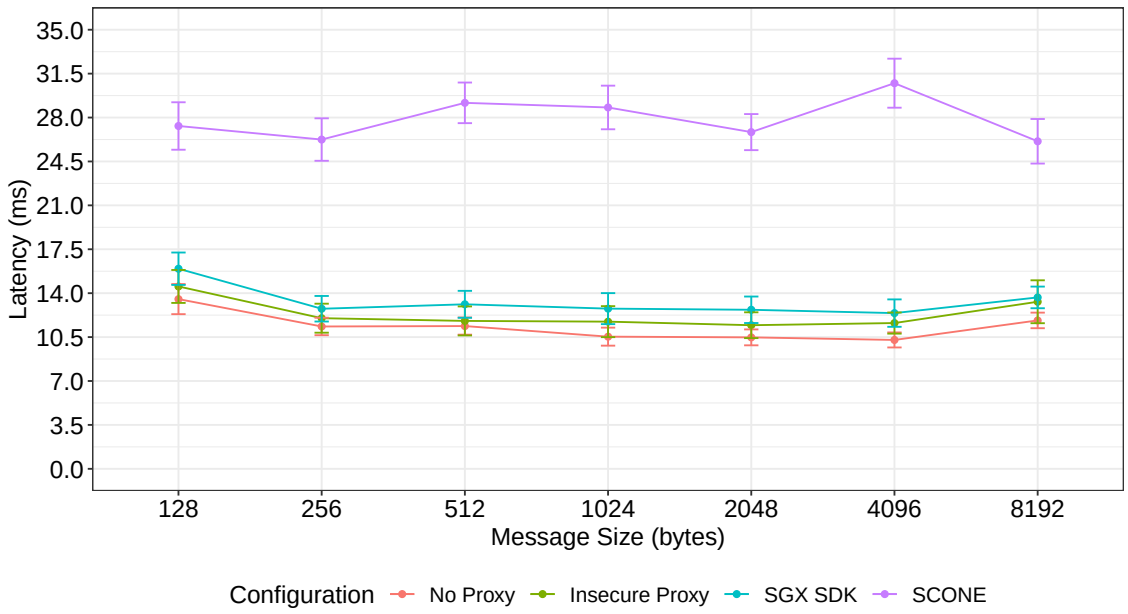

Figure 8. End-to-end latency (ms) per message size (bytes)

anonymization and homomorphic encryption. Anonymization techniques have the advantage of providing analytic-provable guarantees on the information that can be eventually leaked, but requires that the information is trimmed upfront, making it infeasible to later extend the application. In contrast, homomorphic encryption also has limited applicability, but due to practical issues. Efficient solutions exist only for specific computations (requiring 5-6 orders of magnitude of overhead for generic computations) and there are not practical tools for converting arbitrary code. In this work we focus on using protection mechanisms that are based on trusted execution environments as they are computationally efficient and preserve the information on the data.

In the direction of using trusted execution environments to provide confidentiality support to similar application scenarios there are also several related works. For example, Silva et al. [Silva et al. 2016] present the usage of Intel SGX to perform specific power-data analysis and compare resource usage with solutions based on homomorphic encryption. Pires et al. [Pires et al. 2016] propose SCBR, a content-based routing system that runs entirely inside Intel SGX enclaves and Sampaio et al. [Sampaio et al. 2017] leverage SCBR for building a system for processing power data with confidentiality guarantees. The works from Pires et al. and Sampaio et al. provide pub-sub capabilities, but as with the work from Silva et al., put the complete software stack inside the enclave, compromising the scalability of the pub-sub system due to the very limited availability of the protected memory (as discussed in Section 2). In our work, we use a popular and scalable message bus, that may have to be hosted in the cloud in other to support a large scale distributed application. In our solution, the KP would be typically run closer to the producers and, thus, the Kafka cluster and the KPs could be scaled independently.

Finally, as mentioned before, enterprise add-ons to Kafka such as Cloudera Navigator Encrypt and Vormetric Transparent Encryption could add data-at-rest encryption, but would not prevent data or keys to be stolen from memory.

6. Conclusion

In this paper, we proposed a strategy to ensure integrity and confidentiality for publish-subscribe applications that does not require applications to be modified. We designed a set of components that run in trusted execution environments (Intel SGX) for credential management and cryptographic operations. Also, we implemented the proposed components and validated them with experiments to understand the overhead of the security layer offered. Our results show that following such approaches enables leveraging the best of both worlds: On the one hand, off-the-shelf, high performance and easy-to-use Kafka clusters; On the other hand, data-at-rest encryption and simple key management for applications that handle sensitive data. Finally, our experiments evaluate the performance tradeoff of developing applications with the Intel SGX SDK, which provides performance advantages at the cost of leaving more responsibilities to the developers, versus using SCONE, where some performance and TCB space is traded for improved maintainability and support to some SGX limitations (e.g., some side channel attacks).

Acknowledgements

This work has been supported by the RNP Workgroup Program (GT), by Smartiks Ltda., and by the Brasilian Agency for Industrial Innovation (EMBRAPII) under the project GT-LiteCampus.

References

Adebayo, A. O. (2012). A foundation for breach data analysis. *Journal of Information Engineering and Applications*, 2(4):17–23.

Arnautov, S., Trach, B., Gregor, F., Knauth, T., Martin, A., Priebe, C., Lind, J., Muthukumaran, D., O'Keeffe, D., Stillwell, M. L., Goltzsche, D., Eyers, D., Kapitza, R., Pietzuch, P., and Fetzer, C. (2016). SCONE: Secure linux containers with intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 689–703, Savannah, GA. USENIX Association.

Barbosa, P., Freitas, L., Brito, A., and Silva, L. (2016). Privacy preserving techniques in smart metering: An overview. In *Proceedings of the 16th Brazilian Symposium on Information and Computational Systems Security*. Sociedade Brasileira de Computação.

Borges, F. (2017). *On Privacy-Preserving Protocols for Smart Metering Systems: Security and Privacy in Smart Grids*.

Eugster, P. T., Felber, P. A., Guerraoui, R., and Kermarrec, A.-M. (2003). The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131.

Figueiredo, M., de Almeida, A., and Ribeiro, B. (2012). Home electrical signal disaggregation for non-intrusive load monitoring (nilm) systems. *Neurocomputing*, 96:66 – 73. Adaptive and Natural Computing Algorithms.

Maene, P., Götzfried, J., de Clercq, R., Müller, T., Freiling, F., and Verbauwhede, I. (2018). Hardware-based trusted computing architectures for isolation and attestation. *IEEE Transactions on Computers*, 67(3):361–374.

Mukhtar, M. A., Bhatti, M. K., and Gogniat, G. (2019). Architectures for security: A comparative analysis of hardware security features in intel sgx and arm trustzone. In *2019 2nd International Conference on Communication, Computing and Digital systems (C-CODE)*, pages 299–304.

Pires, R., Pasin, M., Felber, P., and Fetzer, C. (2016). Secure content-based routing using intel software guard extensions. In *Proceedings of the 17th International Middleware Conference*, Middleware '16, New York, NY, USA. Association for Computing Machinery.

Sampaio, L., Silva, F., Souza, A., Brito, A., and Felber, P. (2017). Secure and privacy-aware data dissemination for cloud-based applications. In *Proceedings of The10th International Conference on Utility and Cloud Computing*, UCC '17, page 47–56, New York, NY, USA. Association for Computing Machinery.

Silva, L., Marinho, R., Brito, A., and Barbosa, P. (2016). Agregação de dados na núvem com garantias de segurança e privacidade. In *Proceedings of the 16th Brazilian Symposium on Information and Computational Systems Security*. Sociedade Brasileira de Computação.

Weichbrodt, N., Aublin, P.-L., and Kapitza, R. (2018). sgx-perf: A performance analysis tool for intel sgx enclaves. In *Proceedings of the 19th International Middleware Conference*, pages 201–213.