Integrated implementation of the micro-services for distributed big data applications

D4.4

Due date: 31 December 2018
Submission date: 21 January 2019

Start date of project: 1 January 2016

Document type: Deliverable
Work package: WP4

Editor: Marcelo Pasin (UniNE)
Reviewer: Andrey Brito (UFCG)
Stefan Köpsell (TUD)

Dissemination Level

<table>
<thead>
<tr>
<th>PU</th>
<th>Public</th>
<th>√</th>
</tr>
</thead>
<tbody>
<tr>
<td>CO</td>
<td>Confidential, only for members of the consortium (including the Commission Services)</td>
<td></td>
</tr>
<tr>
<td>CI</td>
<td>Classified, as referred to in Commission Decision 2001/844/EC</td>
<td></td>
</tr>
</tbody>
</table>

SecureCloud has received funding from the European Union’s Horizon 2020 research and innovation programme and was supported by the Swiss State Secretariat for Education, Research and Innovation (SERI) under grant agreement No 690111.
<table>
<thead>
<tr>
<th>Task No.</th>
<th>Task description</th>
<th>Partners involved</th>
</tr>
</thead>
<tbody>
<tr>
<td>T4.1</td>
<td>Secure distributed communication mechanisms</td>
<td>UniNE, IMP, CC</td>
</tr>
<tr>
<td>T4.2</td>
<td>Secure distributed data management and storage</td>
<td>CC, IMP, UniNE, UTFPR</td>
</tr>
<tr>
<td>T4.3</td>
<td>Distributed scheduling mechanisms</td>
<td>UniNE, TUD, IMP, CC</td>
</tr>
<tr>
<td>T4.4</td>
<td>Secure map/reduce computation</td>
<td>UniNE, IMP, TUD, CC</td>
</tr>
</tbody>
</table>

*This task list may not be equivalent to the list of partners contributing as authors to the deliverable
*Task leader
## Contents

1 **Introduction** 2

2 **Secure Communication Mechanisms** 3
   2.1 Secure content-based routing 3
   2.1.1 Description 3
   2.1.2 Validation and Integration 4
   2.2 Identity-based broadcast encryption using SGX 9
   2.2.1 Description 10
   2.2.2 Validation and Integration 11

3 **Distributed Data Store** 17
   3.1 An overview of the Secure Key-Value Store 17
   3.2 The role of the Secure Key-Value Store inside SecureCloud 18
   3.2.1 Evaluation 19
   3.2.2 Integration 20
   3.3 Updated components 21
   3.3.1 TLS interceptor 21
   3.3.2 Kubernetes integration 23

4 **Scheduling** 24
   4.1 SGX-Aware scheduler 24
   4.1.1 Description 24
   4.1.2 Validation and Integration 25

5 **Secure computation** 28
   5.1 SecureStreams 28
   5.1.1 Description 28
   5.1.2 Validation and Integration 29
   5.2 Lightweight MapReduce 30
   5.2.1 Description 31
   5.2.2 Validation and Integration 34
   5.3 Secure Apache Spark Executor 38
   5.3.1 Description 38
   5.3.2 Validation and Integration 38

6 **Summary** 41
1 Introduction

This deliverable presents a catalogue of components implemented in Work Package 4 throughout the execution of SecureCloud. Figure 1.1 recalls the global SecureCloud architecture as defined in D1.3, where the components presented in this deliverable belong to the level of the Platform Services. They are a set of secure components for communication, storage, scheduling and big data processing. For distributed communication, we present here SCBR (Secure Content-Based Routing) in Section 2.1 and IBBE (Identity-Based Broadcast Encryption) in Section 2.2. For data management and storage, we present the Distributed Data Store in Chapter 3. For distributed scheduling, we present an extension to Kubernetes in Chapter 4. Finally, for big data processing, we present SecureStreams in Section 5.1, Lightweight MapReduce in Section 5.2 and Secure Apache Spark Executor in Section 5.3.

The components cited above are described in the remaining chapters of this deliverable. All these components have another longer, integrated, description in D1.3, including motivation, solution, and functionality. Each component description follows the same template, with a section containing its description, and a section containing validation, evaluation and integration efforts developed during the project. The Distributed Data Store has evolved in the third year, therefore it presents its new developments in Section 3.3.

In concurrency with this deliverable, for what is produced as open source, we provide all final implementations of the secure communication, distributed storage, map/reduce, and scheduling components. These final versions have been validated using continuous integration and validation mechanisms of WP1 and aiming for the use cases of WP5. Deliverable D1.3 (Requirements & Architecture Specification — Final Version) provides many more details on components validation and integration from a global point of view.
2 Secure Communication Mechanisms

This chapter summarises the communication components developed in Work Package 4 of SecureCloud. It closes the work developed in Task 4.1, Secure distributed communication mechanisms (month 1 to month 36). The components described here are Secure Content-Based Routing (SCBR, Section 2.1) and Identity-Based Broadcast Encryption using SGX (IBBE-SGX, Section 2.2). Although TaLoS is as well a platform-level communication component proposed in SecureCloud, it has been developed in a much lower level, placing standard SSL libraries inside a trusted environment. Therefore, TaLoS is delivered as part of Work Package 3, and has been extensively described in Deliverable D3.2.

2.1 Secure content-based routing

Content-based routing (CBR) is a flexible paradigm for scalable communication among distributed processes. It decouples data producers from consumers, and dynamically routes messages based on their content. While the publish/subscribe communication model has been extensively studied and many implementations have been proposed, it still fails to reach wide deployment and usage in the era of cloud computing.

CBR requires router components to filter messages by matching their content against a (potentially large) collection of subscriptions that act as a reverse index and, hence, must be stored by the filtering engines. This process requires the router to see the content of both the messages and the subscriptions, which represents a major threat for companies which deal with sensitive data.

In SecureCloud we proposed SCBR, which stands for Secure Content-Based Routing. It has been initially described in the revised Deliverable D4.1. SCBR combines a key exchange protocol and a state-of-the-art routing engine to provide both security and performance. To accomplish that, it takes advantage of the trusted execution environment provided by SGX enclaves.

2.1.1 Description

SCBR has been conceived under a simple model with few assumptions. The publish/subscribe system operates as a service under the control of a single provider that publishes data. Consumers are the clients of the service and typically pay recurring fees to have access to the data. We assume that there are ways for the producers to control who the subscribers can be (joining and leaving the service). The publishers operate within the administrative domain of the provider from which data originates, and they are trusted by the clients for the purpose of the considered service. This model closely maps most scenarios developed in SecureCloud use cases.

Given the trust relationships between the different components of the system, publishers and clients must share cryptographic keys that are not known by the infrastructure. There should also be a lightweight mechanisms for the publishers to (dis-)allow clients from accessing new data, independently of whether they had been legitimate customers in the past.

We distinguish three main roles in SCBR architecture, illustrated in Figure 2.1:

- The service provider (or data provider) produces information flows for the clients, typically “as a service” and for a fee. Data may be produced by multiple sources (publishers) operating within the same domain.

- The infrastructure provider hosts the CBR engines in the cloud. It provides secure hardware (SGX-based enclaves) and performs the actual routing through its network. As it operates under a different administrative domain and may share its resources among several customers (in a
multi-tenant configuration), the infrastructure provider is trusted neither by the data provider nor by its clients.

- The clients of the service are the end users who are interested in the actual data and subscribe to information flows via the CBR engines. They trust the data providers but not the infrastructure.

Messages are composed of a payload, which is of interest to the end users but opaque to SCBR, and a header that contains several attributes and associated values. SCBR filters messages based on the attribute values in their header.

Subscriptions are composed of predicates specifying constraints over the attributes. Predicates can include equality constraints or any kind of ranges over the values of the attributes. For example, a subscriber interested in stock quotes for the company HAL when they reach a price level below 50 euros can register a subscription such as "symbol = "HAL" ∧ price < 50". We say that a message matches a subscription if its header satisfies the constraints expressed in the subscription predicate. SCBR uses a storage algorithm that exploits subscription containment to minimize the footprint of the subscriptions stored in the enclave, where only a limited amount of memory is available.

More details concerning implementation were given in Section 4.2.4 of the revised Deliverable D4.1. As presented there, SCBR leverages state-of-the-art techniques for efficient filtering in plaintext, since the trusted perimeter is limited to the CPU die. Outside that boundary, private data is always encrypted and protected from tampering and replay attacks, even from operating systems, hypervisors, and administrators with physical access to machines. As a result, we do not suffer from the prohibitive performance and space overheads of software-based secure approaches, such as homomorphic encryption.

2.1.2 Validation and Integration

Evaluation

We evaluated SCBR with both the producer and consumer running in one machine and the filtering engine in another. Measurements were collected at the machine running the filter, which was equipped with an Intel Skylake CPU model i7-6700 running at 3.4 GHz with an 8 MB cache and 8 GB of main memory. We allocated 128 MB of main memory to EPC (maximum allowed).
To evaluate SCBR and to facilitate comparison, we reused the workloads from previous work [6] composed of 9 datasets. They were built based on real data corresponding to randomly selected stock quotes from the Yahoo! finance website. Approximately 250,000 entries were collected in a period of 5 years, with publications composed by 8 to 11 attributes. The entries collected were used to produce synthetic subscription datasets containing an assortment of equality and range predicates on the quotes’ attributes. Besides, subscriptions were selected according to (1) a uniform random distribution and (2) a Zipfian law with exponent $s = 1$. In order to assess the algorithms’ performance with a greater number of variables and different levels of containment, some workloads were synthesised with twice and four times the number of attributes of the original publications, by merging data from multiple quotes. Table 2.1 summarises the characteristics of the datasets used.

Our first experiment aimed at evaluating the performance overhead caused by executing our filter inside an enclave. We filled the subscription database with datasets with 1,000, 2,500, 5,000, 10,000, 25,000, 50,000 and 100,000 subscriptions, reaching a total memory size of approximately 43MB for the largest dataset. Thereafter, we sent a batch of 1,000 publications to be matched against the subscriptions and measured the time it took to accomplish each filtering operation. We ran an identical setup with and without encryption, inside and outside an enclave, using the same filtering code. When using encryption, publications and subscriptions were encrypted in the producer and decrypted in the filter using AES-CTR. The average results for the first workload ($e100a1$) are shown in Figure 2.2. By considering the proximity of the lines with and without encryption, we can see that encryption overhead is small and nearly constant. Indeed, this overhead remains below $5\mu s$ both inside or outside the enclave, which is negligible when compared to the matching time given a reasonable large database size. The overhead resulting from the enclave is more significant, reaching nearly 40% for the largest set of subscriptions considered in our experiments, which is explained by the occurrence of cache misses (cache size is shown by the vertical line).

We then focused on the influence of the workloads. In order to understand the effect of different datasets on SCBR performance, we first executed each of them without encryption outside secure enclaves. Results are shown in Figure 2.3. The first ($e100a1$) and last ($e100a1zz100$) workloads show the best performance, since all subscriptions contain equality predicates and the subscription set forms deeper

---

<table>
<thead>
<tr>
<th>Workload name</th>
<th>Proportion of equality predicates</th>
<th>Number of attributes</th>
<th>Subs values distribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e100a1$</td>
<td>100% : 1 eq. pred.</td>
<td>8–11 (original)</td>
<td></td>
</tr>
<tr>
<td>$e80a1$</td>
<td>20% : 0 eq. pred.</td>
<td>2x more</td>
<td></td>
</tr>
<tr>
<td>$e80a2$</td>
<td>80% : 1 eq. pred.</td>
<td>4x more</td>
<td></td>
</tr>
<tr>
<td>$e80a4$</td>
<td>15% : 0 eq. pred.</td>
<td>2x more</td>
<td>Uniform</td>
</tr>
<tr>
<td>extsub2</td>
<td>60% : 1 eq. pred.</td>
<td></td>
<td>Zipf on symbol</td>
</tr>
<tr>
<td>extsub4</td>
<td>15% : 2 eq. pred.</td>
<td></td>
<td>Zipf on all attributes</td>
</tr>
<tr>
<td>$e80alz100$</td>
<td>20% : 0 eq. pred.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$e80alzz100$</td>
<td>80% : 1 eq. pred.</td>
<td>8–11 (original)</td>
<td></td>
</tr>
<tr>
<td>$e100alzz100$</td>
<td>100% : 1 eq. pred.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2.1: Workloads description (adapted from [6]).
containment trees. In contrast, datasets with more attributes (e80a4 and extsub4) perform worse because they yield indexes with more roots and shallow trees, therefore inducing more comparisons to traverse the whole database.

Figure 2.3: Performance of the containment-based algorithm applied to the different workloads in plaintext, outside enclaves.

Figure 2.4 displays separate measurements for each workload running SCBR inside and outside an enclave (both using AES encryption). We also measured, for each workload, the performance of our implementation of ASPE [12, 6] as a baseline for a software-only alternative that does not use enclaves. We measured only the matching step, and not the encryption or decryption of ASPE messages. The presented ASPE performance cost was therefore inherent to its matching algorithm, which grows faster than any other strategy when increasing the size of the subscription database. The difference is more substantial for the first and last workloads, although it remains close to at least one order of magnitude in all setups. These observations indicate that the performance penalties of SGX are largely tolerable when considering software-only alternatives for secure filtering, at least when the amount of memory used by the routing engine remains below the EPC assigned size. We will further explore this matter below.
Another interesting aspect is the gap between the curves corresponding to execution inside and outside the enclaves. After approximately 10,000 subscriptions, the versions inside and outside enclaves begin to drift apart due to the number of memory accesses necessary to accomplish every comparison. At some point, the filtering data does not fit completely in the processor’s cache memory and cache misses start to occur more frequently. When this happens, data must be fetched from system memory and, in the case of enclave executions, it must be decrypted and checked for integrity and freshness. Moreover, the evicted enclave’s cache node must be encrypted before being sent to system memory. That behavior is consistent with cache miss rates (measured outside the enclaves), which are also reported in Figure 2.4. We only measured cache misses outside the enclaves, because our Linux version failed to properly monitor the cache performance counters inside enclaves. Since the code running inside and outside the enclaves is the same, it is reasonable to assume that cache miss rates would be similar.

Figure 2.4: Comparison of different approaches with varying workloads.

We finally wanted to observe the performance penalties when exceeding the maximum protected memory size and paging begins to happen. Since EPC memory is limited, whenever it is full and more space is required, pages must be evicted from the protected area to the main (untrusted) memory.
Accordingly, a page swap occurs every time a previously evicted page is accessed. Besides the fact that system memory is slower than the processor’s cache, which already imposes performance costs, memory page swaps are serviced by the operating system and hence incur an even higher overhead.

![Figure 2.5: Loss in performance when exceeding EPC memory limit.](image)

Figure 2.5 shows the combined results of two executions when populating the in-memory subscription storage. In one execution we registered subscriptions inside an enclave, and outside in the other. We used the workload e80a1 in plaintext format, and we executed the same registration code in both experiments. Each point of the graph accounts for an average of 5,000 points, from a set of 500,000 subscriptions. We plotted the page fault rates observed by dividing their numbers from inside and outside enclaves. The values measured outside are very large for the largest database size, reaching up to 40,000 more page faults.

We also divided the time it took to register one subscription inside the enclave by the time required outside. We can clearly see the point where paging starts to take place, when memory consumption reaches just over 90 MB. The vertical line shows the EPC memory limit, which comprises both the enclaved application memory and SGX internal data structures. At the maximum size of our experiment (213 MB), registering a subscription inside the enclave took 18 times more time than doing it outside. These results show that the overhead grows outrageously when paging starts to happen, and they make a strong case for further studies on optimising the memory footprint of applications running inside secure SGX enclaves.

Finally, content-based publish/subscribe appears at first sight to be incompatible with privacy preservation, as by definition messages must be filtered based on their content, i.e., the filtering engine should be able to see both the data of publications and subscriptions. To make the problem even more challenging, many filtering techniques exploit structural properties of the information exchanged between publishers and subscribers. In particular, by structuring the containment relations between subscriptions, one can build efficient data structures to store subscriptions and match publications. Containment allows for a significant reduction of the number of subscriptions stored as well as the number of matching evaluations executed per publication. As a consequence, containment is used in most CBR systems in use today [10, 11, 26], which makes them largely incompatible with classical cryptographic techniques for privacy preservation.
SGX was introduced only very recently and only few practical algorithms using TEEs have been already proposed. As far as we could reach in the available literature, SCBR is the first attempt to demonstrate the practical benefits of SGX for privacy-preserving content-based routing. But if using SGX for CBR is new, secure content-based publish subscribe is not. There is a vast amount of literature on the subject, including major surveys [31, 36]. One of these surveys classifies publish/subscribe systems preserving confidentiality in two categories, depending on whether they leverage traditional security techniques or they rely on special-purpose forms of encryption [31]. All encryption schemes presented in the surveys are either heavier or weaker than standard production-level encryption (e.g., AES). By using symmetric encryption and plaintext matching under trusted execution, SCBR is able to combine the best of both worlds. It provides a novel scheme that (i) supports confidentiality for events and filters; (ii) permits publishers to express further constraints about who can access their events; (iii) handles filters that can express very complex constraints on events even if brokers are not able to access any information in clear on both events and filters; (iv) allows brokers to use state-of-the-art containment-based filtering, and finally (v) does not require publishers and subscribers to share any keys.

Integration
SCBR has been integrated in three use cases implemented in SecureCloud: (1) Smart Meter Reading and Management, (2) Fault Analysis and (3) Fraud Detection. SCBR is used inside the Smart Meter Reading and Management use case, in order to connect the MDC with the modules Archivist and Measurement Report Writer. MDC, who acts as a publisher, is a standard component in legacy Smart Metering applications, usually being responsible for many functions as reading, configuring, controlling, and managing smart meters and interfacing with a database. The first subscriber, the Archivist, is responsible for storing each measurement in the secure key value store, for archival purposes. The second component subscribed to the SCBR is responsible to write the measured values into an unstructured batch of files, that are later used by billing components. Thanks to its fundamental structure (it is able to deal with multiple subscribers), other modules could profit from the flow of messages published by the MDC. Figure 2.6 (see Deliverable D1.3, Chapter 5) shows how all the modules from this use case are connected. Finally, Deliverable D5.4 also describes the integration of SCBR with the metering application demonstrator.

The structure of the implementation of the Fault Analysis use case is presented in Figure 2.7 (see Deliverable D1.3, Chapter 5), where the communication between the services is handled by SCBR. The core of the application consists of fault analysis microservices. The Fraud Detection use case uses the same structure, only replacing the central microservices.

2.2 Identity-based broadcast encryption using SGX
In order to support secure multi-party communication and collaborative editing on private data, we developed another service as part of WP4 that enforce access control policies. This service, which was published in [13], is described in the remaining of this section.
2.2.1 Description

While many cloud storage systems allow users to protect their data by making use of encryption, only few support collaborative editing on that data. A major challenge for enabling such collaboration is the need to enforce cryptographic access control policies in a secure and efficient manner.

To address this challenge, we developed IBBE-SGX, a new cryptographic access control extension that is efficient both in terms of computation and storage even when processing large and dynamic workloads of membership operations, while at the same time offering zero knowledge guarantees. Zero knowledge is guaranteed by executing the cryptographic access control membership operations in a TEE.²

IBBE-SGX builds upon identity-based broadcasting encryption (IBBE). We address IBBE’s impracticality for cloud deployments by exploiting Intel SGX to derive cuts in the computational complexity. Our solution is to execute the membership operations of IBBE within the TEE so that we can make use of a master secret key. The TEE guarantees that this secret stays within the trusted computing boundary. We can therefore propose an optimization of a well-studied IBBE scheme that drastically reduces its computational complexity.

A remaining issue is the computational complexity required for users to derive membership changes. To mitigate this last aspect, we propose a group partitioning mechanism such that the computational cost on the user-side is bound to a fixed constant partition size rather than the potential large group size.

We have implemented and evaluated our new access control extension. Moreover, we deployed our system on a commercially-available public cloud storage. Results highlight that IBBE-SGX performs membership changes 1.2 orders of magnitude faster than the traditional approach of hybrid encryption (HE), producing group metadata that are 6 orders of magnitude smaller than HE, while at the same time offering zero knowledge guarantees.

²Only membership operations rely on the TEE, user operations are done in a conventional execution environment.
2.2.2 Validation and Integration

We consider groups of users who perform collaborative editing on cryptographically-protected data stored on untrusted cloud storage systems. The data is protected using a block cipher encryption algorithm such as AES making use of a symmetric group key $g_k$. As illustrated in Figure 2.8, this work addresses the challenge of designing a system for group access control, in which the group key $g_k$ is cryptographically protected and derivable only by the members of the group. Because groups may become large with a significant turnover in their members, we investigate the implication of numerous member additions and revocations happening throughout their lifetimes.

We distinguish between two types of actors interacting within the system: administrators and users. All group membership operations are performed by administrators. Their duties include creating groups, and adding or revoking group members. The administrators manifest an honest-but-curious behavior, correctly serving work requests but with a possible malicious intent of discovering the group key $g_k$. On
Deliverable 4.4 Integrated implementation of microservices for big data applications

the other hand, users listen to the cloud storage for group membership changes, and derive the new group key \( g_k \) whenever it changes. Users are considered of having a trusted behavior.

The role of the cloud storage is to store the definitions of groups access control, also referred to as groups metadata, together with the list of members composing the group, and the actual group data. Besides being a storage medium, we also use the cloud storage as a broadcasting interface for group access control changes. Administrators are communicating with the cloud each time a group membership operation takes place so that users can be notified of the group membership update. We consider the cloud storage to show a similar behavior than administrators (i.e., honest-but-curious). It correctly services assigned tasks albeit with a possible malicious intent of peeking into the groups secrets. Moreover, when manifesting the curious behavior, the cloud storage could collude with any number of curious administrators or revoked users.

Size-wise, we target a solution that can accommodate groups of a very large-scale nature.\(^3\) It is desired that administrators perform membership changes for multiple groups at a time, therefore the number of administrators is relatively small when compared to the number of users.

We choose to ensure authenticity guarantees only with respect to administrator identities, and therefore authenticate membership changes operations. Also, authenticating the group data created by users is out of scope, the current model being focused on confidentiality guarantees. Finally, we do not consider hiding the identities of group members, nor the type of executed membership operations, as they can be inferred by the cloud storage from traffic access patterns.

Broadcast encryption

Broadcast encryption (BE) [17] is a public-key cryptosystem with a unique public key that envelopes the entire system, contrary to the HE scheme where each user uses a different public key. However, each user in a BE system has a unique private key generated by a trusted authority. To randomly generate a group key \( g_k \) and the associated group metadata (named encrypt operation within BE systems), one makes use of the system-wide public key. On the other side, when a user wants to reveal \( g_k \) (decrypt in BE systems), she makes use of her individual private key.

As broadcast encryption schemes come with different contextual models, we impose a number of conditions. First, to maintain the same threat model as HE, we are only investigating the use of fully collusion-resistant BE schemes [7], in which no coalition of members outside of the group could reveal \( g_k \). Second, the set of users participating in the system is not initially known, thus we rely on the usage of dynamic BE schemes [16]. Third, as in the case of HE, we would prefer constructions that can accommodate the use of identity-based encryption (IBE).

Piercing through the existing research literature, we identified identity-based broadcasting encryption (IBBE) scheme [15] that not only fulfills all the aforementioned requirements, but also operates with group metadata expansions and user private keys of constant sizes. Moreover, the scheme has an additional strategic advantage that proves beneficial in our context: the system-wide public key size is linear in the maximal size of any group.

Upon analyzing the computational complexity of the selected IBBE scheme, one can notice that creating \( g_k \) given a set of members, as well as decrypting it as a user, are operations with a quadratic complexity in the number of members. Therefore, even though the scheme brings a tremendous gain in

\(^3\)Our evaluation uses 1 million users as the largest group size.
the size of group metadata expansion, the computational cost of IBBE might be excessive for practical use.

Figure 2.9 exemplifies the performance of HE-PKI, HE-IBE and IBBE schemes in their raw form, before any integration with SGX is considered. The sub-figure on the left displays the total time taken for the operation of creating a group while the one on the right shows the size occupied by the expansion of group metadata. The optimality of IBBE regarding the size of group metadata expansion is immediately obvious. It always produces 256 bytes of metadata, regardless of the number of users per group. That is preferable compared to HE-PKI and HE-IBE, which produce increasingly larger values, as much as 27 MB for groups of 100,000 users, and 274 MB for the largest benchmarked group size. On the other hand, IBBE performs much worse than HE-PKI when considering the execution time. It is 150× and 144× slower for groups of 10,000 and 100,000 users, respectively. There is no doubt that the original IBBE is inadequate with a large number of participants, as is the case in the target SecureCloud applications for secure Big Data processing.

IBBE-SGX

IBBE-SGX can be broadly described in 3 steps:

1. Trust establishment and private key provisioning;
2. Membership definitions and group key provisioning;
3. Membership changes and key updates.

IBBE schemes generate a single public key that can be paired with several private keys, one per user. Users, in turn, need to be sure that the private key they receive is indeed generated by someone they trust, otherwise they would be vulnerable to malicious entities trying to impersonate the key issuer. To achieve that, we rely upon PKI to provide verifiable private keys to users.

Another security requirement of IBBE-SGX is that the key management must be kept in TEE. Therefore, there must be a way of checking whether that is the case. On that front, Intel SGX makes it possible to attest enclaves. Running this procedure gives the assurance that a given piece of binary code is truly the one running within an enclave, on a genuine Intel SGX-capable processor.
Deliverable 4.4 Integrated implementation of microservices for big data applications

Figure 2.10 illustrates the initial setup of trust that must be executed at least once before any key leaves the enclave. Initially, the enclaved code generates a pair of asymmetric keys. While the private one never leaves the trusted domain, the public key is sent along with the enclave measurement to the Auditor (1), who is both responsible for attesting the enclave and signing its certificate, thus also acting as a certificate authority (CA). Next, the auditor checks with Intel’s attestation service (IAS) (2) if the enclave is genuine. Being the case, it compares the enclave measurement with the expected one, so that it can be sure that the code inside the shielded execution context is trustworthy. Once that is achieved, the CA issues the enclave’s certificate (3), which also contains its public key. Finally, users are able to receive their private keys and the enclave’s certificate (4). The key will be encrypted by the enclave’s private key generated in the beginning. To be sure they are not communicating with rogue key issuers, users check the signature in the certificate and then use the enclave’s public key contained within. All communication channels described in this scheme must be encrypted by cryptographic protocols such as TLS.

Figure 2.10: Initial setup.

In contrast to traditional IBBE that requires the use of a trusted authority to perform the operations for system setup and to extract user secret, we rely on SGX enclaves. Therefore, the master secret key $MSK$ used by the two aforementioned operations can be made available in plaintext exclusively inside the enclave, and securely sealed if stored outside of the enclave for persistence reasons.

Similarly to IBBE, the operations to encrypt broadcast key and decrypt broadcast key rely on the system public key $PK$, and are thus usable by any user of the system.

As opposed to the traditional IBBE usage scenario, our model requires that all group membership changes—generating the group key and metadata—are performed by an administrator. Administrators can use the master secret key $MSK$ to encrypt, set up the system and extract user keys. The decryption operation, however, remains identical to the traditional IBBE approach, being executed by any arbitrary user.

We now describe the computational simplification opportunities introduced by IBBE-SGX compared to IBBE [15]. First, by making use of $MSK$ inside the enclave, the complexity of the encryption operation drops from $O(\vert S\vert^2)$ for IBBE to $O(\vert S\vert)$ for IBBE-SGX, where $\vert S\vert$ is the number of users in the broadcast group set. The reason behind the complexity drop is bypassing a polynomial expansion of quadratic cost, necessary in the traditional IBBE assumptions (see [13] for details). This complexity cut is sufficient to tackle the impracticality of the IBBE scheme emphasized earlier in Figure 2.9. Second, by relying on $MSK$, one can build efficient access control specific operations, such as adding or removing a user from a broadcast group. IBBE-SGX can accommodate $O(1)$ complexities for both operations.

Unfortunately, IBBE-SGX maintains an $O(\vert S\vert^2)$ complexity for the user decrypt operation, during which, similarly to IBBE encryption, the algorithm performs a polynomial expansion of quadratic cost. We address this drawback by introducing a partitioning mechanism, which is described in depth in [13].
Deliverable 4.4 Integrated implementation of microservices for big data applications

Finally, we consider a re-keying operation, for optimally generating a new broadcast key and metadata when the identities of users in the group $S$ do not change. The operation can be performed in $O(1)$ complexity for both IBBE and IBBE-SGX.

**Evaluation**

We benchmarked the performance of the IBBE-SGX scheme from several different perspectives, with both microbenchmarks and realistic workloads. The complete evaluation can be found in [13]. We just show here representative performance results for a simple microbenchmark.

We chose to compare IBBE-SGX to HE only as the latter already shows better computational complexity than IBBE (see Figure 2(a) of [13]). We conducted experiments on a quad-core Intel i7-6600U machine, with a processor running at 3.4 GHz and 16 GB of RAM, using Ubuntu 16.04 LTS.

Figure 2.11 displays the computational cost for operations of creating a group, removing a user from a group, and the storage footprint of the group metadata. One can notice that all three operations are better than their HE counterparts by approximately a constant factor. The computational cost of create and remove operations of IBBE-SGX is on average 1.2 orders of magnitude faster than HE. Compared to the original IBBE scheme, IBBE-SGX is better by 2.4 orders of magnitude for groups of 1,000 users and 3.9 orders of magnitude for one million users. Storage-wise, IBBE-SGX is up to 6 orders of magnitude
Deliverable 4.4 Integrated implementation of microservices for big data applications

better than HE. When analyzing the performance of IBBE-SGX create and remove operations and the storage footprint, and considering different sizes of partitions, one can notice that the remove operation takes half the time than the create operation (detailed results in [13]). Considering the storage footprint, the degradation brought by using smaller partition sizes is fairly small (e.g., 432 vs. 128 bytes for groups of 1 million members).

Application

The IBBE-SGX is implemented as an independent and self-contained service that can be used by SecureCloud applications that rely on broadcast encryption and need to enforce access control policies. It provides an administrator’s API that makes calls to the underlying SGX enclave implementing the functionalities of IBBE-SGX, which is built on top of an IBBE component. Since SGX is not required on the client side, the client API directly calls the functionalities of the IBBE component. Both administrators and clients make use of local in-memory caches in order to save round-trips to the cloud for accessing existing access policies. Administrators make use of the PUT HTTP verb to send data to the cloud, while clients are listening by using HTTP long polling.

A main motivation for this work was to securely share data in a cloud environment by carefully controlling and enforcing access control policies. In our tests, we deployed IBBE-SGX and interfaced it on top of Dropbox. As Dropbox implements long polling at the directory level, we indexed the group metadata as a bi-level hierarchy where the parent folder represents the group and each child stands for a partition.

Besides data sharing, IBBE-SGX can be applied for encrypting arbitrary information that is securely broadcasted to a group of users over any shared media. Applications like streaming media distribution, TV broadcast (pay-per-view), collaborative editing, or peer-to-peer networks are obvious candidates for such mechanisms.
3 Distributed Data Store

The distributed data store of SecureCloud is offered as Key-Value Store (KVS) service, providing basic storage features for applications running on the overall SecureCloud platform. The service allows applications to create datastores and to store key-value pairs (objects) in them. Each datastore has a storage policy associated with it that determines the protection level, and an erasure coding schema that determines the availability and reliability levels with which objects are stored.

Any combination of replication and erasure coding can be specified to achieve the optimal trade-off between storage cost, availability and reliability. A frequently used object can be stored with numerous exact replicas. An infrequently used object could have one exact replica and eight erasure coded fragments, of which only six are necessary for recovering a copy of the object. An archival object can be stored simply by using erasure coding. Additionally, the policy can be changed as object access patterns change. These are unique capabilities of the Secure KVS developed in SecureCloud.

The Secure KVS has several features designed with cloud applications in mind. It supports multi-tenancy, isolating the objects and datastores of different applications. An authentication service translates API keys into access privileges. Data locality can be optional, ensuring that certain objects are stored on specified nodes, allowing for faster data access.

The Secure KVS is a modular platform service, designed based on the same microservices' architectural pattern as supported by the SecureCloud platform. By splitting the work into small services, parts of it can be scaled up or down to meet demands. This is particularly useful for dealing with dynamically changing access patterns as well as providing support for different storage policies as erasure coded policies require additional computational resources. Furthermore, the microservices architecture pattern is ideal for creating highly distributed services, crucial in achieving reliable data storage across the nodes running the SecureCloud platform. Finally, well-defined contracts between the microservices allow for easy extension.

This chapter summarises the final implementation of the distributed data store developed throughout SecureCloud. It wraps up the efforts of Task 4.2, Secure distributed data management and storage (month 1 to month 36). The main component described here is a Key-Value Store, just as introduced above.

3.1 An overview of the Secure Key-Value Store

The Secure Key-Value Store (KVS) offers basic object storage capabilities for applications hosted on the SecureCloud platform. Users create datastores and store key-value pairs in them. Client applications connect to the KVS using HTTPS, the underlying TLS connections are terminated inside an SGX enclave. The value of each object is encrypted before it leaves the enclave. Each datastore has a storage policy associated with it which determines how many replicas of each object will be stored in the system. An erasure coding schema can also be specified for each policy. The KVS currently supports Reed-Solomon and Random Linear Network Coding (RLNC). RNLC has the benefit of reduced network traffic when repairing lost fragments.

Any combination of replication and erasure coding can be specified to create so-called hybrid storage policies. These can achieve arbitrary trade-offs between storage overhead, availability and reliability. For example, frequently accessed data can be stored as three exact replicas for fast and reliable access. Infrequently accessed data could have one replica and eight erasure coded fragments, of which only six are necessary for recovering a copy of the object. Archival data could be stored by using 9 erasure coded fragments, with 6 necessary to rebuild the original data. These examples show comparable reliability in regards to the number of node outages they survive, but have vastly different performance characteristics.
and storage cost. The ability to fine-tune the storage policy to the requirements of the cloud applications it is a unique capability of the KVS and has been specifically developed for the SecureCloud platform.

The KVS supports multi-tenancy, isolating the objects and datastores of different cloud applications. An authentication service translates API keys into access privileges.

The KVS is a modular platform service which uses the same microservice-based architecture provided to user applications by the SecureCloud platform. Figure 3.1 presents an overview. By separating tasks into small, simple parts, services can be scaled up or down individually to meet demand. This is particularly useful for dealing with dynamically changing access patterns and as an efficient way to support different storage policies as erasure coded policies require additional computational resources. Finally, the microservice architecture pattern is ideal for creating highly distributed services, deployed to different failure domains. This is crucial in achieving reliable data storage across the nodes of the system.

### 3.2 The role of the Secure Key-Value Store inside SecureCloud

The KVS makes it possible to store and retrieve key-value pairs for the SecureCloud platform applications in an efficient and secure way using Intel SGX. Many simultaneous applications can take advantage of
Deliverable 4.4 Integrated implementation of microservices for big data applications

the KVS as multi-tenancy is supported. Data storage can be optimized for storage overhead or reliability based on the data access patterns (e.g. warm or cold data) using the configurable storage policy options.

3.2.1 Evaluation

In a joint paper published recently, we evaluated the performance of RLNC encode (when incoming data is coded with redundancy) and decode (when data is reconstructed) operations inside an enclave [28]. Measurements were carried out in CloudSigma’s infrastructure with an Intel SGX capable server. The test device was an Ubuntu-based virtual machine, which runs on top of an Intel(R) Xeon(R) E3-1230 server-grade CPU (v5 @3.40GHz) and has a quota of 8GB DDR4 memory (2133MHz). The hypervisor has been configured to expose Intel SGX capabilities to the virtual machine.

The following network coding implementations were compared:

- NCLib-SGX is a network coding library which runs in an enclave.
- NCLib is the same library but running outside of the enclave (to measure the overhead of SGX in this application).
- Chocolate Cloud’s RLNC storage library is the state-of-the-art in RLNC with fast execution without SGX and using hardware accelerated SIMD instructions.

Furthermore, to compare RLNC with traditional erasure codes, we included measurements for ISA-L Reed Solomon and ChocolateCloud’s Reed-Solomon libraries as well.

Table 3.1: Performance comparison of encoder implementations, averaged over all configurations. Libraries marked with a star use hardware acceleration.

<table>
<thead>
<tr>
<th>Library</th>
<th>Systematic RLNC</th>
<th>Full RLNC</th>
<th>Systematic RLNC</th>
<th>Full RLNC</th>
<th>Systematic RLNC</th>
<th>Full RLNC</th>
</tr>
</thead>
<tbody>
<tr>
<td>NCLib-SGX</td>
<td>19.66 MB/s</td>
<td>137.09 MB/s</td>
<td>19.54 MB/s</td>
<td>29.16 MB/s</td>
<td>183.85 MB/s</td>
<td>1276.95 MB/s</td>
</tr>
<tr>
<td>NCLib</td>
<td>183.96 MB/s</td>
<td>225.92 MB/s</td>
<td>183.96 MB/s</td>
<td>225.92 MB/s</td>
<td>183.96 MB/s</td>
<td>225.92 MB/s</td>
</tr>
<tr>
<td>CC RLNC Storage Library*</td>
<td>7462.76 MB/s</td>
<td>11270.54 MB/s</td>
<td>2099.45 MB/s</td>
<td>3068.59 MB/s</td>
<td>2099.45 MB/s</td>
<td>3068.59 MB/s</td>
</tr>
<tr>
<td>Intel ISA-L*</td>
<td>10096.63 MB/s</td>
<td>9178.24 MB/s</td>
<td>10096.63 MB/s</td>
<td>9178.24 MB/s</td>
<td>10096.63 MB/s</td>
<td>9178.24 MB/s</td>
</tr>
<tr>
<td>CC RS Storage Library*</td>
<td>10499.86 MB/s</td>
<td>12370.72 MB/s</td>
<td>10499.86 MB/s</td>
<td>12370.72 MB/s</td>
<td>10499.86 MB/s</td>
<td>12370.72 MB/s</td>
</tr>
</tbody>
</table>

192 different configuration were measured (12 schemas, 2 codecs and 8 chunk sizes). The measured encode and decode speeds are compared in Table 3.1 and plotted separately on Figure 3.2 and Figure 3.3. The results show an order of magnitude difference between the performance of running inside or outside of an SGX enclave. A similar improvement can be observed when running encode and decode operations using hardware accelerated instructions. Initially, we envisioned running all code that deals with object data inside an enclave. Based on these numbers, we decided to change our architecture slightly and run all such computationally intensive operations outside an enclave. This motivated the development of the TLS interceptor, described in Section 3.3.1.
3.2.2 Integration

We designed the KVS as a user-facing storage service that exposes a simple to use REST API. As such, it is not used directly by the lower level SecureCloud infrastructure and platform services. We present its interoperability with other services and its place in the SecureCloud ecosystem in the final project demonstrator. The KVS serves as the data storage service for a fraud detection application, ingesting and digesting data from a large number of smart meters.

The KVS uses two components developed by our SecureCloud partners. First, to ensure the confidentiality of data stored in the KVS, we created a new component that terminates TLS connections inside an enclave and encrypts/decrypts data with client-supplied keys. We built this on top of TaLoS [4], another component of the SecureCloud ecosystem. We refer to it as TLS interceptor and describe it in greater detail in Chapter 3.3.1.

Second, SecureCloud developed an SGX-aware scheduler able to work with SGX-enabled containers in a heterogeneous Kubernetes cluster of SGX and non-SGX-enabled machines. We configured and developed the operational scripts required to deploy the KVS to a Kubernetes cluster. It has a single service which needs to be run inside an SGX-enabled container, the aforementioned TLS interceptor. The other KVS services do not have this requirement. This makes SecureCloud’s Kubernetes scheduler a perfect fit for our use case. The KVS needs a single addition to its default configuration, we must specify `schedulerName: spread` in the YAML file that describes the deployment.
3.3 Updated components

3.3.1 TLS interceptor

In order to protect the confidentiality and authenticity of objects stored in the KVS, we added a new microservice in front of our REST API. We refer to it as the TLS interceptor. This component has the role of intercepting HTTPS requests and encrypting the data that is written to the KVS. Likewise, it intercepts and decrypts HTTPS responses containing object data before they are returned to the client. The termination of TLS connections and encryption/decryption operations are performed inside an SGX enclave, thus object data is never exposed outside of an enclave. The other microservices which make up the KVS handle only encrypted object data and have no access to the decryption keys.

Positioning the TLS interceptor as the gateway of the KVS removes the need to provide encryption capabilities in each of the other parts of our service. Thus, the other KVS microservices do not need to be rewritten to support secure communications and data processing inside enclaves. As an alternative, we considered running our microservices inside SCONES [3] containers. We decided against this approach for several reasons. Having a single microservice that contains an enclave greatly reduces the need for SGX-compatible hardware. Furthermore, only the TLS interceptor must be remotely attested. We also expect better performance as it is no longer necessary to perform erasure coding operations like encoding, decoding and recoding inside an SGX enclave. We have previously created and evaluated [28] NCLib–SGX, a network coding library for the enclave. While we have shown that NCLib–SGX is sufficiently fast for a wide range of scenarios, a state of the art library that does not run inside an enclave
will perform better. Not to be underestimated as factor, this approach also leads to simpler code. We believe this to be important in creating a robust and secure service.

To enable this functionality, we use NGINX as a reverse proxy to the KVS. NGINX loads the TLS interceptor as the shared library to be used for handling HTTPS connections. The TLS interceptor is built on top of TaLoS [4], the platform-level communication component proposed in SecureCloud (part of Work Package 3, described in Deliverable D3.2). As such it is a fork of LibreSSL v2.4.1, and implements the de-facto standard interface for TLS libraries as defined by OpenSSL. Internally, most of the required modifications to LibreSSL have previously been implemented by TaLoS, which provides a simple way to do further processing of the HTTP data stream as part of a separate module. These modules can register callbacks to events such as data being read from or written to a TLS connection. We have used this convenient method to extend the functionality TaLoS provides. We added further changes to LibreSSL to be able to modify the size of the buffer used to read from/write to the HTTP stream. This was key to enabling encryption and decryption in blocks of 16k bytes as well as to ensure that we could insert and remove the Initialization Vector (IV) and authentication tag into/from the re-encrypted HTTP stream. This modified version of LibreSSL compiles into a shared library that can be signed using Chocolate Cloud’s Intel-supplied development key.

The TLS interceptor parses HTTP headers to determine whether the request of response contains a payload that must be encrypted or decrypted. If this is the case, it employs AES-GCM with a key length of 128 bits, as implemented by the Cryptographic Library included in the Intel SGX SDK [20]. This is the strongest encryption scheme supported by the library which also provides authentication. Thus, the TLS interceptor can verify that data was not tampered during storage or processing by malicious actors. Encryption is performed in 16-kbyte chunks, with an IV randomly generated for each chunk. The IV and the resulting authentication tag are inserted into the HTTP data stream after the end of each chunk. Figure 3.4 shows an example of writing an object to the KVS using the TLS interceptor. To ensure that upon retrieval data is streamed in the same sized chunks, we pad the HTTP response header in our REST frontend microservice. The encryption keys must be provisioned to the enclave either during remote attestation or as part of each HTTPS request, in which case they are removed inside the enclave.

If there is no SGX hardware available, or the application does not require the level of security provided by the TLS interceptor, the KVS can also be accessed using the REST frontend component directly. However, data encrypted using the TLS interceptor cannot be accessed in this way.

In this case, the confidentiality of data is provided by our previously proposed encryption schemes, as described in greater detail in D4.3. The level of security provided by these network coding-based
schemes has been studied previously in detail for various attack types. For example, [40] studies entropy, byzantine, pollution and eavesdropping attacks and proposes defense schemes. [27] characterizes the algebraic security offered by the random coefficient selection process to eavesdroppers. Perhaps of most relevance to our encryption-enhanced scheme is SPOC. In [32], the authors look at SPOC and other approaches to secure network coding and provide an extension to it called eSPOC.

3.3.2 Kubernetes integration

We have designed the KVS to be a distributed, scalable solution. We used Kubernetes to move from a single machine development environment to a storage solution that achieves these goals. Kubernetes enables our microservices to be distributed to separate nodes and greatly simplifies communication, deployment and scheduling. It also allows services to scale individually to meet demand.

The storage daemon service is responsible for the actual data writes and reads to/from disk. Its instances are defined as a Kubernetes Stateful Set\(^1\). This allows the creation of unique and ordered Kubernetes pods, necessary to identify which fragment or replica is stored on which pod. We also specified (with pod anti affinity\(^2\)) that each pod of the StatefulSet is deployed to a separate node. This is crucial in meeting reliability requirements by creating separate failure domains. Thus, we can ensure that if a disk loses data permanently then only one pod was using it. The KVS can then reconstruct onto a backup node the lost replica or erasure coded fragments for each object stored on the failed disk.

We used Persistent Volumes\(^3\) to achieve persistent data storage that is not influenced by the lifecycle of the Kubernetes deployment. The ideal type of storage class for the storage daemons is Local Persistent Volume\(^4\). This exposes local disks directly, hence fast, local SSDs can be used instead of slower remote disks. Furthermore, it enforces the rule that the same pod will always use the same persistent volume. Unfortunately, this is currently a beta feature of Kubernetes and only available from version 1.10. When deploying to an older version of Kubernetes, the storage daemons can be deployed using other storage classes, for example GCE-PD on Google Kubernetes Engine or Cinder on OpenStack.

The ports on which the REST API and the TLS Interceptor accept requests are made available through a Kubernetes Service.

---

1. [https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/](https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/)
2. [https://kubernetes.io/docs/concepts/configuration/assign-pod-node/](https://kubernetes.io/docs/concepts/configuration/assign-pod-node/)
3. [https://kubernetes.io/docs/concepts/storage/persistent-volumes/](https://kubernetes.io/docs/concepts/storage/persistent-volumes/)
4. [https://kubernetes.io/docs/concepts/storage/storage-classes/#local](https://kubernetes.io/docs/concepts/storage/storage-classes/#local)
4 Scheduling

This chapter presents the final summary of an integrated scheduler developed for SecureCloud. It serves as a final report for Task 4.3, Distributed scheduling mechanisms (month 12 to month 36). The scheduler implemented efficiently allocates machines to support containers for executing SecureCloud services. It relies on heterogeneous clusters combining machines that are capable of running trusted elements with machines that cannot. This component was first introduced in Deliverable D4.3, and we present here its evaluation and integration with SecureCloud use cases.

4.1 SGX-Aware scheduler

The SGX-Aware scheduler [37] provides an execution platform based on Kubernetes, in such a way that other containers developed as part of SecureCloud can use it. It allows to efficiently schedule SGX-enabled containers in a cluster of SGX- and non-SGX-enabled machines. At its basis is Kubernetes, a container orchestrator that was designed by Google to automatically deploy, scale in and out, and manage containerized applications on clusters of computers [24]. The sections that follow present the SGX-aware scheduler developed in Work Package 4.

4.1.1 Description

The architecture of our system is shown in Figure 4.1. The complete system implements a container orchestrator that can efficiently schedule SGX-enabled jobs, as well as regular jobs, on a heterogeneous cluster. Technical details regarding this system, including instructions to deploy it, are available in D4.3.

The most important aspect of our system is its ability to guarantee that containers running on a given host always fit within its current SGX-specific memory limits. This is of particular relevance to avoid major performance penalty [2]. The SGX-aware orchestrator is composed of many sub-components that work together to provide decisions regarding where to schedule submitted jobs.

![Figure 4.1: Workflow of our SGX-aware scheduler.](image-url)
First, we leverage a monitoring layer, implemented by means of daemons executed on each node of the cluster. We rely on a time-series database to store and analyse data regarding resource usage. The use of our scheduler is transparent to the users; they simply submit their jobs by using the existing programming interface exposed by Kubernetes. We provide two well-known scheduling algorithms as part of our system; they are described in details in D4.3. Another crucial component is the device plugin that allows to mark a Kubernetes node as able to execute SGX instructions. We also deliver a probe that collects SGX-specific metrics on each machine that is part of the cluster. Finally, we modified the existing SGX driver provided by Intel, as well as the Kubelet node daemon shipped with Kubernetes to enforce SGX-related resource usage limits.

4.1.2 Validation and Integration

Evaluation

We have demonstrated that the design and implementation of our SGX-aware scheduler are sound by performing a detailed evaluation which is described in length in our article [37]. We evaluated our system on a cluster of 5 machines.

Our evaluation uses the Google Borg Trace [39, 34]. The trace was recorded in 2011 on a Google cluster of about 12,500 machines. The nature of the jobs in the trace is undisclosed. We are not aware of any publicly available trace that would contain SGX-enabled jobs. Therefore, we arbitrarily designate a subset of trace jobs as SGX-enabled. The trace specifies the memory usage of each job as a percentage of the largest memory capacity in Google’s cluster. Given the size of that cluster, we have to scale down the trace before being able to replay it on our own cluster setup. Second, the trace reports data for 29 days; as we do not need such extensive data, we use a 1-hour subset that is less job-intensive in terms of concurrent jobs, but still injects an intensive load on the cluster. Moreover, we sampled every 1200th job from the trace, to end up with a number of jobs big enough to cause contention in our system, but that does not clutter it with an incommensurable amount of jobs.

After processing the trace file, we get a timed sequence of jobs with their effective memory usage. In order to materialize the information given by the trace into actual usage, our jobs are built around containers that run STRESS-SGX [38], a fork of the popular STRESS-NG [23] stress tool. This tool was developed by University of Neuchâtel (UniNe) as part of the LEGaTO Horizon 2020 project [25]. Normal jobs use the original virtual memory stressor brought from STRESS-NG, while SGX-enabled jobs use the topical Enclave Page Cache (EPC) stressor. We specify parameters to allocate the right amount of memory for every job, in accordance with the values reported by the trace.

In Figure 4.2, we show the aggregated turnaround time for all jobs submitted during the experiment. This metric refers to the duration elapsed between the instant a job was submitted to the moment when the job finishes and dies. At the top of the figure, the dotted black bar (labelled “Trace”) represents the total useful job duration, as recorded in the cluster. The difference with the other results highlights the total waiting time for each of the different settings. We use runs that only contain one type of job (either all SGX or regular jobs). The binpack strategy, in this specific setting and portion of the trace, achieves the best result (shorter turnaround time). When using the binpack strategy, SGX jobs need slightly less than twice the time of their non-SGX counterparts. This shows that incorporating a reasonable number of SGX jobs has a limited impact on the scheduling. Notably, we expect our deployments to include small percentages of overall jobs requiring SGX instructions.
One of the features of our system is the strict enforcement of limits regarding per-container EPC memory consumption. Resource usage limits are declared by the users themselves, and therefore could be inaccurate with regard to the real usage made by their containers. We identify incentives to lead users into truthfully declaring resource usages: if the user declares too high a limit for his container, then the infrastructure provider will charge him for the additional resources. On the other hand, declaring too low resource usages will lead to the container being denied service due to the enforcement of limits. To show the potential damages that users could cause without this mechanism, we add so-called malicious containers to our system under test (SUT). Their modus operandi is to declare less resources than they actually use. They use as much as up to 50% of the total EPC available on the machine they execute on.

The results, as presented in Figure 4.3, show that, without strict limits being enforced, most honest containers in the system suffer from longer waiting times. Obviously, as the size of the allocations made by malicious containers increases, the effects suffered by honest containers grow as well. Fortunately, we clearly see that enforcing limits on memory allocations annihilates the efforts of the malicious containers.

The hardware in current Intel processors only supports version 1 of SGX. Intel has already published several design documents regarding the second version, SGX 2 [29, 21]. The software development kit (SDK) and the driver adapted to SGX 2 have been recently published [22]. The most important feature that this new version introduces is dynamic EPC memory allocation. Enclaves can ask the operating system for the allocation of new memory pages, and may also release pages they own. Contrary to the current version, these operations can also be done after the enclave was started. Considering the limited amount of EPC that is shared by every enclave running on the same node, this new feature can really improve resource utilization on shared infrastructures.

As far as our scheduler is concerned, we believe that only minor changes need to be performed to fully take these new possibilities into account. Provided that Kubernetes nodes are deployed on SGX 2-compatible hardware, the solution presented here will work out-of-the-box. The only part of our system that we have identified as not yet SGX 2-ready is our implementation of resource usage limits in the Intel software guard extensions (SGX) driver. We believe that the effort required to port it to the new revision of SGX is modest.
Deliverable 4.4 Integrated implementation of microservices for big data applications

Integration

The SGX-aware scheduler developed in SecureCloud has been natively integrated with all containers deployed to support and execute SecureCloud’s microservices. The SGX-aware scheduler has been developed as a plug-in for Kubernetes [24], and we used Kubernetes as a basis for all container deployments. Kubernetes has the advantage of being developed by Google having a very large installed base. Its source code is very stable, and is kept alive by a very large free-software community. We believe that Kubernetes is currently one of the most popular software infrastructures for managing large container installations. By having implemented it as a plug-in to Kubernetes, all SecureCloud components that use SGX inside containers (i.e. almost every component implemented in the project) profit from using the SGX-aware scheduler.

Deliverable D2.4 describes in its Chapter 3 how the deployment tools of SecureCloud integrate with Kubernetes. That integration naturally includes the SGX-aware scheduler in order to properly allocate containers in SGX-capable machines. Deliverable D1.3 presents all use cases developed within SecureCloud. Most use cases interact with a Job Scheduler that uses Kubernetes, and therefore, the SGX-aware scheduler.
5 Secure computation

MapReduce is one of the most widely used programming models for big data processing. Existing MapReduce frameworks provide services for launching computations across distributed servers, including the tasks’ lifecycle management, storage of intermediary results, data transmission, and synchronization. In SecureCloud, we developed three implementations: SecureStreams, described in Section 5.1, Lightweight MapReduce, presented in Section 5.2 and Secure Apache Spark Executor, summarised in Section 5.3. The first two implementations are aimed at supporting execution of jobs in resource-limited conditions, as is the case of edge devices, for example. The third implementation allows the execution of full-fledged Apache Spark jobs, to be used in industrial-grade data processing. All implementations combined offer a complete tool set for implementing secure MapReduce and streaming jobs, at different levels. This chapter presents the three implementations, and serves as a final report for Task 4.4, Secure map/reduce computation (month 1 to month 36).

5.1 SecureStreams

SecureStreams is a middleware framework for developing and deploying secure stream processing on untrusted distributed environments. It supports the implementation, deployment, and execution of stream processing tasks in distributed settings, from large-scale clusters to multi-tenant cloud infrastructures. It has been extensively described in Deliverable D4.3.

5.1.1 Description

SecureStreams is a message-oriented middleware [14], which integrates with the TLS protocol for data communication and the current version of Intel’s software guard extensions (SGX) to deliver end-to-end security guarantees along data stream processing stages. SecureStreams can scale by adding or removing processing nodes at any stage of the pipeline, to dynamically adjust according to any given workload. Its design is inspired by the dataflow programming paradigm [35]: the developer combines together several independent processing components (mappers, reducers, sinks, shufflers, joiners, etc.) to compose specific processing pipelines. Following the same rationale as other SecureCloud components, SecureStreams is packaged and deployed using industrial-grade lightweight virtualization technologies such as Docker containers [30].

SecureStream is built with a combination of two different types of base components: workers and routers. A worker continuously listens for incoming data by means of non-blocking I/O. As soon as data
flows in, it executes an application-dependent business logic. A router acts as a message broker between workers in the pipeline, and transfers data between them according to a given policy. Figure 5.1 shows an example implementation of a SecureStreams dataflow, implementing a classical pipeline with filter, map and reduce workers.

5.1.2 Validation and Integration

SecureStreams has been delivered in SecureCloud’s second year and an extensive evaluation is included in Deliverable D4.3, therefore it will not be repeated here. Please refer to Deliverable 4.3 for all details of the evaluation of SecureStreams.

Evaluation

To summarise, there were three main sets of experiments in the evaluation presented in D4.3. First, we present a set of micro-benchmarks to evaluate performance of the integration of the Lua virtual machine inside SGX enclaves. Following, we deploy a pipeline including mappers, filters and reducers to measure the achievable throughput of SecureStreams, as well the overall network overhead. Finally, we evaluated SecureStreams in terms of scalability, deploying a pipeline like in Figure 5.1 and vary the number of workers deployed on each stage.

![Figure 5.2: SecureStreams scalability: processing time with different numbers of mappers.](image)

As an example, Figure 5.2 shows the total completion time while varying the number of mapper workers in the first stage of the pipeline. This completion time is important because we identified the mapper as the worker consuming most resources. We observe that the speed-up increases accordingly, until the number of deployed containers reaches the number of physical cores.

Integration

In order to demonstrate the usability of SecureStreams, it has been integrated in the Smart Home use case. There, a set of filter workers process input data from sensors in order to preselect potential alarms
or critical conditions. Please refer to Deliverable D1.3 for the integrated description of all use cases. In a more general case, thanks to its Lua implementation, SecureStreams allows to implement streaming jobs having a small footprint in the devices executing it. The so-called edge devices gather data from the Internet-of-things and forward events to the cloud for aggregated processing. These edge devices are usually small and, although they may offer multiprocessing or multithreading, they do not offer large processing capacities as those found in the server-grade machines installed in the cloud. Together with Secure MapReduce, it is very useful when processing big data from the Internet-of-things, as one can build processing jobs that execute on edge devices.

5.2 Lightweight MapReduce

Since it was officially adopted by Google in 2008, the MapReduce programming model consistently gained ground as a viable solution for assuring the necessary scalability of distributed data processing. The generic model, composed of the two main map and reduce functions, was widely used to implement applications that can leverage parallel task processing. In this generic model, the data to be processed is typically located over a series of mapper nodes, which apply in parallel a map function responsible for mapping individual data items to a finite set of predefined keys. The output is redistributed in a shuffle step based on the key mappings to reducer nodes, which execute in parallel a reduce function for processing each set of data items corresponding to a certain key. This model can be used in processing tasks that range from simple operations that can be composed like counting, sorting, and searching data to more complex algorithms like cross-correlation or page rank. MapReduce was adapted in different ways to fit this wide diversity of scenarios, as well as different deployment environments.

Particular settings where the deployment of MapReduce encountered difficulties are services that require security guarantees. For cost savings purposes, data analysis and processing services are more and more often deployed to public cloud infrastructures, which do not offer strong security guarantees and hence make sensitive data prone to privacy and integrity risks. If such services rely on a MapReduce-based implementation, it is of high importance to find a way to adapt the programming model in a manner that provides the required security assurance to the system.

SecureCloud proposes a self-contained framework for securing MapReduce that leverages the benefits of SGX’s trusted execution environment. Our system architecture combines a lightweight virtual machine based on the Lua language, a MapReduce library, and a publish/subscribe service for communication between the client and worker nodes. Unlike cryptographic solutions, our approach is independent of the particular characteristics of the map and reduce functions and can hence be used for any problem parallelizable with MapReduce. The specific code to be executed in the MapReduce service can be integrated in simple scripts, which are run privately and in isolation using SGX over data decrypted only inside the enclave. Our focus is on providing a flexible framework for securely running MapReduce applications that can be easily implemented and deployed. The basic word count MapReduce example, for counting the number of occurrences of different word in a given text, can be implemented in our framework with less than 30 lines of code (LOC).
5.2.1 Description

In Figure 5.3 we display the entities composing our solution architecture: clients, router (SCBR based pub/sub engine) and workers, which can assume the role either of a mapper or a reducer. Clients provide the code to be executed, the data to be processed and gather the results after completion. All communication channels use the 0MQ [18] message passing library, having a central point in the SCBR pub/sub engine (Sec. 2.1).

The SCBR engine is responsible for securely storing subscriptions that contain the conditions under which each message should be forwarded to the corresponding interested party. Subscriptions, used in an initiation protocol for the MapReduce processing as described further below, are stored within the SGX protected memory area and every match processing is done inside enclaves. Data is always encrypted whenever outside of the enclaves. Although such a centralized approach is not suitable to large-scale data processing, it is arguably useful for modest quantities of highly sensitive data that could be, for instance, partitioned from higher amounts of non-sensitive data.

The MapReduce processing is started following an initial message exchange displayed in Figure 5.4. Worker nodes register their availability for MapReduce job openings through JOB_OPENING subscriptions. The client registers his interest in the details of the map and reduce tasks that the workers are capable of doing through the JOBDETAILS subscription. When the client has a new job to execute it advertises it through a JOB_OPENING publication, which is received by workers that previously registered as available. The workers submit then their desired details regarding the job through a JOBDETAILS publication by sending in the message payload their further subscriptions for code and data (particular to the role they choose: mapper or reducer). At the end of this negotiation, if the client decides to hire a worker, it registers on SCBR the received subscriptions for code and data on the worker’s behalf. By doing so, the client established the MapReduce chain and keeps track of how many workers it has hired and of which kind (mappers or reducers).

The provisioning of code and data is shown in Figure 5.5. The client also includes the number of reducers along with the Lua scripts in case of type MAP_CODETYPE, or the number of mappers in case of type REDUCE_CODETYPE. The purpose is to make the workers aware of how many messages signaling the end of the stream they have to wait before considering the work done. This is important because the reduce phase can only start once all data for a given key is routed to the intended worker. Additionally, the number of reducers received by a mapper is used in a hash function that takes as argument the mapped key and returns the indication of which reducer it has to be forwarded to. After sending the code, data is
split by the client among the mappers, line by line. The destination id is included in the header of the MAP_DATATYPE publication.

Workers decrypt the received code and store it inside the enclaves waiting to start the execution. When data arrives, mappers perform the processing and each one of the resultant set of key-value pairs is forwarded to the proper reducer. Its id is obtained after providing the key and number of reducers as arguments to the hash function that comes along with the code of the mapper. The shuffling phase is hence conducted by the mappers. In order to forward data to the following step, all the Lua script has to do is to call a special function called `push(key,value)`, and the framework handles all the communication aspects of forwarding the data.

Regarding the execution environment, since SGX code must be previously signed and no dynamic linking is possible after the enclave creation, we ported a Lua [19] virtual machine to run in protected area. Porting legacy code to SGX means that every system call or input/output instruction have to be dealt with, since these are not allowed inside enclaves. Workers contain a Lua interpreter that runs inside the enclaves, which loads the scripts that contain the processing phases of `map` and `reduce`. This setting provides a very accessible programming environment, which can be easily used and efficiently maintained. We provide below a basic example.

```lua
local json = require "json"

function hash(key,rcount)
    return string.byte(key,1) % rcount
end

function combine(key,value)
    local clist = json:decode(value)
    local sum = 0
    for k,v in pairs(clist) do
        sum = sum + v
    end
end
```

Figure 5.4: Session establishment protocol.
Listing 5.1: Map code in Lua for word count

```lua
end
push(key,sum)
end

function map(key,value)
  for word in value:gmatch("%w+") do
    push(word,1)
  end
end
```

Listing 5.1 shows the sample code of a mapper of a word count application. As it can be noticed, data is encoded in JavaScript Object Notation (json). For convenience, the json Lua parser is pre-loaded in the enclave of worker nodes, and is accessible through the require command. The script can contain as many helper functions as desired. The following special functions, however, are called by the framework:

- **map(key, value)**: Contains the functional implementation of mapper.
- **combine(key, value)**: [Optional] Post-processing on values grouped by keys.
- **hash(key,rcount)**: Returns the Reducer id that is supposed to receive a given key considering that there are rcount Reducers in total.

Likewise, listing 5.2 shows a sample code for the reduce step that contains a single special function:

- **reduce(key, value)**: Contains the functional implementation of reducer.
5.2.2 Validation and Integration

We construct our evaluation using the k-means algorithm, a popular use-case for cluster analysis in data mining and machine learning. In our implementation, we partitioned \( n \) randomly generated bi-dimensional coordinates into \( k \) clusters. In the map step, we assign each observation to the nearest center by computing and comparing their distances (\( n \times k \) operations). Then, we update the centers to become the centroid of assigned coordinates (\( n \) operations) in the reduce step. This process repeats until the average distance between the most recent calculated centroids and the ones from the previous iteration is less than a threshold.

We conducted all the experiments using two SGX capable machines, both with processor Intel i7-6700 64bits, clock of 3.4GHz, 8MB cache, 4 cores, 8 threads, with 8GB of installed memory and SSD of 256GB. In terms of software, we used the Intel SGX SDK 1.7.100 over Ubuntu 14.04.1, kernel 4.2.0-42. Unless mentioned otherwise, messages were all encrypted with AES-CTR with key and input vector both of 128 bits and were decrypted only inside the enclaves. The process placement was made as follows: Machine 1: The Client, entity responsible for providing code and data, 8 mappers and 5 reducers. Machine 2: 8 mappers and 5 reducers. The number of mappers was chosen to be twice as much as the number of cores in each machine to take advantage of parallelism, while the number of reducers was set to be a divisor of input data size to stimulate an even distribution of work among them. To illustrate how small our final code-base is, Table 5.1 shows the memory section sizes of executables and shared libraries that are loaded into enclaves.

<table>
<thead>
<tr>
<th></th>
<th>text</th>
<th>data</th>
<th>bss</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>client</td>
<td>379,968</td>
<td>26,672</td>
<td>376</td>
<td>407,016</td>
</tr>
<tr>
<td>worker</td>
<td>288,557</td>
<td>26,312</td>
<td>768</td>
<td>315,637</td>
</tr>
<tr>
<td>worker enclave</td>
<td>679,175</td>
<td>60,004</td>
<td>83,584</td>
<td>822,763</td>
</tr>
<tr>
<td>scbr</td>
<td>277,468</td>
<td>14,808</td>
<td>72</td>
<td>292,348</td>
</tr>
<tr>
<td>scbr enclave</td>
<td>278,967</td>
<td>7,456</td>
<td>80,608</td>
<td>367,031</td>
</tr>
</tbody>
</table>

Table 5.1: Binaries size in bytes

Figure 5.6 illustrates 4 out of 76 iterations that k-means took to converge when the threshold was set to zero. In this example, we randomly generated 100,000 observations and 10 centers. As it can be seen in the first frame, the initial centroids were all set on a small area in the bottom-left corner by limiting the
domain of generated coordinates. In spite of this, they quickly assume positions that are closer to the best fit. After just 6 iterations, the average distance of centroids decreases more than six times. Besides, even with an initial state closer to the final one, we can get a slower convergence (90 iterations, as shown in Figure 5.7). The final result depends on the initial points and there is no guarantee that it is the global optimum, neither about how fast it will converge.

For the following executions, we arbitrarily set the threshold to be one thousandth of the diagonal of the rectangle that contains all the observed points. That means that the iterative process finishes when the average distance of centroids between two subsequent iterations is less than that fraction, so that we can avoid slow convergences. For the given examples of Figure 5.7, this criteria would stop the executions by the 41st (instead of 76th) and 21st (instead of 90th) iterations. From the figure, we can see that there is not much variation from those points on.

To assess the influence of input data size, namely the number of observations and centroids, on the memory usage and time consumption, we conducted multiple experiments using the two SGX capable machines described above. Figure 5.8 shows the average time it took to complete one iteration of k-means with varying input sizes. It can be noticed that, although the variation on the number of clusters can cause some inflection in the curves, the completion time is mostly affected by the number of observed data points. Moreover, while the two first increments on the number of data points \(n = 10k\) and \(n = 100k\) caused a proportional increase on consumed time with regards to the data growth (ten times), the last one \(n = 1M\) induced a twenty-fold rise. That can be explained by the growth in the occurrence of cache

Figure 5.6: k-means convergence. Iterations 0 (top left), 19 (top right), 38 (bottom left) and 76 (bottom right)

Figure 5.7: k-means convergence.
misses within each worker. When that happens, data must be fetched from main memory. When using SGX protected executions, this means one page has to be evicted from cache (and hence, encrypted), while the one that is fetched must be decrypted and checked for integrity and freshness (that prevents tamper and replay attacks, respectively).

The described cache effect is well documented \cite{33, 9, 2} and can be noticed by the cache miss rates. Since the reduce phase is more memory intensive, we chose to make the average on the cache miss rates per second of all 10 reducers in each execution, i.e., for the same second, cache miss rates of reducers were summed and divided by 10. Wall clocks were synchronized with a common NTP server, so that the resulting skew was at the range of tens of milliseconds and it should not affect the sampling resolution of 1s. Figure 5.9 shows that measurement as reported by the tool `pidstat` when the number of centroids is \( k = 50 \). Note that \( y \) scale is logarithmic, so that the cache miss rates for \( n = 1M \) is at least two orders of magnitude higher than \( n = 100k \).

Finally, we moved to appraising the influence of SGX and encryption when compared to native executions (no hardware protection). We ran the same datasets with a fixed number of clusters \( k = 50 \) and observed points varying from \( n = 1000 \) until \( n = 1M \) along with the four combinations of turning on and off enclaves and encryption. Results are plotted in Figure 5.10. We also included the overhead in percentage. Encryption overhead was obtained by comparing the average time it took to complete one iteration with and without encryption both inside and outside the enclave, and averaging the two values. SGX overhead was established analogously. The time corresponds to the average of all iterations in a
Deliverable 4.4 Integrated implementation of microservices for big data applications

Figure 5.10: Encryption and SGX overhead

<table>
<thead>
<tr>
<th></th>
<th>Split</th>
<th>Shuffle</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n = 1k$</td>
<td>58.7 KB</td>
<td>112.1 KB</td>
<td>4.3 KB</td>
</tr>
<tr>
<td>$n = 10k$</td>
<td>257.2 KB</td>
<td>1.1 MB</td>
<td>4.5 KB</td>
</tr>
<tr>
<td>$n = 100k$</td>
<td>2.2 MB</td>
<td>11 MB</td>
<td>4.6 KB</td>
</tr>
<tr>
<td>$n = 1M$</td>
<td>19.1 MB</td>
<td>96.5 MB</td>
<td>4.6 KB</td>
</tr>
</tbody>
</table>

Table 5.2: Data volume exchanged per iteration

single run of k-means (until the threshold was reached). Coefficient of variation across multiple runs was negligible. We can notice that encryption overhead is quite low, of around 5%. Enclave execution, on the other hand, is kept around 30% until we start to get high occurrence of cache misses, as discussed before, when it reaches more than 200%.

The data volume exchanged in each MapReduce step can be seen in Table 5.2. Numbers correspond to the average per iteration for the same executions as Figure 5.10. We observe an increase in overhead when cache misses rate grows. Therefore, based on the processor’s cache size, we could establish the maximum amount of data a single SGX-capable machine would be able to handle before incurring in too much overhead. In our experiments, we perceived that behavior when processing amounts somewhere in between 11 MB and 96.5MB shared between 2 machines (average of around 54MB, or 27MB per machine). Therefore, a rough estimation would be to limit those amounts to three times the cache size, or 24MB in our case. More scalability could be provided horizontally, with the addition of more machines.

Integration

SecureCloud’s integrated uses cases Fault Analysis and Fraud Detection demonstrate the usability of Lightweight MapReduce. Please recall that the integrated use cases have been described in Deliverable D1.3. There, MapReduce jobs process input smart meter data, each one according to its own specific algorithm. In a more general case, Lightweight MapReduce allows to implement MapReduce jobs imposing a small footprint in the computing resources used. Being similar to SecureStreams, Lightweight MapReduce is very useful when processing big data from the Internet-of-things, as one can build batch processing jobs that execute on small edge devices.
5.3 Secure Apache Spark Executor

The Secure Apache Spark Executor, named SGX-Spark, enables the secure execution of unmodified Spark big data processing jobs. A summary of this component is included in Deliverable D1.3. In this section we present, after a brief description, the validation and integration of this component in the SecureCloud platform.

5.3.1 Description

SGX-Spark increases the security of the regular Apache Spark [1] by implementing several functionalities:

- it executes computation on sensitive data inside an Intel SGX enclave to guarantee the confidentiality of the tasks input and output data as well as the integrity of the job execution. This mechanism is presented in more details in Deliverable D3.2;
- it executes non-sensitive components of the original Apache Spark (such as the Spark scheduler) outside of the Intel SGX enclave for efficiency. By design, the non-sensitive components of Spark can never access sensitive information in clear text;
- it provides a mechanism to securely and transparently retrieve the input of big data processing jobs. This mechanism is based on the encryption and decryption of files on the HDFS distributed file system [8]. This mechanism is presented in more details in Deliverable D1.3;
- it can be integrated with the TaLoS [4] and LibSEAL [5] libraries to respectively offer secure communications as well as job integrity verification. This mechanism is presented in more details in Deliverable D4.3;
- it exposes the same interface as the regular Apache Spark so that developers do not need to modify their existing Spark jobs and can continue to write jobs as usual, in Scala or Java, without having to learn new operators or the existence of the SGX-Spark components.

5.3.2 Validation and Integration

In this section we describe the integration and validation of the SGX-Spark component.

Application

The goal of SGX-Spark with regard to the SecureCloud project is to be able to securely execute big data processing jobs. Apache Spark perfectly fit this goal as it is a big data processing framework.

While Apache Spark can execute numerous jobs by exposing a wide interface, we consider a subset of existing big data processing jobs for the implementation and evaluation of SGX-Spark:

- Word count is a simple big data processing job that counts the number of words present in the given input file. To that end the job uses the map and reduce operators;
• **k-means** is a clustering algorithm that partitions \( n \) elements into \( k \) clusters. For example, if the elements are coordinates, the algorithm will group the points into multiple clusters so that two points belong to the same cluster if they are geographically close to each others. To that end the cluster proceeds to several iterations, which involve multiple Spark operators;

• **Random forest classification** is a statistical method for the classification of elements by constructing decision trees. The random forest method is a basic building block of numerous machine learning systems;

• **Naive Bayes** algorithms are used to construct classifiers in machine learning algorithms. The Naive Bayes algorithms are known to be simple yet scalable and require only a small number of training data. Examples of the usage of Naive Bayes algorithms are to mark whether an email is a spam or not, to automatically classify documents into various categories, for face recognition software, etc.

**Demonstration**

We have developed two demonstrators to show the SGX-Spark component. The first demonstrator runs each SGX-Spark component into a Docker image, in a distributed manner. More precisely, the master node, workers and client can execute independently on several physical machines. More details can be found in Deliverable D3.2.

The second demonstrator shows the usability of SGX-Spark with HDFS file encryption. This demonstrator starts a cluster of HDFS nodes that are in charge of storing the (encrypted) input files as well as the results from the execution of Spark jobs. The Spark job, more precisely the SGX-Spark secure executor, which executes inside an SGX enclave, then retrieves the encrypted input data before securely decrypting it inside the enclave. At the end of the job execution, it can encrypt the result before sending it to the HDFS cluster for storage.

**Performance evaluation**

SGX-Spark is still a work in progress. As such not all the jobs presented in the previous section have been implemented yet. In this section we present the performance of the **word count** and **k-means** jobs.

We deployed Apache Spark and SGX-Spark on a cluster of 5 identical machines connected via a 1GB network connection. Each machine embeds an SGX-enabled 4-cores Intel Xeon E3-1280 @3.90GHz with 64GB of RAM. One machine of the cluster acts as the master node, two machines act as worker and the last one acts as the client submitting the Spark job.

Figure 5.11 presents the performance ratio of SGX-Spark compared to the native Apache Spark as the input size increases, from 1kB to 20MB. First, it can be observed that the performance difference decreases as the input size increase. For example, in the case of **k-means**, SGX-Spark is 158% slower with a 1kB input size but only 33% slower with a 20MB input file. Second, the performance degradation of SGX-Spark with big data jobs (20MB in our experiments) is acceptable: 23% for **word count** and 33% with **k-means**.

Note that SGX-Spark is still under heavy development and we are working on performance improvements.
Integration

Several SecureCloud components can profit from Apache Spark. In particular it is used in two use-cases: Fraud Detection and Smart Metering. These use-cases are detailed in Deliverable D1.3. We briefly present them in the rest of this section.

• The fraud-detection use-case is developed by LACTEC. It is based on end-user requirements provided by the Brazilian utility company COPEL. In this use-case, end-users possess a smart meter that measures their electricity consumption. An important problem faced by COPEL is frauds, where malicious users would tamper with the smart meter in order to reduce their electricity bills. The goal of this use-case is to analyze the data sent by the smart meters to the COPEL company in order to detect such frauds.

• The UTFPR partner is working on a smart metering use-case. Similarly to the fraud-detection use-case, the smart metering use-case aggregates data from smart meters in order to generate electricity consumption bill statements to the users. The data is stored into the ChocolateCloud Key-Value Store component. The smart-metering big-data processing job first has to retrieve the stored measured values, before checking the authenticity of that data and generating and signing the bill statements for the different clients.

Given that these use-cases have to process a large amount of input data, the SGX-Spark component is a legitimate choice.
6 Summary

Deliverable D4.4, Integrated implementation of the micro-services for distributed big data applications, is a report issued by the SecureCloud project. This report presents a catalogue of the components implemented throughout the execution of Work Package 4 (WP4) in SecureCloud.

As specified in the Description of Action, this document presents a set of secure components for communication, storage, scheduling and big data processing. For distributed communication, it presents SCBR (Secure Content-Based Routing) in Section 2.1 and IBBE (Identity-Based Broadcast Encryption) in Section 2.2. For data management and storage, it presents the Distributed Data Store in Chapter 3. For distributed scheduling, it presents an extension to Kubernetes in Chapter 4. Finally, for big data processing, it presents SecureStreams in Section 5.1, Lightweight MapReduce in Section 5.2 and Secure Apache Spark Executor in Section 5.3.

Each component included in D4.4 follows the same template, with a section containing its description, and a section containing validation, evaluation and integration efforts done during the project. Additionally, the Distributed Data Store has evolved in the third year, therefore we presented its new developments in Section 3.3.

All software components mentioned in this document were also included in the Platform Services of SecureCloud’s architecture as described in Deliverable D1.3. The integrated final versions of these components are also delivered with this report. When the components are delivered as open source, their repository is indicated in Deliverable D1.3. There, all components have an integrated description, including motivation, solution, and functionality. D1.3 actually contains the same type of description for all components built in the project. Please refer to D1.3 whenever necessary to find an overall description for any particular component.
Bibliography


Deliverable 4.4  Integrated implementation of microservices for big data applications


[25] LEGaTO. LEGaTO is a low-energy toolset for heterogeneous computing.


43
Deliverable 4.4 Integrated implementation of microservices for big data applications


