Horizon 2020
European Union funding
for Research & Innovation

SecureCloud

Joint EU-Brazil Research and Innovation Action
SECURE BIG DATA PROCESSING IN UNTRUSTED CLOUDS

http://www.securecloudproject.eu/

Specification and design of middleware framework for distributed processing of data using SGX
D4.3

Due date: ... 2017
Submission date: ... 2017

Start date of project: 1 January 2016

Document type: Deliverable
Work package: WP4

Editor: Pascal Felber (UniNE)

Reviewer: ... (...)
... (...)

Dissemination Level

PU	Public	√
CO	Confidential, only for members of the consortium (including the Commission Services)	
CI	Classified, as referred to in Commission Decision 2001/844/EC	

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.

Revision history:

Version	Date	Authors	Institution	Description
0.1	2017/07/17	Aurélien Havet	UniNE	First draft

Tasks related to this deliverable:

Task No.	Task description	Partners involved°
...	...	UniNE*

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

Contents

List of Figures

1 Introduction

This work package aims at designing and implementing building blocks for developing big data applications on top of microservices (WP3), themselves deployed within containers (WP2) in the cloud. The main objectives are to make the development of cloud-based big data application easier, safer, and faster. More specifically, the work package will result in the following outcomes: T4.1: Secure and efficient (i.e., low latency and high throughput) communication mechanisms for transmitting big data between microservices, and between clients and big data applications. T4.2: A secure distributed key/value data store for big data application to store their data, and used by the map/reduce framework of T4.4 to store (intermediary) computation results. T4.3: Distributed scheduling mechanisms designed for executing computation tasks (running in microservices) close to the data they depend on, and for placing data close to associated compute tasks or to related data for better efficiency. T4.4: A generic framework for map/reduce computations with big data across microservices, as well as a collection of pre-defined components for big data processing. This workpackage is led by UniNE, with significant technical contributions from TUD, IMP, and CC.

The data deluge imposed by a world of ever-connected devices, whose most emblematic example is the **Internet of Things** (IoT), has fostered the emergence of novel data analytics and processing technologies to cope with the ever increasing **volume**, **velocity**, and **variety** of information that characterize the big data era. In particular, to support the continuous flow of information gathered by millions of IoT devices, data streams have emerged as a suitable paradigm to process flows of data at scale. However, as some of these data streams may convey sensitive information, stream processing requires support for end-to-end security guarantees in order to prevent third parties accessing restricted data.

This document therefore introduces SECURESTREAMS, our initial work on a middleware framework for developing and deploying secure stream processing on untrusted distributed environments. SECURESTREAMS supports the implementation, deployment, and execution of stream processing tasks in distributed settings, from large-scale clusters to multi-tenant Cloud infrastructures. More specifically, SECURESTREAMS adopts a message-oriented [1] middleware, which integrates with the SSL protocol [2] for data communication and the current version of Intel®'s **software guard extensions** (SGX) [3] to deliver end-to-end security guarantees along data stream processing stages. SECURESTREAMS can scale vertically and horizontally by adding or removing processing nodes at any stage of the pipeline, for example to dynamically adjust according to the current workload. The design of the SECURESTREAMS system is inspired by the dataflow programming paradigm [4]: the developer combines together several independent processing components (*e.g.*, mappers, reducers, sinks, shufflers, joiners) to compose specific processing pipes. Regarding packaging and deployment, SECURESTREAMS smoothly integrates with industrial-grade lightweight virtualization technologies like Docker [5].

In this document, we propose the following contributions: (i) we describe the design of SECURESTREAMS, (ii) we provide details of our reference implementation, in particular on how to smoothly integrate our runtime inside an SGX enclave, and (iii) we perform an extensive evaluation with micro-benchmarks, as well as with a real-world dataset.

The remainder of the document is organized as follows. Some related works to this topic are gathered in Chapter 2. To better understand the design of SECURESTREAMS, Chapter 3 delivers a brief introduction to today's SGX operating mechanisms. The architecture of SECURESTREAMS is then introduced in Chapter 4. The chapter 5 presents the implementation choices for a first prototype, and an example of a SECURESTREAMS program. Chapter 6 discusses our extensive evaluation, presenting a detailed analysis of micro-benchmark performances, as well as more comprehensive macro-benchmarks with real-world

datasets. Finally, Chapter 7 briefly describes our future work, and Chapter 7 presents a user manual of the first SECURESTREAMS prototype.

2 State-of-the-Art

Spark [6] has recently gained a lot of traction as prominent solution to implement efficient stream processing. It leverages Resilient Distributed Datasets (RDD) to provide a uniform view on the data to process. Despite its popularity, Spark only handles unencrypted data and hence does not offer security guarantees. Recent proposals [7] study possible software solutions to overcome this limitation.

Several big industrial players introduced their own stream processing solutions. These systems are mainly used to ingest massive amounts of data and efficiently perform (real-time) analytics. Twitter's Heron [8], and Google's Cloud DataFlow [9] are two prominent examples. These systems are typically deployed on the provider's premises and are not offered **as a service** to end-users.

A few dedicated solutions exist today for distributed stream processing using reactive programming. For instance, REACTIVE KAFKA [10] allows stream processing atop of Apache KAFKA [11, 12]. These solutions do not, however, support secure execution in a trusted execution environment.

More recently, some open-source middleware frameworks (*e.g.*, Apache SPARK [13], Apache STORM [14], INFINISPAN [15]) introduced APIs to allow developers to quickly set up and deploy stream processing infrastructures. These systems rely on the **Java** virtual machine (JVM) [16]. However, SGX currently imposes a hard memory limit of 128 MB to the enclaved code and data, at the cost of expensive encrypted memory paging mechanisms and serious performance overheads [17, 18] when this limit is crossed. Moreover, executing a fully-functional JVM inside an SGX enclave would currently involve significant re-engineering efforts.

DEFCON [19] relies also on the JVM. This event processing system focuses on security by enforcing constraints on event flows between processing units. The event flow control is enforced using application-level virtualisation to separate processing units in a *ad-hoc* JVM.

A few recent contributions tackle privacy-preserving data processing, particularly in a MapReduce scenario. This is the case of Airavat [20] and GUPT [21]. These systems leverage differential-privacy techniques [22] and can face a different threat model than the one supported by SGX and hence by SECURESTREAMS. In particular, when deploying such systems on a public infrastructure, one needs to trust the cloud provider. Our system greatly reduces the trust boundaries, and only requires trust of Intel® and their SGX implementation.

Some authors contest that public clouds may be secure enough some parts of an application. They propose to split the jobs, running only the critical parts in private clouds. A privacy-aware framework on hybrid clouds [23] has been proposed to work on tagged data, at different granularity levels. A MapReduce preprocessor splits data into private and public clouds according to their sensitivity. Sedic [24] does not offer the same tagging granularity, but proposes to automatically modify reducers to optimize the data transfers in a hybrid cloud. These solutions require splitting application and data in two parts (sensitive and not) and impose higher latencies due to data transfers between two different clouds. Yet, they cannot offer better security guarantees that the software stack itself offers, be it public or private.

MrCrypt [25] proposes using homomorphic encryption instead of trusted elements. Through static code analysis, it pinpoints different homomorphic encryption schemes for every data column. Still, some of the demonstrated benchmarks are ten times slower than the unecrypted execution. SECURESTREAMS avoids of complex encryption schemes, decrypts data entering enclaves and processes in plaintext.

The STYX [26] system uses partial homorphic encryption to allow for efficient stream processing in trusted cloud environments. Interestingly, the authors of that system mention Intel® SGX as possible alternative to deploy stream processing systems on trusted hardware offered by untrusted/malicious cloud environments. SECURESTREAMS offers insights on the performances of exactly this approach.

To best of our knowledge, SECURESTREAMS is the first lightweight and low-memory footprint stream processing framework that can fully execute within SGX enclaves.

As we described before, SECURESTREAMS is executing processes taking advantage of SGX enclaves inside Docker containers. SCONE [27], which is not yet openly available, is a recently introduced system that offers a secure container mechanism for Docker to leverage the SGX trusted execution support. It proposes a generic technology to embed any C program to execute inside an SGX enclave. Rather than generic programs, SECURESTREAMS offers support to execute a lightweight LUAVM inside an SGX enclave and securely execute chunks of LUA code inside it. In our experiments, we execute this LUAVM inside Docker containers.

3 Background

The design of SECURESTREAMS revolves around the availability of SGX features in the host machines. It consists in a **trusted execution environment** (TEE) recently introduced into Intel® SkyLake, similar in spirit to ARM TRUSTZONE [28] but much more powerful. Applications create secure **enclaves** to protect the integrity and the confidentiality of the data and the code being executed.

The SGX mechanism, as depicted in Figure 3.1, allows applications to access confidential data from inside the enclave. The architecture guarantees that an attacker with physical access to a machine will not be able to tamper with the application data without being noticed. The CPU package represents the security boundary. Moreover, data belonging to an enclave is automatically encrypted and authenticated when stored in main memory. A memory dump on a victim's machine will produce encrypted data. A **remote attestation protocol** allows one to verify that an enclave runs on a genuine Intel® processor with SGX. An application using enclaves must ship a signed (not encrypted) shared library (a shared object file in Linux) that can possibly be inspected by malicious attackers.

In the current version of SGX, the **enclave page cache** (EPC) is a 128 MB area of memory[1] predefined at boot to store enclaved code and data. At most around 90 MB can be used by application's memory pages, while the remaining area is used to maintain SGX metadata. Any access to an enclave page that does not reside in the EPC triggers a page fault. The SGX driver interacts with the CPU to choose which pages to evict. The traffic between the CPU and the system memory is kept confidential by the **memory encryption engine** (MEE) [30], also in charge of tamper resistance and replay protection. If a cache miss hits a protected region, the MEE encrypts or decrypts data before sending to, respectively fetching from, the system memory and performs integrity checks. Data can also be persisted on stable storage protected by a seal key. This allows the storage of certificates, waiving the need of a new remote attestation every time an enclave application restarts.

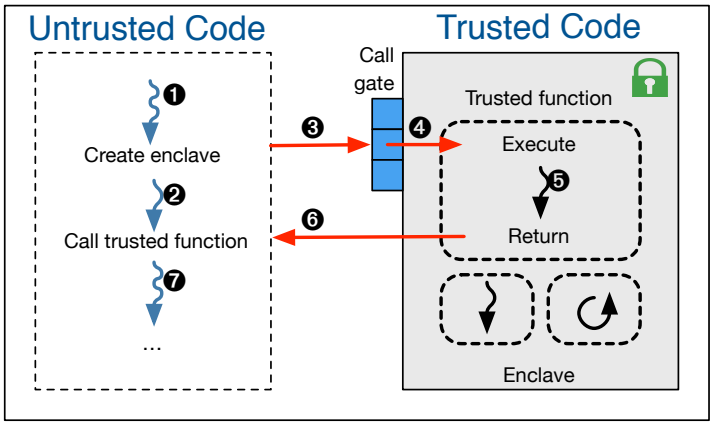


Figure 3.1: SGX core operating principles.

The execution flow of a program using SGX enclaves is like the following. First, an enclave is created (see Figure 3.1-❶). As soon as a program needs to execute a trusted function (❷), it executes SGX's primitive `ecall` (❸). The call goes through the SGX call gate to bring the execution flow inside the enclave (❹). Once the trusted function is executed by one of the enclave's threads (❺), its result is encrypted and sent back (❻) before giving back the control to the main processing thread (❼).

[1] Future releases of SGX might relax this limitation [29].

4 SECURESTREAMS Architecture

The architecture of SECURESTREAMS comprises a combination of two different types of base components: worker and router. A worker component continuously listens for incoming data by means of non-blocking I/O. As soon as data flows in, an application-dependent business logic is applied. A typical use-case is the deployment of a classic filter/map/reduce pattern from the functional programming paradigm [31]. In such a case, worker nodes execute only one function, namely `map`, `filter`, or `reduce`. A router component acts as a message broker between workers in the pipeline and transfers data between them according to a given **dispatching policy**. Figure 4.1 depicts a possible implementation of this dataflow pattern using the SECURESTREAMS middleware.

SECURESTREAMS is designed to support the processing of sensitive data inside SGX enclaves. As explained in the previous section, the **enclave page cache** (EPC) is currently limited to $128\,MB$. To overcome this limitation, we settled on a lightweight yet efficient embeddable runtime, based on the LUA virtual machine (LUAVM) [32] and the corresponding multi-paradigm scripting language [33]. The LUA runtime requires only few kilobytes of memory, it is designed to be embeddable, and as such it represents an ideal candidate to execute in the limited space allowed by the EPC. Moreover, the application-specific functions can be quickly prototyped in LUA, and even complex algorithms can be implemented with an almost 1:1 mapping from pseudo-code [34]. We provide further implementation details of the embedding of the LUAVM inside an SGX enclave in Section 5.

Each component is wrapped inside a lightweight Linux container (in our case, the **de facto** industrial standard Docker [5]). Each container embeds all the required dependencies, while guaranteeing the correctness of their configuration, within an isolated and reproducible execution environment. By doing so, a SECURESTREAMS processing pipeline can be easily deployed without changing the source code on different public or private infrastructures. For instance, this will allow developers to deploy SECURESTREAMS to Amazon EC2 container service [35], where SkyLake-enabled instances will soon be made available [36], or similarly to Google compute engine [37]. The deployment of the containers can be transparently executed on a single machine or a cluster, using a Docker network and the Docker Swarm scheduler [38].

The communication between workers and routers leverages ZEROMQ, a high-performance asynchronous messaging library [39]. Each router component hosts inbound and outbound queues. In particular, the routers use the ZEROMQ's pipeline pattern [40] with the PUSH-PULL socket types.

The inbound queue is a PULL socket. The messages are streamed from a set of anonymous[1] PUSH peers (*e.g.*, the upstream workers in the pipeline). The inbound queue uses a fair-queuing scheduling to deliver the message to the upper layer. Conversely, the outbound queue is a PUSH socket, sending messages using a round-robin algorithm to a set of anonymous PULL peers—*e.g.*, the downstream workers.

This design allows us to dynamically scale up and down each stage of the pipeline in order to adapt it to application's needs or the workload. Finally, ZEROMQ guarantees that the messages are delivered across each stage via reliable TCP channels.

We define the processing pipeline components and their chaining by means of Docker's Compose [41] description language. Listing 4.1 reports on a snippet of the description used to deploy the architecture in Figure 4.1. Once the processing pipeline is defined, the containers must be deployed on the computing infrastructure. We exploit the `constraint` placement mechanisms to enforce the Docker Swarm's scheduler in order to deploy workers requiring SGX capabilities into appropriate hosts. In the example, an `sgx_mapper` nodes is deployed on an SGX host by specifying `"constraint:type==sgx"` in the Compose description.

[1] **Anonymous** refers to a peer without any identity: the server socket ignores which worker sent the message.

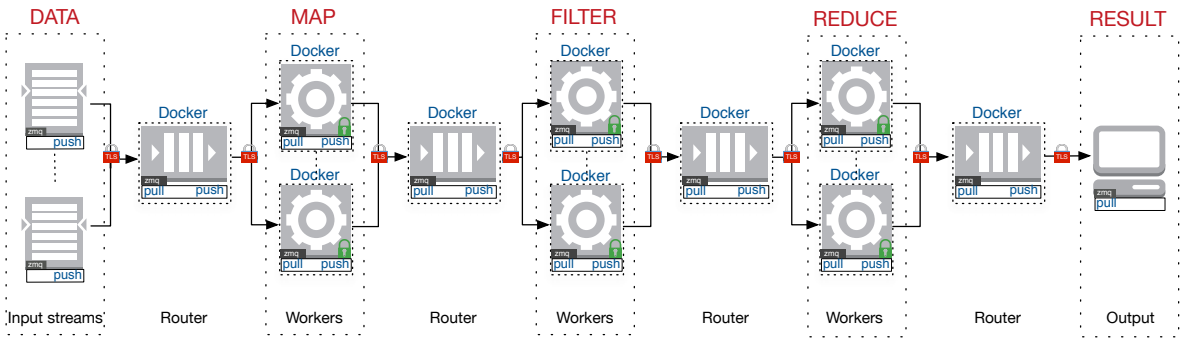


Figure 4.1: Example of SECURESTREAMS pipeline architecture.

Listing 4.1: SECURESTREAMS pipeline examples. Some attributes (`volume`, `networks`, `env_file`) are omitted.

```
sgx_mapper:
  image: "${IMAGE_SGX}"
  entrypoint: ./start.sh sgx-mapper.lua
  environment:
    - TO=tcp://router_mapper_filter:5557
    - FROM=tcp://router_data_mapper:5556
    - "constraint:type==sgx"
  devices:
    - "/dev/isgx"

router_data_mapper:
  image: "${IMAGE}"
  hostname: router_data_mapper
  entrypoint: lua router.lua
  environment:
    - TO=tcp://*:5556
    - FROM=tcp://*:5555
    - "constraint:type==sgx"

data_stream:
  image: "${IMAGE}"
  entrypoint: lua data-stream.lua
  environment:
    - TO=tcp://router_data_mapper:5555
    - "constraint:type==sgx"
    - DATA_FILE=the_stream.csv
```

5 SECURESTREAMS Prototype

SECURESTREAMS is implemented in LUA (v5.3). The implementation of the middleware itself requires careful engineering, especially with respect to the integration in the SGX enclaves (explained later). However, a SECURESTREAMS use-case can be implemented in remarkably few lines of code. For instance, the implementation of the map/filter/reduce accounts for only 120 lines of code (without counting the dependencies). The framework partially extends RXLUA [42], a library for reactive programming in LUA. RXLUA provides to the developer the required API to design a data stream processing pipeline following a dataflow programming pattern [4].

Listing 5.1 provides an example of a RXLUA program (and consequently a SECURESTREAMS program) to compute the average age of a population by chaining `:map`, `:filter`, and `:reduce` functions.[1] The `:subscribe` function performs the subscription of 3 functions to the data stream. Following the **observer** design pattern [43], these functions are observers, while the data stream is an observable.

SECURESTREAMS dynamically ships the business logic for each component into a dedicated Docker container and executes it. The communication between the Docker containers (the router and the worker components) happens through ZEROMQ (v4.1.2) and the corresponding LUA bindings [44]. Basically, SECURESTREAMS abstracts the underlying network and computing infrastructure from the developer, by relying on ZEROMQ and Docker.

Under the SGX threat model where the system software is completely untrusted, system calls are not allowed inside secure enclaves. As a consequence, porting a legacy application or runtime, such as the LUA interpreter, is challenging. To achieve this task, we traced all system calls made by the interpreter to the standard C library and replaced them by alternative implementations that either mimic the real behavior or discard the call. Our changes to the vanilla LUA source code consist of the addition of about 600 lines of code, or $2.5\,\%$ of its total size. By doing so, LUA programs operating on files, network sockets or any other input/output device do not execute as they normally do outside the enclaves. This inherent SGX limitation also reinforces the system security guarantees offered to the application developers. The SECURESTREAMS framework safely ships the data and code to enclaves. Hence, the LUA scripts executed within the SGX enclave do not use (read/write) files or sockets. Wrapper functions are nevertheless installed in the SGX-enabled LUAVM to prevent any of such attempts.

An additional constraint imposed by the secure SGX enclaves is the impossibility of dynamically linking code. The reason is that the assurance that a given code is running inside a SGX-enabled processor is made through the measurement of its content when the enclave is created. More specifically, this measurement is the result of `EREPORT` instruction, an SGX-specific report that computes a cryptographically secure hash of code, data and a few data structures, which overall builds a snapshot of the state of the enclave (including threads, memory heap size, etc.) and the processor (security version numbers, keys, etc.). Allowing more code to be linked dynamically at runtime would break the assurance given by the attestation mechanism on the integrity of the code being executed, allowing for example an attacker to load a malicious library inside the enclave.

In the case of LUA, a direct consequence is the impossibility of loading LUA extensions using the traditional dynamic linking technique. Every extension has to be statically compiled and packed with the enclave code. To ease the development of SECURESTREAMS applications, we statically compiled **json** [45], and **csv** [46] parsers within our enclaved LUA interpreter. With these libraries, the size of the VM and the complete runtime still remains reasonably small, approximately 220 KB ($19\,\%$ larger than the original).

[1] Note that in our evaluation the code executed by each worker is confined into its own LUA file.

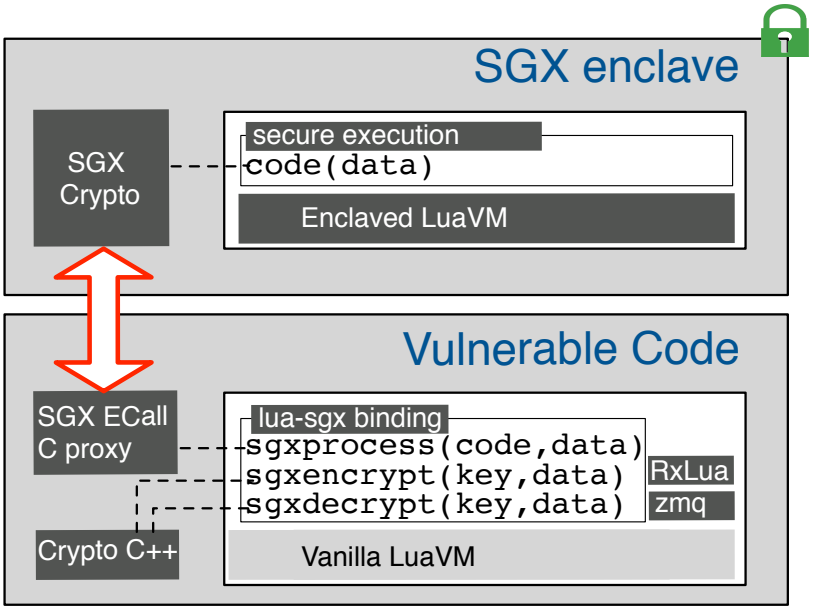


Figure 5.1: Integration between LUA and Intel® SGX.

While this restricted LUA has been adapted to run inside SGX enclaves, we still had to provide a support for communications and the reactive streams framework itself. To do so, we use an external vanilla LUA interpreter, with a couple adaptations that allowed the interaction with the SGX enclaves and the LUAVM therein. Figure 5.1 shows the resulting architecture. We extend the LUA interface with 3 functions: `sgxprocess`, `sgxencrypt`, and `sgxdecrypt`. The first one forwards the encrypted code and data to be processed in the enclave, while the remaining two provide cryptographic functionalities. In this work, we assume that attestation and key establishment was previously performed. As a result, keys safely reside within the enclave. We plan to release our implementation as open-source.

Listing 5.1: Example of process pipeline with RxLua.

```
Rx.Observable.fromTable(people)
 :map(
   function(person)
     return person.age
   end
 )
 :filter(
   function(age)
     return age > 18
   end
 )
 :reduce(
   function(accumulator, age)
     accumulator[count] = (accumulator.count
       or 0) + 1
     accumulator[sum] = (accumulator.sum
       or 0) + age
     return accumulator
   end
 )
 :subscribe(
   function(datas)
     print("Adult people average:",
       datas.sum / datas.count)
   end,
   function(err)
     print(err)
   end,
   function()
     print("Process complete!")
   end
 )
```

6 Evaluation

This chapter reports on our extensive evaluation of SECURESTREAMS. First, we present our evaluation settings. Then, we describe the real-world dataset used in our macro-benchmark experiments. We then dig into a set of micro-benchmarks that evaluate the overhead of running the LUAVM inside the SGX enclaves. Finally, we deploy a full SECURESTREAMS pipeline, scaling the number of workers per stage, to study the limits of the system in terms of throughput and scalability.

6.1 Evaluation Settings

We have experimented on machines using a Intel® Core™ i7-6700 processor [47] and $8\ GiB$ RAM. We use a cluster of 2 machines based on UBUNTU 14.04.1 LTS (kernel 4.2.0-42-generic). The choice of the Linux distribution is driven by compatibility reasons with the Intel® SGX SDK (v1.6). The machines run Docker (v1.13.0) and each node joins a Docker Swarm [38] (v1.2.5) using the Consul [48] (v0.5.2) discovery service. The Swarm manager and the discovery service are deployed on a distinct machine. Containers building the pipeline leverage the Docker overlay network to communicate to each other, while machines are physically interconnected using a switched $1\ Gbps$ network.

6.2 Input Dataset

In our experiments, we process a real-world dataset released by the **American Bureau of Transportation Statistics** [49]. The dataset reports on the flight departures and arrivals of 20 air carriers [50]. We implement a benchmark application atop of SECURESTREAMS to compute average delays and the total of delayed flights for each air carrier (cf. Table 6.1). We design and implement the full processing pipeline, that (i) parses the input datasets (in a comma-separated-value format) to data structure (map), (ii) filters data by relevancy (**i.e.**, if the data concerns a delayed flight), and (iii) finally reduces it to compute the desired information.[1] We use the 4 last years of the available dataset (from 2005 to 2008), for a total of 28 millions of entries to process and $2.73\ GB$ of data.

6.3 Micro-Benchmark: LUA in SGX

We begin our evaluation with a set of micro-benchmarks to evaluate performance of the integration of the LUAVM inside the SGX enclaves. First, we estimate the cost of execution for functions inside the enclave. This test averages the execution time of 1 million function calls, without any data transfer. We compare against the same result without SGX. While non-enclaved function calls took $23.6\ ns$, the performances inside the enclave drop down to on average $2.35\ s$—*i.e.*, approximately two orders of magnitude worse. We then assess the cost of copying data from the unshielded execution to the enclave and we compare it with the time required to compute the same on the native system. We initialize a buffer of $100\ MB$ with random data and copy its content inside the enclave. The data is split into chunks of increasing sizes. Our test executes one function call to transfer each chunk, until all data is transfered. Each point in the plot corresponds to the average of 20 runs. Correctness of the copies was verified by SHA256 digest comparison between reproduced memory areas.

Figure 6.1 shows the results for 4 different variants, comparing the native and the SGX version to only copy the data inside the enclave (**in**) or to copy it inside and copying it back (**in/out**). When using smaller

[1] This experiment is inspired by Kevin Webber's blog entry **diving into Akka streams**: `https://blog.redelastic.com/diving-into-akka-streams-2770b3aeabb0`.

System layer	Size (LoC)
DELAYEDFLIGHTS app	86
SECURESTREAMS library	350
RXLUA runtime	1,481
Total	1,917

Table 6.1: Benchmark app based on SECURESTREAMS.

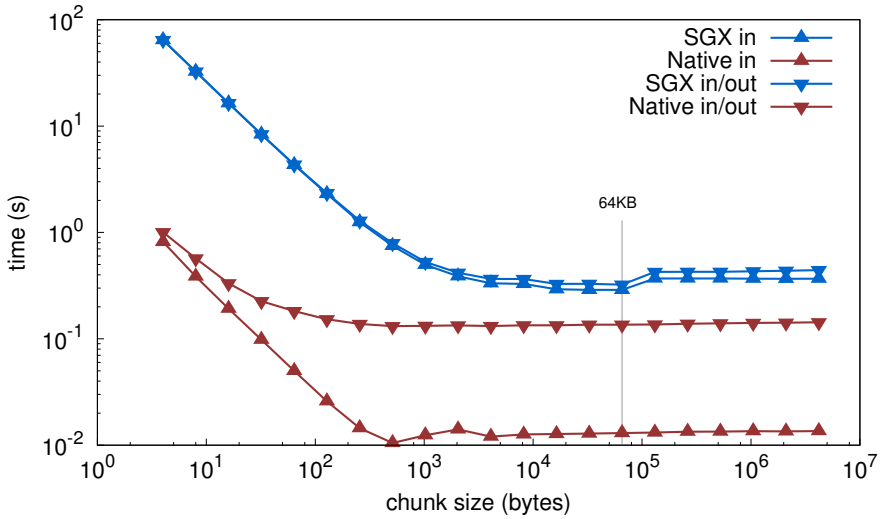


Figure 6.1: Execution time to copy 100 MB of memory inside an SGX enclave (**in**) or to copy it back outside **in/out**.

chunks, the function call overhead plays an important role in the total execution time. Moreover, we notice that the call overhead steadily drops until the chunk size reaches the size of 64 KB (vertical line). We can also notice that copying data back to non-SGX execution imposes an overhead of at most 20 % when compared to the one-way copy. These initial results are used as guidelines to drive the configuration of the streaming pipeline, in particular with respect to the size of the chunks exchanged between the processing stages. The larger the chunks, the smaller the overhead induced by the transfer of data within the SGX enclave.

Once the data and the code are copied inside the enclave, the LUAVM must indeed execute the code before returning the control. Hence, we evaluate here the raw performances of the enclaved SGX LUAVM. We select 6 available benchmarks from a standard suite of tests [51]. We based this choice on their library dependencies (by selecting the most standalone ones) and the number of input/output instructions they execute (selecting those with the fewest I/O). Each benchmark runs 20 times with the same pair of parameters of the original paper, shown in the even and odd lines of Table 6.2. Figure 6.2 depicts the total time (average and standard deviation) required to complete the execution of the 6 benchmarks. We use a bar chart plot, where we compare the results of the **Native** and **SGX** modes. For each of the 6 benchmarks, we present two bars next to each other (one per executing mode) to indicate the different

	configuration parameter	memory peak	ratio SGX/Native
dhrystone	50 K	275 KB	1.14
	5 M	275 KB	1.04
fannkuchredux	10	28 KB	0.99
	11	28 KB	1.04
nbody	2.5 M	38 KB	0.99
	25 M	38 KB	1.00
richards	10	106 KB	1.02
	100	191 KB	0.97
spectralnorm	500	52 KB	1.00
	5 K	404 KB	0.99
binarytrees	14	25 MB	1.18
	19	664 MB	4.76

Table 6.2: Parameters and memory usage for LUA benchmarks.

configuration parameters used. Finally, for the sake of readability, we use a different y-axis scale for the binarytrees case (from 0 to 400 s), on the right-side of the figure.

We note that, in the current version of SGX, it is required to pre-allocate all the memory area to be used by the enclave. The most memory-eager test (binarytrees) used more than 600 MB of memory, hence using the wall clock time comparison would not be fair for smaller tests. In such cases, almost the whole execution time is dedicated to memory allocation. Because of that, we subtracted the allocation time from the measurements of enclave executions, based on the average for the 20 runs. Fluctuations on this measurement produced slight variations in the execution times, sometimes producing the unexpected result of having SGX executions faster than native ones (by at most 3 %). Table 6.2 lists the parameters along with the maximum amount of memory used and the ratio between runtimes of SGX and Native executions. When the memory usage is low, the ratio between the Native and SGX versions is small—*e.g.*, less than 15 % in our experiments. However, when the amount of memory usage increases, performance drops to almost 5× worse, as reflected in the case of the **binarytrees** experiment. The smaller the memory usage, the better performance we can obtain from SGX enclaves.

Synthesis. To conclude this series of micro-benchmarks, taming the overhead of secured executions based on SGX requires balancing the size of the chunks transfered to the enclave with the memory usage within this enclave. In the context of stream processing systems, SECURESTREAMS therefore uses reactive programming principles to balance the load within processing stages in order to minimize the execution overhead.

6.4 Benchmark: Streaming Throughput

The previous set of experiments allowed us to verify that our design, implementation, and the integration of the LUAVM into the SGX enclaves is sound. Next, we deploy a SECURESTREAMS pipeline which includes mappers, filters and reducers. To measure the achievable throughput of our system, as well

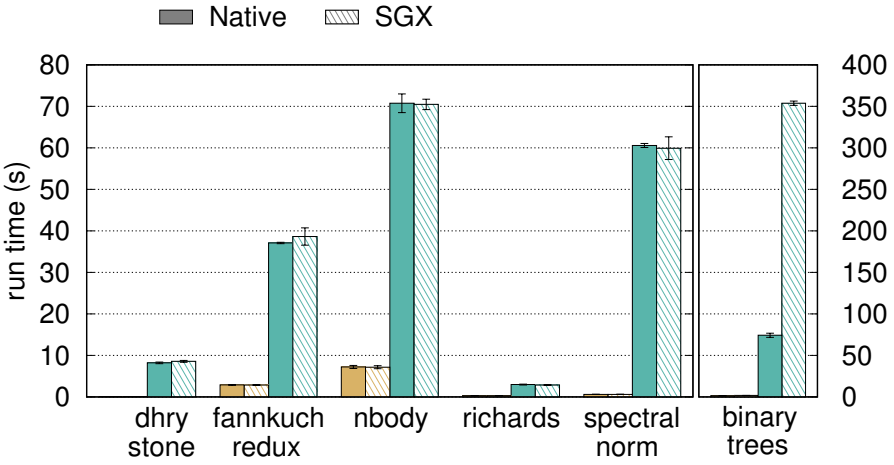


Figure 6.2: Enclave versus native running times for LUA benchmarks.

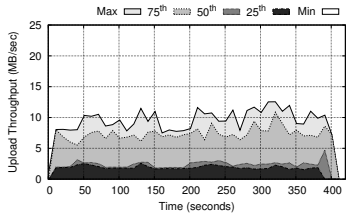
the network overhead of our architecture, we deploy the SECURESTREAMS pipeline in 3 different configurations. In each case, the setup of the pipeline architecture, **i.e.** the creation of the set of containers, has been done in $11\,s$ for the lightest configuration, in $15\,s$ for the heaviest one.

The first configuration allows the streaming framework to blindly bypass the SGX enclaves. Further, it does not encrypt the input dataset before injecting it into the pipeline. This mode operates as the baseline, yet completely **unsafe**, processing pipeline. The second mode encrypts the dataset but lets the encrypted packets skip the SGX enclaves. This configuration requires the deployers to trust the infrastructure operator. Finally, we deploy a fully secure pipeline, where the input dataset is encrypted and the data processing is operated inside the enclaves. The data nodes inject the dataset, split into 4 equally-sized parts, as fast as possible. We gather bandwidth measurements by exploiting Docker's internal monitoring and statistical module.

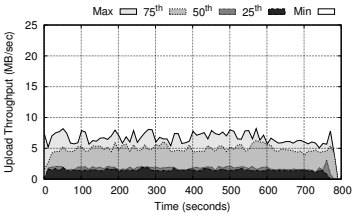
The results of these deployments are presented in Figure 6.3. For each of the mentioned configurations, we also vary the number of workers per stage, from one (Figure 6.3-a,d,g), two (Figure 6.3-b,e,h), or four (the remaining ones.) We use a representation based on stacked percentiles. The white bar at the bottom represents the minimum value, the pale grey on top the maximal value. Intermediate shades of grey represent the 25th-, 50th-, and 75th-percentiles. For instance, in Figure 6.3-a (our baseline) the median throughput at $200\,s$ into the experiment almost hits $7.5\,MB/s$, meaning that $50\,\%$ of the nodes in that moment are outputting data at $7.5\,MB/s$ or less. The baseline configuration, with only 1 worker per stage, completes in $420\,s$, with a peak of $12\,MB/s$. By doubling the number of workers reduces the processing time down to $250\,s$ (Figure 6.3-d), a speed-up of $41\,\%$. Scaling up the workers to 4 in the baseline configuration (Figure 6.3-g) did not produce a similar speed-up.

As we start injecting encrypted datasets (Figure 6.3-b and follow-up configurations with 2 and 4 workers), the processing time almost doubles ($795\,s$). The processing of the dataset is done after the messages are decrypted. We also pay a penalty in terms of overall throughput—**i.e.**, the median value rarely exceeds $5\,MB/s$. On the other hand, now we observe substantial speed-ups when increasing the workers per stage, down to $430\,s$ and $300\,s$ with 2 and 4 workers, respectively.

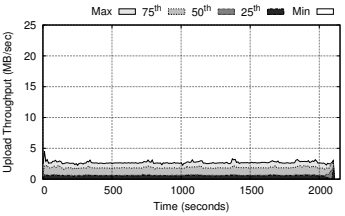
The deployment of the most secure set of configurations (right-most column of plots in Figure 6.3) shows that when using encrypted datasets and executing the stream processing inside SGX enclaves one must expect longer processing times and lower throughputs. This is the (expected) price to pay for higher-security guarantees across the full processing pipeline. Nevertheless, one can observe that the more workers the less penalty is imposed by the end-to-end security guarantees provided by SECURESTREAMS.



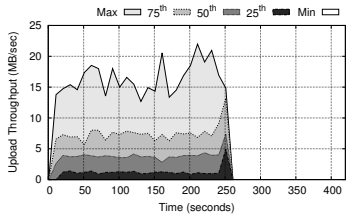
Data in clear text, no SGX, 1 worker per stage.



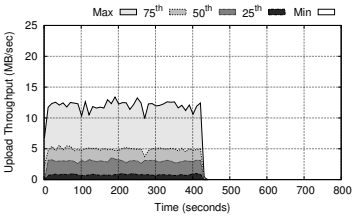
Encrypted data, no SGX, 1 worker per stage.



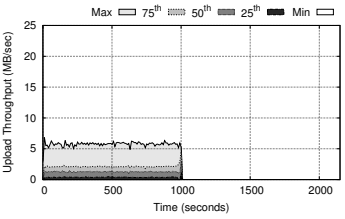
Encrypted data, processing SGX, 1 worker per stage.



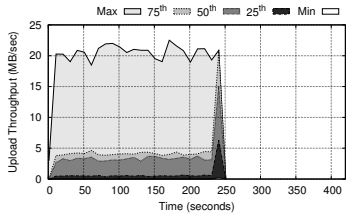
Data in clear text, no SGX, 2 workers per stage.



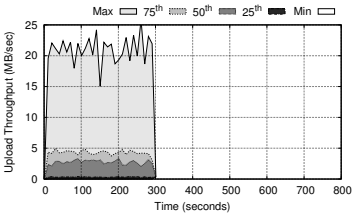
Encrypted data, no SGX, 2 workers per stage.



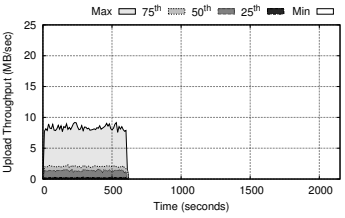
Encrypted data, processing SGX, 2 workers per stage.



Data in clear text, no SGX, 4 workers per stage.



Encrypted data, no SGX, 4 workers per stage.



Encrypted data, processing SGX, 4 workers per stage.

Figure 6.3: Streaming throughput.

6.5 Benchmark: Workers' Scalability

To conclude our evaluation, we study SECURESTREAMS in terms of scalability. We consider a pipeline scenario similar to Figure 4.1 with some variations in the number of workers deployed for each stage. We do so to better understand to what extents the underlying container scheduling system can exploit the hardware resources at its disposal.

First, we increase the number of workers for each stage of the pipeline, from 1 to 4. For each of the configurations, the experiment is repeated 5 times. We present average and standard deviation of the total

completion time to process the full dataset in Figure 6.4. As expected, we observe ideal speed-up from a configuration using 1 worker to that using 2 workers. However, in the configuration using 4 workers by stage, we do not reach the same acceleration. We explain this because, in this latter case, the number of deployed containers (which equals the sum of input data streams, workers, and routers, hence 20 containers) is greater than the number of physical cores of the hosts (8 for each of the 2 hosts used in our deployment—**i.e.**, 16 cores on our evaluation cluster).

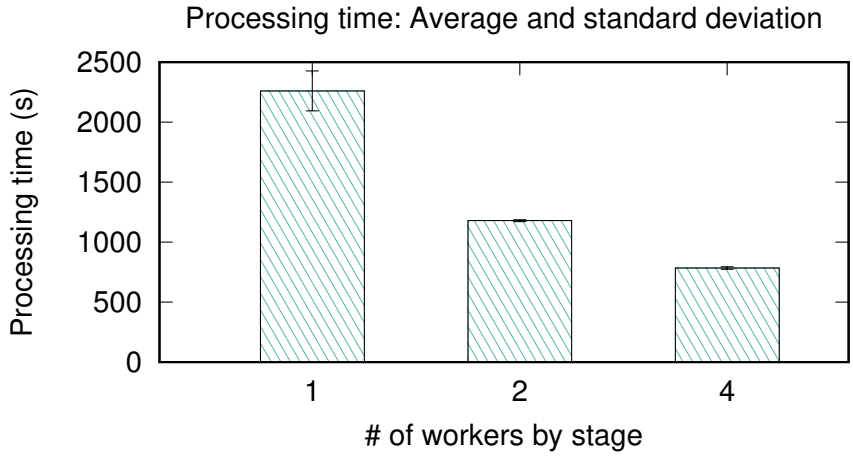


Figure 6.4: Scalability: processing time, average and standard deviation. The experiment is repeated 5 times, with a variation on the number of workers for each stage, each worker using SGX.

We also study the total completion time while increasing only the number of mapper workers in the first stage of the pipeline (which we identified as the one consuming most resources) from 1 to 16 and maintaining the numbers of filters and reducers in the following stages constant. As in the previous benchmark, the experiment is repeated 5 times for each configuration and we measure the average and standard deviation of the total completion time. Figure 6.5 presents the results. Here again, we observe ideal speed-up until the number of deployed containers reaches the number of physical cores. Beyond this number, we do not observe further improvements. These two experiments clearly show that the scalability of SECURESTREAMS according the number of deployed workers across the cluster is primarily limited by the total number of physical cores available.

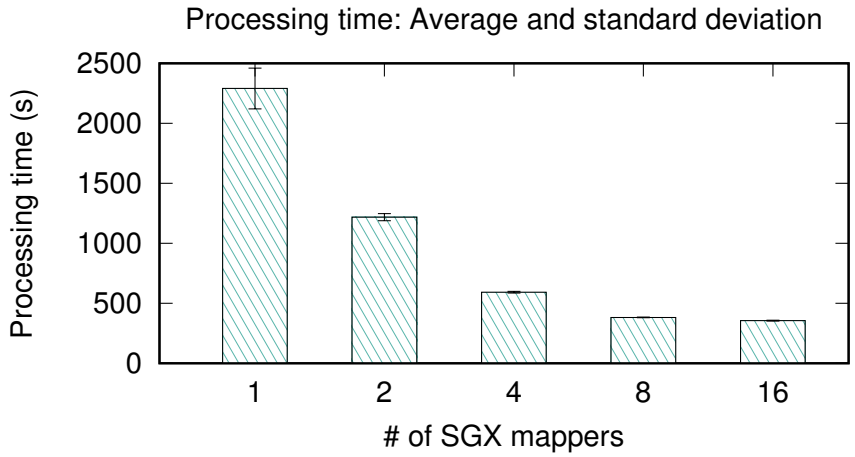


Figure 6.5: Scalability: processing time, average and standard deviation. The experiment is repeated 5 times, with a variation on the number of mappers SGX, other workers—1 filter worker and 1 reduce worker—do not use SGX.

Apart from this scalability limitation, there are other factors that reduce the observed streaming throughput, with or without involving the SGX enclaves. For instance, our throughout experiments highlight that the system does not manage to saturate the available network bandwidth in all cases. We believe this behaviour can be explained by the lack of optimizations in the application logic as well as possible tuning options of the inner ZEROMQ queues.

As part of our future work, we therefore plan to further investigate these effects and to build on this knowledge to only scale the appropriate workers in order to maximize the overall speed-up of the deployed application. In particular, we intend to leverage the elasticity of workers at runtime in order to cope with the memory constraints imposed by SGX and the configuration of the underlying hardware architecture, for each of the available nodes, in order to offer the best performances for secured execution of data stream processing applications built atop of SECURESTREAMS.

7 Summary

Secure stream processing is becoming a major concern in the era of the Internet of Things and big data. This document introduces our design and evaluation of SECURESTREAMS, an concise and efficient middleware framework to implement, deploy and evaluate secure stream processing pipelines for continuous data streams. The framework is designed to exploit the SGX **trusted execution environments** readily available in Intel®'s commodity processors, such as the latest SkyLake. We implemented the prototype of SECURESTREAMS in LUA and based its APIs on the reactive programming approach. Our initial evaluation results based on real-world traces are encouraging, and pave the way for deployment of stream processing systems over sensitive data on untrusted public clouds.

We plan in our future work to further extend and thoroughly evaluate SECURESTREAMS against other known approaches on secure stream processing, like STYX [26], MrCrypt [25] or DEFCON [19]. In particular, we plan to extend SECURESTREAMS with full automation of container deployments, as well as enriching the framework with a library of standard stream processing operators and efficient yet secure native plugins, to ease the development of complex stream processing pipelines.

8 User Manual

At the time of writing, SECURESTREAMS is not fully implemented, and some automations are yet missing in the launch of the process of a data stream. The following sections explain how the current implementation works, and how to run the proof-of-concept which SECURESTREAMS have been evaluated on.

SECURESTREAMS is running on a legacy standalone DOCKER SWARM cluster using an overlay network[1]. First of all, a DOCKER SWARM cluster have to be deployed on top of a group of hosts. Then each processing code has to be defined in separated files. Finally, the process of the stream can be launched from the manager of the DOCKER SWARM cluster using a DOCKER COMPOSE file. Have a look to the Section 8.4 to get an idea of how SECURESTREAMS can be used.

8.1 Infrastructure

The DOCKER SWARM cluster can be built with some hosts using:

- UBUNTU 16.04.2 LTS;
- DOCKER v. 1.13.0 (cf. Subsection 8.1.1);
- DOCKER SWARM v. 1.2.5 with the DOCKER image labeled `swarm:1.2.5`[2];
- the key-value store CONSUL using its latest DOCKER image[3].

[1] DOCKER overlay network: `https://docs.docker.com/engine/userguide/networking/#overlay-networks-in-swarm-mode`.

[2] DOCKER SWARM on DOCKERHUB: `https://hub.docker.com/_/swarm/`.

[3] CONSUL on DOCKERHUB: `https://hub.docker.com/r/progrium/consul/`.

8.1.1 Install DOCKER v. 1.13.0

Run the following commands to install DOCKER v. 1.13.0 on UBUNTU 16.04.2 LTS:

- Add the APT key:

```
$ sudo apt-key adv --keyserver hkp://p80.pool.sks-keyservers.net:80 \
  --recv-keys 58118E89F3A912897C070ADBF76221572C52609D
```

- Add the DOCKER repository in the sources list of APT:

```
$ sudo bash \
  -c 'echo "deb␣https://apt.dockerproject.org/repo␣ubuntu-xenial␣main" \
  > /etc/apt/sources.list.d/docker.list'
```

- Update APT:

```
$ sudo apt-get update
```

- Be sure to have this package installed:

```
$ sudo apt-get install linux-image-extra-$(uname -r)
```

- You can then list all available package of DOCKER and should find out the version 1.13.0:

```
$ apt-cache policy docker-engine
```

- Install DOCKER v. 1.13.0:

```
$ sudo apt-get install docker-engine=1.13.0-0~ubuntu-xenial
```

8.1.2 Configure DOCKER

DOCKER has to be configured correctly on each host with the key-value store CONSUL. If your host is using the boot manager SYSTEMD (like UBUNTU 16.04.2 LTS does by default), you can edit the file `/lib/systemd/system/docker.service` and change the value of `ExecStart` to `/usr/bin/docker daemon` the following options:

- `-H tcp://0.0.0.0:2375`: bind DOCKER on `0.0.0.0` for it to be reachable from outside the host;
- `--cluster-store=consul://172.16.0.2:8500`: provide the IP address (*e.g.* here `172.16.0.2`) and the port (*e.g.* here `8500`) of the key-value store CONSUL;
- `--cluster-advertise=enp0s31f6:2375`: provide the network interface (*e.g.* here `enp0s31f6`) and the port (*e.g.* here `2375`) of the DOCKER daemon;
- `--label type="sgx"`: the label to give to the DOCKER daemon in the DOCKER SWARM cluster (it is mandatory to target the correct host when launching the containers with DOCKER COMPOSE, as we will explain later).

Note that all this configuration can be defined easily using the tiny CLI tool `remote` (cf. Section 8.2).

8.2 Cluster Setup

A tiny CLI tool has been implemented to setup easily a DOCKER SWARM cluster and its overlay network. This one can be found in the directory `remote` of the repository `https://gitlab.securecloud.works/aurelien.havet/securestreams-poc`. Once the configuration for the different hosts used to build the cluster is given (cf. Subsection 8.2.1), the CLI can be used to build a DOCKER SWARM cluster (cf. Subsection 8.2.2). Note that the example of configuration given here is present in the file `config.yml.example`: you can copy this file to `config.yml` and modify it with your own configuration.

When the CLI is called without any command, it returns the list of the available ones:

```
$ ./remote.rb
Welcome to the remote manager !

Version: 0.1.0
Use one or several (you can chain them) of the following commands:

- help            Print this usage notice
- version         Print used version of Swarm
- ping            Check if connexions for each VM are well configured
- create          Create cluster
- network         Create Docker overlay network
- hostnames       Set node hostnames
- config-docker   Upgrade Docker with the last release, and configure
- test            Run hello world on each Docker node using the Swarm manager
- init-xp         Initialize nodes for experimental POC by getting repository

For example:      ./remote.rb ping config_docker create test network
```

8.2.1 Cluster Configuration

The configuration has to be given into the YAML file `config.yml` at the root of tool's code. This file has to be build like in the example 8.1. All configurations options are detailed below.

Listing 8.1: Configuration file example for configuring the DOCKER SWARM cluster using the CLI.

```yaml
ssh:
  user: 'host_user'
  identity_file: 'path/to/ssh_keys/id_rsa'
  public_key_file: 'path/to/ssh_keys/id_rsa.pub'
  proxy:
    user: 'proxy_user'
    host: 'proxy_host'

cluster:
  manager: 172.16.0.2
  manager_docker_port: 2380
  node_docker_port: 2380
  consul_ip: 172.16.0.3
  consul_port: 8500
  network_name: default_network

  nodes:
    -
      ip: 172.16.0.4
      name: 'sgx-1'
      network_if: 'enp0s31f6'
      type: sgx
      roles:
        - sgx-worker
    -
      ip: 172.16.0.5
      name: 'regular-1'
      network_if: 'enp0s31f6'
      type: regular
      roles:
        - regular-worker

swarm:
  image: swarm:1.2.0
  strategy: spread
```

SSH options

The CLI is using the protocol SSH to access to the different hosts of your cluster.

- `user` (**mandatory**): the user you want to use to connect to hosts;
- `identity_file` (**mandatory**): the path to the private key to use to connect to hosts;
- `public_key_file` (**optional**): the path to the public key to use to connect to hosts - required only for using CLI's feature for exporting a public key on remote hosts;
- `proxy` (**optional**): the proxy to use to reach the different hosts - required only to use a proxy, it will use the same key than the one for hosts.

Cluster options

Here are defined all the parameters related to the infrastructure of the cluster:

- `manager` (**mandatory**): IP address of host where the DOCKER SWARM manager will be deployed;
- `manager_docker_port` (**optional**): port where the DOCKER SWARM manager will listen (default value: `2381`);
- `node_docker_port` (**optional**): port where the DOCKER SWARM nodes will listen (default value: `2375`);
- `consul_ip` (**mandatory**): IP address of host where the CONSUL service will be deployed;
- `consul_port` (**optional**): port where the CONSUL service will listen (default value: `8500`);
- `network_name` (**optional**): name of the DOCKER overlay network (default value: `default_network`);
- `nodes` (**mandatory**): the list of the DOCKER SWARM nodes to set in the cluster, with for each:
  - `ip` (**mandatory**): IP address of the host;
  - `cpu` (**mandatory**): number of CPUs of the host;
  - `network_if` (**mandatory**): name of the network interface the host use over the cluster;
  - `type` (**mandatory**): the value to give to the label `type` which will be set on the DOCKER daemon of the current node;
  - `roles` (**optional**): a list of roles given as strings for the current node (useful for some specific operations like the provision of data to use for the POC experiment).

Swarm options

Finally, few parameters about DOCKER SWARM can be set:

- `image` (**optional**): the image of `Docker Swarm` to use (default value: `swarm:1.2.5`);
- `strategy` (**optional**): the strategy of scheduling to give to `Docker Swarm` (default value: `spread`).

8.2.2 Manage a cluster

We are assuming that your hosts are all reachable by SSH from the CLI. You can ensure that by running:

```
$ ./remote.rb ping
```

Once the configuration of the hosts to use for creating the cluster has been given, here are the few steps needed to create a cluster:

- **Create nodes**: this command will configure DOCKER on the nodes to use in the DOCKER SWARM cluster, by resetting the configuration file of the DOCKER service of each host, like explained in Subsection 8.1.2:

  ```
  $ ./remote.rb config-docker
  ```

- **Create the cluster**: this command will deploy the DOCKER SWARM cluster:

  ```
  $ ./remote.rb create
  ```

- **Create an overlay network**: this command will create a DOCKR overlay network:

  ```
  $ ./remote.rb network
  ```

All these commands can be run at once:

```
$ ./remote.rb ping config-docker create network
```

You can ensure that your cluster have been well created by running:

```
$ ./remote.rb test
```

If you want to recreate a cluster, use the command `remove` before, *e.g.*:

```
$ ./remote.rb remove
```

8.3 SECURESTREAMS Usage

The code of SECURESTREAMS can be retrieved from the repository `https://gitlab.securecloud.works/aurelien.havet/securestreams-poc`.

The main idea behind SECURESTREAMS is to run each steps of a processing pipeline (*e.g.* Listing 8.2) on different nodes of a cluster. To do this, SECURESTREAMS provides an API built on top of RXLUA, the Reactive Extensions for Lua[4] (see Subsection 8.3.1), and relies on ZEROMQ by the implementation of an extension of the RXLUA (see Subsection 8.3.2). Finally, SECURESTREAMS provides also another extension of RXLUA to use seamlessly the LUAVM modified to process some code in an SGX enclave (see Subsection 8.3.3).

8.3.1 RXLUA API

RXLUA is an API to implement a processsing pipeline in the way of the dataflow programming paradigm. This library permits to describe a processsing pipeline like shown in Listing 8.2.

Listing 8.2: Example of process pipeline with RxLua.

```
1  local Rx = require 'rx'
2  local csv = require 'csv'
3
4  local input = 'data.csv'
5  local output = 'output.log'
6  local start = os.time()
7  local results = {}
8
9  io.output(file)
10
11 Rx.Subject.fromFileByLine(input)
12   :map(
13     function(value)
14       if not value then return {} end
15
16       local array = csv.parse(value)
17       local event = {}
18       event.uniquecarrier = array[9]
19       event.arrdelay = array[15]
20
21       return event
22     end
23   )
24   :filter(
25     function(event)
```

[4] RXLUA, Reactive Extensions for Lua: `https://github.com/bjornbytes/RxLua`.

```
26        return (tonumber(event.arrdelay) or 0) > 0
27      end
28    )
29    :reduce(
30      function(accumulator, event)
31        carrier = accumulator[event.uniquecarrier] or {}
32        accumulator[event.uniquecarrier] = { count = (carrier.count
              or 0) + 1, total = (carrier.total or 0) + event.arrdelay }
33
34        return accumulator
35      end, {}
36    )
37    :subscribe(
38      function(datas)
39        for k,v in pairs(datas) do
40          carrier = results[k] or {}
41          results[k] = { count = (carrier.count or 0) + v.count,
                total = (carrier.total or 0) + v.total }
42        end
43      end,
44      function(error)
45        print(error)
46      end,
47      function()
48        io.write((os.time() - start) .. "\n")
49        io.close(ouput)
50
51        for k,v in pairs(results) do
52          print('Delays for carrier ' .. k .. ' -> ' .. (v.count == 0
               and 0 or (v.total / v.count)) .. ' average mins - ' ..
               math.tointeger(v.count) .. " delayed flights")
53        end
54
55        ZmqRx.sendZmqCompleted()
56        print('completed!')
57      end
58    )
```

8.3.2 RxLua extension for ZeroMQ

We have implemented a layer on top of RxLua to define a communication channel using ZeroMQ between two steps of a processing pipeline. This layer has been written in the file `zmq-rx.lua` and provides 2 functions:

- **Rx.Observable.fromZmqSocket(`socket`)**: defines a ZeroMQ where to get a data stream;
- **Rx.Observable:subscribeToSocket(`socket`)**: defines a ZeroMQ where to send a data stream.

The parameter `socket` is a formatted string giving the protocol, the address and the port of the socket (*e.g.* `'tcp://localhost:5555'`).

Using this layer for ZeroMQ, we can split the 5 steps of the processsing pipeline shown in Listing 8.2 on 5 independant files of Lua code:

- The data input steam:

Listing 8.3: Process pipeline with Zmq-RxLua: data stream.

```
1  local ZmqRx = require 'zmq-rx'
2
3  local input = 'data.csv'
4  local to_socket = 'tcp://localhost:5555'
5
6  ZmqRx.Subject.fromFileByLine(input)
7    :subscribeToSocket(to_socket) -- send to 'tcp://localhost:5555'
```

- The mapper from CSV line to data event:

Listing 8.4: Process pipeline with Zmq-RxLua: mapper.

```
1  local ZmqRx = require 'zmq-rx'
2  local csv = require 'csv'
3
4  local from_socket = 'tcp://localhost:5555'
5  local to_socket = 'tcp://localhost:5556'
6
7
8  ZmqRx.Subject.fromZmqSocket(from_socket) -- receive from
       'tcp://localhost:5555'
9    :map(
10     function(value)
11       if not value then return {} end
12
13       local array = csv.parse(value)
14       local event = {}
15       event.uniquecarrier = array[9]
```

```
16        event.arrdelay = array[15]
17
18        return event
19      end
20    )
21    :subscribeToSocket(to_socket) -- send to 'tcp://localhost:5556'
```

- The event filter:

Listing 8.5: Process pipeline with Zmq-RxLua: filter.

```
1  local ZmqRx = require 'zmq-rx'
2
3  local from_socket = 'tcp://localhost:5556'
4  local to_socket = 'tcp://localhost:5557'
5
6  ZmqRx.Subject.fromZmqSocket(from_socket) -- receive from
       'tcp://localhost:5556'
7    :filter(
8      function(event)
9        return (tonumber(event.arrdelay) or 0) > 0
10     end
11   )
12   :subscribeToSocket(to_socket) -- send to 'tcp://localhost:5557'
```

- The events reducer:

Listing 8.6: Process pipeline with Zmq-RxLua: reducer.

```
1  local ZmqRx = require 'zmq-rx'
2
3  local from_socket = 'tcp://localhost:5557'
4  local to_socket = 'tcp://localhost:5558'
5
6  ZmqRx.Subject.fromZmqSocket(from_socket) -- receive from
       'tcp://localhost:5557'
7    :reduce(
8      function(accumulator, event)
9        carrier = accumulator[event.uniquecarrier] or {}
10       accumulator[event.uniquecarrier] = { count = (carrier.count
             or 0) + 1, total = (carrier.total or 0) + event.arrdelay }
11
12       return accumulator
13     end, {}
14   )
```

```
15    :subscribeToSocket(to_socket) -- send to 'tcp://localhost:5558'
```

- The results printer:

Listing 8.7: Process pipeline with Zmq-RxLua: printer.

```
1  local ZmqRx = require 'zmq-rx'
2
3  local from_socket = 'tcp://localhost:5558'
4
5  local output = 'output.log'
6  local start = os.time()
7  local results = {}
8
9  io.output(file)
10
11 ZmqRx.Subject.fromZmqSocket(from_socket) -- receive from
       'tcp://localhost:5558'
12   :subscribe(
13     function(datas)
14       for k,v in pairs(datas) do
15         carrier = results[k] or {}
16         results[k] = { count = (carrier.count or 0) + v.count,
               total = (carrier.total or 0) + v.total }
17       end
18     end,
19     function(error)
20       print(error)
21     end,
22     function()
23       io.write((os.time() - start) .. "\n")
24       io.close(output)
25
26       for k,v in pairs(results) do
27         print('Delays for carrier ' .. k .. ' -> ' .. (v.count == 0
               and 0 or (v.total / v.count)) .. ' average mins - ' ..
               math.tointeger(v.count) .. " delayed flights")
28       end
29
30       ZmqRx.sendZmqCompleted()
31       print('completed!')
32     end
33   )
```

Between each step of this processing pipeline, data are streamed through some routers using the code from the file `router.lua`. These routers forward the data from a step to another one, without any process on them, using some ZEROMQ channels.

8.3.3 RxLua extension for SGX

SecureStreams can execute the code of a function `map`, `filter`, or `reduce` inside an SGX enclave. To do that, it uses the primitives provided by the modified LuaVM (which are `sgxprocess`, `sgxencrypt`, and `sgxdecrypt`, like described in Chapter 5). These primitives are called from the library implemented in the file `sgx.lua`. This library provides also some mocks for these primitives, in case the code has to be executed in a regular LuaVM.

SecureStreams provides finally a layer built on top of RxLua, and which provides the function `mapSGX`, `filterSGX`, and `reduceSGX` with whom the steps of mapping, filtering and reducing (described in Subsection 8.3.2) can be rewritten as following to be executed inside a SGX enclave:

- The mapper from CSV line to data event:

Listing 8.8: Process pipeline with Zmq-RxLua using SGX: mapper.

```
1  local ZmqRx = require 'sgx-rx'
2  local csv = require 'csv'
3
4  local from_socket = 'tcp://localhost:5555'
5  local to_socket = 'tcp://localhost:5556'
6
7
8  ZmqRx.Subject.fromZmqSocket(from_socket) -- receive from
       'tcp://localhost:5555'
9    :mapSGX(
10     function(value)
11       if not value then return {} end
12
13       local array = csv.parse(value)
14       local event = {}
15       event.uniquecarrier = array[9]
16       event.arrdelay = array[15]
17
18       return event
19     end
20   )
21   :subscribeToSocket(to_socket) -- send to 'tcp://localhost:5556'
```

- The event filter:

Listing 8.9: Process pipeline with Zmq-RxLua using SGX: filter.

```
1  local ZmqRx = require 'sgx-rx'
2
3  local from_socket = 'tcp://localhost:5556'
4  local to_socket = 'tcp://localhost:5557'
```

```
5
6  ZmqRx.Subject.fromZmqSocket(from_socket) -- receive from
       'tcp://localhost:5556'
7    :filterSGX(
8      function(event)
9        return (tonumber(event.arrdelay) or 0) > 0
10     end
11   )
12   :subscribeToSocket(to_socket) -- send to 'tcp://localhost:5557'
```

- The events reducer:

Listing 8.10: Process pipeline with Zmq-RxLua using SGX: reducer.

```
1  local ZmqRx = require 'sgx-rx'
2
3  local from_socket = 'tcp://localhost:5557'
4  local to_socket = 'tcp://localhost:5558'
5
6  ZmqRx.Subject.fromZmqSocket(from_socket) -- receive from
       'tcp://localhost:5557'
7    :reduceSGX(
8      function(accumulator, event)
9        carrier = accumulator[event.uniquecarrier] or {}
10       accumulator[event.uniquecarrier] = { count = (carrier.count
             or 0) + 1, total = (carrier.total or 0) + event.arrdelay }
11
12       return accumulator
13     end, {}
14   )
15   :subscribeToSocket(to_socket) -- send to 'tcp://localhost:5558'
```

Note that the code of the function passed to each step of the processing pipeline does not change between the regular version and the one usig SGX. Only the name of the required file for `ZmqRx` and the name of the used functions are different.

8.3.4 DOCKER images for SECURESTREAMS

SECURESTREAMS executes each step of the processing pipeline and each router in some DOCKER containers using either the image `lorel/zmqrxlua:lua` or the one `lorel/zmqrxlua:sgx`, depending if the code to execute uses the regular LUAVM or the one modified for SGX. These images have to be provided to the configuration file used by DOCKER COMPOSE to run all the containers of the pipeline on the DOCKER SWARM cluster (see Listing 8.11 for a complete configuration).

For each container, the following values are mandatory:

- **image**: the DOCKER image to use, either `lorel/zmqrxlua:lua` or `lorel/zmqrxlua:sgx`;
- **entrypoint**: the command to run when the container is started, either `lua` for using the regular LUAVM, or `./start.sh` for using the one modified for SGX, each of them followed by the name of the file with the code to execute for this node (*i.e.* the one for the corresponding step of the processing pipeline);
- **environment**: it expects a list of the environment variables to pass to the container, typically `TO` and `FROM` with the address for the corresponding sockets, and a constraint used by the DOCKER SWARM scheduler to launch the given container on a specific host of the cluster;
- **env_file**: pass a file to the container with some environment wariables, typically the ones for setting the logger and the endpoint for the common controller of the pipeline (here the printer) - convenient for giving some variables shared by all containers;
- **volumes**: the list of the volumes to mount into the container - here have to be mount the directory with the common code of SECURESTREAMS and the file with the code of the processing pipeline step to be executed by the current container;
- **networks**: the list of the networks joined by the container - it must contain the overlay network shared by all containers, and defined at the end of this DOCKER COMPOSE file.

Listing 8.11: Configuration file example for DOCKER COMPOSE.

```
version: '2'

services:
  printer:
    image: "lorel/zmqrxlua:lua"
    entrypoint: lua print-results.lua
    environment:
      - FROM=tcp://routerreduceprinter:5562
      - OUTPUT=$RESULT_OUTPUT
      - "constraint:type==printer"
    env_file: global.env
    volumes:
```

```
      - "/path/to/securestream/code:/root/worker"

    networks:
      xp:
        aliases:
          - controller

  routerreduceprinter:
    image: "lorel/zmqrxlua:lua"
    hostname: routerreduceprinter
    entrypoint: lua router.lua
    environment:
      - TO=tcp://*:5562
      - FROM=tcp://*:5561
      - "constraint:type==worker"
    env_file: global.env
    volumes:
      - "/path/to/securestream/code:/root/worker"
    networks:
      - xp

  reduce:
    image: "lorel/zmqrxlua:lua"
    entrypoint: lua reduce-events.lua
    environment:
      - TO=tcp://routerreduceprinter:5561
      - FROM=tcp://routerfilterreduce:5560
      - "constraint:type==worker"
    env_file: global.env
    volumes:
      - "/path/to/securestream/code:/root/worker"
    networks:
      - xp

  routerfilterreduce:
    image: "lorel/zmqrxlua:lua"
    hostname: routerfilterreduce
    entrypoint: lua router.lua
    environment:
      - TO=tcp://*:5560
      - FROM=tcp://*:5559
      - "constraint:type==worker"
    env_file: global.env
```

```
    volumes:
      - "/path/to/securestream/code:/root/worker"
    networks:
      - xp

  filter:
    image: "lorel/zmqrxlua:lua"
    entrypoint: lua filter-event.lua
    environment:
      - TO=tcp://routerfilterreduce:5559
      - FROM=tcp://routermapperfilter:5558
      - "constraint:type==worker"
    env_file: global.env
    volumes:
      - "/path/to/securestream/code:/root/worker"
    networks:
      - xp

  routermapperfilter:
    image: "lorel/zmqrxlua:lua"
    hostname: routermapperfilter
    entrypoint: lua router.lua
    environment:
      - TO=tcp://*:5558
      - FROM=tcp://*:5557
      - "constraint:type==worker"
    env_file: global.env
    volumes:
      - "/path/to/securestream/code:/root/worker"
    networks:
      - xp

  mapper:
    image: "lorel/zmqrxlua:lua"
    entrypoint: lua map-csv-to-event.lua
    environment:
      - TO=tcp://routermapperfilter:5557
      - FROM=tcp://routerdatamapper:5556
      - "constraint:type==worker"
    env_file: global.env
    volumes:
      - "/path/to/securestream/code:/root/worker"
    networks:
```

```
      - xp

  routerdatamapper:
    image: "lorel/zmqrxlua:lua"
    hostname: routerdatamapper
    entrypoint: lua router.lua
    environment:
      - TO=tcp://*:5556
      - FROM=tcp://*:5555
      - "constraint:type==worker"
    env_file: global.env
    volumes:
      - "/path/to/securestream/code:/root/worker"
    networks:
      - xp

  data:
    image: "lorel/zmqrxlua:lua"
    entrypoint: lua data-stream.lua
    environment:
      - TO=tcp://routerdatamapper:5555
      - "constraint:type==data"
      - DATA_FILE=data/2008.csv
    env_file: global.env
    volumes:
      - "/path/to/securestream/code:/root/worker"
    networks:
      - xp


networks:
  xp:
    external:
      name: default_network
```

8.4 How to run the Proof-of-Concept

You can find here all the steps to run the proof-of-concept of SECURESTREAMS.

8.4.1 Create a DOCKER SWARM cluster and its overlay network

After a correct configuration of the remote CLI (see Subsection 8.2.1 for more explanations), use this one to configure a DOCKER SWARM cluster and its overlay network:

```
$ ./remote.rb ping config-docker create network
```

8.4.2 Import SECURESTREAMS code on each node

The proof-of-concept is based of data which have to be downloaded on the nodes of the cluster where the data streams will be run. You have to indicate in the file `config.yml` of the remote CLI which nodes need to download this code by using the attributes `roles` for that:

Listing 8.12: Configure some roles for a node of the cluster

```
cluster:
  ...

  nodes:
    -
      ip: 172.16.0.4
      name: 'sgx-1'
      network_if: 'enp0s31f6'
      type: sgx
      roles:
        - data1
        - data2
        - data3
        - data4
```

You have also to provide in the same file some details about the repository of SECURESTREAMS:

Listing 8.13: Configure the repository of SECURESTREAMS

```
poc:
  working_dir: '/home/ubuntu/zmqrxlua'
  project_name: 'zmqrxlua-poc'
  experiment_path: 'experiment'
  remote_repo: repo_url
```

Give as `remote_repo` value the address of the repository of SECURESTREAMS: `https://gitlab.securecloud.works/aurelien.havet/securestreams-poc`.

After a correct configuration of the remote CLI, use this one to import the code of SECURESTREAMS on each node:

```
$ ./remote.rb init-xp
```

8.4.3 Run XP on the cluster

Finally, you only have to connect to the host of your DOCKER SWARM manager, go to the directory `experiment/test` of the cloned repository, and execute the script `run.sh`.

Bibliography

[1] E. Curry, **Message-Oriented Middleware**. John Wiley & Sons, Ltd, 2005, pp. 1–28.

[2] A. Freier, P. Karlton, and P. Kocher, "The secure sockets layer (SSL) protocol version 3.0," 2011.

[3] V. Costan and S. Devadas, "Intel® SGX explained," Cryptology ePrint Archive, Report 2016/086, 2016., Tech. Rep.

[4] T. Uustalu and V. Vene, **The Essence of Dataflow Programming**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 135–167. [Online]. Available: http://dx.doi.org/10.1007/11894100_5

[5] "Docker," https://www.docker.com, Accessed on 12/03/2017.

[6] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in **Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles**, ser. SOSP '13. New York, NY, USA: ACM, 2013, pp. 423–438. [Online]. Available: http://doi.acm.org/10.1145/2517349.2522737

[7] S. Y. Shah, B. Paulovicks, and P. Zerfos, "Data-at-rest security for Spark," in **2016 IEEE International Conference on Big Data (Big Data)**, Dec 2016, pp. 1464–1473.

[8] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja, "Twitter Heron: Stream Processing at Scale," in **Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data**, ser. SIGMOD '15. New York, NY, USA: ACM, 2015, pp. 239–250. [Online]. Available: http://doi.acm.org/10.1145/2723372.2742788

[9] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle, "The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-scale, Unbounded, Out-of-order Data Processing," **Proc. VLDB Endow.**, vol. 8, no. 12, pp. 1792–1803, Aug. 2015. [Online]. Available: http://dx.doi.org/10.14778/2824032.2824076

[10] "Reactive Streams for Kafka," https://github.com/akka/reactive-kafka, Accessed on 12/03/2017.

[11] "Apache Kafka," https://kafka.apache.org, Accessed on 12/03/2017.

[12] J. Kreps, N. Narkhede, J. Rao **et al.**, "Kafka: A distributed messaging system for log processing," in **Proceedings of the NetDB**, 2011, pp. 1–7.

[13] "Spark Streaming," https://spark.apache.org/streaming, Accessed on 12/03/2017.

[14] "Apache Storm," http://storm.apache.org, Accessed on 12/03/2017.

[15] "Infinispan," http://infinispan.org, Accessed on 12/03/2017.

[16] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, **The Java Virtual Machine Specification: Java SE 8 Edition**. Pearson Education, 2014.

[17] R. Pires, M. Pasin, P. Felber, and C. Fetzer, "Secure content-based routing using intel® software guard extensions," in **Proceedings of the 17th International Middleware Conference**, ser. Middleware '16. New York, NY, USA: ACM, 2016, pp. 10:1–10:10. [Online]. Available: http://doi.acm.org/10.1145/2988336.2988346

[18] S. Brenner, C. Wulf, D. Goltzsche, N. Weichbrodt, M. Lorenz, C. Fetzer, P. Pietzuch, and R. Kapitza, "SecureKeeper: Confidential ZooKeeper Using Intel® SGX," in **Proceedings of the 17th International Middleware Conference**, ser. Middleware '16. ACM, 2016, pp. 14:1–14:13.

[19] M. Migliavacca, I. Papagiannis, D. M. Eyers, B. Shand, J. Bacon, and P. Pietzuch, "Defcon: High-performance event processing with information security," in **Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference**, ser. USENIXATC'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–1. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855840.1855841

[20] I. Roy, S. T. V. Setty, A. Kilzer, V. Shmatikov, and E. Witchel, "Airavat: Security and Privacy for MapReduce," in **Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation**, ser. NSDI'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 20–20. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855711.1855731

[21] P. Mohan, A. Thakurta, E. Shi, D. Song, and D. Culler, "GUPT: Privacy Preserving Data Analysis Made Easy," in **Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data**, ser. SIGMOD '12. New York, NY, USA: ACM, 2012, pp. 349–360. [Online]. Available: http://doi.acm.org/10.1145/2213836.2213876

[22] C. Dwork, F. McSherry, K. Nissim, and A. Smith, "Calibrating noise to sensitivity in private data analysis," in **Theory of Cryptography Conference**. Springer, 2006, pp. 265–284.

[23] X. Xu and X. Zhao, "A framework for privacy-aware computing on hybrid clouds with mixed-sensitivity data," in **IEEE International Symposium on Big Data Security on CloudIEEE International Symposium on Big Data Security on Cloud**. IEEE, 2015, pp. 1344–1349.

[24] K. Zhang, X. Zhou, Y. Chen, X. Wang, and Y. Ruan, "Sedic: privacy-aware data intensive computing on hybrid clouds," in **Proceedings of the 18th ACM conference on Computer and communications security**. ACM, 2011, pp. 515–526.

[25] S. D. Tetali, M. Lesani, R. Majumdar, and T. Millstein, "Mrcrypt: Static analysis for secure cloud computations," **ACM Sigplan Notices**, vol. 48, no. 10, pp. 271–286, 2013.

[26] J. J. Stephen, S. Savvides, V. Sundaram, M. S. Ardekani, and P. Eugster, "STYX: Stream Processing with Trustworthy Cloud-based Execution," in **Proceedings of the Seventh ACM Symposium on Cloud Computing**, ser. SoCC '16. New York, NY, USA: ACM, 2016, pp. 348–360. [Online]. Available: http://doi.acm.org/10.1145/2987550.2987574

[27] P. R. Pietzuch, S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. Stillwell, D. Goltzsche, D. Eyers, K. RÃ¼diger, and C. Fetzer, "SCONE: Secure Linux Containers with Intel SGX," in **12th USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2016**. USENIX, Nov. 2016. [Online]. Available: http://spiral.imperial.ac.uk/handle/10044/1/42263

[28] "Building a Secure System using TrustZone® Technology," **ARM Limited**, 2009.

[29] F. McKeen, I. Alexandrovich, I. Anati, D. Caspi, S. Johnson, R. Leslie-Hurd, and C. Rozas, "Intel® software guard extensions (intel® sgx) support for dynamic memory management inside an enclave," in **Proceedings of the Hardware and Architectural Support for Security and Privacy 2016**, ser. HASP 2016. New York, NY, USA: ACM, 2016, pp. 10:1–10:9. [Online]. Available: http://doi.acm.org/10.1145/2948618.2954331

[30] S. Gueron, "A Memory Encryption Engine Suitable for General Purpose Processors." **IACR Cryptology ePrint Archive**, vol. 2016, p. 204, 2016.

[31] R. Bird and P. Wadler, **Introduction to Functional Programming**. Prentice Hall, Mar. 1988.

[32] R. Ierusalimschy, L. H. de Figueiredo, and W. C. Filho, "Lua - An Extensible Extension Language," **Software: Practice and Experience**, vol. 26, no. 6, pp. 635–652, Jun. 1996.

[33] "Lua," www.lua.org, Accessed on 12/03/2017.

[34] L. Leonini, E. Rivière, and P. Felber, "Splay: Distributed systems evaluation made simple (or how to turn ideas into live systems in a breeze)," in **Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation**, ser. NSDI'09. Berkeley, CA, USA: USENIX Association, 2009, pp. 185–198. [Online]. Available: http://dl.acm.org/citation.cfm?id=1558977.1558990

[35] "Amazon EC2 Container Service," https://aws.amazon.com/ecs/, Accessed on 12/03/2017.

[36] "Amazon EC2 and SkyLake CPUs," http://amzn.to/2nmIiH9, Accessed on 12/03/2017.

[37] "Google Compute Engine and SkyLake CPUs," http://for.tn/2lLdUtD, Accessed on 12/03/2017.

[38] "Docker Swarm," https://www.docker.com/products/docker-swarm, Accessed on 12/03/2017.

[39] "ZeroMQ," http://zeromq.org, Accessed on 12/03/2017.

[40] "ZeroMQ Pipeline," https://rfc.zeromq.org/30/, Accessed on 12/03/2017.

[41] "Docker Comopse," https://docs.docker.com/compose/, Accessed on 12/03/2017.

[42] "Reactive Extensions for Lua," https://github.com/bjornbytes/RxLua, Accessed on 12/03/2017.

[43] C. Szallies, "On using the observer design pattern," **XP-002323533,(Aug. 21, 1997)**, vol. 9, 1997.

[44] "Lua binding to ZeroMQ," https://github.com/zeromq/lzmq, Accessed on 12/03/2017.

[45] "The JavaScript Object Notation (JSON) Data Interchange Format," RFC 7159, Mar. 2014. [Online]. Available: https://rfc-editor.org/rfc/rfc7159.txt

[46] Y. Shafranovich, "Common Format and MIME Type for Comma-Separated Values (CSV) Files," RFC 4180, Oct. 2005. [Online]. Available: https://rfc-editor.org/rfc/rfc4180.txt

[47] "Intel® Core™ i7-6700," http://ark.intel.com/products/88196/Intel-Core-i7-6700-Processor-8M-Cache-up-to-4_00-GHz, Accessed on 12/03/2017.

[48] "Consul," http://consul.io, Accessed on 12/03/2017.

[49] "RITA | BTS," http://www.transtats.bts.gov/OT_Delay/OT_DelayCause1.asp, Accessed on 12/03/2017.

[50] "Data Expo'09 ASA Statistics Computing and Graphics," http://stat-computing.org/dataexpo/2009/the-data.html, Accessed on 12/03/2017.

[51] C. F. Bolz and L. Tratt, "The impact of meta-tracing on vm design and implementation," **Science of Computer Programming**, vol. 98, pp. 408–421, 2015.