Introducing SECURESTREAMS: Scalable Middleware for Reactive and Secure Data Stream Processing

Aurelien Havet*, Valerio Schiavoni*, Pascal Felber*, Romain Rouvoy†
*University of Neuchâtel, Switzerland. Email: first.last@unine.ch
†Univ. Lille / Inria / IUF, France. Email: romain.rouvoy@univ-lille.fr

***Abstract*—We introduce SECURESTREAMS, a middleware framework for secure stream processing. Its design builds on Intel's *Secure Guard Extensions* (SGX) to guarantee the privacy and the integrity of the data being processed. Our initial experimental results of SECURESTREAMS are promising: the framework is easy to use, and delivers high throughput, enabling developers to implement complex processing pipelines in a few lines of scripting code.**

***Keywords*-streaming; reactive programming; Lua; SGX**

I. INTRODUCTION

Data streams are more and more predominant in today's era of big data. In the world of all-connected and the *Internet-of-Things* (IoT), on market places, or elsewhere, data is continuously produced and consumed. However, such data streams requires more and more to be processed with reliability, scalability and security in mind. This paper introduces SECURESTREAMS, our initial work on a middleware framework for developing secure stream processing in the Cloud. SECURESTREAMS supports the implementation, deployment and the execution of stream processing tasks in distributed settings, from clusters to large-scale Cloud infrastructures. SECURESTREAMS adopts of message-oriented, responsive, resilient to faults middleware design. Finally, it scales horizontally and vertically seamlessly. Briefly, its design is inspired by the dataflow programming paradigm [1]: the developer combines together several independent processing components (*e.g.*, mappers, reducers, sinks, shufflers, joiners) to compose several processing pipes.The framework integrates an abstraction of the required deployment infrastructure. It smoothly integrate with industrial-grade lightweight virtualization technologies (*e.g.*, Docker [2]). It aims to be secure: communication channels use the SSL protocol for data communication. Finally, our design intends to exploit the secure processing capabilities offered by trusted hardware *enclaves*, nowadays widely available into mass-market thanks to the introduction of Intel's *Software Guard eXtensions* (SGX) [3] in the SkyLake processors [4].

Few mainstream solutions exist today for distributed stream processing. Reactive Kafka [5] allows stream processing atop of Apache Kafka. More recently, few open-source solutions (*e.g.*, Apache Spark Streaming [6], Apache Storm [7], Infinispan [8]) introduced APIs to allow developers to quickly setup and deploy stream processing infrastructures. These systems rely on the *Java Virtual Machine* (JVM). However, SGX currently imposes a hard memory limit of 128 MB to the enclaved code, at the cost of expensive encrypted memory paging mechanisms and serious performance leaks [9] when this limit is crossed. SECURESTREAMS proposes a lightweight and low-memory footprint framework that can fully execute within SGX enclaves. We detail our implementation choices in Section III.

To recap, our **contributions** are the followings: (i) an architecture of SECURESTREAMS, (ii) details of our implementation choices, and (iii) results of our preliminary evaluation based on a real-world dataset.

This paper is organized as follows. The architecture of SECURESTREAMS is described in Section II. Our implementation choices and an example of SECURESTREAMS program are presented in Section III. Section IV reports on our preliminary evaluation with respect to throughput and scalability results. Before concluding, we wrap up by briefly describing our future work in Section V.

II. ARCHITECTURE

The architecture of SECURESTREAMS comprises a combination two different types of components: worker and router. A worker component continuously listens for incoming data by means of non-blocking I/O. As soon as a data flows in, some application-dependent business logic is applied. A typical use-case is the deployment of a processing pipeline, with worker nodes that execute `map`, `filter` and `reduce` functions. A router acts as a message broker between workers in the pipeline and transfers data between workers according to some dispatching policy. Figure 1 depicts a possible implementation of MapReduce using the SECURESTREAMS middleware.

SECURESTREAMS is designed to allow processing of sensible data inside SGX enclaves. However, the *Enclave Page Cache* (EPC) is currently limited at 128 MB. To overcome these limitations, we settled on a lightweight yet efficient embeddable runtime based on the Lua Virtual Machine [10] and the corresponding multi-paradigm scripting language [11]. Note that the Lua runtime requires only few kilobytes of memory, and thus representing an

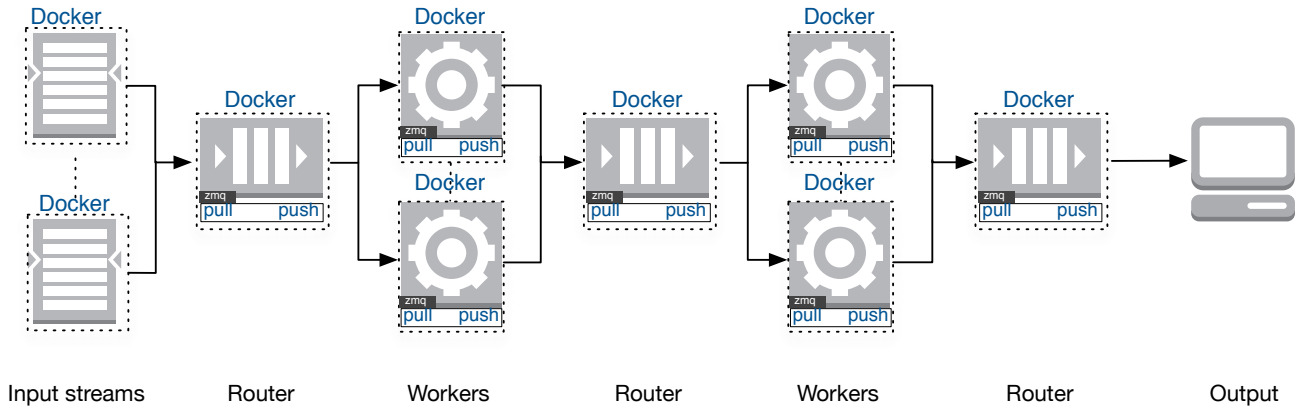

Figure 1. Example of SECURESTREAMS pipeline architecture.

ideal candidate to execute in the limited space allowed by the EPC. The application-specific functions can be quickly prototyped in Lua.

Each component is wrapped inside a Docker container to easily embed all required dependencies and ensure the correctness of their configuration. By doing so, a processing pipeline can be easily deployed on different infrastructures, either on a single machine or a cluster, using a Docker network and the Docker Swarm [12] scheduler.

The communication between workers and routers leverages ZEROMQ, a high-performance asynchronous messaging library [13]. Each router component hosts inbound and outbound queues. In particular, we use the ZEROMQ's pipeline pattern with the *PUSH-PULL* protocol [14]. 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, the downstream workers. This design allows to dynamically scale up and down each stage of the pipeline. Finally, ZEROMQ guarantees that the messages are delivered across each stage via reliable TCP channels.

III. IMPLEMENTATION

SECURESTREAMS is implemented in Lua (v5.3). Our implementation is compact, as it consists of only 120 lines of code (without counting the dependencies). The framework partially extends RXLUA [15], 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 [1]. Listing 1 shows 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. The `:subscribe` function performs the subscription of 3 functions to the data stream. These functions are observers, and the stream is an observable: this is precisely an implementation of the *Observer Design Pattern* [16].

SECURESTREAMS dynamically ships the business logic for each component into a dedicated Docker container and executes it. The communication between the Docker container happens through ZEROMQ (v4.1.2) and the corresponding Lua bindings [17]. In summary, SECURESTREAMS abstracts the underlying infrastructure from the developer, relying on ZEROMQ and Docker. We plan to release our code as open-source.

```
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 + 1
     accumulator[sum] = accumulator.sum + age
     return accumulator
   end, { count = 0, sum = 0 }
 )
 :subscribe(
   function(datas)
     print("Average:", datas.sum / datas.count)
   end,
   function(err)
     print(err)
   end,
   function()
     print("Process complete!")
   end
 )
```

Listing 1. Process pipeline example, using the library RxLua

[1]Anonymous is said about a peer without any identity, the server socket does not know which worker sent the message

IV. Preliminary Evaluation

This section reports on our preliminary evaluation of SecureStreams in the Cloud. First, we present our evaluation settings. Secondly, we detail the real-world dataset used in our experiments. Finally, we analyze some preliminary benchmarks, namely throughput and scalability.

Evaluation settings. We deploy a cluster of *virtual machines* (VM) based on Ubuntu 16.04 LTS and running a daemon Docker (v.1.13.0-rc3). Each VM is set with 2 CPU cores and 2GB RAM, and interconnected using a switched 1 Gbps network. Nodes join a Docker Swarm cluster [12] (v1.2.5). Each VM only executes one single Docker instance, to prevent cross-container interferences [18]. Containers leverage the Docker overlay network to communicate to each other.

Dataset. In our experiments, we process a real dataset released by the *American Bureau of Transportation Statistic*. The dataset reports on the flight departures and arrivals of 20 air carriers [19]. We implement a benchmark application atop of SecureStreams to compute average delays and the total of delayed flights for each air carrier. 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 wanted informations.[2] 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.

Benchmark: throughput. This benchmark shows the upload throughput observed across the whole cluster while streaming the dataset as fast as possible from the source nodes into the processing pipeline. We gather bandwidth measurements by exploiting Docker's own monitoring and statistical module. The statistics are gathered at runtime while the experiment is executing. We report on our results in Figure 2. In this scenario, 4 nodes concurrently inject the input dataset into the processing pipeline, each one using a subset of the full dataset. However, only one worker process is used for each step of the processing pipeline. 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–, median–, and 75th percentiles. For instance, the median throughput at 200 seconds into the experiment almost hits 2,500 kB/s, meaning that 50 % of the nodes in that moment are outputting data at 2,500 kB/s or less. With the current implementation, we observe a peak of 10 MB/s upload throughput into the processing stages.

Benchmark: scalability. We conclude this preliminary by presenting scalability results of the SecureStreams framework. In particular, we scale up each stage of the processing pipeline, up to 4 workers per stage. For each of the configurations, the experiment is repeated 20 times. We show average and standard deviation of the overall completion time to process the full dataset. Figure 3 depicts these results. We observe that by doubling the number of

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

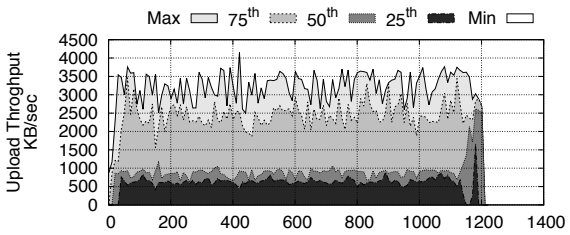

Figure 2. Upload Throughput, single source. The middleware framework completes the processing of the dataset in 1,200 seconds, with a peak of 4 MB/s and an overall average throughput of 2.3 MB/s

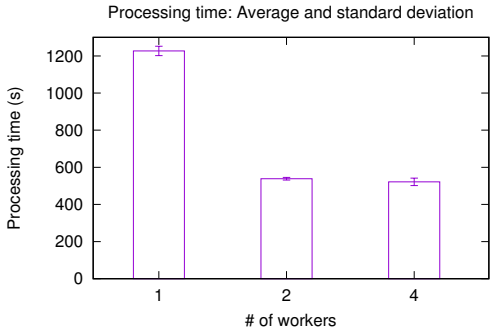

Figure 3. Scalability: processing time, average and standard deviation. The experiment is repeated 20 times.

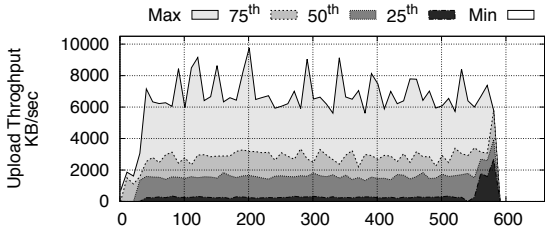

Figure 4. Upload throughput, 2 concurrent processing workers by pipeline stage. Peak throughput at 10MB/s.

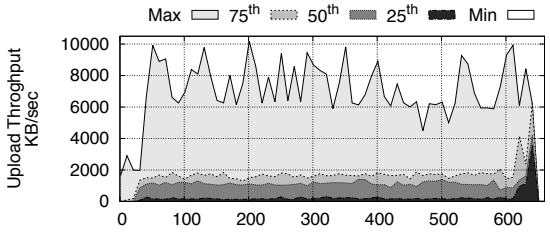

Figure 5. Upload throughput, 4 concurrent processing workers by pipeline stage. Peak throughput at 10MB/s.

workers from the initial configuration achieves a 2× speed-up of the overall processing time, that is from 20 minutes to less than 10 minutes. Conversely, we do not observe similar improvements when using 4 workers.

We believe this behavior can be explained by existing bottlenecks in the pipelining infrastructure, lack of optimization in the application logic as well as tuning options of the ZEROMQ queues. These hypotheses are confirmed by observing the throughput of the system once we increase the processing workers to 2 and 4 in Figure 4 and Figure 5, respectively. We observe the following facts. First, the system is far from saturating the network's available bandwidth, hitting a peak of 10 MB/s. Second, a small percentage of nodes consumes much more bandwidth than the other components. We intend to further investigate these effects as part of our future work, as we are going to elaborate in the following section.

V. FUTURE WORK

We plan to extend SECURESTREAMS along the following directions, with the common goal of making the middleware framework more secure. First, the communication between the processing components will rely on secure channels. To this end, we plan to integrate CURVEZMQ [20], an open-source authentication and encryption protocol for ZEROMQ. CURVEZMQ is based on a fast, secure elliptic-curve cryptographic primitive provided by the libraries CURVECP and NACL. Moreover, SECURESTREAMS will generate the authentication certificates in use by the containers on-the-fly at the moment of their deployment.

Secondly, the SECURESTREAMS will exploit the SGX enclaves to both process and route data. The computation will be executed inside a trusted enclave, ensuring the encryption of the data and the integrity of the executed code.

We intend to evaluate the impact of these modifications on the performance of the system, in particular with respect to the overall throughput, memory and CPU usage.

The current implementation of SECURESTREAMS does not provide full automation of container deployments yet. In particular, we will supersede the current approach based on Docker Compose and instead will rely on the more portable Docker APIs.

Finally, SECURESTREAMS will allow to embed Lua scripts or native (*e.g.*, C/C++) components inside a node of the processing pipeline. This feature will improve the usability of the framework.

VI. CONCLUSION

Secure stream processing is becoming a major concern in the era of IoT. This paper introduces our design and evaluation of SECURESTREAMS, an easy-to-use and efficient middleware framework to implement, deploy and evaluate Cloud-based stream processing pipelines for continuous data streams. The framework is designed to exploit the Trusted Execution Environments nowadays available in Intel's 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. We plan to further extend and throughly evaluate SECURESTREAMS in our future work.

ACKNOWLEDGMENTS

This project is a part of a doctoral research on stream processing in large scale distributed systems. Aurelien Havet's PhD research is co-supervised by Pascal Felber from the University of Neuchâtel and Romain Rouvoy from the University of Lille. The research leading to these results has received funding from the European Commission, Information and Communication Technologies, H2020-ICT-2015 under grant agreement number 690111 (SecureCloud project).

REFERENCES

[1] T. Uustalu and V. Vene, "The essence of dataflow programming," in *Central European Functional Programming School*. Springer, 2005, pp. 135–167.
[2] "Docker," https://www.docker.com.
[3] V. Costan and S. Devadas, "Intel SGX explained," Cryptology ePrint Archive, Report 2016/086, 2016., Tech. Rep.
[4] "Intel SkyLake," https://ark.intel.com/products/88195/Intel-Core-i7-6700K-Processor-8M-Cache-up-to-4_20-GHz.
[5] "Reactive Streams for Kafka," https://github.com/akka/reactive-kafka.
[6] "Spark Streaming," https://spark.apache.org/streaming.
[7] "Apache Storm," http://storm.apache.org.
[8] "Infinispan," http://infinispan.org.
[9] 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.
[10] 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.
[11] "Lua," http://www.lua.org.
[12] "Docker Swarm," https://www.docker.com/products/docker-swarm.
[13] "ZeroMQ," http://zeromq.org.
[14] "ZeroMQ Pipeline," https://rfc.zeromq.org/spec:30/PIPELINE/.
[15] "Reactive Extensions for Lua," https://github.com/bjornbytes/RxLua.
[16] C. Szallies, "On using the observer design pattern," *XP-002323533,(Aug. 21, 1997)*, vol. 9, 1997. [Online]. Available: http://www.wohnklo.de/patterns/observer.ps
[17] "Lua binding to ZeroMQ," https://github.com/zeromq/lzmq.
[18] Y. Koh, R. Knauerhase, P. Brett, M. Bowman, Z. Wen, and C. Pu, "An analysis of performance interference effects in virtual environments," in *Performance Analysis of Systems & Software, 2007. ISPASS 2007. IEEE International Symposium on*. IEEE, 2007, pp. 200–209.
[19] "Data Expo 09. ASA Statistics Computing and Graphics," http://stat-computing.org/dataexpo/2009/the-data.html.
[20] "CurveZMQ," http://curvezmq.org/.