SecureStreams: A Reactive Middleware Framework for Secure Data Stream Processing

Aurélien Havet, Rafael Pires, Pascal Felber, Marcelo Pasin, Romain Rouvoy, Valerio Schiavoni

▶ **To cite this version:**

Aurélien Havet, Rafael Pires, Pascal Felber, Marcelo Pasin, Romain Rouvoy, et al.. SecureStreams: A Reactive Middleware Framework for Secure Data Stream Processing. DEBS'17 - The 11th ACM International Conference on Distributed and Event-Based Systems, Marta Patiño, Jun 2017, Barcelona, Spain. pp.124-133, 10.1145/3093742.3093927 . hal-01510699v2

HAL Id: hal-01510699
https://inria.hal.science/hal-01510699v2

Submitted on 20 Feb 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

SecureStreams: A Reactive Middleware Framework for Secure Data Stream Processing

Aurélien Havet[1] Rafael Pires[1] Pascal Felber[1]
Marcelo Pasin[1] Romain Rouvoy[2] Valerio Schiavoni[1]

[1] University of Neuchâtel, Switzerland (email firstname.lastname@unine.ch)
[2] University of Lille / INRIA / IUF, France (email firstname.lastname@univ-lille.fr)

Abstract

The growing adoption of distributed data processing frameworks in a wide diversity of application domains challenges end-to-end integration of properties like security, in particular when considering deployments in the context of large-scale clusters or multi-tenant Cloud infrastructures.

This paper therefore introduces SecureStreams, a reactive middleware framework to deploy and process secure streams at scale. Its design combines the high-level reactive dataflow programming paradigm with Intel®'s low-level *software guard extensions* (SGX) in order to guarantee privacy and integrity of the processed data. The experimental results of SecureStreams are promising: while offering a fluent scripting language based on Lua, our middleware delivers high processing throughput, thus enabling developers to implement secure processing pipelines in just few lines of code.

1 Introduction

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 paper 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 [28] middleware, which integrates with the SSL protocol [30] for data communication and the current version of Intel®'s *software guard extensions* (SGX) [27] 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 [48]: 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 [9].

In this paper, 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 paper is organized as follows. To better understand the design of SecureStreams, Section 2 delivers a brief introduction to today's SGX operating mechanisms. The architecture of SecureStreams is then introduced in Section 3. Our implementation choices and an example of a SecureStreams program are reported in Section 4. Section 5 discusses our extensive evaluation, presenting a detailed analysis of micro-benchmark performances, as well as more comprehensive macro-benchmarks with real-world datasets. Some related works to this topic are gathered in Section 6. Finally, Section 7 briefly describes our future work and concludes.

2 SGX Lightning Tour

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 [2] 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 1, al-

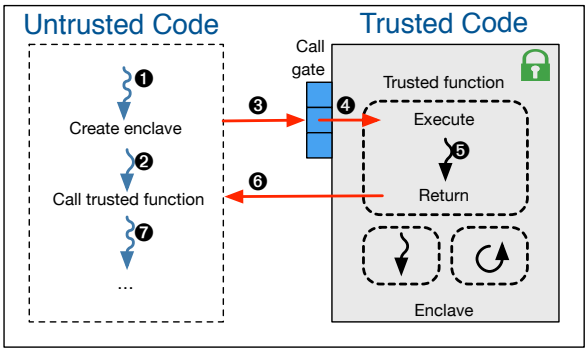


Figure 1: SGX core operating principles.

lows 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) [31], 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.

The execution flow of a program using SGX enclaves is like the following. First, an enclave is created (see Figure 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 [37].

3 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 [24]. 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 2 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 [15]. 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 pseudocode [35]. We provide further implementation details of the embedding of the LuaVM inside an SGX enclave in Section 4.

Each component is wrapped inside a lightweight Linux container (in our case, the *de facto* industrial standard Docker [9]). 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 [1], where SkyLake-enabled instances will soon be made available [4], or similarly to Google compute engine [12]. 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 [11].

The communication between workers and routers leverages ZeroMQ, a high-performance asynchronous messaging library [21]. Each router component hosts inbound and outbound queues. In particular, the routers use the ZeroMQ's pipeline pattern [22] with the Push-Pull socket types.

The inbound queue is a Pull socket. The messages are streamed from a set of anonymous[2] 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

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

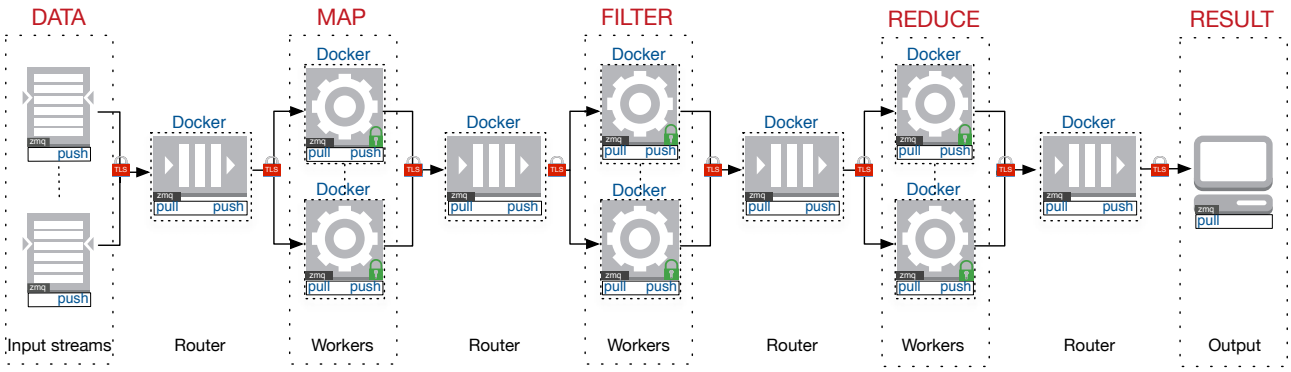


Figure 2: Example of SecureStreams pipeline architecture.

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 [10] description language. Listing 1 reports on a snippet of the description used to deploy the architecture in Figure 2. 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.

```
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
```

Listing 1: SecureStreams pipeline examples. Some attributes (`volume`, `networks`, `env_file`) are omitted.

4 Implementation Details

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 [17], 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 [48].

Listing 2 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.[3] The `:subscribe` function performs the subscription of 3 functions to the data stream. Following the *observer* design pattern [46], these functions are observers, while the data stream is an observable.

```
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
```

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

```
15          or 0) + 1
16        accumulator[sum] = (accumulator.sum
17          or 0) + age
18        return accumulator
19      end
20    )
21    :subscribe(
22      function(datas)
23        print("Adult people average:",
24          datas.sum / datas.count)
25      end,
26      function(err)
27        print(err)
28      end,
29      function()
30        print("Process complete!")
31      end
32    )
```

Listing 2: Example of process pipeline with RxLua.

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 [16]. 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.

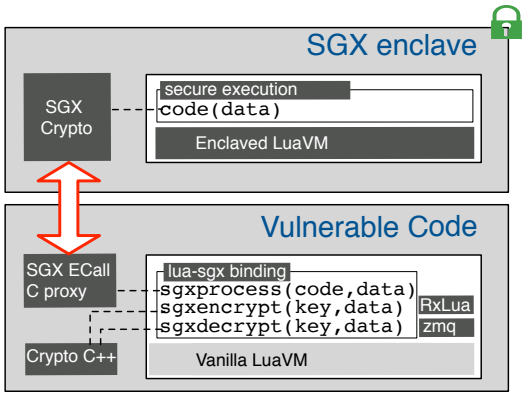


Figure 3: Integration between Lua and Intel® SGX.

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* [3], and *csv* [43] 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).

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 3 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.[4]

5 Evaluation

This section 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.

[4] Omitted due to DEBS17 double-blind policy.

System layer	Size (LoC)
DelayedFlights app	86
SecureStreams library	350
RxLua runtime	1,481
Total	1,917

Table 1: Benchmark app based on SecureStreams.

5.1 Evaluation Settings

We have experimented on machines using a Intel® Core™ i7-6700 processor [14] 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 [11] (v1.2.5) using the Consul [7] (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.

5.2 Input Dataset

In our experiments, we process a real-world dataset released by the *American Bureau of Transportation Statistics* [19]. The dataset reports on the flight departures and arrivals of 20 air carriers [8]. We implement a benchmark application atop of SecureStreams to compute average delays and the total of delayed flights for each air carrier (cf. Table 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.[5] 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.

5.3 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

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

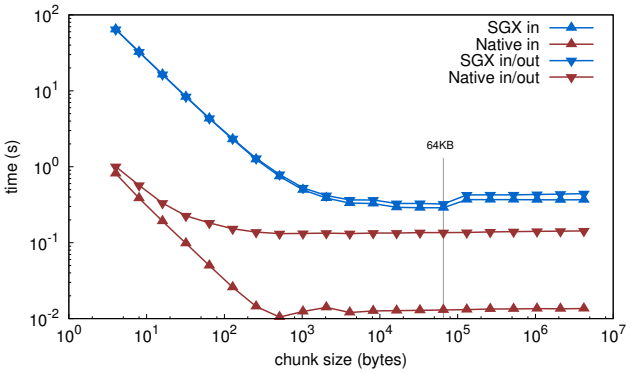


Figure 4: Execution time to copy 100 *MB* of memory inside an SGX enclave (*in*) or to copy it back outside *in/out*.

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 4 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 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 [25]. 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 2. Figure 5 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 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

	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 2: Parameters and memory usage for Lua benchmarks.

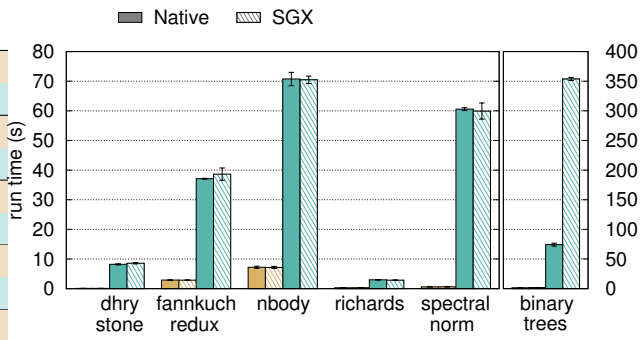


Figure 5: Enclave versus native running times for Lua benchmarks.

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 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.

5.4 Benchmark: Streaming Throughput

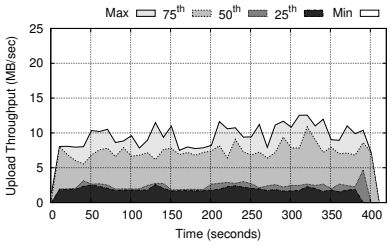
The previous set of experiments allowed us to verify that our design, implementation, and the integration of the LuaVM into the SGX enclaves is sound. Next, we deploy a SecureStreams pipeline which includes mappers, filters and reducers. To measure the achievable throughput of our system, as well the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

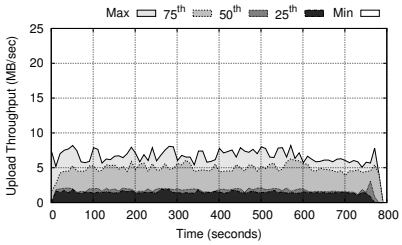
The results of these deployments are presented in Figure 6. For each of the mentioned configurations, we also vary the number of workers per stage, from one (Figure 6-a,d,g), two (Figure 6-b,e,h), or four (the remaining ones.) We use a representation based on stacked percentiles. The white bar at the 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-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-d), a speed-up of 41 %. Scaling up the workers to 4 in the baseline configuration (Figure 6-g) did not produce a similar speed-up.

As we start injecting encrypted datasets (Figure 6-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.

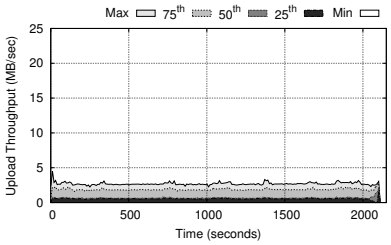
The deployment of the most secure set of configurations (right-most column of plots in Figure 6) shows that when using encrypted datasets and executing the stream processing inside SGX enclaves one must expect longer processing times and lower throughputs. This is the (expected) price to pay for higher-security 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.



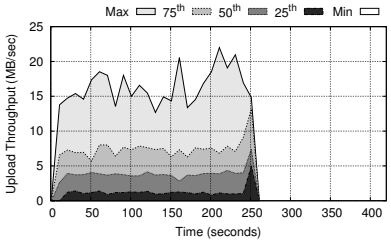
(a) Streaming throughput. Data in clear text, no SGX, 1 worker per stage.



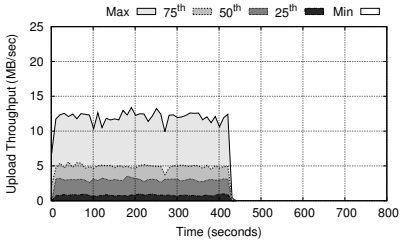
(b) Streaming throughput. Encrypted data, no SGX, 1 worker per stage.



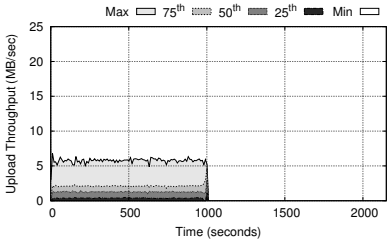
(c) Streaming throughput. Encrypted data, processing SGX, 1 worker per stage.



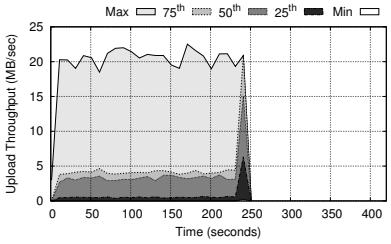
(d) Streaming throughput. Data in clear text, no SGX, 2 workers per stage.



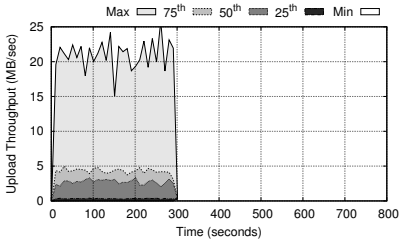
(e) Streaming throughput. Encrypted data, no SGX, 2 workers per stage.



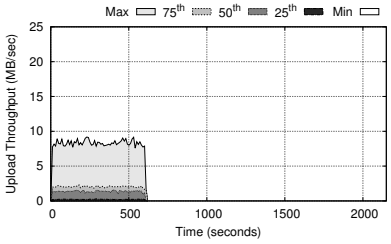
(f) Streaming throughput. Encrypted data, processing SGX, 2 workers per stage.



(g) Streaming throughput. Data in clear text, no SGX, 4 workers per stage.



(h) Streaming throughput. Encrypted data, no SGX, 4 workers per stage.



(i) Streaming throughput. Encrypted data, processing SGX, 4 workers per stage.

Figure 6: Throughput comparison between normal processing (with cleartext data and no encryption), encrypted data but without enclaves, and with encrypted data and SGX processing. We scale the number of worker nodes per stage from 1 (left-most column), 2 (center colum) and 4 (right-most column).

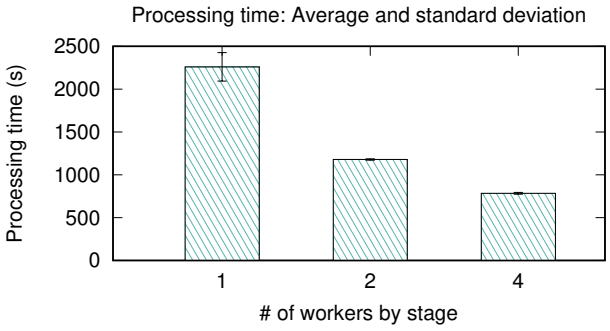


Figure 7: Scalability: processing time, average and standard deviation. The experiment is repeated 5 times, with a variation on the number of workers for each stage, each worker using SGX.

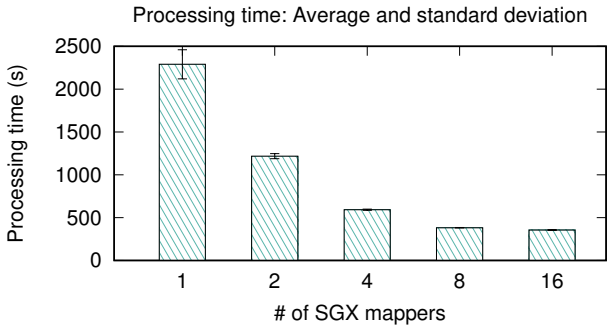


Figure 8: Scalability: processing time, average and standard deviation. The experiment is repeated 5 times, with a variation on the number of mappers SGX, other workers—1 filter worker and 1 reduce worker—do not use SGX.

5.5 Benchmark: Workers' Scalability

To conclude our evaluation, we study SecureStreams in terms of scalability. We consider a pipeline scenario similar to Figure 2 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 7. 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).

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 8 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.

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.

6 Related Work

Spark [50] 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 [44] 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 [34], and Google's Cloud DataFlow [23] 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 [18] allows stream processing atop of Apache Kafka [5, 33]. These solutions do not, however, support secure execution in a trusted execution environment.

More recently, some open-source middleware frameworks (*e.g.*, Apache Spark [20], Apache Storm [6], Infinispan [13]) introduced APIs to allow developers to quickly set up and deploy stream processing infrastructures. These systems rely on the *Java* virtual machine (JVM) [36]. 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 [41, 26] when this limit is crossed. Moreover, executing a fully-functional JVM inside an SGX enclave would currently involve significant re-engineering efforts.

DEFCon [38] 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 [42] and Gupt [39]. These systems leverage differential-privacy techniques [29] 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 [49] 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 [51] 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 [47] 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 [45] 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 [40], 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.

7 Conclusion

Secure stream processing is becoming a major concern in the era of the Internet of Things and big data. This paper 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 [45], MrCrypt [47] or DEFCon [38]. 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.

Acknowledgments

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). Rafael Pires is also sponsored by CNPq, National Counsel of Technological and Scientific Development, Brazil.

References

[1] Amazon EC2 Container Service. https://aws.amazon.com/ecs/.

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

[3] The JavaScript Object Notation (JSON) Data Interchange Format. RFC 7159, March 2014.

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

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

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

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

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

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

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

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

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

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

[14] 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.

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

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

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

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

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

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

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

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

[23] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam 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.*, 8(12):1792–1803, August 2015.

[24] Richard Bird and Philip Wadler. *Introduction to Functional Programming*. Prentice Hall, March 1988.

[25] Carl Friedrich Bolz and Laurence Tratt. The impact of meta-tracing on vm design and implementation. *Science of Computer Programming*, 98:408–421, 2015.

[26] Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Christof Fetzer, Peter Pietzuch, and Rüdiger Kapitza. SecureKeeper: Confidential ZooKeeper Using Intel® SGX. In *Proceedings of the 17th International Middleware Conference*, Middleware '16, pages 14:1–14:13. ACM, 2016.

[27] Victor Costan and Srinivas Devadas. Intel® SGX explained. Technical report, Cryptology ePrint Archive, Report 2016/086, 2016.

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

[29] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. Calibrating noise to sensitivity in private data analysis. In *Theory of Cryptography Conference*, pages 265–284. Springer, 2006.

[30] Alan Freier, Philip Karlton, and Paul Kocher. The secure sockets layer (SSL) protocol version 3.0. 2011.

[31] Shay Gueron. A Memory Encryption Engine Suitable for General Purpose Processors. *IACR Cryptology ePrint Archive*, 2016:204, 2016.

[32] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes Filho. Lua—An Extensible Extension Language. *Software: Practice and Experience*, 26(6):635–652, June 1996.

[33] Jay Kreps, Neha Narkhede, Jun Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, pages 1–7, 2011.

[34] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. Twitter Heron: Stream Processing at Scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 239–250, New York, NY, USA, 2015. ACM.

[35] Lorenzo Leonini, Étienne Rivière, and Pascal 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*, NSDI'09, pages 185–198, Berkeley, CA, USA, 2009. USENIX Association.

[36] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine Specification: Java SE 8 Edition*. Pearson Education, 2014.

[37] Frank McKeen, Ilya Alexandrovich, Ittai Anati, Dror Caspi, Simon Johnson, Rebekah Leslie-Hurd, and Carlos 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*, HASP 2016, pages 10:1–10:9, New York, NY, USA, 2016. ACM.

[38] Matteo Migliavacca, Ioannis Papagiannis, David M. Eyers, Brian Shand, Jean Bacon, and Peter Pietzuch. Defcon: High-performance event processing with information security. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 1–1, Berkeley, CA, USA, 2010. USENIX Association.

[39] Prashanth Mohan, Abhradeep Thakurta, Elaine Shi, Dawn Song, and David Culler. GUPT: Privacy Preserving Data Analysis Made Easy. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 349–360, New York, NY, USA, 2012. ACM.

[40] 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. Rüdiger, and C. Fetzer. SCONE: Secure Linux Containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2016*. USENIX, November 2016.

[41] Rafael Pires, Marcelo Pasin, Pascal Felber, and Christof Fetzer. Secure content-based routing using intel® software guard extensions. In *Proceedings of the 17th International Middleware Conference*, Middleware '16, pages 10:1–10:10, New York, NY, USA, 2016. ACM.

[42] Indrajit Roy, Srinath T. V. Setty, Ann Kilzer, Vitaly Shmatikov, and Emmett Witchel. Airavat: Security and Privacy for MapReduce. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, pages 20–20, Berkeley, CA, USA, 2010. USENIX Association.

[43] Yakov Shafranovich. Common Format and MIME Type for Comma-Separated Values (CSV) Files. RFC 4180, October 2005.

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

[45] Julian James Stephen, Savvas Savvides, Vinaitheerthan Sundaram, Masoud Saeida Ardekani, and Patrick Eugster. STYX: Stream Processing with Trustworthy Cloud-based Execution. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, pages 348–360, New York, NY, USA, 2016. ACM.

[46] Constantin Szallies. On using the observer design pattern. *XP-002323533,(Aug. 21, 1997)*, 9, 1997.

[47] Sai Deep Tetali, Mohsen Lesani, Rupak Majumdar, and Todd Millstein. Mrcrypt: Static analysis for secure cloud computations. *ACM Sigplan Notices*, 48(10):271–286, 2013.

[48] Tarmo Uustalu and Varmo Vene. *The Essence of Dataflow Programming*, pages 135–167. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[49] Xiangqiang Xu and Xinghui 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*, pages 1344–1349. IEEE, 2015.

[50] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 423–438, New York, NY, USA, 2013. ACM.

[51] Kehuan Zhang, Xiaoyong Zhou, Yangyi Chen, XiaoFeng Wang, and Yaoping Ruan. Sedic: privacy-aware data intensive computing on hybrid clouds. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 515–526. ACM, 2011.