2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing

A lightweight MapReduce framework for secure processing with SGX

Rafael Pires*, Daniel Gavril†, Pascal Felber*, Emanuel Onica† and Marcelo Pasin*
*University of Neuchâtel, Switzerland;†Alexandru Ioan Cuza University of Iaşi, Romania

***Abstract*—MapReduce is a programming model used extensively for parallel data processing in distributed environments. A wide range of algorithms were implemented using MapReduce, from simple tasks like sorting and searching up to complex clustering and machine learning operations. Many of these implementations are part of services externalized to cloud infrastructures. Over the past years, however, many concerns have been raised regarding the security guarantees offered in such environments. Some solutions relying on cryptography were proposed for countering threats but these typically imply a high computational overhead. Intel, the largest manufacturer of commodity CPUs, recently introduced SGX (software guard extensions), a set of hardware instructions that support execution of code in an isolated secure environment. In this paper, we explore the use of Intel SGX for providing privacy guarantees for MapReduce operations, and based on our evaluation we conclude that it represents a viable alternative to a cryptographic mechanism. We present results based on the widely used k-means clustering algorithm, but our implementation can be generalized to other applications that can be expressed using MapReduce model.**

***Keywords*-distributed processing; security; MapReduce.**

I. Introduction

Since it was officially adopted by Google [1] in 2008, the MapReduce programming model consistently gained ground as a viable solution for assuring the necessary scalability of distributed data processing. The generic model, composed of the two main *map* and *reduce* functions, was widely used to implement applications that can leverage parallel task processing. In this generic model, the data to be processed is typically located over a series of mapper nodes, which apply in parallel a *map* function responsible for mapping individual data items to a finite set of predefined keys. The output is redistributed in a shuffle step based on the key mappings to reducer nodes, which execute in parallel a *reduce* function for processing each set of data items corresponding to a certain key. This model can be used in processing tasks that range from simple operations that can be composed like counting, sorting, and searching data to more complex algorithms like cross-correlation or page rank. MapReduce was adapted in different ways to fit this wide diversity of scenarios, as well as different deployment environments.

Particular settings where the deployment of MapReduce encountered difficulties are services that require security guarantees. For cost savings purposes, data analysis and processing services are more and more often deployed to public cloud infrastructures, which do not offer strong security guarantees and hence make sensitive data prone to privacy and integrity risks. If such services rely on a MapReduce-based implementation, it is of high importance to find a way to adapt the programming model in a manner that provides the required security assurance to the system. As we refer in Section II, some solutions use complex cryptographic techniques for protecting the privacy of the data being processed in the *map* and *reduce* functions. Such techniques are typically complex: they range from secret sharing to homomorphic encryption, depending on the exact operations that need to be executed over the sensitive data in each use case. In particular the choice of the encryption is largely determined by the task to be executed in the reducers. Therefore, a common limitation is that the customized data encryption schemes used often have limited expressiveness and are applicable to only a small class of applications. Another drawback generally encountered is the high performance overhead of complex encryption schemes, homomorphic encryption in particular being notorious for its lack of practicality. Finally such solutions address mostly the privacy of the data, but much less often the privacy or the integrity of the code itself, which is a key concern when externalizing a service to a public cloud.

A different approach to security is to rely on trusted execution environments (TEE) supported in hardware. A TEE provides an isolated space for executing code where confidentiality and integrity are assured. *Software guard extensions* (SGX) is a powerful TEE included in the Intel's commodity processors starting with the Skylake generation in late 2015. The SGX-capable processors provide the possibility of running code in *enclaves* that are isolated from the other memory used by system processes. This facilitates the provision of privacy and integrity for proprietary algorithm implementations and for sensitive data. The code designated to run in the enclave space is not accessible from outside. Sensitive data can be protected using state-of-the-art encryption algorithms while outside the enclave and decrypted only inside where is processed efficiently in plaintext form—but shielded from the rest of the system.

In our work we propose a self-contained framework for securing MapReduce that leverages the benefits of SGX's trusted execution environment. Our system architecture combines a lightweight virtual machine based on the Lua language, a MapReduce library, and a publish/subscribe service for communication between the client and worker nodes. Unlike cryptographic solutions, our approach is independent of the particular characteristics of the *map* and *reduce* functions and can hence be used for any problem parallelizable with MapReduce. The specific code to be executed in the MapReduce service can be integrated in simple scripts, which are run privately and in isolation using SGX over data decrypted only inside the enclave. Our focus is on providing a flexible framework for securely running MapReduce applications that can be easily implemented and deployed. The basic *word count* MapReduce example, for counting the number of occurrences of different word in a given text, can be implemented in our framework with less than 30 lines of code (LOC). We present in this paper an evaluation of our approach using the widely used k-means clustering algorithm, showing that the overhead incurred is minimal and that our solution is applicable to other use cases.

Our paper is structured as follows. We overview related work in Section II, including other approaches on securing MapReduce based services. We then present additional details on SGX and k-means in Section III, and we describe the architecture of our solution in Section IV. Section V covers the evaluation of the k-means use case by comparing our SGX-secured architecture with a basic unsecured implementation. We finally conclude in Section VI.

II. Related Work

VC3 [2] is an early proposal of a secure framework for MapReduce services using an SGX emulator instead of real SGX-enabled processors. The general idea behind the framework design is relatively close to our proposal, the user having the possibility to write its own *map* and *reduce* functions that can execute in SGX-enabled machines. The user code has to be written in C++, which can make the implementations prone to potential faults like illegal memory accesses. In this respect the authors provide an optional compiler through which the programmers can enforce self-integrity invariants for memory regions. The main advantage in our proposed framework, which we more accurately evaluate effectively on SGX-enabled processors, is the additional flexibility and ease-of-use of the high-level Lua-based programming environment. The MapReduce implementation in our case is enclosed in scripts, which are executed by a standard Lua interpreter that is less prone to faults, easier to maintain, and safer thanks to its small code base. The VC3 solution is further extended in [3], which focuses on security issues generated by traffic analysis attacks on the exchanges between mapper and reducer nodes. We do not consider such attacks in our current work, but we believe that the analysis and the solutions proposed are also applicable to our system.

M^2R [4] is another proposal of a secure framework for MapReduce, which takes a more general approach with a generic design that can be implemented on any trusted execution environment. The authors refer to SGX-enabled processors as one potential target, but the evaluation is conducted as in case of VC3 above on a TEE simulation using a trusted Xen-4.4.3 kernel-based hypervisor. As in [3], M^2R specifically focuses on attacks exploiting the leakage between mappers and reducers. M^2R's generic design formalizes the execution of the MapReduce architecture in four *trusted code base* (TCB) components: *mapT*, *reduceT*, *mixT*, and *groupT*, with the security of the latter two being particularly critical as they are meant to contain the mappers-to-reducers shuffle implementation. Evaluation using the HiBench MapReduce benchmark suite [5] yields the interesting observation that *k-means clustering* (which we also consider in our tests) and *grep*, two computationally intensive cases, do not leak extra information in the baseline implementation without the shuffle phase's extra security. Therefore, in such situations, adding additional protection between mappers and reducers will produce unnecessary overheads.

In [6] the authors propose a solution that leverages Shamir secret sharing [7] for executing queries over an outsourced database using the MapReduce model. Each field in a database record is split in secret shares that are constructed for each letter in the field value. Each letter in the alphabet is associated with a unary vector of 26 bits, with one index bit set and the rest being zeroes. The secret shares are obtained for each bit by applying 26 polynomials of identical degree. Shares of numerical values, as well as the privacy preserving queries submitted by users, are represented in the same fashion. Mappers and reducers execute specific operations depending on the queries on these shares for obtaining the result. The individual computations seem rather simple and efficient, since many of the initial bits are zeroes, but the total number of values obtained in the encoding (26 per letter) still inflicts a high overhead. Furthermore, the technique is limited to four query types: count, search, equijoin, and range.

SecureMR [8] focuses on MapReduce integrity. In this purpose some of the MapReduce tasks are replicated and assigned to different mappers and reducers. The MapReduce architecture is enhanced with several secure components: a secure manager and a secure scheduler for task duplication and assignment, a secure task executor for DoS and replay attacks prevention, a secure committer for consistency checking of mappers intermediate results, and a secure verifier that detects attacks based on results inconsistencies. The solution takes in account neither the privacy of data nor of the code.

Airavat [9] proposes to secure MapReduce using differential privacy, which is also combined with access control policies provided by SELinux in the proposed implementation. The mappers can be both trusted and untrusted in the proposed architecture, the approach relying on adding noise in this phase in order to minimize leaks of sensitive data in the *map* function output. Among the limitations of the approach, one can mention that reducers should be always trusted and that there is no possibility to chain multiple

MapReduce cycles on the same input data. Moreover, the accuracy of the results depends on parameters that should be carefully tweaked for obtaining a sound output.

Tagged MapReduce [10] considers the execution of MapReduce computations over a hybrid cloud, where a part of the infrastructure is public and untrusted and a part is private and trusted. The sensitive data can be processed securely only in the private part and is identified through tags added to the tuples. For this purpose, the system architecture involves a scheduler that assigns tasks to workers and control the flow based on tags. The programmer can implement in *map* and *reduce* functions specific policies that dictate how data sensitivity changes during the processing. The solution does not address a case where only a public untrusted cloud is sufficient for deployment.

III. Preliminaries

The MapReduce model includes three main sequential phases in processing the input data, which we also briefly described in Section I. In the first phase the mapper nodes independently execute a *map* function in which each individual item in the distributed input data is mapped to a key. In the second phase the (key, value) tuples obtained after mapping are shuffled according to their keys and redistributed to reducer nodes, such that all tuples corresponding to the same key arrive on the same node. Optionally, in this phase multiple tuples with the same key can be aggregated by a combiner function in order to optimize the redistribution. Finally, in the third phase reducer nodes process independently in a local *reduce* function all data associated to a key and output the result of their computation. In our work, we are concerned with assuring the privacy and integrity of the data processed within the *map* and *reduce* functions and also of the code of these functions, while also offering to the programmer an accessible lightweight environment of implementing various use-cases. In the current solution we do not consider any attacks targeting the in-between shuffle phase like traffic analysis, leaving this for future work. In the framework we propose data is encrypted outside of the two main *map* and *reduce* functions, which are executed securely in SGX enclaves.

In our architecture we use the publish/subscribe (pub/sub) communication model [11], one of the most effective ways of disseminating information in distributed systems. In the generic model, publisher nodes submit data to a pub/sub routing service as publications formed of a header describing the data and a payload containing the effective data. The pub/sub routing service matches the publications header with subscriptions registered by subscriber nodes and further routes the matching publications towards their destinations. Our solution is based on a previous Secure Content-Based Routing (SCBR) implementation [12] that evaluated the pub-/sub communication model in an SGX secured environment. In brief, in SCBR the matching and routing of publications is determined inside secure SGX enclaves provided by the pub/sub service, which is typically deployed in a public cloud. All subscriptions and publication messages are encrypted using a symmetric cypher while outside the SGX enclaves. The subscriptions and publication headers are decrypted inside the enclave, where subscriptions are stored. Then, the service routes the publication payloads (encrypted with a different key) to matching subscribers. For simplicity we leave out the details related to the key provisioning, which can be consulted in [12].

We adapt the SCBR pub/sub implementation to a MapReduce service by designing a pub/sub communication protocol between the nodes involved in the distributed data processing. Worker nodes will act as subscribers registering queries to find out about new MapReduce job openings, and also as publishers submitting their desired job details. The client of the MapReduce service, which is also the data owner will also act as both subscriber and publisher, registering subscriptions for job details and publishing announcements on new MapReduce jobs to be executed. The client will also coordinate the routing from mapping to reducers using the pub/sub system, and will finally publish the code and data to the registered workers and obtain the results. We detail our solution architecture, describing the exact steps executed in the pub/sub based communication protocol in IV.

For testing our solution we consider a use case where the target of the MapReduce service is to classify data using the k-means clustering method. K-means is an unsupervised learning algorithm widely used and adapted for data classification purposes since its first mention in [13], which operates as following: (i) a certain k number of clusters is fixed *a priori* and k corresponding centers are defined for each one of them; (ii) data items are iterated and assigned to the nearest cluster center (typically through Euclidean distance); and (iii) every cluster center is recomputed as the centroid of the assigned data items (the mean of those points). Steps (ii) and (iii) repeat until a termination criteria is reached (e.g. the sum of distances between the old and the new cluster centers is below a given threshold). For implementing k-means in the MapReduce model we put (ii) in the *map* function and (iii) in the *reduce* function as displayed in Figure 1.

For providing an easily programmable and efficiently maintainable environment our framework offers the possibility to write the *map* and *reduce* code in simple Lua scripts, which are run in the enclave using a Lua interpreter, currently in version 5.3.2. Lua [14] is a lightweight multi-paradigm programming language. The language API provides the possibility to call Lua functions from C/C++ code. Using Lua is particularly interesting in an SGX secured environment due to the reduced overhead, which fits with the enclave restrictions. We discuss in further detail the evaluation of the k-means use case with our Lua implementation in Section V.

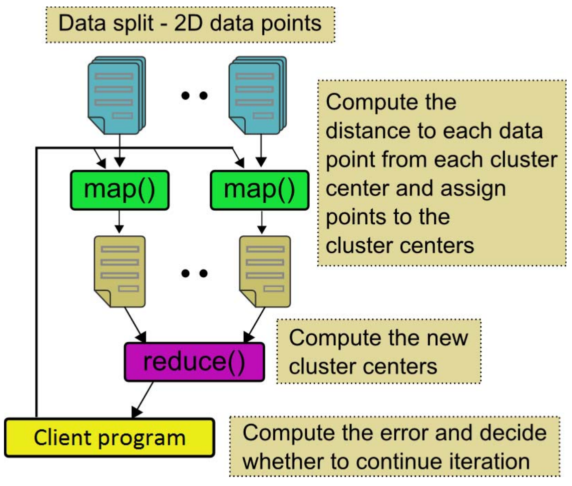
Data split - 2D data points
map()
map()
Compute the distance to each data point from each cluster center and assign points to the cluster centers
reduce()
Compute the new cluster centers
Client program
Compute the error and decide whether to continue iteration

Figure 1. Flow of K-means with MapReduce.

IV. Solution Architecture

In Figure 2 we display the entities composing our solution architecture: clients, router (SCBR based pub/sub engine) and workers, which can assume the role either of a mapper or a reducer. Clients provide the code to be executed, the data to be processed and gather the results after completion. All communication channels use the 0MQ [15] message passing library, having a central point in the SCBR pub/sub engine, which we adapted from previous work [12] as briefly described in Section III.

The SCBR engine is responsible for securely storing subscriptions that contain the conditions under which each message should be forwarded to the corresponding interested party. Subscriptions, used in an initiation protocol for the MapReduce processing as described further below, are stored within the SGX protected memory area and every match processing is done inside enclaves. Data is always encrypted whenever outside of the enclaves. Although such a centralized approach is not suitable to large-scale data processing, it is arguably useful for modest quantities of highly sensitive data that could be, for instance, partitioned from higher amounts of non-sensitive data. Nevertheless, it has been demonstrated [16], [17] that it is possible to elastically scale a pub/sub engine by specializing its functional steps into replicable operators. That could dramatically improve the network performance of such a centralized approach.

The MapReduce processing is started following an initial message exchange displayed in Figure 3. Worker nodes register their availability for MapReduce job openings through `JOB_OPENING` subscriptions. The client registers his interest in the details of the *map* and *reduce* tasks that the workers are capable of doing through the `JOB_DETAILS` subscription. When the client has a new job to execute it advertises it through a `JOB_OPENING` publication, which is received by workers that previously registered as available.

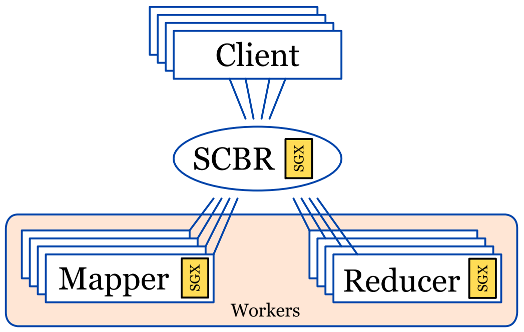

Figure 2. Entities.

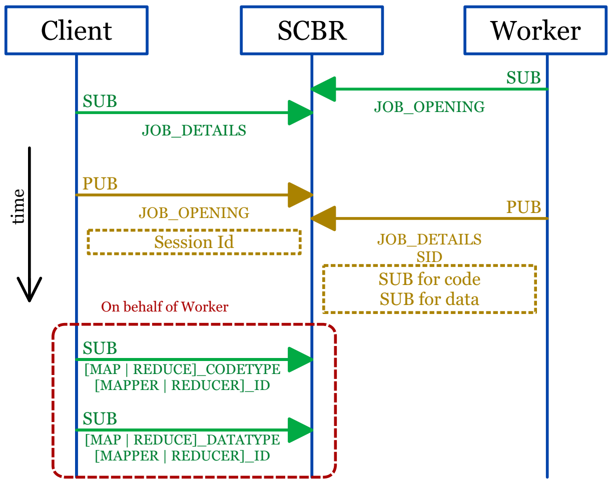

Figure 3. Session establishment protocol.

The workers submit then their desired details regarding the job through a `JOB_DETAILS` publication by sending in the message payload their further subscriptions for code and data (particular to the role they choose: mapper or reducer). At the end of this negotiation, if the client decides to hire a worker, it registers on SCBR the received subscriptions for code and data on the worker's behalf. By doing so, the client established the MapReduce chain and keeps track of how many workers it has hired and of which kind (mappers or reducers).

The provisioning of code and data is shown in Figure 4. The client also includes the number of reducers along with the Lua scripts in case of type `MAP_CODETYPE`, or the number of mappers in case of type `REDUCE_CODETYPE`. The purpose is to make the workers aware of how many messages signaling the end of the stream they have to wait before considering the work done. This is important because the reduce phase can only start once all data for a given key is routed to the intended worker. Additionally, the number of reducers received by a mapper is used in a hash function that takes as argument the mapped key and returns the indication

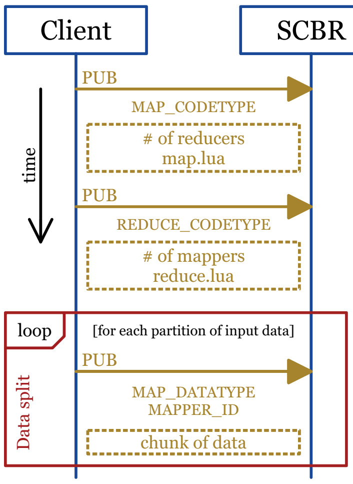

Figure 4. Provisioning of code and data.

of which reducer it has to be forwarded to. After sending the code, data is split by the client among the mappers, line by line. The destination id is included in the header of the `MAP_DATATYPE` publication.

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

Regarding the execution environment, since SGX code must be previously signed and no dynamic linking is possible after the enclave creation, we ported a Lua [14] virtual machine to run in protected area. Porting legacy code to SGX means that every system call or input/output instruction have to be dealt with, since these are not allowed inside enclaves. Workers contain a Lua interpreter that runs inside the enclaves, which loads the scripts that contain the processing phases of *map* and *reduce*. This setting provides a very accessible programming environment, which can be easily used and efficiently maintained. We provide below a basic example.

```
local json = require "json"

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

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

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

Listing 1. Map code in Lua for word count

Listing 1 shows the sample code of a mapper of a word count application. As it can be noticed, data is encoded in JavaScript Object Notation (*json*). For convenience, the json Lua parser is pre-loaded in the enclave of worker nodes, and is accessible through the *require* command. The script can contain as many helper functions as desired. The following special functions, however, are called by the framework:

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

Likewise, listing 2 shows a sample code for the reduce step that contains a single special function:

reduce(key, value)	Contains the functional implementation of reducer.

```
local json = require "json"

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

Listing 2. Reduce code in Lua for word count

V. Evaluation

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

We conducted all the experiments using two SGX capable machines, both with processor Intel i7-6700 64bits, clock of 3.4GHz, 8MB cache, 4 cores, 8 threads, with 8GB of installed memory and SSD of 256GB. In terms of software, we used the Intel SGX SDK 1.7.100 over Ubuntu 14.04.1, kernel 4.2.0-42. Unless mentioned otherwise, messages were all encrypted with AES-CTR [18] with key and input vector both of 128 bits and were decrypted only inside the enclaves. The process placement was made as follows: Machine 1: The Client, entity responsible for providing code and data, 8 mappers and 5 reducers. Machine 2: 8 mappers and 5 reducers. The number of mappers was chosen to be twice as much as the number of cores in each machine to take advantage of parallelism, while the number of reducers was set to be a divisor of input data size to stimulate an even distribution of work among them. To illustrate how small our final code-base is, Table I shows the memory section sizes of executables and shared libraries that are loaded into enclaves.

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

Table I
BINARIES SIZE IN BYTES

Figure 5 illustrates 4 out of 76 iterations that K-means took to converge when the threshold was set to zero. In this example, we randomly generated 100,000 observations and 10 centers. As it can be seen in the first frame, the initial centroids were all set on a small area in the bottom-left corner by limiting the domain of generated coordinates. In spite of this, they quickly assume positions that are closer to the best fit. After just 6 iterations, the average distance of centroids decreases more than six times. Besides, even with an initial state closer to the final one, we can get a slower convergence (90 iterations, as shown in Figure 6). The final result depends on the initial points and there is no guarantee that it is the global optimum, neither about how fast it will converge.

For the following executions, we arbitrarily set the threshold to be one thousandth of the diagonal of the rectangle that contains all the observed points. That means that the iterative process finishes when the average distance of centroids between two subsequent iterations is less than that fraction, so that we can avoid slow convergences. For the given examples of Figure 6, this criteria would stop the executions by the 41st (instead of 76th) and 21st (instead of 90th) iterations. From the figure, we can see that there is no much variation from those points on.

To assess the influence of input data size, namely the number of observations and centroids, on the memory usage and

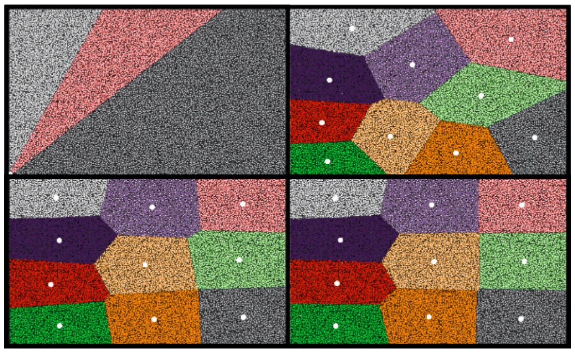
Figure 5. K-means convergence. Iterations 0 (top left), 19 (top right), 38 (bottom left) and 76 (bottom right)

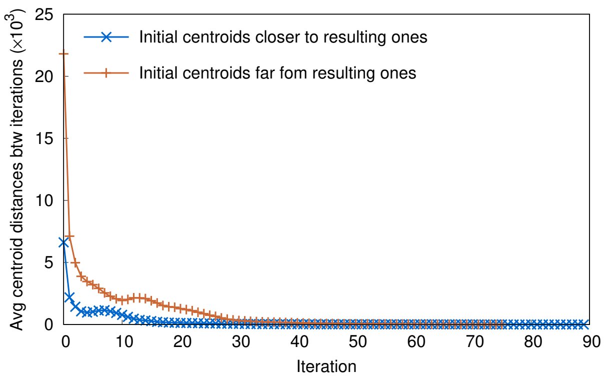

Figure 6. K-means convergence.

time consumption, we conducted multiple experiments using the two SGX capable machines described above. Figure 7 shows the average time it took to complete one iteration of kmeans with varying input sizes. It can be noticed that, although the variation on the number of clusters can cause some inflection in the curves, the completion time is mostly affected by the number of observed data points. Moreover, while the two first increments on the number of data points ($n = 10k$ and $n = 100k$) caused a proportional increase on consumed time with regards to the data growth (ten times), the last one ($n = 1M$) induced a twenty-fold rise. That can be explained by the growth in the occurrence of cache misses within each worker. When that happens, data must be fetched from main memory. When using SGX protected executions, this means one page has to be evicted from cache (and hence, encrypted), while the one that is fetched must be decrypted and checked for integrity and freshness (that prevents tamper and replay attacks, respectively).

The described cache effect is well documented [12], [19], [20] and can be noticed by the cache miss rates. Since the *reduce* phase is more memory intensive, we chose to make the average on the cache miss rates per second of all 10 reducers in each execution, i.e., for the same second, cache miss rates of reducers were summed and divided by

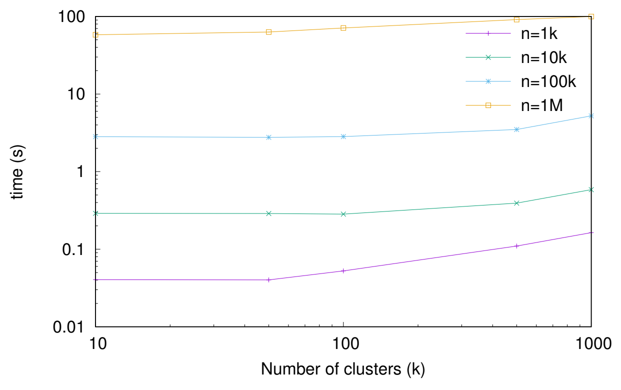

Figure 7. Average time to run one iteration with varying input sizes

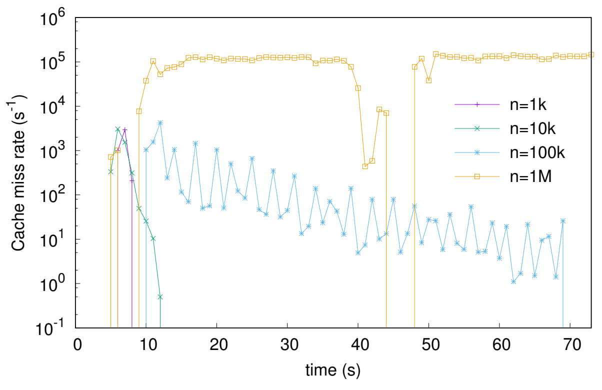

Figure 8. Cache miss rates for different input sizes

10. Wall clocks were synchronized with a common NTP server, so that the resulting skew was at the range of tens of milliseconds and it should not affect the sampling resolution of 1s. Figure 8 shows that measurement as reported by the tool `pidstat` when the number of centroids is $k = 50$. Note that *y* scale is logarithmic, so that the cache miss rates for $n = 1M$ is at least two orders of magnitude higher than $n = 100k$.

Finally, we moved to appraising the influence of SGX and encryption when compared to native executions (no hardware protection). We ran the same datasets with a fixed number of clusters $k = 50$ and observed points varying from $n = 1000$ until $n = 1M$ along with the four combinations of turning on and off enclaves and encryption. Results are plotted in Figure 9. We also included the overhead in percentage. Encryption overhead was obtained by comparing the average time it took to complete one iteration with and without encryption both inside and outside the enclave, and averaging the two values. SGX overhead was established analogously. The time corresponds to the average of all iterations in a single run of k-means (until the threshold was reached). Coefficient of variation across multiple runs was negligible. We can notice that encryption overhead is quite low, of around 5%. Enclave execution, on the other hand,

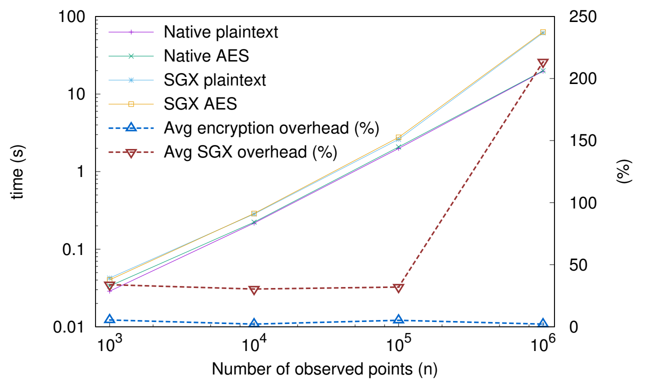

Figure 9. Encryption and SGX overhead

	Split	Shuffle	Output
$n = 1k$	58.7 KB	112.1 KB	4.3 KB
$n = 10k$	257.2 KB	1.1 MB	4.5 KB
$n = 100k$	2.2 MB	11 MB	4.6 KB
$n = 1M$	19.1 MB	96.5 MB	4.6 KB

Table II
DATA VOLUME EXCHANGED PER ITERATION

is kept around 30% until we start to get high occurrence of cache misses, as discussed before, when it reaches more than 200%.

The data volume exchanged in each MapReduce step can be seen in Table II. Numbers correspond to the average per iteration for the same executions as Figure 9. We observe an increase in overheads when cache misses rate grows. Therefore, based on the processor's cache size, we could establish the maximum amount of data a single SGX-capable machine would be able to handle before incurring in too much overhead. In our experiments, we perceived that behavior when processing amounts somewhere in between 11 MB and 96.5MB shared between 2 machines (average of around 54MB, or 27MB per machine). Therefore, a rough estimation would be to limit those amounts to three times the cache size, or 24MB in our case. More scalability could be provided horizontally, with the addition of more machines.

VI. Conclusion

We have proposed in the current work a lightweight framework for implementing secure MapReduce applications in an untrusted environment. Our approach does not require any particular knowledge from the programmer of cryptographic mechanisms for implementing the *map* and *reduce* functions. Also, preserving privacy and integrity is not dependent in any way on the specific characteristics of the functions, simply taking advantage in this purpose of the standard TEE characteristics of SGX enclaves. Besides these advantages, our solution is easily maintainable relying on a standard Lua interpreter for code execution.

The main objective of the current implementation was to show its viability and to assess its behavior under different conditions. We consider as a next step testing our prototype at a larger scale and comparing it with other solutions for securing MapReduce (e.g., [2], [4]). We are also considering taking in account the shuffle phase security in respect to attacks based on traffic analysis. We believe that detection of tampering at network level and denial of service attacks could be easily implemented through well known techniques (e.g. sequential numbers, MACs, nonces), which we also leave for future work.

Acknowledgments

This work is partly funded from the *European Union's Horizon 2020 research and innovation programme* under grant agreement No 692178 (EBSIS project) and H2020-ICT-2015 under grant agreement 690111 (SecureCloud project). This work was partly supported by a grant of the Romanian National Authority for Scientific Research and Innovation, CNCS/CCCDI - UEFISCDI, project number 10/2016, within PNCDI III. Rafael Pires is also sponsored by CNPq, National Counsel of Technological and Scientific Development, Brazil.

References

[1] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[2] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich, "VC3: Trustworthy data analytics in the cloud using SGX," in *2015 IEEE Symposium on Security and Privacy*, 2015, pp. 38–54.

[3] O. Ohrimenko, M. Costa, C. Fournet, C. Gkantsidis, M. Kohlweiss, and D. Sharma, "Observing and preventing leakage in MapReduce," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. ACM, 2015, pp. 1570–1581.

[4] T. T. A. Dinh, P. Saxena, E.-C. Chang, B. C. Ooi, and C. Zhang, "M2R: Enabling stronger privacy in MapReduce computation," in *24th USENIX Security Symposium (USENIX Security 15)*. USENIX Association, 2015, pp. 447–462.

[5] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The HiBench benchmark suite: Characterization of the MapReduce-based data analysis," in *2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010)*, 2010, pp. 41–51.

[6] S. Dolev, Y. Li, and S. Sharma, "Private and secure secret shared MapReduce (extended abstract)," in *Data and Applications Security and Privacy XXX: 30th Annual IFIP WG 11.3 Conference, DBSec 2016, Trento, Italy, July 18-20, 2016. Proceedings*, S. Ranise and V. Swarup, Eds. Springer International Publishing, 2016, pp. 151–160.

[7] A. Shamir, "How to share a secret," *Commun. ACM*, vol. 22, no. 11, pp. 612–613, Nov. 1979.

[8] W. Wei, J. Du, T. Yu, and X. Gu, "SecureMR: A service integrity assurance framework for MapReduce," in *2009 Annual Computer Security Applications Conference*, 2009, pp. 73–82.

[9] 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. USENIX Association, 2010, pp. 20–20.

[10] C. Zhang, E. C. Chang, and R. H. C. Yap, "Tagged-MapReduce: A general framework for secure computing with mixed-sensitivity data on hybrid clouds," in *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2014, pp. 31–40.

[11] P. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *ACM Computing Surveys*, vol. 35, no. 2, pp. 114–131, 2003.

[12] 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. ACM, 2016, pp. 10:1–10:10.

[13] J. MacQueen, "Some methods for classification and analysis of multivariate observations," in *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Statistics*. Berkeley, Calif., USA: University of California Press, 1967, pp. 281–297.

[14] A. Hirschi, "Traveling light, the Lua way," *IEEE Software*, vol. 24, no. 5, pp. 31–38, 2007.

[15] P. Hintjens, *ZeroMQ: Messaging for Many Applications*. O'Reilly Media, Inc., 2013.

[16] R. Barazzutti, P. Felber, C. Fetzer, E. Onica, J. Pineau, M. Pasin, E. Rivière, and S. Weigert, "StreamHub: a massively parallel architecture for high-performance content-based publish/subscribe," in *The 7th ACM International Conference on Distributed Event-Based Systems, DEBS '13*, 2013, pp. 63–74.

[17] R. Barazzutti, T. Heinze, A. Martin, E. Onica, P. Felber, C. Fetzer, Z. Jerzak, M. Pasin, and E. Rivire, "Elastic scaling of a high-throughput content-based publish/subscribe engine," in *2014 IEEE 34th International Conference on Distributed Computing Systems*, 2014, pp. 567–576.

[18] J. Daemen and V. Rijmen, *The Design of Rijndael*. Springer-Verlag New York, Inc., 2002.

[19] 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.

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