Monitoring Data Integrity in Big Data Analytics Services

Konstantinos Mantzoukas
Department of Computer Science
City, University of London
London, United Kingdom
Konstantinos.Mantzoukas.1@city.ac.uk

Christos Kloukinas
Department of Computer Science
City, University of London
London, United Kingdom
C.Kloukinas@city.ac.uk

George Spanoudakis
Department of Computer Science
City, University of London
London, United Kingdom
G.E.Spanoudakis@city.ac.uk

***Abstract*—Enabled by advances in Cloud technologies, Big Data Analytics Services (BDAS) can improve many processes and identify extra information from previously untapped data sources. As our experience with BDAS and its benefits grows and technology for obtaining even more data improves, BDAS becomes ever more important for many different domains and for our daily lives. Most efforts in improving BDAS technologies have focused on scaling and efficiency issues. However, an equally important property is that of security, especially as we increasingly use public Cloud infrastructures instead of private ones. In this paper we present our approach for strengthening BDAS security by modifying the popular Spark infrastructure so as to monitor at run-time the *integrity* of data manipulated. In this way, we can ensure that the results obtained by the complex and resource-intensive computations performed on the Cloud are based on correct data and not data that have been tampered with or modified through faults in one of the many and complex subsystems of the overall system.**

***Keywords*-big data services, security, run-time monitoring, data integrity.**

I. Introduction

The Internet of Things (IoT), enabled by the Cloud, is generating data at an ever increasing rate. Gartner, a research and advisory company, estimates that by 2020 the number of connected devices will be in the vicinity of 20 billion [1], whereas IHS, a financial market company, forecasts that this number will skyrocket to 125 billion devices in 2030 [2]. Many data processing frameworks have been implemented to easily model and effectively process large datasets, with a main focus on fault tolerance, efficiency, and scalability [3]. These properties are vital but we also need to address other non-functional requirements, especially security. An indicative use case that highlights this point is the current debacle over the release of heat-maps from a social network for runners called Strava [4]. In that specific incident, researchers and journalists used publicly available data from the daily activities of soldiers to identify the locations of secret USA military bases around the world.

Data integrity is a topic that has been examined in great depth in the domain of databases, especially for semantic integrity, to ensure that certain data constrains are respected, *e. g.*, range of values. It has also been studied in the context of digital signatures for asymmetric encryption to support the authentication of entities and the encryption of their messages. In this work we focus instead on supporting the runtime monitoring of data integrity to monitor whether data has been tampered with from an external source during service execution. In our solution, as data gets transformed by different service operations, we keep track of the checksums of the intermediate data produced and compare them between service operations to decide whether data integrity has been violated. Violations indicate either faults in the cloud infrastructure or unauthorized modifications by attackers.

Section II gives a detailed account of our design and system architecture. Section III compares our approach with related work. and finally section IV summarizes our work and discusses possible improvements.

II. Design & System Architecture

This section presents an overview of the proposed monitoring specification, reviews the components of the proposed monitoring system, and describes how they can be used to monitor data integrity at runtime. Our monitoring specification is composed of three main artifacts:

i Monitoring rules of the specific property of interest;
ii Monitoring events of interest; and
iii Event captors that need to be installed.

We have chosen Apache Spark [5] as the data processing framework, since it is an open source project that allows us to experiment with its code. In addition, Apache Spark has one of the largest communities of users and contributors, making our solution of potential benefit to a larger part of the big data community compared to other alternatives [6], [7], [8], [9]. Lastly, Apache Spark's unified API for addressing multiple types of processing activities in the big data processing space such as machine learning, batch processing, stream processing and graph processing, allows our work to impact multiple classes of data processing challenges.

In order to derive the aforementioned artifacts we first need to quickly overview what the code of an Apache Spark service [5] actually does. Spark code operates on a set of *Resilient Distributed Dataset* (RDD) structures to hold the data produced by each Spark operation. RDDs are computed lazily — when one needs to be read it is computed at that point if it does not exist already, causing other RDDs on which it depends to be computed as well. Spark splits RDDs into partitions across different nodes to distribute the data and computations respectively. Spark RDD operations are of two types: *actions*, *e.g.*, `foreach()`, which cause an RDD to be computed by means of applying a user function to every data item of the RDD and returning a non-RDD value to the main program, and *transformations*, *e.g.*, `flatMap()`, which define how the partitions of an RDD will be produced by means of applying a user function on every data item of the partitions of another RDD. Actions return data to their caller, whereas transformations produce new RDDs lazily.

RDD transformations themselves can be divided into two groups. *1-to-1* transformations or transformations with *narrow dependencies* as Spark calls them [1], *e.g.*, `flatMap()`, transform an RDD's partition directly into another RDD's partition.These are applied directly to the contents of each input RDD partition to obtain the contents of the respective output RDD partition. *N-to-1* transformations or transformations with *wide dependencies* as Spark calls them [2], *e.g.*, `combineByKey()`, require input from multiple partitions of another RDD to compute an RDD's partition. N-to-1 transformations group input partition data by the ID of the consuming transformation node and store them in separate partition *segments*.

A. Monitoring Rules

To support monitoring of data integrity, we must ensure that whenever a Spark operation reads data then some other Spark operation had actually written these data. In Event Calculus [10], the input language of the Everest Monitor [11] we use, this is expressed as in formula 1. $O_{r(w)}$ identifies some Spark operation that reads (resp. writes) an RDD R, and d is the value of the data read/written.

$$\begin{aligned}&\texttt{Happens(read(}O_r\texttt{, }R\texttt{, }d\texttt{), }t_r\texttt{)}\\ &\rightarrow \texttt{Happens(write(}O_w\texttt{, }R\texttt{, }d\texttt{), }t_w\texttt{)} \wedge t_w \leq t_r\end{aligned} \quad (1)$$

Formula 1 checks that data written has not been corrupted/manipulated/injected, since Spark does not allow RDDs to be updated, so a write operation that precedes a read operation is unique and we do not need to identify "the most recent" write before a read. Therefore if the data do not match it is due to an external entity that has altered them either intentionally or unintentionally. Of course, formula 1 is too high-level. As RDDs are split into partitions, R is really a pair of (R, P), where R identifies the RDD and P the specific partition a Spark operation is currently operating on. The data identifier d is also an abstract identifier — what exactly it should be used for is a design choice. For example, the most basic and unsophisticated solution would use each complete data-tuple written/read by a Spark operation. However, this would create too much strain on the network and the monitoring engine, which needs to unify rule variables with event values. A more refined solution uses a *hash* for each one of the partitions that the data-tuples are grouped into, speeding up the matching of event values with the monitoring rule event variables in the engine.

[1] spark.apache.org/docs/2.2.1/api/java/org/apache/spark/Dependency.html
[2] spark.apache.org/docs/2.2.1/api/java/org/apache/spark/rdd/RDD.html

B. Events of Interest

As aforementioned, the events of interest for the rule in formula 1 are these writing/reading data in RDD partitions.

Events per Data Tuple: In a simple design, events can be produced for each different data-tuple as it is being written/read, by viewing the internal representation of the tuple as a byte sequence and computing a hash value to distinguish it from other tuples (with some small probability of clashes — dependent on the exact hashing function). The byte sequence is obtained by serializing the tuple, *i. e.*, representing the tuple as a sequence of bytes that can be stored in a file or sent over the network. This is an action that is actually carried out by Spark itself, as it needs to have a serialized version of the data to store them in an RDD (which can be seen as a file in a distributed file-system).

Events per Partition — Less Resource Usage: A more sophisticated design we have explored produces a single hash value for all the data held in one RDD partition as a single unit, so as to reduce the network traffic and the load of the monitor substantially. For 1-to-1 transformations this approach does not require any changes in the monitoring rule we employ — it simply creates a single event per RDD partition read/written instead of creating as many events from that partition as the data-tuples it contains. However, an issue with this design arises when we face an N-to-1 transformation. Now the data held inside a partition are not read in their entirety by some node computing the N-to-1 transformation. Instead only a subset of them are read — those in the segment assigned to that specific node. In such instances it is required to use the internal Spark structures in order to identify the segments that will be read by each consuming node and compute hash values (and respective events) separately for each of them. In these situations, the component R of the rule in formula 1 does not correspond to an (RDD ID, partition ID) pair but to an (RDD ID, segment ID). In addition, now each RDD partition leads to the creation of not just a single read/write event but as many of them as there are partition segments.

C. Event Captors

In order to support the run-time monitoring we need to also add code inside Spark so that we can:

1) Identify when a new RDD is written/read; and
2) Construct the event messages for data writes/reads.

These captors work differently depending on the type of transformation used. From an implementation point of view, we need to intercept the creation of the partitions of RDDs and calculate a hash value for every data-tuple or every partition, depending on the monitoring strategy that we want to employ. By design, each type of RDD extends a base abstract class called `RDD`. One of the abstract methods of the class `RDD` that each type of RDD has to implement is called `compute()`. Method `compute()` is responsible for the computation of all the partitions of the RDD.

1) 1-to-1 Transformations: As aforementioned, for these transformations each output RDD partition can be computed by reading only one input RDD partition. This enables Spark to execute those computations in parallel at the locations where the partitions reside. In addition, Spark for efficiency, can fuse together multiple 1-to-1 transformations that are invoked sequentially and execute them within a single unit of work, called a *task*.

2) N-to-1 Transformations: The approach described so far works well for 1-to-1 transformations mapping one RDD into another. If we were to follow the simple approach, whereby each data-tuple becomes a single event, these transformations would work just as the 1-to-1 ones. As aforementioned however, we have also considered another approach that is more coarse-grained — instead of considering a single data-tuple as the unit for checking integrity, we instead consider a whole RDD partition as the unit, producing a single checksum for each RDD partition. This approach produces far fewer events, speeding up the monitoring and also reducing the strain on the memory and network bandwidth. For 1-to-1 transformations this approach is more straight forward to implement, since an RDD's partition is created by data coming from a single partition of another RDD. For N-to-1 transformations however, this approach is more complex, since now an RDD's partition is read only for its segment of interest to the specific RDD partition under construction — the other partitions of the RDD currently being read provide the remaining segments. For this reason we need specialized event captors in N-to-1 transformations that use Spark's internal representation to identify the RDD partition segments and produce a hash value for each of them. In essence N-to-1 transformations take place in two steps. In the first step the previous map transformation groups data into segments according to the N-to-1 transformation's RDD partition ID that they belong to. In the second step, the N-to-1 transformation consumes the segment data and applies a user defined combinator function. We implemented event-capturing by intercepting the action of writing data into segments and also intercepting the action of reading the data from them when invoking method `compute()` of the N-to-1 transformation. N-to-1 transformations create a special type of RDD, called `ShuffledRDD` [3].

[3]https://spark.apache.org/docs/2.2.1/api/scala/#org.apache.spark.rdd.ShuffledRDD

Capture write events for N-to-1 transformations: All write operations in Spark are performed by an object implementing interface `ShuffleWriter` that has two methods — `write()` to write data in segments and `stop()` to interrupt the process of writing. Method `write()` does not actually keep track of the RDD and partition it is writing in. However this information is important to us, so we need to pass it along. To enable this we declare a method with additional RDD and partition parameters in Spark's default implementation of `ShuffleWriter` is class `SoftShuffleWriter` where our custom `write()` method is implemented. The implementation of our `write()` method is identical to Spark's original method, apart from the fact that when it internally invokes method `writePartitionedFile()` to group data into segmensts it also passes along a reference to the RDD. Method `writePartitionedFile()` is declared in class `ExternalSorter`, which combines the values of each input partition segment using a user function. Inside `writePartitionedFile()` data is written in the segments and appropriate indexes to the segments are stored as well. Spark does this by iterating through the data-tuples and writing the segments without returning anything to the caller method, namely `writePartitionedFile()`. However, for monitoring we need to return the written value and capture the checksum of the tuple or the partition segment according to the monitoring strategy used. This is done by implementing our own `write()` method, called `writeAndReturnValue()`, which also returns the value written so as to produce a checksum for it afterwards.

Capture read events for N-to-1 transformations: Similar to how data-tupes are written by N-to-1 transformations, reading is performed by an object implementing interface `ShuffleReader` that has a single method called `read()`. Method `read()` returns an iterator with all the data that the `compute()` method of the current N-to-1 `ShuffledRDD` will read from its input RDD. `BlockStoreShuffleReader`, Spark's default implementation of this interface, takes no arguments. Since we need to keep track of the RDD and the segment that the data was read from, we again have to pass as extra arguments the RDD and segment IDs that it reads data from. This is implemented in class `BlockStoreShuffleReader`, whose `read()` method opens up multiple streams to the input RDD partitions and reads the corresponding segments.

III. Related Work

The security of distributed computations executed in public clouds has made its way into the top ten Big Data security and privacy challenges published by the Cloud Security Alliance (CSA) [12]. A significant body of work has been produced to address the issue of the preservation of data integrity for data outsourced in the cloud. jMonAtt [13] addresses the issue of application integrity preservation and describes an architecture that provides guarantees for cloud applications by enforcing dynamic attestation through the HotSpot virtual machine. Efforts to target the challenges of runtime data integrity monitoring has been done in [14] and [15], in both of which the computation runs more than once to verify the integrity of the data at runtime. In [14] data is processed both in a public and in a private cloud and the two results are compared whereas in [15] a reputation-based rating system is built through redundant computations, where every computation node is rated on the basis of the quality of the results that it produces. Both approaches have been implemented in map/reduce types of computations.

Our approach to data integrity runtime monitoring is different in three main ways. Firstly, it takes into account not only map/reduce operations but also 1-to-1 transformations. Secondly, it avoids re-computations, minimizing the overhead of the BDAS infrastructure, and, thirdly, it allows for two separate degrees of granularity with respect to the monitoring strategy, namely per data-tuple and per partition.

IV. Conclusions

In this paper we have presented the design of two approaches for monitoring data integrity in Big Data Analytics Services (BDAS). More specifically, we have analyzed how Spark [5] transfers data from one computation node to another and presented what extra code needs to be added in Spark to allow us to monitor these data exchanges and to identify inconsistencies between what one node produces and another node consumes from it. To achieve this we had to consider the different types of Spark primitives (*actions* and *transformations*) and how these interact with the distributed data structures (called RDD in Spark) that they produce and consume.

Currently we are working on improving different aspects of our work. First, we are working on speeding up our approach by performing all event-captor work in parallel to what Spark normally does. Second, we are working on making our monitoring more easy to fine-tune and pinpoint the exact parts of a BDAS system where monitoring should be applied, since one may wish to turn off monitoring in particular parts of the system and focus on other parts only. Third, we are working on linking with the fault-tolerance mechanisms of Spark, so as to be able to ask it to re-compute the data sets where integrity was violated (potentially with the finer grained monitoring enabled) and have a system that can not only identify the integrity security property violation but also react to it. Finally, we are working on extending our technique so as to support other security properties as well.

References

[1] Gartner, "Leading the IoT, Gartner insights on how to lead in a connected world," Tech. Rep., 2017.

[2] IHS, "The Internet of Things: A movement, not a market," Tech. Rep., 2017.

[3] M. Armbrust, T. Das, A. Davidson, A. Ghodsi, A. Or, J. Rosen, I. Stoica, P. Wendell, R. Xin, and M. Zaharia, "Scaling Spark in the real world: Performance and usability," *Proc. VLDB Endow.*, vol. 8, no. 12, pp. 1840–1843, Aug. 2015.

[4] "Strava data heat maps expose military base locations around the world — Wired," https://www.wired.com/story/strava-heat-map-military-bases-fitness-trackers-privacy/, 2018, [Online; accessed 15-Feb-2018].

[5] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 2–2.

[6] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache Flink: Stream and batch processing in a single engine," *IEEE Data Eng. Bull.*, vol. 38, pp. 28–38, 2015.

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

[8] "Apache Storm," http://storm.apache.org/, 2018, [Online; accessed 15-Feb-2018].

[9] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. Henry, R. Bradshaw, and Nathan, "FlumeJava: Easy, efficient data-parallel pipelines," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2 Penn Plaza, Suite 701 New York, NY 10121-0701, 2010, pp. 363–375.

[10] M. Shanahan, "The event calculus explained," in *Artificial Intelligence Today*, 1999, pp. 409–430. [Online]. Available: doi.org/10.1007/3-540-48317-9_17

[11] G. Spanoudakis, C. Kloukinas, and K. Mahbub, "The SERENITY runtime monitoring framework," in *Security and Dependability for Ambient Intelligence*, ser. Advances in Information Security. Springer, 2009, vol. 45, pp. 213–237.

[12] R. Behrends, L. K. Dillon, S. D. Fleming, and R. E. K. Stirewalt, "Expanded top ten big data security and privacy challenges," Cloud Security Alliance, Tech. Rep. MSU-CSE-06-2, January 2013.

[13] H. Ba, H. Zhou, S. Bai, J. Ren, Z. Wang, and L. Ci, "jMonAtt: Integrity monitoring and attestation of jvm-based applications in cloud computing," in *2017 4th International Conference on Information Science and Control Engineering (ICISCE)*, July 2017, pp. 419–423.

[14] Y. Wang, Y. Shen, and X. Jiang, "Practical verifiable computation — A MapReduce case study." *IEEE Trans. Information Forensics and Security*, vol. 13, no. 6, pp. 1376–1391, 2018.

[15] Z. Gao, N. Desalvo, P. D. Khoa, S. H. Kim, L. Xu, W. W. Ro, R. M. Verma, and W. Shi, "Integrity protection for big data processing with dynamic redundancy computation," in *2015 IEEE International Conference on Autonomic Computing*, July 2015, pp. 159–160.