Bagheri *et al. J Big Data* (2020) 7:86
https://doi.org/10.1186/s40537-020-00362-1

Journal of Big Data

RESEARCH Open Access

Efficient verification of parallel matrix multiplication in public cloud: the MapReduce case

Ramtin Bagheri, Morteza Amini* and Somayeh Dolatnezhad Samarin

*Correspondence:
amini@sharif.edu
Department of Computer Engineering, Sharif University of Technology, Azadi Ave., Tehran, Iran

Abstract

With the advent of cloud-based parallel processing techniques, services such as MapReduce have been considered by many businesses and researchers for different applications of big data computation including matrix multiplication, which has drawn much attention in recent years. However, securing the computation result integrity in such systems is an important challenge, since public clouds can be vulnerable against the misbehavior of their owners (especially for economic purposes) and external attackers. In this paper, we propose an efficient approach using Merkle tree structure to verify the computation results of matrix multiplication in MapReduce systems while enduring an acceptable overhead, which makes it suitable in terms of scalability. Using the Merkle tree structure, we record fine-grained computation results in the tree nodes to make strong commitments for workers; they submit a commitment value to the verifier which is then used to challenge their computation results' integrity using elected input data as verification samples. Evaluation outcomes show significant improvements comparing with the state-of-the-art technique; in case of 300*300 matrices, 73% reduction in generated proof size, 61% reduction in the proof construction time, and 95% reduction in the verification time.

Keywords: MapReduce, Computation integrity, Matrix multiplication

Introduction

Thanks to the existence of the Internet and its global popularity, the pace of generating data has been escalated significantly in recent years. In 2014, a research showed that the volume of generated data is doubled every year, and this multiplier will increase up to ten until 2020 [1]. Therefore, challenges regarding storing and processing the massive generated data has motivated researchers to find efficient solutions. Cloud-based services have emerged to solve mentioned challenges, giving its users the ability to process their big data.

The expansion of public clouds has drawn much attentions to itself since users can simply rent their required resources without worrying about staggering costs of buying and maintaining them. In order to speed up the data processing in cloud services, many hardware-based and software-based approaches have been proposed. Parallel processing is one of the latter type of techniques, by which input data is divided into several groups, and each group is processed independently. MapReduce [2] is a parallel processing model of programming that have been implemented and used by a diverse set of companies and academic researchers for big data processing purposes such as machine learning and data analysis. There exist several software frameworks implemented based on MapReduce model, among of which Hadoop MapReduce and Spark are the most prominent ones.

Setting aside the benefits, cloud services have caused some security issues for their customers; failures such as Byzantine, outside attacks, and even cloud owners' economic motivations (i.e., saving processing power by faking computations) leads to incorrect results in those systems. Therefore, without being assured of the real behavior of the cloud during the processing stage, users cannot consider the outputs as trustworthy.

Among the MapReduce applications, matrix multiplication is one of the popular ones. It is an essential operation in linear algebra, and has numerous applications in many areas such as [3–5]. Public clouds offering MapReduce services can nicely handle matrix multiplication computations since the MapReduce model is fully compatible with them. Nevertheless, considering the security issues mentioned for cloud computing, finding an efficient approach for ensuring result integrity has been a challenge for outsourced computations. In this regard, several approaches addressing the issue have been proposed; however, since multiplying big matrices is a heavy job, the proposed approaches do not have a reasonable efficiency in this application and also other similar applications.

In this paper, we propose a result verification approach for matrix multiplication computations in the MapReduce model using the Merkle tree data structure; however, our approach is also applicable on any other applications having the same set of computation features as matrix multiplication.

Briefly, our contributions in this paper are as follows:

- Proposing a new approach for verifying the matrix multiplication computation results in MapReduce.
- Performing analysis on the system security.
- Implementing the proposed approach on a MapReduce system, measuring its performance, and comparing it with related works.

In the rest of this paper, "Background" section gives some background information about the concepts we used in our work. "Related Work" section reviews previous related works and describe why they do not meet our needs. "Basic Definitions and attack model" section introduces some basic definitions, which we are going to use in our explanations and talks about the attack model we consider for our approach, as well as the types of the errors that we want to detect. "Methods and proposed approach" section explicates our main work. "Evaluation and analysis section focuses on analyzing and evaluating the approach and its comparison with a recent related work in terms of performance. Finally, "Conclusion" section concludes the paper and introduces new research challenges in this field.

Background

Some basic concepts and definitions, which are required to be understood before studying the proposed approach, are presented in the following.

MapReduce

MapReduce [2] is a programming model for processing big data clusters in distributed environments. It has two main functions, Map and Reduce, and three main entity types: *Master*, *Worker*, and *Distributed File System* (DFS). The master manages job flows and interactions among other entities. Workers, consisting of mappers and reducers, do the computation tasks assigned by the master. Finally, DFS acts as a distributed disk storage in the system.

The process flow in MapReduce, which is shown in Fig. 1, consists of three main phases: *Map*, *Shuffle*, and *Reduce*. First, input data is divided into several specific-sized chunks and each one is assigned to a mapper. Having called the inputs, mappers perform the computation determined in the map function and generate outputs, called *intermediate records*, in form of key-value pairs and store them on their local storages. Then, in the shuffle phase, the intermediate records are sorted, grouped, and assigned to the reducers defined by the shuffler function. Afterwards, reducers apply the reduce function on their assigned data, generate final results, and write them back on DFS.

There are different types of computations that can be performed using the MapReduce model; however, they need to share a set of predefined characteristics [6] in order to benefit the utilization brought by the model. The most important characteristic is that the computation should be able to be divided into independent subcomputations.

Matrix multiplication in MapReduce

If $A = [a_{ij}]$ is an $m \times n$ matrix and $B = [b_{ij}]$ is an $n \times p$ matrix, $C = [c_{ij}]$, the product of AB, is an $m \times p$ matrix where each element c_{ij} is calculated as

$$c_{ij} = \vec{r}_i^{\,A} \cdot \vec{c}_j^{\,B} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj} \tag{1}$$

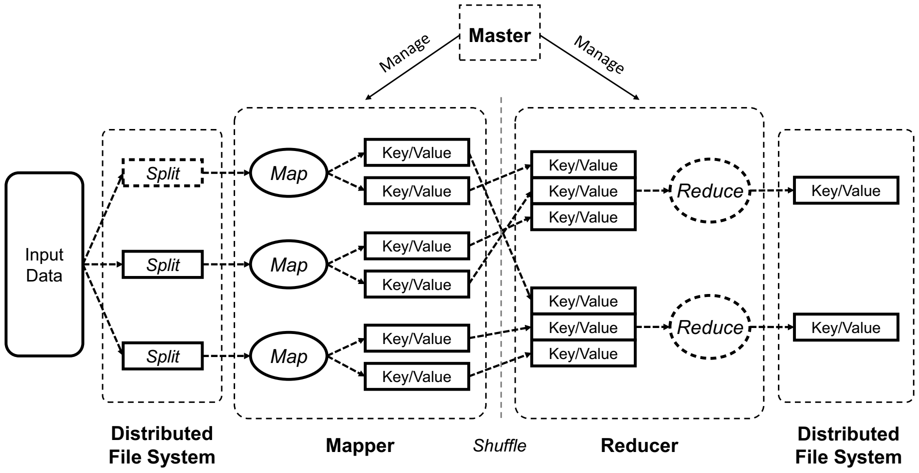


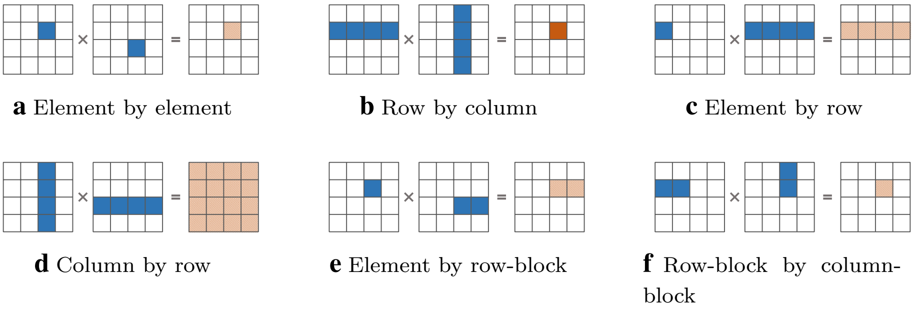
Fig. 1 Data flow of a job in MapReduce [2]

In other words, element c_{ij} of matrix C is derived from the inner product of ith row of matrix A (denoted as $\vec{r}_i^{\,A}$) and jth column of matrix B (denoted as $\vec{c}_j^{\,B}$).

The basic way of performing the computations is using the sequential algorithm, which has the time complexity of $\mathcal{O}(n^3)$. In this regard, several approaches such as parallel algorithms [7–9] have been proposed over the years with the aim of reducing the overall overhead and speeding up the computations.

Implementing matrix multiplication algorithms using the MapReduce model is one of the efficient approaches as the calculations fit nicely into the MapReduce style of computing. The element-to-element strategy (Fig. 2a), same as the sequential algorithm, imposes significant communication overhead to the system. In order to mitigate the problem and increase the utilization, several strategies are proposed over the years. Sun [10] presented the row-to-column (Fig. 2b) strategy to decompose the first and the second matrix into row and column vectors, respectively. Increasing the computation load on mappers' side, his technique aims to reduce the communication overhead. Zheng [11] suggested decomposing the first matrix into elements or columns and the second matrix into rows (Fig. 2c, d) for efficient computations in case of sparse matrices. Deng and Wu [12] made a comparison between element-to-element, row-to-column, and element-to-row strategies and claimed that the element-to-row strategy has the highest efficiency among the others for both dense and sparse matrix types. Using a balanced number of mappers, Kadhum [13] proposed element-to-row-block (Fig. 2e) and row-block-to-column-block (Fig. 2f) strategies as the general cases of the previous works to make a balance between the processing and the I/O overhead.

In overall, the aforementioned strategies try to keep a balance between communication and computation overheads. These strategies differ in how the input elements are contributed in the matrix multiplication computation, which determine the placement of the elements in the input blocks. In this regard, there are two main approaches for matrix multiplication in MapReduce; double-job and single-job approaches. In the former, the first job is responsible for labeling and grouping the elements resulted by the mappers, whereas the second one is responsible for performing the main computations (i.e., inner products of the matrices' elements) jointly done by the mappers and reducers. In the latter approach, a preprocessing stage is required for organizing and grouping the input data. Therefore, the main computations can be done in a single job by the mappers and reducers.



a Element by element **b** Row by column **c** Element by row

d Column by row **e** Element by row-block **f** Row-block by column-block

Fig. 2 Schemes of different strategies for matrix multiplication

Merkle tree-based verification

The Merkle tree-based verification method is based on a data structure called *Merkle tree* (aka *hash tree*) [14]. This method was introduced in [15] to ensure result integrity in grid computing. A Merkle tree is a complete binary tree that utilizes two functions; a one-way collision resistant hash function, and mapping function Φ, which maps a set of nodes to a set of constant-sized strings that are generated using the hash function. In a Merkle tree, leaves contain the computation results of the input data. The tree construction process begins from the leaves to the root, building each internal node using its children.

In a Merkle tree, each node has a Φ value. Assume that $L_1, \ldots, L_n$ are leaves constructed based on computation outputs $f(x_1), \ldots, f(x_n)$. In order to build a Merkle tree, the Φ value for leaf L_i is defined as

$$\Phi(L_i) = f(x_i), i \in [1, n] \tag{2}$$

Then, the participant builds a complete binary tree with these leaves. We use V to denote an internal node, and V_{left} and V_{right} to denote V's two children. The Φ value of each internal node is defined as

$$\Phi(V) = hash\big(\Phi(V_{left}) \parallel \Phi(V_{right})\big) \tag{3}$$

where "$\|$" represents concatenation of two strings, and *hash* is the one-way collision resistant hash function (e.g., SHA-1). Using Eq. 3, the hash value of the root node, marked as $\Phi(R)$, can be computed in a bottom-up manner.

To facilitate our expression, we use Λ as an abstract representation of employing Eqs. 2 and 3 to compute the $\Phi(R)$ for inputs $x_1, ..., x_n$ as follows:

$$\Lambda(x_1, \ldots, x_n) = \Phi(R) \tag{4}$$

In other words, $\Lambda(x_1, \ldots, x_n)$ returns the Φ value of a Merkle tree's root node if the Merkle tree's leaves are results of applying the f function on $x_1, \ldots, x_n$.

The verification process begins when the computations are over and the results are ready. Two roles participate in this process: *prover*, who is responsible for performing the assigned computations, and *checker*, who wants to ensure the integrity of the results. The process consists of four steps; *Commit*, *Challenge*, *Prove*, and *Verify*. FigureDefinition 3 represents an example of a Merkle tree built on arbitrary computation results $f(x_1), \ldots, f(x_n)$, which we use during the explanation in order to better understanding the process. Those steps are described in the following:

Commit. Using Eqs. 2, and 3, the prover builds a Merkle tree on its computation results (i.e., $f(x_1), \ldots, f(x_n)$), obtains Φ value of the root node (i.e., the hash of the root node), and sends it to the checker. We refer to the sent value in this step as the commitment value. In Fig. 3, R is the root node of the Merkle tree. Therefore, $\Phi(R)$ is submitted as the commitment value.

Challenge. Having received the commitment value, the checker generates some samples to challenge the prover. To be clear, sample s_i is equal to $f(x_i)$, which is resulted from applying the computation on randomly selected input data x_i. Then, the checker sends the generated samples to the prover. In the example shown in Fig. 3,

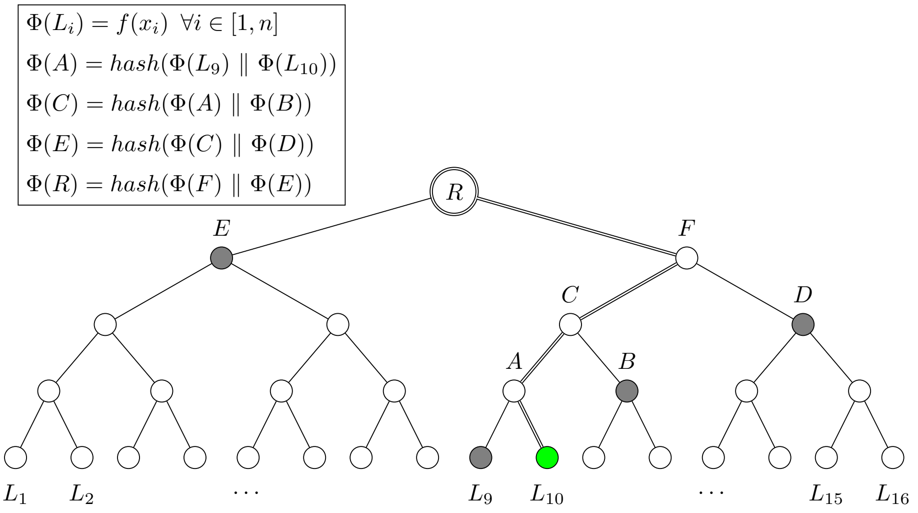

$\Phi(L_i) = f(x_i)\ \forall i \in [1, n]$
$\Phi(A) = hash(\Phi(L_9) \parallel \Phi(L_{10}))$
$\Phi(C) = hash(\Phi(A) \parallel \Phi(B))$
$\Phi(E) = hash(\Phi(C) \parallel \Phi(D))$
$\Phi(R) = hash(\Phi(F) \parallel \Phi(E))$
R
E
F
C
D
A
B
L_1
L_2
...
L_9
L_{10}
...
L_{15}
L_{16}

Fig. 3 An example of Merkle tree-based verification scheme

x_{10} is selected among the set of input data, and $s_{10} = f(x_{10})$ is sent to the prover as the verification sample.

Prove. For each sample s_i, the prover finds the corresponding leaf L_i and the nodes in its path to the root, extracts the complementary node of each, stores the Φ value of the extracted nodes in a sequence, and finally sends the sequence to the checker. This sequence of nodes is called *proving path* and is denoted as λ. The nodes in a proving path are required to regenerate the hash value of the root node using the corresponding sample. To show the derivation, we extend Eq. 4 as follows.

$$\Lambda(\lambda_i, s_i) = \Lambda(x_1, \ldots, x_n) = \Phi(R) \tag{5}$$

where s_i is a sample, λ_i is the corresponding proving path, and $\Phi(R)$ is the commitment value.

In the example shown in Fig. 3, the prover receives $f(x_{10})$ from the checker, finds the corresponding leaf L_{10}, retrieves its proving path $\lambda_{10} = \langle \Phi(L_9), \Phi(B), \Phi(D), \Phi(E) \rangle$, and then, sends λ_{10} to the checker. The proving path for each sample has $\log n$ nodes where n is the number of leaves in the Merkle tree.

Verify. For each sample s_i and its proving path λ_i, the checker regenerates the Φ value of the root node (i.e., $\Phi(R')$) for each sample. The verification is passed successfully if and only if for each generated sample, the recalculated Φ value of the root node and the commitment value (generated in the commit step) are equal (i.e., $\Phi(R') = \Phi(R)$). If the equality is not respected, the checker knows that the computation on the selected input data has not been done correctly.

Related work

Several works have been done on the result integrity of general applications in the MapReduce model. We can categorize them based on the techniques they use as *log-based*, *hardware-based*, *watermark-based*, and *replication-based*.

Log-based solutions (e.g., [16, 17]) analyze logs generated by the system to build profiles for workers. Then, the profiles are compared against a normal one to detect delinquent workers. Yoon et al. [17] suggested that Hadoop framework logs, as well as system calls, can be collected for analysis purposes. Nonetheless, since system call logs are very rich information and not all of them are going to be used for the verification process, only specific details of these logs, such as access logs to the file system and mapping/reducing task logs are considered. Despite the low overhead and less modification to the system, these solutions are vulnerable against mimicry attacks [18]. In this type of attack, attacker sends fake commands and generates unreal logs and reports them in order to impersonate its captured workers as honest ones.

Hardware-based solutions (e.g., [19–21]) require the installation of secure module at the infrastructure of the cloud. They use secure hardware chip Trusted Platform Modules (TPMs) installed on computational nodes, which can verify the integrity of their software stack ranking from the bios configuration at the booting procedure to the application that executes the mapping or reducing task. The system is initialized by registering each worker with a unique identifier on the master side, who is communicating with them during the job processing phase. However, trusted hardware-based solutions may affect the flexibility of big data computation, since the required modules need to be integrated with the system core.

Watermark-based solutions (e.g., [22, 23]) inject some data with predefined results, called watermarks, to the main input data and measure the workers' honesty by comparing the predefined results with the outputs that workers produce on the watermarks. The main assumption for these solutions is that workers should not know which data is watermark or they can lure the mechanism by submitting correct results for watermark data while submitting faulty outputs for the rest. More injected watermarks leads to a more accurate fault detection, yet the computation overhead imposed by processing watermarks increases as well. In addition, Ding et al. [22] leverage EigenTrust [24], a reputation-based trust management system (RTMS), to track the source of faults. Nevertheless, detecting faulty nodes is still accompanied with considerable amount of false positives. Lack of an efficient watermark generator is another unanswered yet important challenge in these methods.

Replication-based solutions (e.g., [25–30]) are among the most practical solutions for this purpose, since they do not need the MapReduce model to be altered fundamentally. In order to verify a task result, the task is replicated among multiple, randomly selected workers, and then, their results are compared together in order to detect inconsistencies. We also call them *voting-based* solutions since workers' results are like votes, and the correct result is considered as the one with the most number of votes. The major assumption in these solutions is that the majority of workers should be benign and always do their tasks honestly. SecureMR framework [25] is the first solution to address the result integrity issue in MapReduce model, which is based on task replications. Ding et al. [26] proposed an approach (named VAWS) using weighted undirected graphs for finding inconsistencies among workers' submitted results. Bendahmane et al. [27] suggest using an RTMS to keep the account of workers' computational behavior history for further decision makings. Khan et al. [28] proposed Hatman as the method that leverages an RTMS implementation based on EigenTrust. Producing correct results by honest workers in replication-based result verification systems is not guaranteed, because the failure reason is sometimes out of workers' control, such as incorrect computations or buggy codes. Samuel et al. [29] suggest inserting watermark-like quizzes (i.e., fake predefined tasks with the same structure as the normal ones) to workers in random time periods to calibrate their honesty rating in the system.

Despite the improvements, voting-based methods still are not entirely invulnerable against collusion attacks and cannot perform well enough in extreme faulty environments. Moreover, these methods waste lots of computation power and noticeably increase the time of performing the tasks. To remedy this problem, some approaches (e.g., [31, 32]) use trusted computational nodes, called *verifiers*, for recalculating the workers' tasks. In order to avoid computational bottlenecks, verifiers do not verify all the tasks. Instead, some tasks are selected and assigned to them in a probabilistic manner. In addition, the verification target can be subtasks' results, which requires smaller input data for verifiers. That being said, compared to the number of other workers, verifiers are in the minority, since completely trusted entities are expensive to have.

A similar work to ours is Wang et al.'s MtMR [33], which uses the Merkle tree-based verification technique to increase the utilization of verifiers in the system. In MtMR, verifiers check mappers' and reducer's results by sending samples obtained from arbitrary input blocks, receiving the corresponding proving paths, and verifying the consistency of the proving paths and the commitment value. Although mappers' results are verified in $O(\log n)$ in each round, reducers' are verified in $O(n)$ because of the heavy sample building process and communication overhead. This problem has greater effects in case of specific applications, where the reducers' inputs are heavy, such as matrix multiplication. In our approach, we show that the overall time complexity of the result verification process do not exceed $O(\log n)$ per verification sample.

Freivalds [34] proposed a novel approach to verify the matrix multiplication result in 1979. Utilizing randomization, his algorithm can verify a matrix product in $\mathcal{O}(kn^2)$ with probability of failure less than 2^{-k} provided that the verification process is executed for k rounds. In 2013, Thaler et al. [35] described an interactive proof protocol that can be specifically used for verifying the result of matrix multiplication. The time complexity of the prover and the checker in this protocol is $\mathcal{O}(n^2)$ and $\mathcal{O}(n^2 \log n)$, respectively. This shows no improvement compared to the Freivalds' classic protocol; however, it can be implemented in practice and used as a primitive for computations containing multiple runs of matrix multiplications. In addition, in terms of complexity, using the methods proposed in verifiable computations for verifying the results in MapReduce framework is more complicated than the probabilistic algorithms specifically proposed for the MapReduce framework.

There are also several works such as [36–39] addressing the result integrity of matrix multiplication in public cloud using cryptographic methods. Since they do not assume any restricting rules on the cloud side, they can be applied on the MapReduce model as well. However, they assume that clients can iterate over all the elements in the input matrices, which has time complexity of $\mathcal{O}(n^2)$.

Basic definitions and attack model

Before introducing the proposed approach, some concepts should be defined precisely and the considered attack model for our approach should be described.

Basic definitions

For every two matrices $A_{m \times n}$ and $B_{n \times p}$ and their multiplication result $C_{m \times p}$, we introduce the following terms.

Definition 1

(Result Element). The element c_{ij} is a result element if it is an element placed in the row i and column j of the multiplication result.

Definition 2

(Element Pair). The pair $\langle a_{ij}, b_{jk} \rangle$ is called an element pair , where $i \in [1, m]$, $j \in [1, n]$, and $k \in [1, p]$.

Definition 3

(Intermediate Product). The multiplication result of the two elements in each element pair is called an intermediate product c_{ij}^{k} where $\langle a_{ij}, b_{jk} \rangle$ is the corresponding element pair.

Definition 4

(Sibling Intermediate Product). Sibling intermediate products S_{ij} is the set of all intermediate products c_{ij}^{k} ($k \in [1, n]$) which are required for calculating result element c_{ij}.

Having introduced these fundamental definitions, we present a simple lemma, which is used later in this paper.

Lemma 1

The sum of all the members of sibling intermediate products S_{ij} is equal to the result matrix element c_{ij}.

Proof

The members of S_{ij} are $c_{ij}^{1}, \ldots, c_{ij}^{n}$ where n is the number of columns in the first matrix in the multiplication. By aggregating them together we have

$$\sum_{k=1}^{n} c_{ij}^{k} = c_{ij}$$

which is the same equation as the summation operation in an inner product of two vectors derived from the two matrices. □

For example, assume that we want to multiply two matrices A and B as

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} \tag{6}$$

to produce matrix C where

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \quad B = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$$

In Eq. 7, row $\vec{r}_1^{\,A}$ and column $\vec{c}_2^{\,B}$ are contributing in an inner product operation to produce result matrix element c_{12}. The pairs $\langle a_{11}, b_{12} \rangle$ and $\langle a_{12}, b_{22} \rangle$ are element pairs. Also, $c_{12}^1 = 5$ and $c_{12}^2 = 14$ are members of sibling intermediate products S_{12}.

$$\begin{aligned} \vec{r}_1^{\,A} \cdot \vec{c}_2^{\,B} &= (a_{11} \times b_{12}) + (a_{12} \times b_{22}) \\ &= (1 \times 6) + (2 \times 8) \\ &= c_{12}^1 + c_{12}^2 = c_{12} \\ &= 5 + 14 = 19 \end{aligned} \tag{7}$$

Attack and error model

Workers (i.e., mappers and reducers) in MapReduce can be classified into three different types based on their behavior:

Honest. They do their computation tasks correctly without any deviation from the predetermined way.

Semi-honest (Lazy) Cheater. In order to reduce the cost, these workers are inclined to commit cheat at their tasks whenever it is considered as a rational move, i.e., when it is cheaper than being honest. Since saving computational power is the main goal here, the cloud owner is responsible for managing the process. Hence, attempts to cover cheating behaviors should not result in blaming other workers.

Malicious. There are situations where a number of workers are compromised by attackers and therefore they become faulty nodes who deliberately make errors in the computations. Malicious workers not only attempt to cover their dishonest behavior traces at any cost, they can sabotage the system independently or in a team including some other same-type workers.

As we explain later, detecting malicious worker behavior imposes significant overhead to the system. Therefore, the focus in our proposed approach is on detecting faulty behaviors performed by semi-honest workers since their tendency to maximize their cost-saving attitude is aligned to the approach needs.

Moreover, possible errors emerging in matrix multiplication computations can be related to at least one of the following reasons:

- Element pairs' multiplication. Intermediate elements are not correctly computed, or there is at least one element pair that does not contribute in calculating intermediate elements.
- Intermediate products' summation. The summation of intermediate products is not correctly computed, or there is a sibling intermediate product that is not contributed in calculating the final result.

The integrity of matrix multiplication result is assured if there exist no instances of these errors in the computations.

Methods and proposed approach

In this section, we discuss our proposed approach, its architecture model, and process steps in more details.

Architecture model

Our approach uses a hybrid cloud model structure, which is shown in Fig. 4. The main computations and verification process are performed inside the public and private cloud, respectively. Workers and DFS (Distributed File System) are inside the public cloud while the master and verifiers are inside the private cloud. The cloud domains are connected to each other using a public communication infrastructure like Internet; however, for the sake of privacy, the data transmission can be secured using encryption over data. Our approach is not depended on encryption technique which is used for maintaining privacy Since we focus on integrity of computations in this paper, no further discussion will be placed on the required features of the encryption technique.

Assumptions

Being an isolated system, the private cloud is easy to be secured and become a trusted domain. The public cloud, however, is not trusted and so do its entities. Since there exist methods to guarantee DFS integrity (e.g., [40]), we can make an exception that DFS is a trusted entity. Therefore, that leaves the workers in the public cloud as the untrusted entities in the system.

Despite the fact that there are several verifiers to perform the verification in the private cloud, we assume verifiers as a single entity to avoid considering the inner relations between them.

The main computations in matrix multiplication using the MapReduce model are performed on element pairs and sibling intermediate products. No matter which implementation of matrix multiplication (explained in "Matrix multiplication in MapReduce" section) is used, element pairs are not going to change. For example, in Eq. 6, $\langle a_{11}, b_{11}\rangle$ is always a valid element pair without considering which multiplication technique is going to be applied. Therefore, our approach can be applied on any implementation of matrix

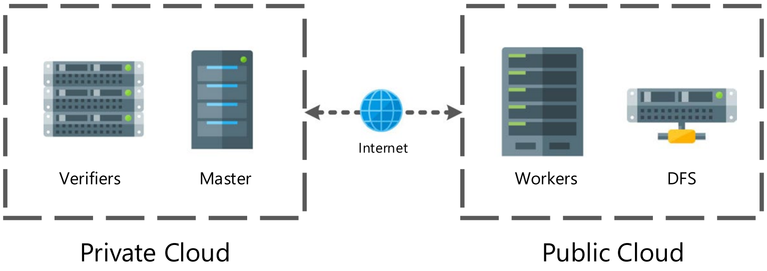


Fig. 4 The architecture model of the proposed approach

multiplication in the MapReduce model. Nevertheless, for the sake of simplicity, we used the column-by-row technique in our examples in this paper.

Moreover, we assume that the whole process is a single MapReduce job (e.g., preparation process is performed on input matrices). The fact that matrix multiplication is performed in single-job or double-job manner only affects the time in which the whole computation is done. That means it does not affect the process flow in our approach nor the verification results.

The result verification process

As we mentioned in "Related Work" section, each proposed solution in the literature has its own advantages and disadvantages; however, in our opinion, replication-based solutions are more interesting than the others based on two reasons: Firstly, they do not need any modification to be made in the infrastructure layer. Secondly, desirable fault detection accuracy is achievable by adjusting proper amount of computation power, which the granularity increases using the Merkle-tree based verification method. In our approach, which utilizes the Merkle tree verification method, we want to reduce the overhead while keeping the fault detection accuracy at the same level as the state-of-the-art solution's.

The verification process is carried out between the verifier, as the checker, and reducers, as the provers, to ensure the integrity of both map and reduce outputs. The process has the following main phases: *Preparation*, *Initialization*, *Sampling*, *Proof Construction*, and *Proof Verification*. In what follows, we elaborate the procedure in each phase, separately.

Initialization

The initialization phase runs concurrently with the reduce phase. During this phase, each reducer builds a Merkle tree M_{ij} per each result matrix element c_{ij}. We denote the set of result matrix elements computed in Reducer r as E_r, which we will use later. The leaves of each Merkle tree M_{ij} are the sibling intermediate products S_{ij}. In the following, we call those Merkle trees as *computation trees*.

During the reduce phase, reducers sum intermediate products in each sibling group together to produce the final matrix elements. By using the Merkle-tree based verification method in the way that is introduced in [15], the hash value of root node is used to verify that sibling intermediate products, are only considered in the tree construction process. In other words, there is no guarantee that all the intermediate products are actually used to produce the result elements. As the result, semi-honest reducers can ignore a proportion of the input data while applying the reduce function. A naive solution would be scheduling a deep probe through replicating a proportion of reduce tasks, like what MtMR [33] already does. Nonetheless, since the verifiers are the minority, the solution does not scale with the size of input data, thereby causing bottlenecks in the system in case of processing over big data.

To rectify the problem, we need the ability to verify that for each arbitrary input data, required computations are truly performed. Two types of computations are performed in matrix multiplication; multiplying elements in each pair, and summing sibling intermediate products. The integrity of the former can be verified with the current configuration of

Merkle tree (i.e., by checking whether a corresponding leaf exists in the tree). However, in order to verify the integrity of the latter, we need to modify the default configuration.

In matrix multiplication, each intermediate product c_{ij}^{k}, where $k \in [1, n]$, participates in a summation operation with other sibling intermediate products to produce c_{ij}. We change how the summation is performed: if we assume sibling intermediate products S_{ij} are placed in the leaves of a complete binary tree and every inner node contains the sum of its children's value, then the sum of all values in the leaves is placed in the root node. To verify that whether an intermediate product has contributed in the computations, we should retrieve nodes in the corresponding proving path and perform $\log(n)$ summations until we reach the root node. We call the path of performing summation operations for each leaf as *contribution trace*. Therefore, we can say that by making the required modifications, the whole process is verifiable for each value in the leaves in time complexity of $\log(n)$.

We introduce assignment function Θ to take the results of summation operations into account. Assume that $L_{ij}^{1}, \ldots, L_{ij}^{n}$ are the leaves of Merkle tree M_{ij}. L_{ij}^{k}, where $k \in [1, n]$, is corresponded to c_{ij}^{k} and its Θ equals to c_{ij}. In other words, we have:

$$\Theta(L_{ij}^{k}) = c_{ij}^{k} = a_{ik} \times b_{kj}, k \in [1, n] \tag{8}$$

Subsequently, for each inner node V, its Θ value is calculated as

$$\Theta(V) = \Theta(V_{left}) + \Theta(V_{right}) \tag{9}$$

where V_{left} and V_{right} are V's left and right children, respectively.

Using Θ values, verifier have access to the contribution trace of selected samples. That being said, simply using Θ value does not give the intended assurance, since cheater workers can alter contribution traces whenever it is necessary (e.g., when the committed cheats are going to be revealed).

The solution here is to ask each reducer to take its Merkle tree nodes' Θ values into account while producing its commitment value. As a result, if any reducer cheats in its sum operations, it becomes undeniable (we discuss it in Theorem 2), since the commitment value is calculated using Θ values as well. In order to make the dependency, we alter function Φ and denote the modified version as Φ'. For each leaf node L_{ij}^{k} in Merkle tree M_{ij}, the Φ' value is defined as

$$\Phi'(L_{ij}^{k}) = hash\Big(\Theta(L_{ij}^{k}) \parallel i \parallel j \parallel k\Big), k \in [1, n] \tag{10}$$

Subsequently, the Φ' value of each inner node V is defined as

$$\Phi'(V) = hash\big(\Phi'(V_{left}) \parallel \Theta(V_{left}) \parallel \Phi'(V_{right}) \parallel \Theta(V_{right}) \parallel \Theta(V)\big) \tag{11}$$

As a result, the hash value of root node in each computation tree is depended on (1) the multiplication result of each element pair, and (2) the contribution trace of each intermediate product.

By the end of reduce phase, each reducer r has constructed $|E_r|$ computation trees and have to submit the hash values of their root nodes as the commitment. To avoid extra overhead, it is desirable to merge computation trees in each reducer and produce a single commitment value. Therefore, each reducer builds another Merkle tree, called as the *joiner tree*, on top of its computation trees. From now on, we call the Merkle tree constructed in this way as a *joined tree*, which is combination of computation trees and a joiner tree. By ending the initialization phase, each reducer r has calculated its commitment value $\Phi'(R_r)$. Figure 5 depicts the schema of the reducer r's joined tree with $R_{x_1y_1}, \ldots R_{x_ny_n}$ as its computation trees, where $n = |E_r|$ and (x_i, y_j) are row and column number of the result matrix.

During the construction of each node V in a joined tree, we have $\Theta(V) = 0$ provided that V is an inner node in the joiner tree. Each node in the proving path is also labeled to guide the verifier whether the node belongs to the joiner tree or not.

For instance, the data flow of a simple matrix multiplication job in MapReduce is depicted in Fig. 6a where the matrices A and B (provided in Eq. 6) are its input. Mapper 1 has respectively the first column and the first row of matrices A and B while mapper 2 has the second column and the second row of these matrices. As we mentioned earlier, for the sake of simplicity, we apply the column-to-row multiplication technique and single-job MapReduce computation mode in the example. Also, Fig. 6b shows the Reducer 1's joined tree where M^1_{11} and M^1_{12} are its computation trees and R_1 is the root node.

At the end of initialization phase, each reducer r sends value $\Phi(R_r)$ to the verifier. As it was explained in the initialization phase, the value of $\Phi(R_r)$ is depended on the results of all the multiplication and summation operations contributed in computing the elements in E_r.

Sampling

In the sampling phase, the verifier generates and sends some samples to the reducers to challenge them. Sampling process consists of two parts: selecting input blocks, and generating the samples.

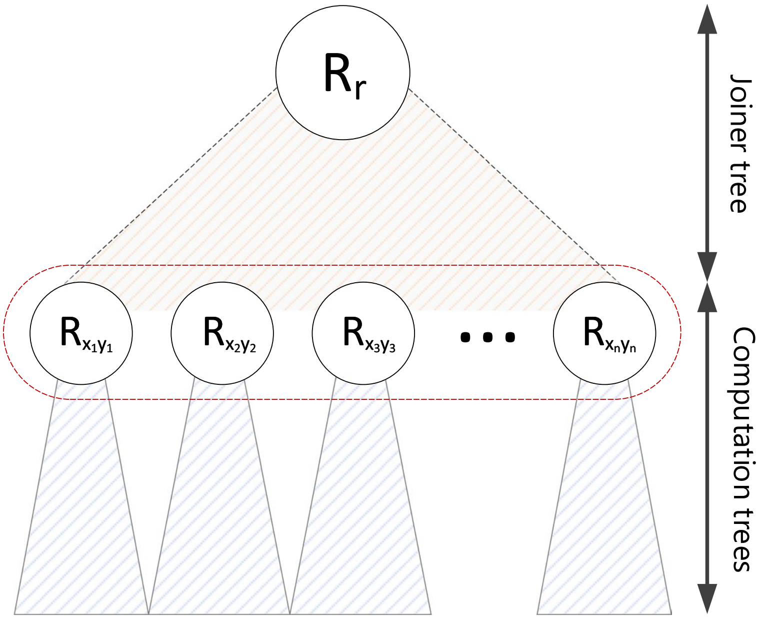
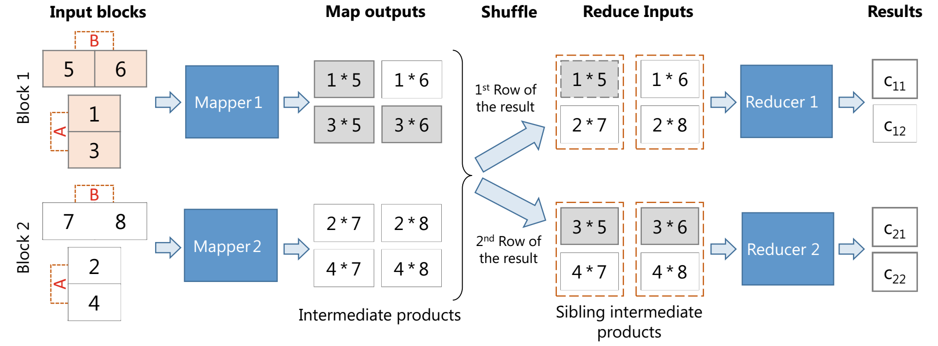
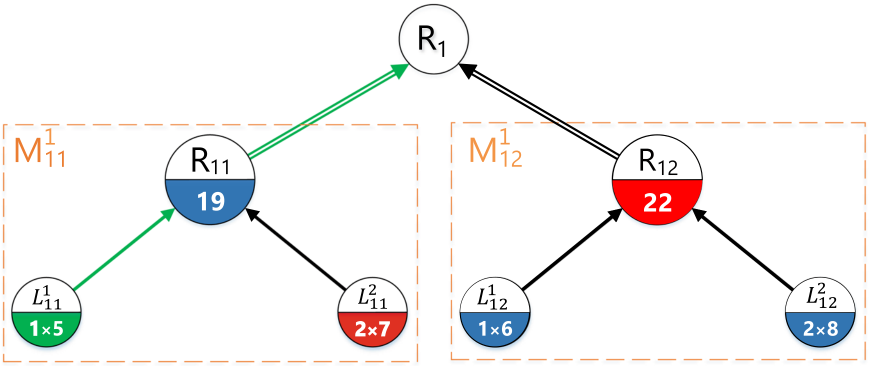


Fig. 5 Schema of a joined tree



a The data flow of the example



b The joined tree in Reducer 1

Fig. 6 An example of matrix multiplication in MapReduce [2]

The verifier randomly selects some mappers and collects their input blocks. In case of existing multiple blocks per each mapper, the verifier again can randomly pick some of the blocks in each mapper. Having picked input blocks, the verifier performs the map computations and generates some intermediate products. Since the size of each block is so small compared to the size of input data (for example, in Hadoop MapReduce, block sizes are 64 MB by default), the computation overhead caused by replicating each map task is negligible. Each sample s_{ij}^{k} consists of intermediate product c_{ij}^{k} and the corresponding auxiliary information is added to guide the reducer to find the related leaf in its joined tree. These information consists of the location of the element in the input matrices. The verifier sends each sample s_{ij}^{k} to the reducer responsible for producing c_{ij}, whom can be known by examining the shuffler function.

For instance, in the example depicted in Fig. 6, the verifier randomly selects mapper 1, fetches its input block (i.e., the first row of the matrix B and the first column of A), runs the map function on the block data to reach the corresponding intermediate products (i.e., $1 * 5, 3 * 5, 1 * 6$, and $3 * 6$), and randomly selects samples $s_{11}^{1} = 5$, $s_{21}^{1} = 15$, and $s_{21}^{2} = 18$ among the calculated intermediate products. Finally, it sends the first two samples to reducer 1 and the third one to reducer 2. It is worth mentioning that if selected mappers have multiple input blocks, the verifier can randomly pick arbitrary number of them.

Proof construction

For each sample s_{ij}, reducers collect and send evidence to the verifier in order to convince it of the two following claims:

(1) Mappers have correctly calculated the intermediate product related to the sample, and
(2) All the sibling intermediate products are contributed correctly in the summation operation.

In other words, by verifying reducers' computations, the verifier can ensure the mappers' computations, too.

For each received sample s_{ij}^k, the related reducer searches c_{ij}^k among the leaves of its computation trees and finds leaf L_{ij}^k using the auxiliary information accompanied the sample. Then, it finds the path from the leaf to the root node R^r, and builds the proving path λ_{ij}^k. Then, for each node V in the proving path, it replaces $(\Phi'(V), \Theta(V))$ with $\Phi(V)$. Finally, the reducer replies the verifier with the new proving path.

In the example shown in Fig. 6b, Reducer 1 receives sample $s_{11}^1 = 5$ and finds the corresponding leaf L_{11}^1 (the half-green node) in its Merkle tree M_{11}^1. Then, it retrieves L_{11}^1's proving path which is

$$\lambda_{11}^1 = \langle\, (\Phi'(L_{11}^2), \Theta(L_{11}^2)),\ (\Phi'(R_{12}), \Theta(R_{12}))\,\rangle$$

and sends it to the verifier.

Proof verification

For each sample, the verifier ought to rebuild the Φ value of the root node in the related joined tree. For each sample s_{ij}^k, the corresponding λ_{ij}^k received from the reducer is verified using Algorithm 1. In the algorithm, we respectively use variables *Phi* and *Theta* for storing the values Φ and Θ during the node reconstruction procedure, as well as *PTheta* for computing the Θ value for the parent node using its child nodes. Lines 1–2 initialize the required variables. Lines 3–15 loop through the nodes in the proving path to reconstruct the root node. Lines 4–8 calculate the Θ values of the nodes in the root path according to the label that the corresponding reducer attached to the nodes in the proving path. Lines 9–13 calculates the Φ values of the nodes placed in the left or right subtree (depending on which side the complementary nodes are placed). Line 14 stores the Θ value of the parent node in the main variable, *Theta*. Finally, lines 16–20 checks whether the constructed root node is eligible or not.

If the verification algorithm on λ_{ij}^k finishes successfully, two things are guaranteed about the intermediate product c_{ij}^k:

1. It has been existed in the reducer's input data. Since the verifier knows which mapper sent c_{ij}^k to the reducer, it can also know that the mapper has correctly produced c_{ij}^k using element pair $\langle a_{ij}, b_{jk}\rangle$.
2. It has correctly contributed in the summation operation performed in the targeted reducer.

For instance, let's resume the example trace in Fig. 6. Having received $\lambda_{11}^{1} = \langle\, (\Phi'(L_{11}^{2}), \Theta(L_{11}^{2})), (\Phi'(R_{12}), \Theta(R_{12}))\, \rangle$ from Reducer 1, the verifier starts computing the nodes using Algorithm 1. First, using $\Phi'(L_{11}^{2})$ and $\Theta(L_{11}^{2})$, $\Phi'(R_{11})$ and $\Theta(R_{11})$ are computed. Finally, $\Phi'(R_1)$ and $\Theta(R_1)$ are computed using the generated values and $\Phi'(R_{12})$ and $\Theta(R_{12})$. If $\Phi'(R_1)$ is equal to the commitment value from the corresponding reducer and $\Theta(R_1)$ is the same as c_{11}, the verifier ensures the correct contribution of c_{11}^{1} in computing the result element c_{11}.

Evaluation and analysis

In this section, we provide different types of analysis for our proposed approach. First, we prove the security of the approach. Then, we formally look into the detection ratio analysis, as well as overhead it imposes to the system. For further clarification, we define "cheating in a node" as cheating in the computation that its result, in the form of Θ value, is placed inside the node.

Algorithm 1 Proving path verification.

```
Input: Sample $s_{ij}^{k}$, sample location information $i$, $j$, and $k$, proving path $\lambda_{ij}^{k}$,
    commitment value $\Phi'(R)$, and result matrix element $c_{ij}$
Theta ← $s_{ij}^{k}$
Phi ← hash(Theta $\|$ $i$ $\|$ $j$ $\|$ $k$)
for each node $V$ in $\lambda_{ij}^{k}$ do
    if $V$ belongs to the joiner tree then
        PTheta ← 0
    else
        PTheta ← Theta + $\Theta(V)$
    end if
    if $V$ is the left child node of its parent then
        Phi ← hash($\Phi'(V)$ $\|$ $\Theta(V)$ $\|$ Phi $\|$ Theta $\|$ PTheta)
    else
        Phi ← hash(Phi $\|$ Theta $\|$ $\Phi'(V)$ $\|$ $\Theta(V)$ $\|$ PTheta)
    end if
    Theta ← PTheta
end for
if Phi = $\Phi'(R)$ and Theta = $c_{ij}$ then
    Correct contribution of $\langle a_{ik}, b_{kj} \rangle$ is verified
else
    Cheating behavior detected
end if
```

Security analysis

In order to prove that our approach is secure, we provide the security proof for each stage in MapReduce, i.e., map and reduce.

Theorem 1

(Security of element pairs multiplication - map phase). Assume that there exists mapper M who announces the faulty output set O'_M *instead of expected set* O_M *as its computation results. If the verifier chooses sample* $S \in (O_M - O'_M)$*, the fault is detected.*

Proof

According to the assumptions, consider that the verifier picks sample $s \in (O_M - O'_M)$*. Moreover, assume that reducer r is associated with the sample through the shuffler function. The verifier, sends s to the reducer and receives* λ_s *as the related proof path. In order to ensure the sample's contribution in computations, the verifier needs to check*

$$\Lambda(s, \lambda_s) \overset{?}{=} \Phi'(R_r)$$

where Λ is defined in Eq. 5 and $\Phi'(R_r)$ is the hash value of the root node in the joined tree of reducer r (i.e., its commitment value) which its generation process consists of one-way collision resistant hash computations. Since reducer r is not aware of s before generating the commitment value $\Phi'(R_r)$, s is not contributed in its generation. The only way for covering up the fault is to generate proof path λ'_s that

$$\Lambda(s, \lambda'_s) = \Lambda(I'_r) = \Lambda(I_r) = \Phi'(R_r)$$

where I_r and I'_r are the expected and faulty input set for reducer r, respectively.

Because of H characteristics, generating the same hash value as $\Phi'(R_r)$, with different set of inputs such that $\Lambda(I_r) = \Lambda(I'_r)$ is computationally infeasible. In addition, since map tasks do not overlap with each other, no one except mapper m would produce mapping result set equal to $O_m - O'_m$. Therefore, the cheat is detected upon the selection of the proper sample s. □

According to Theorem 1, since submitting the commitment occurs before the revealing of the samples, reducers cannot cover up the possible faults made by the mappers. Likewise, we can use the same proof when a portion of data is missing instead of being corrupted. Therefore, the wisest decision for mappers is to correctly compute the intermediate products.

Theorem 2

(Security of intermediate product summation - reduce phase). Assume that semi-honest reducer r receives the correct input set I_r *for its reduce task; however, instead of expected output set* O_r*, it announces the faulty result* O'_r*, which is the result of cheating on computations of input portion* $i_r \subset I_r$*. If the verifier chooses sample* $s \in i_r$*, the cheat is detected.*

Proof

Consider that the verifier picks sample $s \in i_r$, sends it to the reducer r, and receives the proving path λ_i, which then is used to check

$$\Lambda(s, \lambda_i) \overset{?}{=} \Phi'(R_r)$$

If the input portion i_r is not correctly contributed in the computations, there is at least one inner node V in the reducer r's joined tree that

$$\Theta(V) \neq \Theta(V_{left}) + \Theta(V_{right}) \tag{12}$$

Since the miscalculation is originated from input set i_r and we have $s \in i_r$, then Eq. 12's condition would be applied to at least one of the constructed nodes in the verifier. Therefore, for the same reason mentioned in Theorem 1, the commitment value and the hash value of the constructed root node would not be equal.

It is worth mentioning that there is no computationally feasible way for the reducer to hide the cheats it made. The reason is the construction method we use for the inner nodes (Eq. 11). The existence of Θ values of children nodes in constructing each node guarantees that reducer is aware of those values. Therefore, it cannot change them in the future, because it makes the submitted commitment value invalid, and thus, reveals the cheat with an appropriate sample. The only way to cover up the cheats is to make an equal Φ' value with different inputs for the faulty node. As this operation means reversing a one-way hash function, it is assumed to be computationally infeasible. □

Now, we can define the best place to cheat by semi-honest reducers.

Lemma 2

For a strategic semi-honest reducer, the best places to cheat are in the nodes at level $H - 1$ of the computation trees, where H is the height of the tree.

Proof

If a cheating is committed at a node placed closer to the root, the node will be ancestor of more leaves; that means the cheating can be exposed with more samples. Assume that in Fig. 7, marked nodes A and E are the two candidate places for a reducer to cheat. Cheating in node A is revealed using an arbitrary sample among eight nodes (i.e., F to M). However, cheating in node E is revealed only by using N or O as the sample. □

Detection ratio analysis

To reach the overall detection probability equation, first, we need to define some notations described in Table 1. Considering Lemma 2, the worst case scenario in calculating detection ratio is when reducers act as semi-honest strategists.

Theorem 3

(Cheat detection probability). In a MapReduce job with N input records for reducers (mappers' outputs) and assuming reducers as semi-honest strategists, cheat detection probability D is equal to

$$D = 1 - \left(C(1-S)^2 + (1-C)(1-SC)^2\right)^{N_p} \tag{13}$$

and the average undetected error number E is

$$\begin{aligned} E =& N_p C(1-S)\left(C(1-S)^2 + (1-C)(1-SC)^2\right)^{N_p - 1}. \\ & \left((1-S)(1+2C) + 2(1-SC)(1-C)(1-SC)^2\right) \end{aligned} \tag{14}$$

Proof

We name the computation tree nodes at level $H-1$, as parent nodes, in their computation trees. Meanwhile, each parent node has two leaves, which we call as its child nodes. We look to compute the probability that no cheat is detected during the verification phase, denoted as D', whether it happened or not. It happens if the following two conditions hold for each parent node:

(a) The reducer cheats in a parent node, yet its children are not selected as samples. In this case, the probability is

$$p = C(1-S)^2$$

(b) The reducer does not cheat in a parent node. However, its child nodes may be faulty since those are mappers' output and mappers can cheat in their outputs as well. Therefore, the probability is equal to

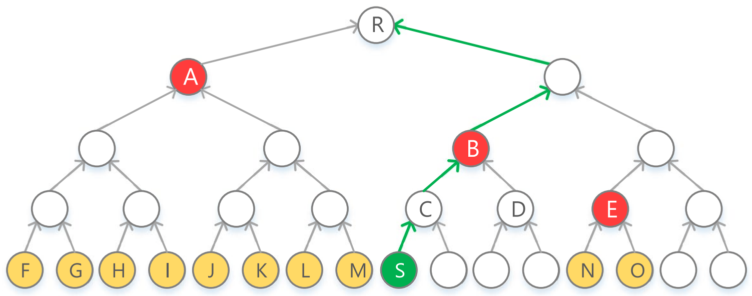


Fig. 7 Example of different places to cheat in a reduce task

Table 1 Notations used in the evaluation and analysis discussions

Notation	Description
H	The height of computation trees in reducers
N	The total number of input records for all reduce jobs.
N_p	The total number of nodes in level $H-1$of each joined tree (=N/2).
S	The ratio of randomly selecting reduce input records
C	The cheating probability during the map and reduce phase
T_s	Total number of samples used for the verification process

$$\begin{aligned} q &= (1-C)\sum_{j=0}^{2} P(j,2) \\ &= (1-C)\sum_{j=0}^{2}\binom{2}{j}(C(1-S))^{j}(1-C)^{2-j} \\ &= (1-C)(1-S.C)^2 \end{aligned}$$

Where $P(j, 2)$ is the number of 2-permutations of j objects. Reminding that $N_p = N/2$ nodes are placed in $H-1$ level, we have:

$$\begin{aligned} D' &= \sum_{i=0}^{N_p} P(i,N_p) \\ &= \sum_{i=0}^{N_p}\binom{N_p}{i}p^i q^{N_p-i} \\ &= \sum_{i=0}^{N_p}\binom{N_p}{i}\left(C(1-S)^2\right)^i\left((1-C)(1-S.C)^2\right)^{N_p-i} \end{aligned}$$

Using binomial theorem, we have:

$$D' = \left(C(1-S)^2 + (1-C)(1-S.C)^2\right)^{N_p}$$

Therefore, cheat detection probability D is

$$\begin{aligned} D &= 1 - D' \\ &= 1 - \left(C(1-S)^2 + (1-C)(1-S.C)^2\right)^{N_p} \end{aligned}$$

Now, we want to find the average number of errors (mathematic expectation) that is not detected in the verification process. To do so, we examine cases that were assumed in the beginning.

If case a) happens, it means that the parent node is faulty. Also, its children can be faulty as well since they are not samples, and therefore, the cheats will not be detected. In this case, The expectation for the number of faulty children is $2C$. Therefore, if case a) happens, $(1+2C)i$ cheats are not detected. Moreover, each time case b) happens, in average, mappers can inject $2C(1-S)(1-S.C)$ (mathematic expectation of probability q) faulty computations into the reducer's inputs without being noticed by the verifier. Therefore, If case a) and b) happens i and $N_p - i$ times, respectively, $3i + (N_p - i)2C(1-S)(1-S.C)$ faulty nodes can be existed in a computation tree without being detected. Hence, average number of undetected errors, denoted as E, can be computed as:

$$E = \sum_{i=0}^{N_p} \left(i(1+2C) + \left(N_p - i\right)2C(1-S)(1-S.C)\right)\binom{N_p}{i}. \\ \left(C(1-S)^2\right)^i \left((1-C)(1-S.C)^2\right)^{N_p - i}$$

All the multipliers except i can come out of the summation. Therefore, the equation is simplified to (we use p and q from cases a and b to save space):

$$E = (1+2C-2C(1-S)(1-S.C)) \sum_{i=0}^{N_p} i\binom{N_p}{i} p^i q^{N_p - i} \\ + N_p 2C(1-S)(1-S.C) \sum_{i=0}^{N_p} \binom{N_p}{i} p^i q^{N_p - i}$$

On the other hand, we have:

$$\sum_{i=0}^{N_p} i\binom{N_p}{i} p^i q^{N_p - i} = N_p p(p+q)^{N_p - 1}$$

Thus, after simplifications and factoring operations, we end up with:

$$E = N_p C(1-S)\left(C(1-S)^2 + (1-C)(1-S.C)^2\right)^{N_p - 1}. \\ \left((1-S)(1+2C) + 2(1-S.C)(1-C)(1-S.C)^2\right)$$

□

Figure 8 depicts D and E in case of different cheat rate values using single values for N_p and S. In case of detection rate, our approach and MtMR [33] are the same even though MtMR need to use double number of samples to achieve the same detection rate (pre-reduce and reduce phase). However, the results show that the average number of undetected errors in our approach is less than half of the MtMR's. Since pre-reduce and reduce phases in MtMR are two distinct phases, the average number of errors in each phase are aggregated together to be comparable with our approach.

Complexity analysis

In this section, the proposed approach overhead is analyzed in terms of time and space complexity. The results of the analysis are listed in Table 2. The notations used in this table are derived from Table 1.

In the initialization phase, reducers built the tree structures on N input records (i.e., intermediate products). Therefore, $2N$ nodes are built in total, which makes the space complexity for this phase equal to $O(N)$. Moreover, each reducer sends its commitment value to the verifier which results in the space complexity of $O(1)$. During the tree construction, Φ' and Θ values are calculated for each node. Thus, $4N$ calculations are required, which results in the time complexity of $O(N)$.

In the sampling phase, the verifier prepares T_s input records and sends them to the reducers. Therefore, the time and space complexities for the calculations and transmission are $O(T_s)$.

In the proof construction phase, the reducers generate proving paths for the given samples. Each generation process needs looping over $\log N$ nodes in the corresponding tree. Totally, the time complexity of computing proving paths is $O(T_s \log N)$. The space complexity of the transmission is also $O(T_s \log N)$.

Finally, in the proof verification phase, T_s proving paths are used to partially verify the computations. Each proving path needs $2 \log N$ calculations (for computing Θ and Φ' values for each node) to produce the expected commitment value. Therefore, the time complexity is $O(T_s \log N)$.

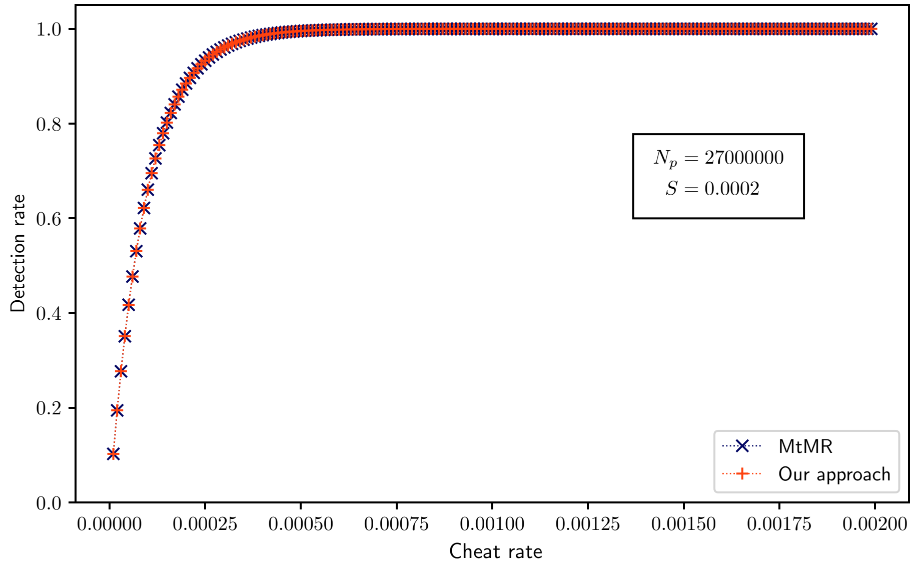
Results and discussion

Since we were curious to measure the expected performance improvements, we implemented our approach on a custom MapReduce system using Python as the main programming language. Experiments are conducted on Intel Core i7-10510U and 16 GB DDR3 SDRAM. To gain the highest computation power during experiments, we enforced the serial task processing, meaning that only one task is get processed at any moment. For running the MapReduce jobs in the system, we deployed 25 mappers, 10 reducers, and a verifier. Input matrices are generated randomly for each experiment. Moreover, we executed each test scenario 20 times to increase the accuracy.

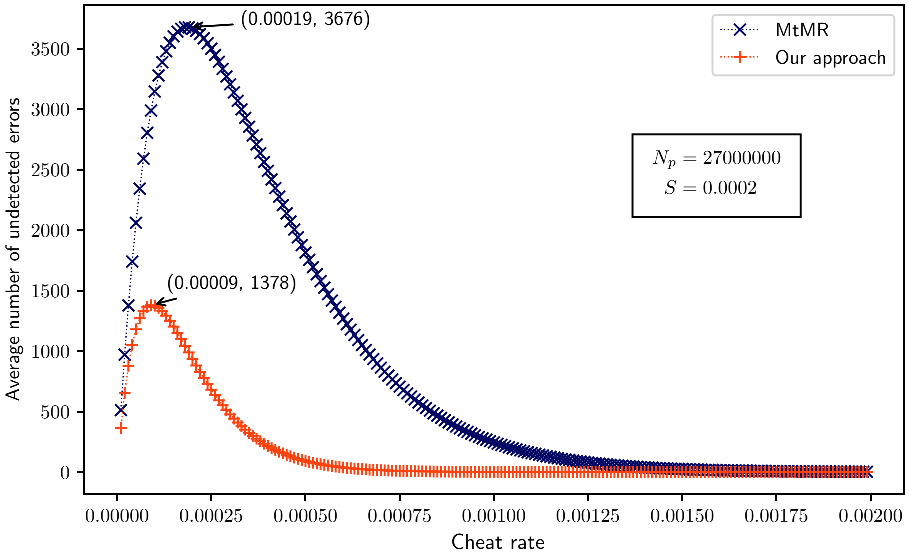
We evaluated our approach in three parts: *Setup*, where the required structures are built in reducers' side, *Proof Construction*, where the reducers generate the proofs required for given verification samples, and *Proof Verification*, where the generated proofs are verified. In each phase, we compare our results with the ones reported for the state-of-the-art technique for this purpose, i.e. MtMR [33]. In MtMR, the verification process is performed in two separate phases (e.g., pre-reduce and reduce). In order to reach a fair comparison, we represent the related performance results in MtMR as the aggregation.

The verification process for each sample is independent of the others. In other words, variation in the sampling ratio has a linear effect on the result. Therefore, we fixed the sampling ratio to 1% of intermediate products in all tests. Also, we realized that the most important parameter is the size of input matrices. Thus, we used matrices in the range of 10x10 to 300x300 as the input. Finally, since we are measuring the worst case performance, the cheating probability is not considered in workers.

Figure 9 shows the results in setup phase; by growing the input size, the required setup time and storage space grow as well. The setup time for our approach and MtMR is roughly the same; however, as the size of the inputs grows, the required storage space in our approach becomes slightly bigger than MtMR. That makes sense, since we record extra information in each Merkle tree node.



a Cheat detection probability



b Average number of undetected errors

Fig. 8 Comparison of our approach with MtMR [33] in terms of cheat detection probability and average number of undetected errors

Table 2 Time and space complexity analysis for the proposed approach

Stage	Storage	Public cloud computations	Private cloud computations	Data transmissions
Initialization	$O(N)$	$O(N)$	–	$O(1)$
Sampling	$O(T_s)$	–	$O(T_s)$	$O(T_s)$
Proof construction	–	$O(T_s \log N)$	–	$O(T_s \log N)$
Proof verification	–	–	$O(T_s \log N)$	–

For the proof construction phase, we measured the time and space that a reducer needs to construct the proof for given samples. The results are shown in Fig. 10. Here we can see tangible differences between the two approaches. In terms of proof construction time, our approach is much faster than MtMR since we provide only $\log_N$ node information per sample, where N is the number of reduce intermediate products for each reducer; however, MtMR needs to prepare the full input records (i.e., N) per each sample. For the same reason, the size of the proof increases with the number of given samples.

The evaluation results for the proof verification phase, where we measured the required time to verify the proofs received from the reducers, are represented in Fig. 11. In our approach, the verifier only needs to do $\log N$ hashes to rebuild the targeted reducer's commitment value. That being said, in MtMR, the verifier needs to do an extra reduce task. Hence, compared to MtMR, we can check many more samples in the same period of time.

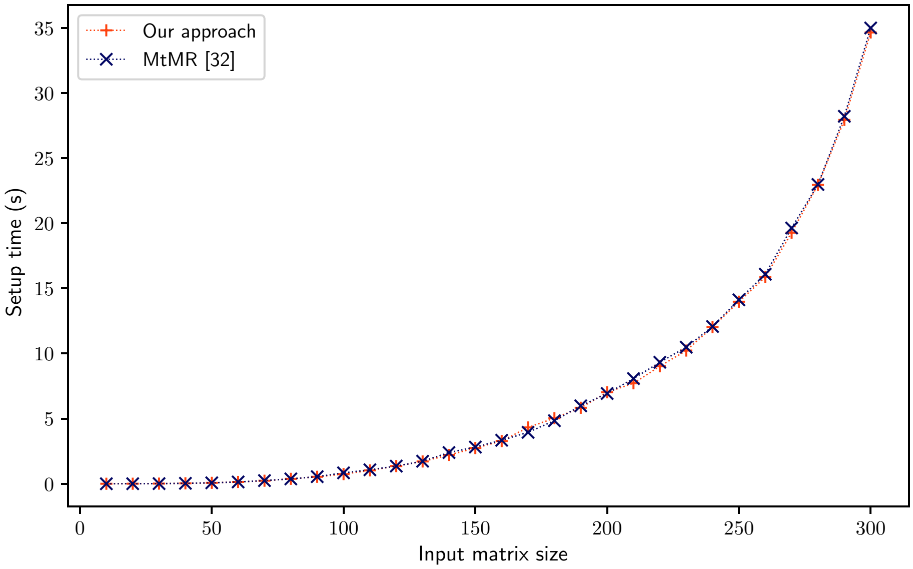
Conclusion

In this paper, we proposed an approach for efficient result verification in systems where parallel processing techniques, such as MapReduce, are utilized. There exists several solutions for this purpose that can be divided into four categories: log-based, watermark-based, hardware-based, and replication-based. Each solution has its own merits and demerits; log-based methods are vulnerable against mimicry attacks, hardware-based methods impose fundamental modification to the system, watermark-based methods suffer from false-positive detection and lack of efficient generator, and replication-based methods waste lots of computation power. Also, verification-based methods, which is grouped in the previous category, benefit from trusted nodes, which are expensive to have.

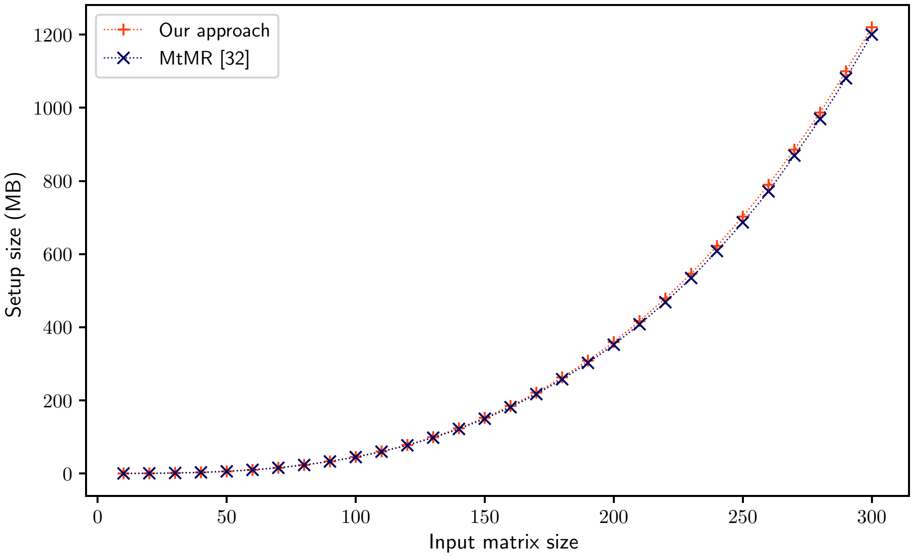
Utilizing Merkle tree structure, we aimed to mitigate verification-based disadvantages by efficiently using verifier entities. To better illustrate our work, we chose matrix multiplication, since it is one of the most popular applications in the MapReduce and its computations fits nicely into the model. The novelty of our work is derived from the fact that fine-grained computation results are contributed to produce a commitment value. In our approach, each reducer constructs a Merkle tree using map phase output data, and submits the hash value of the root node to the verifier as the commitment value. In the tree construction phase, we insert extra information in node, which helps to make the commitment value more powerful against the attempts to deceive the verification mechanism.

To reveal the cheating behavior, a proportion of input data is selected as the samples and used in the verification process. During the process, each sample is sent to the corresponding reducer to generate proving paths. per each sample and its proving path, the verifier rebuild the hash value of the root node and compare it with the commitment value in order to find probable inconsistencies. More cheating behavior performed by the workers, higher probability of detecting the errors.

Although our approach work for both types of malicious and semi-honest workers, we do not recommend using it for detecting the former type of workers, because of the overhead it imposes to the system (i.e., high sampling ratio). In fact, malicious



a Setup time

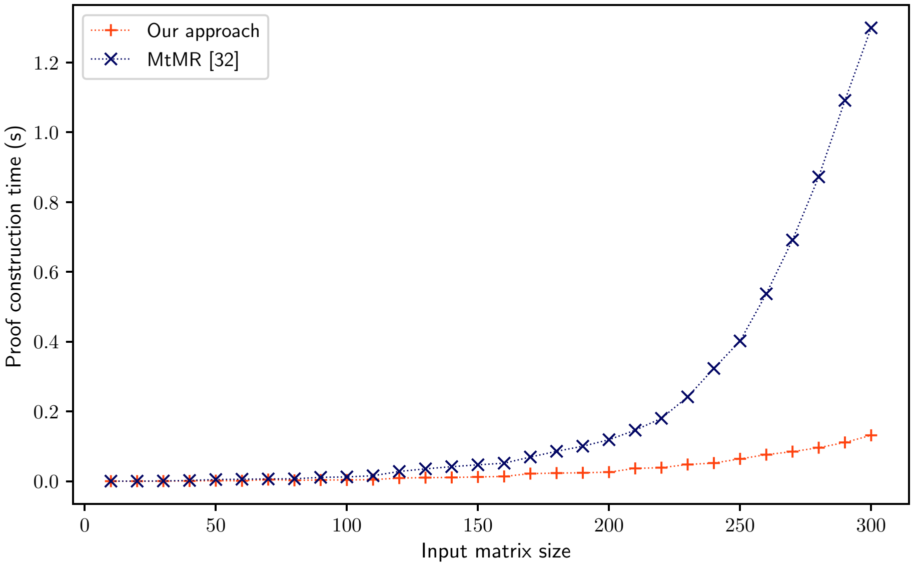


b Setup storage size

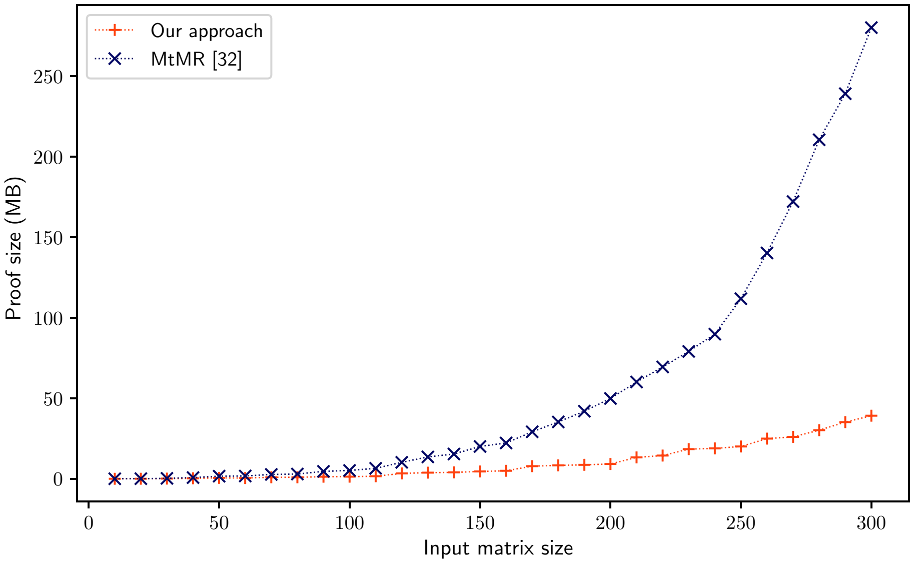
Fig. 9 Required time and space for the setup phase

workers do not have cost-effectiveness motivations. Therefore, they do not risk increasing the cheating probability while they can sabotage the system with a minimum number of miscalculations. Moreover, our approach can detect faulty results in two types of computation in matrix multiplication: multiplying element pairs, and summing intermediate products.

Evaluation results show significant improvements over the state-of-the-art technique. For example, in case of multiplying two 300x300 sized matrices, we achieved



a Proof construction time



b Generated proof size

Fig. 10 Required time and space for the proof construction phase (sampling ratio = 0.01)

86% reduction in the average generated proof size, 90% reduction in the overall proof construction time, and 98% reduction in the verification time.

it is worth mentioning that not only does the proposed approach suit the matrix multiplication application well, but also it is compatible with the applications that their computations on the inputs can be modeled using a Merkle tree.

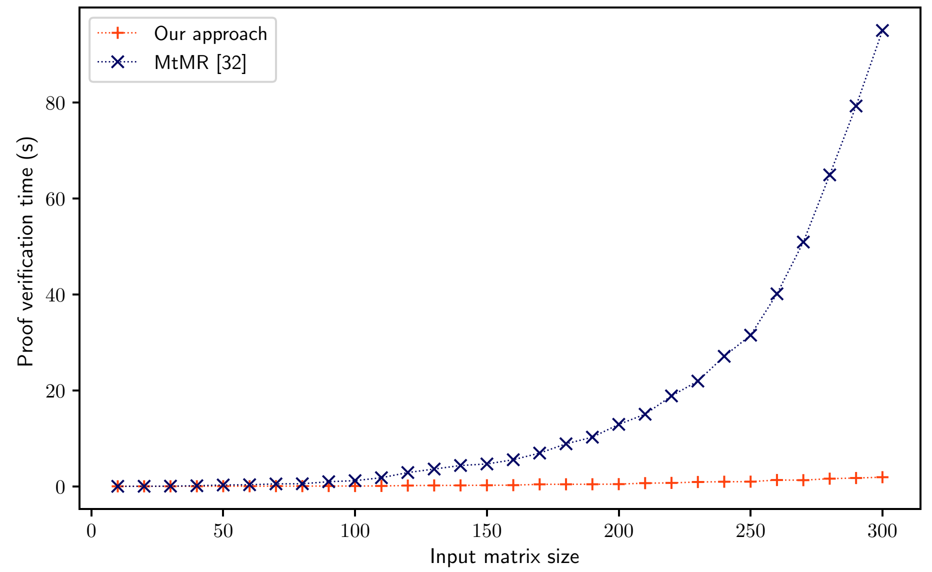


Fig. 11 Proof verification time (sampling ratio = 0.01)

Abbreviations
DFS: Distributed file system; TPM: Trusted platform modules; RTMS: Reputation-based trust management system.

Acknowledgements
We would like to thank the reviewers in the Data and Network Security Laboratory at Sharif University of Technology.

Authors' contributions
RB conceived and designed the research, performed the implementation and experimentation, and performed the evaluation and validation. MA acted as the advisor and challenged the research results in each step. SDS introduced this topic to RB and drafted the security analysis section. All authors provided feedback to RB and helped shape the research. All authors read and approved the final manuscript.

Funding
The conducted research was fully self-funded.

Availability of data and materials
Not applicable.

Ethics approval and consent to participate
Not applicable.

Consent for publication
Not applicable.

Competing interests
The authors declare that they have no competing interests.

Received: 23 June 2020 Accepted: 27 September 2020

References

1. IDC, Digital Universe: The Speed of Data Creation is Rapidly Accelerating. 2014. https://www.emc.com/collateral/analyst-reports/idc-digital-universe-2014.pdf.
2. Dean J, Ghemawat S. Mapreduce. Commun ACM. 2008;51(1):107. https://doi.org/10.1145/1327452.1327492. arXiv:10.1.1.135.4448.
3. Russkov A, Shchur L. Matrix multiplication and universal scalability of the time on the intel scalable processors, In: Journal of Physics: Conference Series, Vol. 1163, IOP Publishing, 2019;0 12079.
4. Vasudevan A, Anderson A, Gregg D. Parallel multi channel convolution using general matrix multiplication, In: 2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP), IEEE, 2017;19–24.

5. AL-Laham MM. Encryption-decryption rgb color image using matrix multiplication. Int J Comput Sci Inform Technol. 2015;7(5):109–19.
6. Highland F, Stephenson J. Fitting the problem to the paradigm: algorithm characteristics required for effective use of mapreduce. Procedia Comput Sci. 2012;12:212–7.
7. Ballard G, Demmel J, Holtz O, Lipshitz B, Schwartz O. Communication-optimal parallel algorithm for strassen's matrix multiplication, In: Proceedinbgs of the 24th ACM symposium on Parallelism in algorithms and architectures - SPAA '12, ACM Press, 2012. https://doi.org/10.1145/2312005.2312044.
8. Choi J, Walker DW, Dongarra JJ. Pumma: parallel universal matrix multiplication algorithms on distributed memory concurrent computers. Concurrency Pract Exp. 1994;6(7):543–70. https://doi.org/10.1002/cpe.4330060702.
9. Fox GC, Otto SW, Hey AJ. Matrix algorithms on a hypercube I: matrix multiplication. Parallel Comput. 1987;4(1):17–31.
10. Sun Z, Li T, Rishe N. Large-scale matrix factorization using MapReduce. Proceed IEEE Int Conf Data Min ICDM. 2010;33199(2):1242–8. https://doi.org/10.1109/ICDMW.2010.155.
11. Zheng J, Zhu R, Shen Y. Sparse matrix multiplication algorithm based on mapreduce. J Zhongkai Univ Agri Engineer. 2013;26:11.
12. Deng S, Wenhua W. Efficient matrix multiplication in hadoop. Int J Comput Sci Appl. 2016;13(1):93–104.
13. Kadhum M, Qasem M H, Sleit A, Sharieh A. Efficient MapReduce Matrix Multiplication with Optimized Mapper Set, In: R. Silhavy, R. Senkerik, Z. Kominkova Oplatkova, Z. Prokopova, P. Silhavy (Eds.), Cybernetics and Mathematics Applications in Intelligent Systems, Vol. 574, Springer International Publishing, 2017;186–196.
14. Merkle R C. A Digital Signature Based on a Conventional Encryption Function, In: C. Pomerance (Ed.), Advances in Cryptology — CRYPTO '87, Vol. 293, Springer Berlin Heidelberg, 1988. https://doi.org/10.1007/3-540-48184-2_32.
15. Du W, Jia J, Mangal Manish, Murugesan Mummoorthy, Uncheatable grid computing, In: 24th International Conference on Distributed Computing Systems, 2004. Proceedings., IEEE, 2004;4–11.
16. Liao C, Squicciarini A. Towards provenance-based anomaly detection in MapReduce, Proceedings - 2015 IEEE/ACM 15th International Symposium on Cluster. Cloud Grid Comput CCGrid. 2015;2015:647–56. https://doi.org/10.1109/CCGrid.2015.16.
17. Yoon E, Squicciarini A. Toward Detecting Compromised MapReduce Workers through Log Analysis, In: Proceedings of the IEEE/ACM 14th International Symposium on Cluster, Cloud and Grid Computing, IEEE, 2014;41–50. https://doi.org/10.1109/CCGrid.2014.120.
18. Wagner D, Soto P. Mimicry attacks on host-based intrusion detection systems, In: Proceedings of the 9th ACM Conference on Computer and Communications Security, ACM, 2002;255–264.
19. Ruan A, Martin A. TMR: Towards a Trusted MapReduce Infrastructure, In: Proceedings of the IEEE 8th World Congress on Services, IEEE. 2012;141–148. https://doi.org/10.1109/SERVICES.2012.28.
20. Bissiriou C A A, Zbakh M. Towards Secure Tag-MapReduce Framework in Cloud, In: Proceedings of the IEEE 2nd International Conference on Big Data Security on Cloud, IEEE. 2016;96–104. https://doi.org/10.1109/BigDataSecurity-HPSC-IDS.2016.78.
21. Zhang C, Chang E C, Yap R H C. Tagged-mapreduce: A general framework for secure computing with mixed-sensitivity data on hybrid clouds, In: Proceeding of the IEEE/ACM 14th International Symposium on Cluster, Cloud and Grid Computing, 2014;31–40. https://doi.org/10.1109/CCGrid.2014.96.
22. Ding Y, Wang H, Chen S, Tang X, Fu H, Shi P. PIIM: Method of Identifying Malicious Workers in the MapReduce System with an Open Environment, In: Proceedings of the IEEE 8th International Symposium on Service Oriented System Engineering, IEEE, 2014;326–331. https://doi.org/10.1109/SOSE.2014.47.
23. Huang C, Zhu S, Wu D. Towards Trusted Services: Result Verification Schemes for MapReduce, In: Proceedings of the IEEE/ACM 12th International Symposium on Cluster, Cloud and Grid Computing, IEEE, 2012;41–48. https://doi.org/10.1109/CCGrid.2012.77.
24. Kamvar S D, Schlosser M T, Garcia-Molina H. The Eigentrust algorithm for reputation management in P2P networks, In: Proceedings of the ACM 12th international conference on World Wide Web, ACM Press, New York, 2003;640. https://doi.org/10.1145/775240.775242.
25. Wei W, Du J, Yu T, Gu X. SecureMR: A Service Integrity Assurance Framework for MapReduce, In: Proceedings of the IEEE Annual Computer Security Applications Conference, IEEE, 2009;73–82. https://doi.org/10.1109/ACSAC.2009.17.
26. Ding Y, Wang H, Wei L, Chen S, Fu H, Xu X. VAWS: constructing trusted open computing system of mapReduce with verified participants E97.D. IEICE Transact Informat Syst. 2014;. https://doi.org/10.1587/transinf.E97.D.721.
27. Bendahmane A, Essaaidi M, Moussaoui A E, Younes A. Result verification mechanism for MapReduce computation integrity in cloud computing, In: Proceedings of the IEEE International Conference on Complex Systems (ICCS), IEEE, 2012;1–6. https://doi.org/10.1109/ICoCS.2012.6458583.
28. Khan S M, Hamlen K W. Hatman: Intra-cloud Trust Management for Hadoop, in: Preceedings of the IEEE 5th International Conference on Cloud Computing, IEEE, 2012;494–501. https://doi.org/10.1109/CLOUD.2012.64.
29. Samuel T A, Abdul Nizar M. Credibility-based result verification for Map-reduce, In: Proceedings of the IEEE Annual India Conference (INDICON), IEEE, 2014;1–6. https://doi.org/10.1109/INDICON.2014.7030682.
30. Wang Y, Wei J. VIAF: Verification-Based Integrity Assurance Framework for MapReduce, In: Proceedings of the IEEE 4th International Conference on Cloud Computing, IEEE, 2011;300–307. https://doi.org/10.1109/CLOUD.2011.33.
31. Wang Y, Wei J, Srivatsa M. Result Integrity Check for MapReduce Computation on Hybrid Clouds, In: Proceedings of the IEEE 6th International Conference on Cloud Computing, IEEE, 2013;847–854. https://doi.org/10.1109/CLOUD.2013.118.
32. Xiao Z, Xiao Y. Achieving accountable MapReduce in cloud computing. Futur Generat Comput Syst. 2014;30(1):1–13. https://doi.org/10.1016/j.future.2013.07.001.
33. Wang Y, Shen Y, Wang H, Cao J, Jiang X. MtMR: ensuring MapReduce computation integrity with Merkle tree-based verifications. IEEE Transact Big Data. 2016;14(8):1. https://doi.org/10.1109/TBDATA.2016.2599928.
34. Freivalds R. Fast probabilistic algorithms, In: International Symposium on Mathematical Foundations of Computer Science, Springer, 1979;57–69.

35. Thaler J. Time-optimal interactive proofs for circuit evaluation, In: Annual Cryptology Conference, Springer, 2013;71–89.
36. Kumar M, Vardhan M. Engineering Vibration, Communication and Information Processing, vol. 478. Singapore: Springer; 2019 10.1007/978-981-13-1642-5.
37. Zhang S, Li H, Dai Y, Li J, He M, Lu R. Verifiable outsourcing computation for matrix multiplication with improved efficiency and applicability. IEEE Int Things J. 2018;5(6):5076–88. https://doi.org/10.1109/JIOT.2018.2867113.
38. Bultel X, Ciucanu R, Giraud M, Lafourcade P. Secure Matrix Multiplication with MapReduce, In: Proceedings of the 12th International Conference on Availability, Reliability and Security - ARES '17, ACM Press, New York, New York, USA, 2017;1–10. https://doi.org/10.1145/3098954.3098989.
39. Zhang S, Li H, Jia K, Dai Y, Zhao L. Efficient secure outsourcing computation of matrix multiplication in cloud computing, 2016 IEEE Global Communications Conference, GLOBECOM 2016 - Proceedings https://doi.org/10.1109/GLOCOM.2016.7841783.
40. Bowers K D, Juels A, Oprea A. Hail: A high-availability and integrity layer for cloud storage, In: Proceedings of the 16th ACM conference on Computer and communications security, ACM, 2009;187–198.

Publisher's Note