Result Integrity Check for MapReduce Computation on Hybrid Clouds

Yongzhi Wang, Jinpeng Wei
Florida International University
Miami, USA
ywang032@cis.fiu.edu, weijp@cis.fiu.edu

Mudhakar Srivatsa
IBM T.J. Watson Research Center
Yorktown Heights, USA
msrivats@us.ibm.com

***Abstract*— Large scale adoption of MapReduce computations on public clouds is hindered by the lack of trust on the participating virtual machines, because misbehaving worker nodes can compromise the integrity of the computation result. In this paper, we propose a novel MapReduce framework, Cross Cloud MapReduce (CCMR), which overlays the MapReduce computation on top of a hybrid cloud: the master that is in control of the entire computation and guarantees result integrity runs on a private and trusted cloud, while normal workers run on a public cloud. In order to achieve high accuracy, CCMR proposes a result integrity check scheme on both the map phase and the reduce phase, which combines random task replication, random task verification, and credit accumulation; and CCMR strives to reduce the overhead by reducing cross-cloud communication. We implement our approach based on Apache Hadoop MapReduce and evaluate our implementation on Amazon EC2. Both theoretical and experimental analysis show that our approach can guarantee high result integrity in a normal cloud environment while incurring non-negligible performance overhead (e.g., when 16.7% workers are malicious, CCMR can guarantee at least 99.52% of accuracy with 33.6% of overhead when replication probability is 0.3 and the credit threshold is 50).**

Keywords*— *MapReduce, Integrity Assurance, Hybrid Cloud

I. Introduction

MapReduce [1] has become the dominant paradigm for large-scale data processing applications such as web indexing, data mining, and scientific simulation. However, MapReduce applications normally are running on a cluster of hundreds or thousands of computation nodes. Most MapReduce customers cannot afford or do not want to invest in computer clusters of such a large scale. The emergence of Cloud Computing [2][3] provides an economical alternative for getting a large-scale cluster on demand, thus MapReduce in the cloud has been embraced by the market with enthusiasm. For example, various services such as Amazon Elastic MapReduce [12] and Microsoft Daytona [13] are provided to facilitate the transition of MapReduce applications to the cloud.

However, MapReduce applications running on the cloud suffer from the integrity vulnerability problem: the malicious participant can render the overall computation result useless. While the cloud vendors can be trusted and the cloud infrastructure (i.e., the virtualization layer) can be assumed to be secure, the virtual machines and the MapReduce applications installed in the virtual machines cannot be trusted to always return correct results. For instance, [14][15] points out a security vulnerability that Amazon EC2 suffers from: some members of the EC2 community can create and upload malicious Amazon Machine Images (AMIs), which, if widely used, could flood the EC2 cloud with virtual machine instances that contain malicious applications, including MapReduce. The above threat puts a MapReduce customer in a dilemma: using public clouds has economic advantage but incurs the risk of getting wrong computation results; on the other hand, avoiding the public cloud completely (i.e., running everything "in house" or in the private cloud) can guarantee result accuracy, but there will be less economic benefit.

In this paper, we propose *Cross Cloud MapReduce* (CCMR for short) that combines the benefits of private clouds and public clouds. CCMR overlays the MapReduce framework on top of a hybrid cloud which consists of a private cloud and a public cloud. The master that is in control of the entire computation and guarantees result integrity runs on a private and trusted cloud, while normal workers run on the public cloud and are untrusted. We further introduce a special type of workers (called *verifiers*) on the private cloud to detect collusive malicious workers on the public cloud. The key rationale of our solution is to retain control and trust "at home", while delegating the more resource-intensive computations to the public cloud.

We explore the design space of result integrity checking in both phases of MapReduce: the map phase and the reduce phase. We extend the capability of the master to propose the result integrity check mechanism, which combines several integrity assurance techniques (replication [8][10], verification [8][9], and credit-based trust management [8]). Due to the different properties of map and reduce phases, CCMR uses the result integrity check on different objects. In the map phase, integrity check is performed on map tasks. In the reduce phase, CCMR factors each reduce task into multiple sub-tasks and applies the integrity check on sub-tasks.

We make the following contributions in this paper: 1) we propose a novel cross-cloud MapReduce architecture that combines the benefits of private clouds and public clouds; 2) we propose a result integrity check mechanism that combines several integrity assurance technique to enhance the result integrity of MapReduce on both the map and reduce phases; 3) we analyze the security of CCMR and quantitatively measure its accuracy and overhead; 4) we implement CCMR based on Apache Hadoop MapReduce [7] and run a series of experiments over the commercial public cloud (Amazon EC2 [2]). We show that CCMR is an efficient framework to guarantee high computation integrity.

The rest of this paper is organized as follows. Section II describes the system assumptions and attacker model. Section

III presents the system design of CCMR. Section IV makes the theoretical analysis in terms of security, accuracy, and overhead. Section V describes and analyzes the experiment result. Section VI discusses related work, and Section VII concludes the paper and points out future work.

II. System Assumptions and Attacker Model

A. System Assumptions

In CCMR, we assume the private cloud is trusted since it is deployed within the user's organization. Therefore, the master and the verifiers are trusted, since they are deployed on the private cloud. On the public cloud, we assume the infrastructure provided by the cloud provider, such as the virtualized hardware and network, is trusted. However, we assume the virtual image used by the customer is untrusted. That makes the MapReduce entities running on the public cloud untrusted. Since our paper only focuses on the MapReduce computing, we assume Distributed File System (DFS) of MapReduce is trusted. For example, the integrity of DFS can be guaranteed by the techniques proposed in [1][6].

In CCMR, the master requires each worker who runs a task/sub-task submit the hash value of its computation result to the master. We further assume the hash value is consistent with the actual task output. Such an assumption can be realized by applying the commitment-based protocol proposed in [4] (i.e., check the correctness of the hash value by the later task that takes the current task output as its input). Finally, we assume that the tasks running on each worker are deterministic. We use this assumption to guarantee that multiple executions of the same task/sub-task by honest workers return the same result.

B. Attacker Model

We model the attacker as an intelligent adversary that controls the malicious nodes on the public cloud. It receives and correlates information collected by the malicious nodes and coordinates them to cheat at the right time in order to introduce as many errors as possible to the final result without detection. For example, if the master replicates the same task on two malicious workers, the adversary can instruct them to return the same erroneous results (i.e., to collude) so that simply comparing the results cannot detect the error. We call such malicious workers *collusive workers*.

III. System Design

A. System Overview and Architecture

CCMR overlays MapReduce on a hybrid cloud which consists of one private cloud and one public cloud, as shown in Fig. 1. The master node and a small number of slave nodes (called *verifiers*) are deployed on the trusted private cloud within the customer's organization. Other slave nodes (called *workers*) and Distributed File System (DFS) are deployed on the public cloud. According to our assumption, the verifiers, the master and the DFS are trusted, yet the workers are not.

In both the map and the reduce phases, CCMR defines three types of tasks: the *original task*, the *replication task*, and the *verification task*. Both the original and replication task are executed by the workers on the public cloud. While the verification task is executed by the verifier on the private cloud. The replication task repeats the original task's work to validate the original task result. While the verification task repeats the original task's work to verify the result returned by the replication task, since the replication task is not trusted. In the map phase, the replication and verification task completely repeat the original map task's work. While in the reduce phase, the replication and verification reduce task repeats only a portion of original reduce task. Each repeated portion in a reduce task is called a *sub-task*.

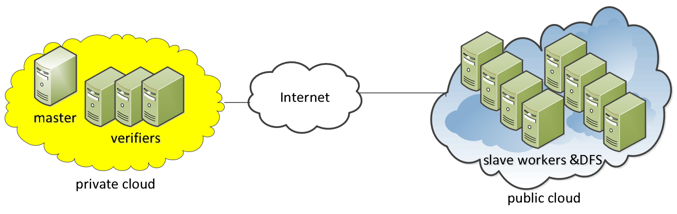

Fig. 1. Architecture of CCMR.

In both Map and Reduce phase, CCMR applies a *two-layer check* on each returned original task/sub-task result: replication and verification. In order to achieve high accuracy, credit based trust management is also applied on each worker. The master only accepts a worker's task/sub-task results when it achieves certain credit threshold.

The task/sub-task execution in CCMR differs from the original MapReduce in both the map and reduce phase. Rather than passively waiting for the worker to ask for task/sub-task, the master of CCMR randomly selects the worker to execute a certain task/sub-task. When a task/sub-task is finished, CCMR requires the worker to return the result. In order to reduce the communication cost, the worker only returns the hash value of the result. The actual result of replication and verification task/sub-tasks will not be stored back to the DFS.

Given the different characteristic of map and reduce phases, we propose different integrity check solutions.

B. Map Phase Integrity Check

CCMR applies two-layer check on each returned original map task result. In the first layer, CCMR creates a replication task and assigns the task to another worker. When the worker returns the replication task result, CCMR compares the original and replication task results. If the results are not consistent, at least one of the workers are cheating, so CCMR will create a verification task and assign it to a verifier to detect the malicious mapper(s). If the original and replication task results are consistent, CCMR launches the second layer check. In the second layer check, CCMR creates a verification task and assigns it to a verifier to verify the consistent results. If the consistent results are different from the verification task result, the two mappers providing the results are all determined as malicious. The reason for the second layer check is to detect collusive workers. To reduce overhead, CCMR creates replication and verification tasks with certain probabilistically. Each original map task is replicated with *replication probability*, and each consistent result is verified with *verification probability*.

Since replication or verification is not performed for every task, there is a possibility that some bad results can evade the detection of the two-layer check. In order to overcome this drawback, CCMR performs the credit based trust management to boost the job result accuracy. Initially, the master sets the credit for each mapper as zero, and maintains a history cache for each mapper to record the id and result (hash value) of original map tasks the mapper has executed. When a mapper passes one two-layer check, the master increments the credit for this mapper and updates the mapper's history cache. The actual task result is buffered in the mapper's local storage before it becomes trusted. When a mapper's credit achieves certain threshold (called *credit threshold*), the mapper becomes trusted temporarily. The task results buffered in its local storage are accepted by the master in a batch. At the same time, the credit and the history cache of this mapper are reset and this mapper becomes untrusted again. The mapper has to earn credit again in order to submit the next batch results to the master. If a mapper fails any two-layer check before it achieves credit threshold, it is determined to be malicious and is added to a black list. The actual results buffered in its local storage are discarded, and the tasks cached in its history cache will be re-executed.

Fig. 2 presents the control flow of CCMR. In the figure, W1 and W2 are two slave workers randomly chosen from the public cloud. The "Arbitrate/Verify task" step is completed by the verifier on the private cloud, and the remaining components in the figure are all performed on the master. Notice that in the figure, instead of assigning the replication and original task simultaneously, the "replication" decision (step 3) is made after W1 returns the original task result R2 (step 2). We call such a technique *hold-and-test*, and it makes it harder for malicious workers to collude because the adversary cannot predict whether the replication task will be assigned to another collusive worker. A detailed discussion of the benefit of *hold-and-test* is deferred to section IV.A.

If the total number of original map tasks in a job is less than the credit threshold, CCMR directly assigns all tasks to verifiers since the computing workload is not significant. Therefore, the accuracy in this case is still guaranteed. If the total number of original map tasks is large enough, a higher credit threshold would guarantee a higher accuracy, as our theoretic analysis shows (Section IV.A).

C. *Reduce Phase Integrity Check*

In the reduce phase, the approach presented in section III.B can be directly applied if the number of reduce tasks is big enough (i.e., bigger than the credit threshold). However, in some applications, the reduce task number is smaller than the credit threshold, even though the computing workload for each reduce task is significant. For example, the word count application in section V.B contains only one original reduce task. However, this single task processes 2.7M of records (1.07GB of data) in the input and generates 598K of records in the output, and it takes 262 seconds. In this case, directly verifying the entire reduce task is expensive in terms of computation and communication cost. Therefore, we break down an original reduce task into many sub-tasks and apply two-layer check on each sub-tasks to achieve high accuracy.

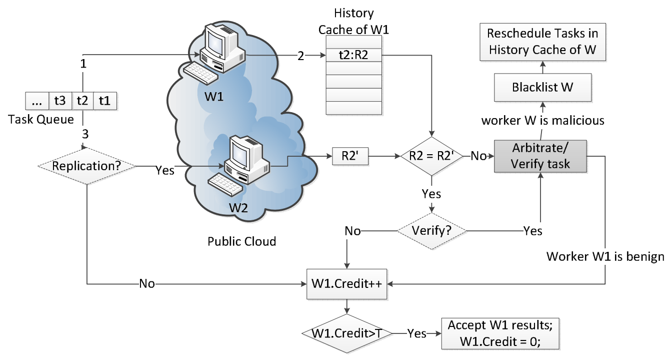

Fig. 2. Control Flow of CCMR

In MapReduce, each map task stores its output locally in the form of <key, value> tuples, which are sorted by key. Each reduce task has to fetch tuples from each map task output and aggregates tuples that contain the same key. If we can precisely fetch a portion of map output that only contains a certain key, we can reproduce the portion of the original reduce task that is only related to that specific key. By applying two-layer check to the result of each portion (called a *sub-task*), we can guarantee high accuracy of the original reduce task.

Our reduce phase integrity check uses the same high-level ideas as the map phase. Each original sub-task returns its result to the master in the form of a hash value (we call each returned sub-task result as a *report*). The master applies first-layer (replication) and second-layer (verification) check on each report with *replication probability* and *verification probability*, respectively. The replication sub-task is generated after the result of original sub-task is returned to the master (*hold-and-test*). In addition, an original reduce task result is accepted by the master only when all its sub-tasks pass two-layer check, which is essentially a credit-based trust management. The credit threshold is the number of sub-tasks in the original reduce task. In the case that the number of sub-tasks in a reduce task is smaller than the credit threshold, we simply assign the verifier to verify the entire original reduce task.

Due to the special characteristic of reduce phase, in order to make the above idea practical, we need to overcome three challenges. 1) Creating a sub-task for each key would incur significant overhead because in many cases, a reduce task can generate many keys (e.g., 598K keys in the word count application in Section V.B). 2) So far our two-layer check only checks the sub-task reports submitted by the reducer. If a malicious reducer cheats on some sub-tasks but does not send these reports to the master, the master would have no way to detect the error. 3) The replication and verification sub-tasks should efficiently locate the portion of map task output with the key they are interested in.

We address the first challenge by requiring each report to cover a range of (instead of only one) consecutive keys. With this improvement, the number of sub-tasks can be reduced.

For the second challenge, CCMR requires that consecutive reports must overlap in one key. Since the reduce task result is sorted by the key, this requirement ensures that no key in the output is skipped in the reports. In case that the master does not know the range of keys in the original reduce task output, the master can insert dummy records in the job input data, which will generate reduce result tuples with predictable smallest and largest keys. For example, when the type of the key is integer, the master can insert records with keys Integer.MIN_VALUE and Integer.MAX_VALUE. When the reduce task is finished, the output can be sanitized to remove the dummy records.

For the third challenge, each map task in CCMR builds a *key table* to facilitate record look up in the map task output, as shown in Fig. 3. Each map task stores the map output file locally, which consists of key-value pairs sorted by key. Notice that multiple key-value pairs can have the same key (e.g., Key 3), and for simplicity we call such key-value pairs a *block*. Each record in the key table corresponds to a unique key and it stores information about the block that contains this key, including the position and length of the key, as well as the length of the block. The length of each key table record is fixed, while the length of key and values in the map output file varies. Hence, the access of each key on the map output file needs to go through the key table. Since the key table is sorted by key, CCMR employs binary search to find the biggest and smallest key contained in the report key range (e.g., key 2 to key 9). When the keys (key 3 through key 8) are found, the key position and the record length directs the reducer to fetch the portion of map output containing the keys of interest.Since each key table is generated by a mapper, a malicious mapper could manipulate its content to fool CCMR. As a defense, CCMR requires each map task to submit the hash value of its key table to the master along with that of task result, and the consistency of the hash value with the key table can be achieved by the commitment-based protocol [4].

Fig. 4 depicts how CCMR works in the reduce phase of *word count* application. The word count application calculates the frequency of each word appeared in a collection of text files. For simplicity, our example only has two map tasks (map 0 and map 1) and one original reduce task (reduce 0). As Fig. 4 shows, each map task creates a key table (Step 1). When original reduce task (reduce 0) starts to output (step 3), sub-task reports (e.g., report 1 and 2) are sent to the master sequentially. The report format is <start key, end key, hash value of the output records covered in the key range> (Step 4). Since consecutive reports must overlap in one key (According to the solution of challenge 2), the key "Driver" appears in both report 1 and report 2.When the master receives report 1(Step 5), it launches the first layer check (with replication probability)

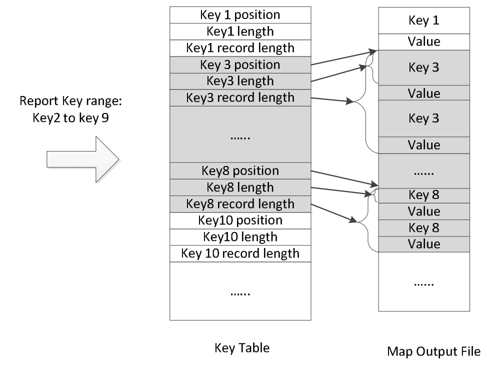

Fig. 3. Locating Map Output Between Key 2 and Key 9 using Key Table

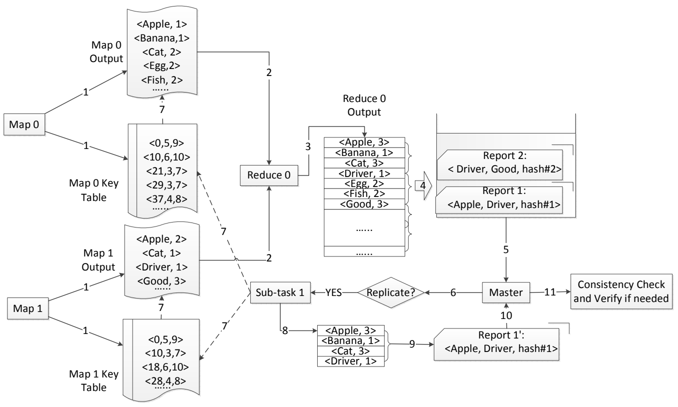

Fig. 4. Control Flow of CCMR on Reduce Phase

on report 1 by initiating a replication sub-task. The replication sub-task fetches input with a key range of (Apple, Driver) from each map task (map 0 and map 1) with the help of key tables (Step 7). When it finishes reducing (Step 8), the replication sub-task sends a report to the master (Step 9), and the master compares the report with the original sub-task report (Step 10). If they are consistent, a second layer check is performed to verify the consistent result (Step 11). The verification sub-task also resorts to the key tables to fetch the input.

IV. SYSTEM ANALYSIS

A. *Quantitive Analysis*

We give quantitative analysis on CCMR. Since the major differences of map and reduce phase in CCMR is only the object used to perform two-layer check (task in map phase and sub-task in reduce phase), we can use the same model to analyze both phases. In our analysis, we denote the *malicious worker fraction* on the public cloud as *m*. We assume that the adversary controls all malicious workers. In other words, each malicious worker is collusive, and there exists only one collusive group.

Adversary Strategy: Assuming the goal of the adversary is to inject as many errors as possible and yet not reveal the malicious worker, we analyze the strategy of the adversary under CCMR as follows. Suppose a task/sub-task is assigned to a malicious worker.

Case 1. If the adversary has not seen a similar task/sub-task (i.e., the one with the same input) before, it has to make a decision on whether to cheat, and remembers the decision, the current task/sub-task and the returned result. Due to the existence of hold-and-test (section III.B), the adversary is not allowed to defer the decision to the time that it see the replica of the current task/sub-task. If the decision is not to cheat, the worker is obviously safe (i.e., not to be caught). If the decision is to cheat, the malicious worker survives the first-layer check only when either the current task/sub-task is not replicated, or the replica of the current task/sub-task is assigned to another malicious worker.

Case 2. If the adversary has seen a similar task/sub-task before, it is assured that the current task/sub-task is a replication task/sub-task. It can simply ask the worker to take the same action for the current task/sub-task as the one it has seen

TABLE I. CCMR SYSTEM SETTING PARAMETERS

Notation	Name	Comment
m	malicious worker fraction	The fraction of malicious workers on the public cloud.
c	cheat probability	The probability that the adversary decides to cheat in Case 1 of the Adversary Strategy.
r	replication probability	The probability that an original task/sub-task is replicated.
v	verification probability	The probability that consistent task/sub-task results are verified.
T	credit threshold	The credit a mapper/reducer has to achieve in order for its batch of results to be accepted by the master.
L	survival length	The expected number of batches a malicious worker can submit to the master before it is detected.
E	batch error number	The expected number of incorrect task/sub-task results in one accepted batch.
e	batch error rate	The fraction of incorrect task/sub-task results in one batch of results.
J	job error rate	The ratio of incorrect task/sub-task results number to the total task/sub-task results number in one job.
O	overhead	The expected number of extra executions for each task/sub-task performed on the public cloud.
V	verifier overhead	The expected number of extra executions for each task/sub-task performed on the private cloud.

before. In this case, it is guaranteed that the malicious worker will survive the first-layer check.

Since in case 2 the adversary just follows its decision made previously in case 1, the risk of revealing a malicious worker is essentially determined by the adversary's decision in case 1. Because the master controls task assignment and replication in a randomized manner, the adversary in case 1 cannot predict whether cheating at the current task is safe or not. On the other hand, since the master constantly applies the two-layer check on tasks/sub-tasks in a randomized manner, the adversary cannot tell whether cheating at the current task/sub-task has a smaller chance of detection than cheating at other tasks/sub-tasks. Therefore, the only thing the adversary can do in case 1 is to make a random guess/predict in terms of whether cheat can be detected. We model the adversary's decision making behavior in case 1 as a random variable, *cheat probability c*. Note that adversaries who cheat rarely (e.g., only once out of hundreds of tasks) can still fit in our model because we can set *c* as a small value close to 0.

We define several metrics to measure the accuracy and overhead of CCMR in both map and reduce phases, and summarize the model in parameters in TABLE I. We provide the analysis result in Theorem 1.

Theorem 1: Assuming that the assignment of tasks/sub-tasks is uniformly distributed across all workers on the public cloud, and the detected malicious workers are *not* added to the black list, the probability for a malicious mapper/reducer to survive after executing *n* original tasks/sub-tasks is

$$S_n = \sum_{i=0}^{n}\sum_{j=0}^{n-i}\binom{n}{i}\binom{n-i}{j}(1-c)^i(c(1-r))^j(crm(1-v))^{n-i-j} \qquad (1)$$

The *survival length of a malicious mapper/reducer* is

$$L = S_T/(1-S_T) \qquad (2)$$

The *batch error number* is

$$E = \sum_{k=0}^{T}(k\times\sum_{i=0}^{k}\binom{T}{T-k}\binom{k}{i}(1-c)^{T-k}(c(1-r))^i(crm(1-v))^{k-i}) \qquad (3)$$

The *batch error rate* is $e = E/T$ (4)

The *job error rate* is $J = me$ (5)

Let $U = 1-(1-rc-rv+rcv+rcm-rcmv)rc(1-m+mv)m\sum_{k=0}^{T-2}S_k$

The *overhead* for each task/sub-task is

$$O = (m+r-mr-m(1-r)S_{T-1} + mr(1-v)(1-c+cm)(rc-rcm+rcmv)\sum_{k=0}^{T-2}S_k)/U \qquad (6)$$

The *verifier overhead* for each task/sub-task is

$$V = ((1-m)r(cm+v-vmc)+mr(c+v-vc-cm+cmv)S_{T-1} + rcm(1-m+mv)(1+rv-rvc)\sum_{k=0}^{T-2}S_k)/U \qquad (7)$$

The derivation of S_n is the foundation of Theorem 1. S_n is the probability summation of all permutations on *n* independent events. Each event falls into one of three cases. For each original task/sub-task it executes, a malicious worker can pass the two-layer check in the following three cases: 1) the worker does not cheat; 2) the worker cheats, but the task/sub-task is not replicated; 3) the worker cheats, the task/sub-task is replicated, but the replication task/sub-task is assigned to another malicious worker and their results are not verified. The probabilities of the above cases are (1-c), c(1-r), and crm(1-v), respectively. By summing up the probabilities of different permutations, we get (1). Further details of the proof of Theorem 1 are elided to save space but are available upon request.

We present several simulations based on Theorem 1 to quantitatively analyze the accuracy and overhead of CCMR. We first simulate the job error rate under different system parameters in Fig. 5(a). The four curves show that when other parameters (c, v, r and m) are fixed, increasing credit threshold T would reduce the job error rate J, and when T is greater than 200, J is close to 0 for any parameter combinations in the figure. Moreover, when T and other parameters are fixed, J would increase if malicious worker fraction m is increased or the replication probability r is decreased. For example, when T is 50 and r is 0.5, J increases from near 0 to 0.06 when m increases from 0.5 to 1.0; when T is 50 and m is 1.0, J increases from 0.06 to 0.15 when r drops from 0.5 to 0.3.

Fig. 5(b) shows the relationship between cheat probability c and job error rate J with fixed T and v. According to the simulation, when T is 50, r is 0.5, and m is 0.5, the maximum J an adversary can achieve is less than 0.01. When m is 1.0, setting r as 0.5 can limit J to less than 0.09. The simulation also shows an interesting tradeoff between c and J: if c is too big, the malicious worker would be detected easily and thus its injected errors are rejected, resulting in a small J; if c is too small, the number of injected errors is reduced also, again resulting in a small J. Fig. 5(c) shows the relationship between c and J when T is 600, v is 0.07, and r is 0.16. With this configuration, even under the most extreme case where m is 1.0, the maximum J the adversary can achieve is less than 0.06; when m is no larger than 0.5, the maximum J is close to 0.

Fig. 5(d) shows the relationship between cheat probability c and survival length L when T is 50 and v is 0.15. We can see that L is generally very small when c is bigger than 0.02, which means that a malicious worker cannot survive CCMR checks for a very long time. Our experiment in section V.A and Fig. 6 confirms this observation. However, L increases exponentially when c decreases from 0.02 to 0, which suggests that CCMR cannot remove very low-profile malicious workers (those that rarely cheat) quickly, but since such workers inject very few errors at the same time, CCMR can still guarantee very low job error rate in that case.

Fig. 5(e) shows the tradeoff between job error rate J and overhead O, and Fig. 5(f) shows the tradeoff between job error rate J and verifier overhead V, given different credit threshold T. For each curve in the figures, the top-left most point corresponds to the setting where T is 0, and the bottom-right most point corresponds to the case where T is 600. The difference of T values between adjacent points on a curve is 50. The figures show that when T is small (e.g., 50), a higher value of r results in a lower job error rate and higher overhead and verifier overhead. When T is big enough (e.g., greater than 200), different values of r do not make much difference in job error rate. However, a smaller value of r would bring a smaller

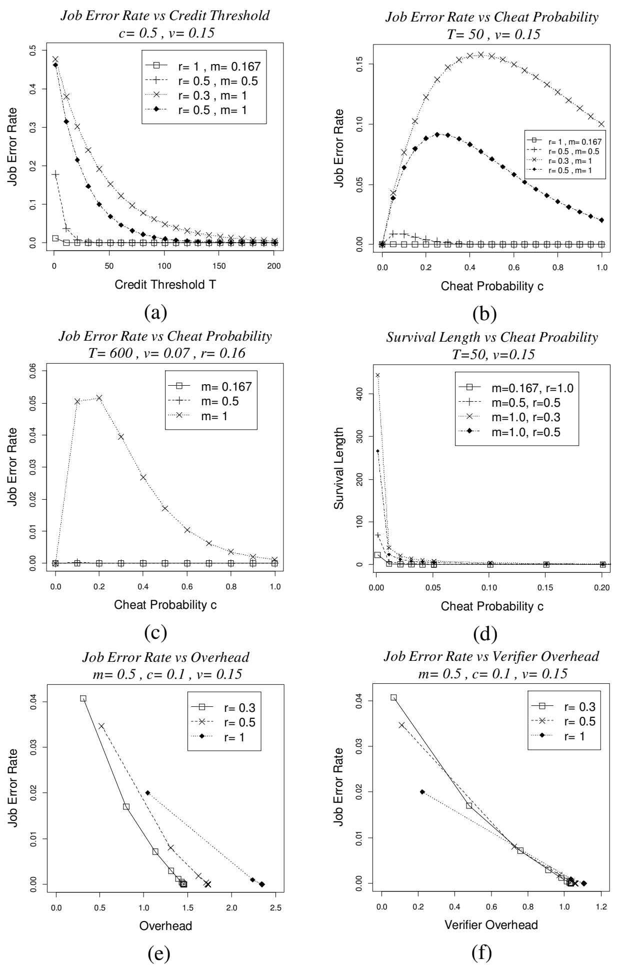

Fig. 5. Simulation of CCMR Analysis Model

overhead and verifier overhead limit. We find that on each curve, the points become denser with the increase of T and eventually concentrate to their outmost limits. This suggests that when T is big enough (e.g., bigger than 200), further increasing T would bring neither additional accuracy benefit, nor additional overhead and verifier overhead cost.

We should point out that Theorem 1 assumes that malicious worker fraction m is constant, i.e., detected malicious workers are not eliminated. However, in our real implementation, detected malicious workers are eliminated, which will cause fewer errors. As a result, task/sub-task reschedule will be reduced, and eventually the overhead and verifier overhead should be lower than the simulation result.

B. *Communication Cost Analysis*

In order to reduce the cross-cloud communication cost, we only deploy DFS nodes on the public cloud. Such a deployment not only avoids DFS data synchronization traffic across clouds but also reduces the cross-cloud communication incurred by MapReduce tasks. Since each mapper fetches input from DFS and stores task output to its local storage, the only major cross-cloud communication in the map phase happens when a verification map task needs to fetch input data from the DFS. Since each reducer fetches input from the mappers' local storage, and only the original reduce task outputs to the DFS (According to section III.A), the only major cross-cloud communication in the reduce phase happens when a verification reduce task fetches input data. Since the number of map/reduce verification tasks is usually very small compared to the number of original and replication task, such cross-cloud communication is not significant.

Other sources of cross-cloud communication includes task scheduling instructions from the master to workers and the task results (hash value) returned from workers to the master. However, network traffic caused by such communication messages is not significant due to their small sizes (e.g., a hash value of a task result contains only a few bytes).

V. EXPERIMENTAL RESULT

We implement a prototype system based on Hadoop MapReduce and deploy it across our private cloud and Amazon EC2. The experiment environment consists of the following entities: a Linux server (2.93 GHz, 8-core Intel Xeon CPU and16 GB of RAM) is deployed on a private cloud, running both the master and the verifier. Twelve Amazon EC2 micro instances are running as slave workers (Amazon Linux AMI 32-bit, 613 MB memory, Shared ECU, Low I/O performance).

We perform experiments on map and reduce phase separately to measure the job error rate, overhead, verifier overhead, and performance overhead. To compare the performance overhead, we set the baseline as a standard MapReduce cluster consisting of thirteen nodes deployed on Amazon EC2. Each node is a micro instance. Out of the 13 nodes, one is running as the master, and the other 12 nodes running as workers.

A. *Map Phase*

We measure job error rate, overhead and verifier overhead of CCMR by running a word count MapReduce job (Section III.C) in an environment with malicious MapReduce workers. We simulate such malicious workers by implementing the adversary's strategy described in section IV.A. The word count job consists of 800 map tasks and one reduce task. We fix T and v and vary other parameters with different value combinations.

The experiment result in TABLE II. indicates that in all parameter combinations, CCMR can keep a very low job error rate. Overall, the maximum job error rate is 2.25% and the minimum is 0. The changing trend of experiment result is consistent with the theoretical analysis in Section IV.A. For example, when m and c are fixed (m is 0.167 and c is 0.1), job error rate drops from 0.48% to 0 when r increases from 0.3 to 1.0. When m and r are fixed (m is 0.5 and r is 1.0), the job error rate drops from 0.14% to 0 when c increases from 0.1 to 1.0. On the other hand, a higher value of r incurs a higher overhead and verifier overhead. For example, when r is 1.0, the average overhead is 112%, and when r is 0.3, the average overhead is 41%. We note that the experiment overhead and verifier overhead is lower than the simulation result in Fig. 5(e) and 5(f), respectively. This fact affirms our prediction in section IV.A: Since Theorem 1 assumes m as a constant value, its estimation of overhead and verifier overhead should be higher than the experiment result.

We observe that in each of the 18 parameter combinations, CCMR is able to eliminate all malicious workers during the execution of the word count job. In Fig. 6, we show three representative combinations in terms of how soon each malicious worker is detected and thus removed. We could see that under the first two combinations, CCMR can remove all malicious workers very quickly (within less than 15% of the total job execution time). Under the third combination, the malicious workers are very stealthy (cheat with a probability of 10%) and the replication frequency is low (30%), but CCMR can still remove all six malicious workers before 50% of the job has finished. Such observations suggest that CCMR is effective in detecting malicious workers even if the adversary implements its best strategy.

We also measure the execution time delay introduced by CCMR in the map phase by running the same 800-map task word count job. Since here our focus is map phase overhead, we disable the reduce phase integrity check. We also disable the combine phase. In addition, we do not introduce malicious workers. The reason is that we believe the customer is willing to pay extra cost to detect malicious workers. However, they are reluctant to spend extra money for CCMR when there are no malicious workers. The experiment result is shown in TABLE III. It indicates that the extra running time compared to the baseline grows with r. When r grows from 0.3 to 1.0, the extra running time grows from 19.75% to 83.26%.

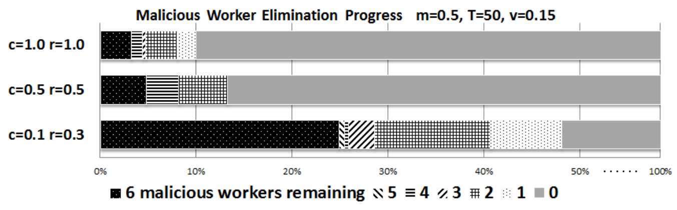

Fig. 6. Malicious Worker Elimination Progress.

B. *Reduce Phase*

To measure the reduce phase, we set the replication probability r as 0.16, the verification probability v as 0.07, and the credit threshold T as 600. Our accuracy test shows that such a configuration guarantees 0 job error rate when m is 0.5 and c changes among 0.1, 0.5 and 1.0, which is consistent to our simulation in Fig. 5(c). Given the space limit, we only present the performance experiment result in this section. We use two applications to measure the performance overhead of reduce phase integrity check: word count and mahout twenty newsgroups classification 0. For a similar reason as the map phase, we introduce neither map integrity check nor malicious nodes in this performance test.

The word count job is the same job described in Section V.A, which consists of 800 map tasks and one reduce task. We compare the running time of CCMR with baseline. On average, CCMR with reduce integrity check takes 1398 seconds to finishes the job. It produces 88 replication reduce sub-tasks and six verification reduce sub-tasks. Compared with the standard MapReduce, which takes 979 seconds to finish the same job (we enable the combine phase to accelerate the execution), the execution delay is 43%. We attribute such delay to the execution of big number of replication reduce sub-tasks.

We also run the Mahout twenty newsgroups classification example on CCMR. Such application consists of five jobs. Each of the first three jobs produces more than 100,000 keys. Hence, CCMR still sets the credit threshold T to 600. The last two jobs produce less than 600 keys, so CCMR sets the report number of those two jobs as one and directly generates one verification reduce task for each job. The total execution time under CCMR is 1892 seconds. Compared to 1304 seconds on the baseline, the execution delay is 45%. The details of each job execution time are shown in TABLE IV. Interestingly, the CCMR execution times of the last two jobs are smaller than their baseline execution time. This is because compared with the baseline, the cluster of CCMR has an extra node on the private cloud to run as the master, which shares the workload of task scheduling and management.

VI. Related Work

Several existing solutions have been proposed using replication sampling, and verification techniques to address result

TABLE II. Accuracy and Overhead of Wordcount Application with Map Phase Integrity Check

Other settings	**T=50, v=0.15**																	
m	**0.5**									**0.167**								
c	**0.1**			**0.5**			**1.0**			**0.1**			**0.5**			**1.0**		
r	**0.3**	**0.5**	**1.0**	**0.3**	**0.5**	**1.0**	**0.3**	**0.5**	**1.0**	**0.3**	**0.5**	**1.0**	**0.3**	**0.5**	**1.0**	**0.3**	**0.5**	**1.0**
J_{map} (%)	1.31	0.58	0.14	1.08	0.28	0.05	2.25	0.76	0.0	0.48	0.04	0.0	0.19	0.45	0.02	0.11	0.0	0.0
O_{map} (%)	46.3	66.3	126.7	47.5	70.1	122.6	51.4	74.6	118.8	33.6	53.7	100.6	32.8	51.5	103.6	34.9	53.5	103.6
V_{map} (%)	4.9	8.3	18.1	6.3	11.5	21.5	9.5	15.4	24.5	4.7	7.8	15.8	4.6	7.3	15.1	10.4	8.5	16.9

TABLE III. PERFORMANCE OF WORDCOUNT APPLICATION WITH MAP PHASE INTEGRITY CHECK

Configuration	**baseline**	**v=0.15, T=50**		
		r=0.3	**r=0.5**	**r=1.0**
Running time(s)	1728	2069	2323	3167
Extra running time (%)	---	19.75	34.41	83.26

TABLE IV. PERFORMANCE OF MAHOUT TWENTY-NEWSGROUP CLASSIFICATION WITH REDUCE PHASE INTEGRITY CHECK

Job#	**Baseline Execution time (s)**	**CCMR Execution time (s)**	**Replication Reduce Sub-ask Number**	**Verification Reduce Sub-task Number**
1	517	983	95	7
2	331	482	108	8
3	210	221	80	6
4	85	63	0	1
5	161	143	0	1

integrity problems in other distributed environments such as P2P Systems and Grid Computing. Golle et al. [10] propose to guarantee correctness of the distributed computation result by duplicating computations. Zhao et al. [8] proposed a sampling based idea of inserting indistinguishable Quizzes to the task package, which is going to be executed by the untrusted worker and verify the returned result for those Quizzes. Their simulation result shows by combining reputation system, Quiz approach gains higher accuracy and lower overhead than replication based approach. However, suggested by their simulation, the reputation accumulation is a long-term process so that in order to accumulate reliable reputation, it takes as many as 10^5 tasks. Du et al. [9] proposed to insert several sampled tasks to the task package, and check the sampled task returns using Merkle-tree based commitment technique. The analysis in the paper showed that in order to detect error from a malicious worker who cheats with low probability such as 0.1, it takes more than 75 samples to be inserted to each worker.

For MapReduce, Wei et al. [4] proposed an integrity assurance framework SecureMR to enforce the *commitment protocol* and the *verification protocol*. SecureMR employs task duplication to defeat collusive workers. The design difference from our paper is that the number of duplication task for each original task is non-deterministic. Such an approach guarantees 90% of detection rate in defeating periodical collusive attacker with 40% of duplication rate when the malicious worker fraction is below 0.15 and malicious cheat probability is 0.5. (According to (2) in [4]) However, (2) in [4] also shows that when malicious worker fraction is 0.5, malicious cheat probability is 0.1, 40% of duplication rate can achieve only 25% of detection rate. The maximum detection rate SecureMR can achieve under this environment setting is 80%, with a duplication rate more than 500%. Wang et al. [11] proposed the VIAF framework that uses full replication and non-deterministic verification. Such approach eliminates all non-collusive workers and removes collusive worker with certain probability. However, their work does not consider practical factors when deployed on a real cloud, such as cross-cloud communication. In addition, both above works cannot handle the case where the reduce task number is very small (e.g. only one reduce task).

VII. CONCLUSION AND FUTURE WORK

We propose a novel framework, CCMR, which overlays MapReduce on top of a hybrid cloud to offer high result integrity. Based on such framework, we propose the result integrity check scheme in order to boost the accuracy meanwhile to reduce the delay. Our theoretical analysis and experimental result suggests CCMR can achieve low job error rate while incurring non-negligible performance overhead.

Even though CCMR suggests a promising result in guaranteeing high result accuracy in MapReduce, the performance delay still needs improving. In addition, the trade-off between accuracy and overhead in CCMR involves several system parameters. Automating the parameter selection to achieve higher accuracy and lower overhead would be another important research topic.

VIII. ACKNOWLEDGEMENT

This material is based upon work supported by the U.S. Department of Homeland Security under grant Award Number 2010-ST-062-000039. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the U.S. Department of Homeland Security.

REFERENCES

[1] Dean, Jeffrey, et al. "MapReduce: simplified data processing on large clusters". *Communications of the ACM*, *51*(1), 107-113.

[2] "Amazon Elastic Compute Cloud (Amazon EC2)", http://aws.amzaon.com/ec2/

[3] "Windows Azure Compute", https://www.windowsazure.com/en-us/home/features/compute

[4] Wei, Wei, et al. "Securemr: A service integrity assurance framework for mapreduce." Computer Security Applications Conference, 2009. ACSAC'09. Annual. IEEE, 2009.

[5] Bowers, Kevin, et al. "HAIL: a high-availability and integrity layer for cloud storage." Proceedings of the 16th ACM conference on Computer and communications security. ACM, 2009.

[6] Popa, Raluca Ada, et al. "Enabling security in cloud storage SLAs with CloudProof." Proceedings of 2011 USENIX Annual Technical Conference, Portland, OR. 2011.

[7] "What is Apache Hadoop", http://hadoop.apache.org/

[8] Zhao, Shanyu, et al. "Result verification and trust-based scheduling in peer-to-peer grids." Peer-to-Peer Computing, 2005. P2P 2005. Fifth IEEE International Conference on. IEEE, 2005.

[9] Du, Wenliang, et al. "Uncheatable grid computing,"in Proceedings of International Conference on Distributed Computing Systems. 2004.

[10] Golle, Phillippe, et al. "Secure distributed computing in acommercial environment," in 5th International Conference Financial Cryptography.

[11] Wang, Yongzhi, et al. "VIAF: Verification-based Integrity Assurance Framework for MapReduce", in the 4th IEEE International Conference on Cloud Computing

[12] "Amazon Elastic MapReduce (Amazon AMR)", http://aws.amazon.com/elasticmapreduce/

[13] "Project Daytona: Iterative MapReduce on Windows Azure", http://research.microsoft.com/en-us/downloads/cecba376-3d3f-4eaf-bf01-20983857c2b1/default.aspx

[14] "Cloud Security: Amazon's EC2 serves up 'certified pre-owned' server images" http://dvlabs.tippingpoint.com/blog/2011/04/11/cloud-security-amazons-ec2-serves-up-certified-pre-owned-server-images

[15] Bugiel, Sven, et al., "AmazonIA: when elasticity snaps back". In Proceedings of the 18th ACM conference on Computer and communications security.

[16] "Twenty news group example", https://cwiki.apache.org/confluence/display/MAHOUT/Twenty+Newsgroups