CROSS CLOUD MAPREDUCE: A RESULT INTEGRITY CHECK FRAMEWORK ON HYBRID CLOUDS

Yongzhi Wang*, Jinpeng Wei*, Mudhakar Srivatsa§
* Florida international University, Miami, USA
§ IBM T.J. Watson Research Center, Yorktown Heights, USA
ywang032@cis.fiu.edu, weijp@cis.fiu.edu, 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, we propose a result integrity check scheme on both the map phase and the reduce phase. On the other hand, we strive to reduce the performance overhead by reducing the cross-cloud communication and merging sub-tasks. We implement CCMR based on Apache Hadoop MapReduce and evaluate it on Amazon EC2. Both theoretical and experimental analysis show that our approach can guarantee high result integrity in a hybrid 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

1. INTRODUCTION

MapReduce (Dean & Ghemawat, 2008) 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 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 and Microsoft Daytona 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: malicious participants 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, (Balduzzi, Zaddach, Balzarotti, Kirda, & Loureiro, 2012) and (Bugiel, Nürnberger, Pöppelmann, Sadeghi, & Schneider, 2011) point 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 (Golle & Stubblebine, 2002), verification (Du, Jia, Mangal, & Murugesan, 2004), and credit-based trust management (Zhao, Lo, & Dickey, 2005)). 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. In order to improve performance of the reduce phase, we propose the *request*

bucketing technique to further reduce the performance overhead.

Our theoretical simulation (in Section 4.3) shows that when credit threshold T is set to 50 (in the map phase), and replication probability r is set to 0.5, CCMR can guarantee a job error rate of less than 1% when less than half of workers on the public cloud are malicious, and a job error rate of less than 9% when all the workers on the public cloud are malicious. When T is set to 600 (in the reduce phase) and r is set to 0.16, CCMR can guarantee a job error rate of 0% when less than half of workers are malicious, and a job error rate of less than 6% when all the workers are malicious.

Our experiment result shows that CCMR introduces 19% to 83% of delay depending on the replication probability r in the map phase. It also shows that CCMR can introduce 29% of delay on average in the reduce phase when applying the request bucketing technique.

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 and run a series of experiments over the commercial public cloud (Amazon EC2). We show that CCMR is an efficient framework to guarantee high computation integrity.

The rest of this paper is organized as follows. Section 2 describes the system assumptions and attacker model. Section 3 presents the system design of CCMR. Section 4 makes the theoretical analysis in terms of security, accuracy, and overhead. Section 5 describes and analyzes the experiment result. Section 6 discusses the related work, and Section 7 concludes the paper.

2. System Assumptions and Attacker Model

2.1 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, which are deployed on the private cloud, are trusted. 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 (Popa, Lorch, Molnar, Wang, & Zhuang, 2011) and (Bowers, Juels, & Oprea, 2009).

In CCMR, the master requires that each worker who runs a task/sub-task submit the hash value of its computation results to the master. We assume the hash value to be consistent with the actual task/sub-task output. Such an assumption can be realized by applying the commitment-based protocol proposed in (Wei, Du, Yu, & Gu, 2009) (i.e., the previous worker commits the task output to the master by the hash value. The later worker who takes the previous task's output as input will return the input hash value to the master. The master compares the hash values to ensure the consistency of the hash value and the corresponding task output). Finally, we assume that the tasks running on each worker are deterministic. This assumption guarantees that multiple executions of the same task/sub-task by honest workers return the same result.

2.2 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*.

3. System Design

3.1 System Overview and Architecture

CCMR overlays MapReduce on a hybrid cloud consisting of one private cloud and one public cloud, which is shown in Figure 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 untrusted.

In both the map and the reduce phases, CCMR defines three types of tasks: the *original task*, the *replication task*, and the *verification task*. The original and replication tasks are executed by the workers on the public cloud. The verification tasks are executed by the verifier on the private cloud. The replication task repeats the original task's work to validate the original task result. Since the replication tasks are executed by the untrusted public cloud worker, the verification tasks can check the replication task result by re-executing the task on the verifier. 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 part of the original reduce task. Each replication/verification reduce task consists of one or several small portions of original reduce task. Each portion of task is called a *sub-task*.

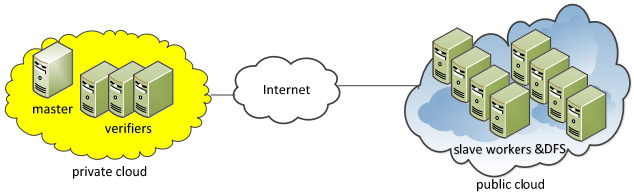


Figure 1. Architecture of CCMR.

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 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. Since the replication and verification tasks are only used to evaluate the correctness of the original tasks, 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.

3.2 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. The replication task assignment is applied with a technique called *hold-and-test*, which will be introduced later. 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 probability. Each original map task is replicated with *replication probability*, and each pair of consistent results 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

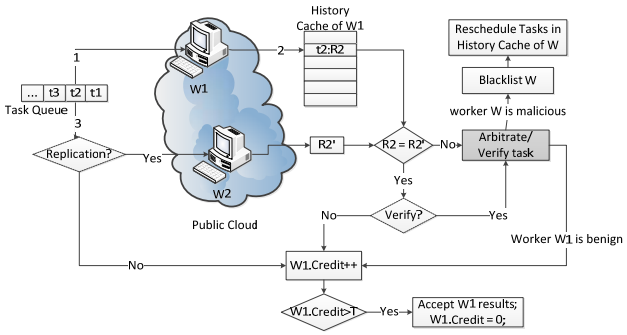


Figure 2. Control Flow of CCMR in Map Phase

overcome this drawback, CCMR performs the credit based trust management to improve 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 the mapper's 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 achieving 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.

Figure 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 4.1.

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 simulation shows (Section 4.3).

3.3 Reduce Phase Integrity Check

In the reduce phase, the approach presented in Section 3.2 can be directly applied only if the number of reduce tasks is large 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 instance, the word count application in Section 5.2 contains only one original reduce task. However, this single task will process 2.7M of records (1.07GB of data) in the input and generate 598K of records in the output, and take 262 seconds to complete. In this case, directly verifying the entire reduce task is expensive in terms of computation and communication cost. Therefore, we propose to break down each original reduce task into many sub-tasks and apply two-layer check on each sub-tasks.

In the original MapReduce, the master breaks the job input into multiple blocks. Each map task processes one block and generates the task output in the format of <key, value> tuples, which are sorted by key. Reduce tasks will process the output of map tasks and also generate <key, value> tuples as reduce task output. The output is also sorted by key. Each reduce task only processes certain map output tuples with specific keys, which are determined by the *partition* function.

Our reduce phase integrity check design is based on the following intuition. We observe that both the map tasks and the reduce tasks outputs are sorted by key. For each key in the reduce output, if we can precisely pinpoint the map output tuples that are related to that key, we can reproduce the portion of the reduce task that is related to that reduce output key. We call each portion of reduce task as *sub-task*. Therefore, each original reduce task can be divided into multiple *sub-tasks*, each of which is related to one key. By applying two-layer check to each sub-task, we can guarantee high accuracy of the original reduce task. Here, we temporarily define each sub-task to cover one key. We will extend this concept later for practicality reason.

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 the first-layer (replication) and second-layer (verification) check on each report with *replication probability* and *verification probability*, respectively. The generation of a replication sub-task is decided after the report of an original sub-task is returned to the master (*hold-and-test*). We regulate that an original reduce task result is accepted by the master only when all its sub-tasks pass the 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. When the number of sub-tasks in an original reduce task is smaller than the credit threshold, CCMR directly generates a verification reduce task to verify the entire original reduce task.

The above idea only works conceptually. However, in order to make it practical, we need to extend the concept of *sub-tasks* and *reports*, and overcome the following three challenges. 1) Creating a sub-task for each key would incur significant overhead because in many cases, the amount of keys in a reduce task can be huge (e.g., 598K keys in the word count application in Section 5.2). 2) The accuracy only relies on the two-layer checks of 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 extending the concept of report and sub-task to cover a range of consecutive keys, instead of just one. In other words, we require each report to cover a range of (instead of only one) consecutive keys. With this improvement, the number of sub-tasks will be reduced.

For the second challenge, CCMR requires that consecutive reports in the original reduce task must overlap in one key. In addition, the first and last report in each original reduce task should cover the first and last key of the task output, respectively. The master will check those requirements when it receives the reports. 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 first and last key 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 complete, the output can be sanitized by removing the output records related to the dummy input records.

For the third challenge, each map task in CCMR builds a *key table* to facilitate the record look up in the map task output, as shown in Figure 3. When a sub-task wants to fetch the map output within a certain key range (e.g., Key2 to Key 9) from the map task output file, it will locate the position through the key table of that task. In the original MapReduce, each map task stores the map output file locally, which consists of key-value pairs sorted by key. Notice that the lengths of keys and values vary and multiple key-value pairs can have the same key (e.g., Key 3). For brevity, we call consecutive key-value pairs in the map output file with the same key as a *block*. Each record in the key table corresponds to a block in the map output file. It has three fields, indicating the start position of the block, the length of the key, and the length of the block. Since the lengths of keys and values vary, each access to a key on the map output file needs to go through the key table, which has the fixed record length. The key table records are sorted by key, CCMR can apply binary search to look up the records

within the request key range. However, each key comparison in the binary search needs to fetch the key in the map output file through the key table. When the binary search finds the keys (e.g., key 3 through key 8) between the request range (e.g., key 2 to key 9), the key position and the record length direct the reducer to fetch the portion of map output. Since it is the mapper who creates the key table, 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. The consistency between the hash value and the actual key table can be guaranteed by the commitment-based protocol (Wei, Du, Yu, & Gu, 2009).

Figure 4 shows 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 Figure 4 shows, each map task creates a key table (Step 1). When the 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 by initiating a replication sub-task (with replication probability). The replication sub-task fetches input with a key range of (Apple, Driver) from each map task (map 0 and map 1) through key tables (Step 7). When it completes 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 results. The verification sub-task will be created if necessary (Step 11).

3.4 REQUEST BUCKETING

One draw back of reduce phase design in Section 3.3 is that in order to achieve high accuracy, CCMR will generate a large number of sub-tasks. Each sub-task will be executed as an individual reduce task. Such a reduce task needs to connect to each map tasks, locate the map output position, and fetch only a small portion of data. In addition, its setup and tear down will consume a certain amount of resource.

As a result, a big number of sub-tasks can introduce a big performance delay. For example, according to the word count experiment in Section 5.2, the reduce phase will generate 88 replication sub-tasks and six verification sub-tasks. The execution delay is 43%. We hope to merge multiple sub-tasks into one reduce task to improve performance, while without sacrificing the accuracy. Therefore, we propose the *request bucketing* technique to achieve such goal.

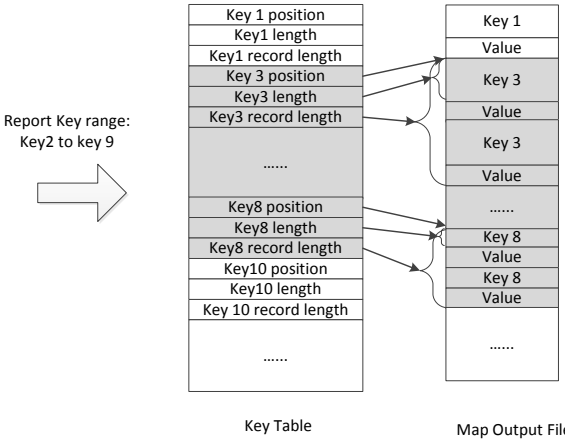


Figure 3. Locating Map Output Between Key 2 and Key 9 using Key Table

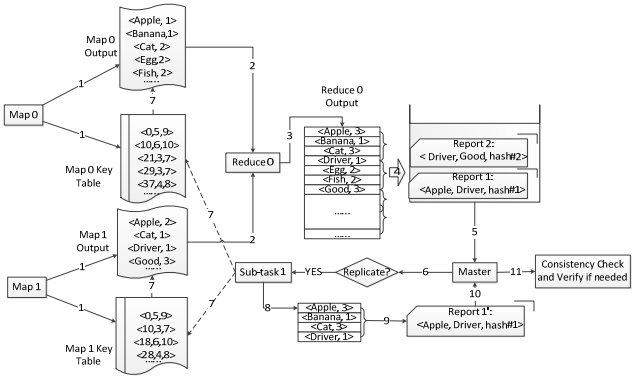


Figure 4. Control Flow of CCMR in Reduce Phase

The idea of request bucketing is as follows. We create several *replication buckets* ready to receive replication sub-task requests. Each buckets has a limited capacity.

Whenever the master receives a sub-task report from original reduce task and decides to generate a replication sub-task (first-layer check), instead of generating a reduce task just for that single sub-task, the master randomly chooses a bucket and stores the sub-task request to this bucket. When a bucket achieves its capacity, the master generates a replication reduce task, which will contain all the sub-task requests in that bucket. Meanwhile, the bucket is emptied. The master will randomly choose a worker to execute this replication task. The replication task will send multiple replication sub-task reports back to the master, each of which will correspond to one original sub-task report. When a report passes the first-layer check, the master will apply the second-layer check to those sub-tasks. Similarly to the replication buckets, the master also creates several verification buckets to accumulate the verification sub-task requests. When a verification bucket is full, the master will generate a verification reduce task to merge all the requests in that bucket.

Although the request bucketing technique merges multiple sub-tasks into on reduce task, the two-layer check

is still performed on each original reduce sub-task. The analysis in Section 4.2 and experiment in Section 5.2 will show that request bucketing technique does not undermine the accuracy of CCMR, while it reduces the execution delay.

4. System Analysis

4.1 Quantitive Analysis on Map Phase

Since the major differences of map and reduce phase in CCMR is the object used to perform two-layer check (task in map phase and sub-tasks in reduce phase), we could use similar model to analyze both phases. In this section we give quantitative analysis on the map phase. In the next section, we will discuss how to adapt the theorem in this section to the reduce phase.

We first analyze the adversary strategy of malicious worker. Based on that, we will perform quantitative analysis on accuracy and overhead.

Adversary Strategy: We denote the *malicious worker fraction* on the public cloud as *m*. We assume that the adversary controls all malicious workers. In other words, all malicious workers are collusive, and there exists only one collusive group. Assuming that the goal of the adversary is to inject as many errors as possible and yet not reveal the malicious workers, we analyze the strategy of the adversary under CCMR as follows. Suppose a task is assigned to a malicious worker, two cases are possible for the adversary.

Case 1. If the adversary has not seen a similar 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 and the returned result. Due to the existence of hold-and-test (Section 3.2), the adversary is not allowed to defer the decision to the time that it sees the replica of the current 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 can survive the first-layer check only when either the current task is not replicated, or the replica of the current task is assigned to another malicious worker.

Case 2. If the adversary has seen a similar task before, it is assured that the current task is a replication task. It can simply ask the worker to take the same action for the current task as the one it has seen 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 in a randomized manner, the adversary cannot tell whether cheating at the current task has a smaller chance of detection than cheating at other 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 cheat once in hundreds of tasks) can still fit in our model because we can set *c* as a small value close to 0.

Item	Name	Definition
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 to make 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 results in one batch of results.
J	job error rate	The ratio of incorrect results number to the total 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.
K	task key number*	The number of keys (records) generated by an original reduce task.
S	sub-task key number*	The number of keys covered by a sub-task.
R	report number*	The number of reports an original reduce task sends to the master.
B	bucket size*	The maximum number of sub-task requests contained in a bucket.

* is only applicable to reduce phase

Table 1 CCMR System Setting Parameters

We define several metrics to measure the accuracy and overhead of CCMR for both the map and reduce phases, and summarize the parameters in Table 1. Notice that the first eleven items are applicable to both phases. The metrics in the map phase is with regard to the tasks. When we adapt such metrics into reduce phase analysis, we only need to substitute the "task" with "sub-task". The last four items are only applicable to the reduce phase. We will discuss them in Section 4.2. We perform a series of probabilistic analysis and present our analysis result in Theorem 1. This theorem can also be applied to the reduce phase by replacing the "task" with "sub-tasks". The argument of such substitution is provided in Section 4.2.

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* is

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

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

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

The *batch error number* is

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

The *batch error rate* is

$$e = E/T \quad (4)$$

The *job error rate* is

$$J = me \quad (5)$$

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

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

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

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

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, if executed on a malicious worker, it can pass the two-layer check in one of 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 of above three cases, we get (1). The complete proof of Theorem 1 can be found in the Appendix.

4.2 Analysis on Reduce phase

The difference between map phase and reduce phase is that the reduce phase performs the two-layer check to the original reduce sub-tasks instead of tasks. Although request bucketing causes multiple sub-tasks to be replicated or verified in a batch, we can still adapt the model of map phase (Section 4.1) for the following reasons.

Firstly, the way that the master performs two-layer checks to each original reduce sub-task is the same as it does to the tasks in the map phase. Second, even though sub-tasks are merged together to execute, the assignment of sub-task requests to the buckets and the assignment of buckets to the workers are all randomized, which is equivalent to the effect of assigning each sub-task request to workers randomly. Third, the adversary strategy in Section 4.1 is also applicable to the reduce phase. If the reducer executing the original reduce task is malicious, when it sends an original sub-task report to the master, it cannot predict whether the replication sub-task will be assigned to its colluder. Therefore, the malicious reducer still needs to make a random guess on each original sub-tasks.

The reduce phase design in Section 3.3 (without request bucketing) is only a special case where each bucket has a capacity of one. Once a sub-task request is assigned to the bucket, a replication/verification reduce task is created immediately. Hence, the above arguments are also applicable to the design without request bucketing technique.

Therefore, to adapt the theoretical analysis result to the reduce phase, we only need to change the definition in Table 1 by replacing the "tasks" with "sub-tasks". The Theorem 1 can be also adapted by substituting the "tasks" with "sub-tasks".

We introduce four parameters specific for the reduce phase, shown in Table 1 (The last four items). The total number of reports a master receives from one original reduce task is $R = \lceil K/S \rceil$. In our design, an original reduce task result is accepted by the master when its credit achieves credit threshold. In other words, we can adjust the parameters to have credit threshold T to be equal to the report number (i.e., $T = R$). When K is big enough, we can set T to a big value by adjusting S to ensure high accuracy. For example, in the word count application in Section 5.2, the single original reduce task output contains 598,000 keys. In this case, in order to set T to 600 to ensure high accuracy, we can set S as $\lceil K/R \rceil = 997$.

When request bucketing is applied, the two-layer check is still applied to each sub-task. Also, the assignment of sub-tasks is still random. Therefore, request bucketing will not undermine the CCMR accuracy. In addition, the number of sub-tasks is not reduced with the introduction of request bucketing. The only thing that is affected is the number of

replication/verification reduce tasks. Suppose the number of replication/verification reduce sub-tasks generated in a job is $\overline{R}$, without request bucketing, the replication/verification reduce task number is $\overline{R}$, since each sub-task will be executed in an independent task. However, with request bucketing, the number of replication/verification reduce task will be $\lceil \overline{R} / B \rceil$.

4.3 Simulation Result

We present several simulation results based on Theorem 1 to analyze the relationships among accuracy, overhead and other system parameters.

We first simulate the job error rate under different system parameters in Figure 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 will be increased 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.

Figure 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 smaller J; if c is too small, the number of injected errors is reduced, also resulting in a smaller J.

Figure 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.

Figure 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 5.1 and Figure 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.

Figure 5(e) shows the tradeoff between job error rate J and overhead O, and Figure 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 each 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

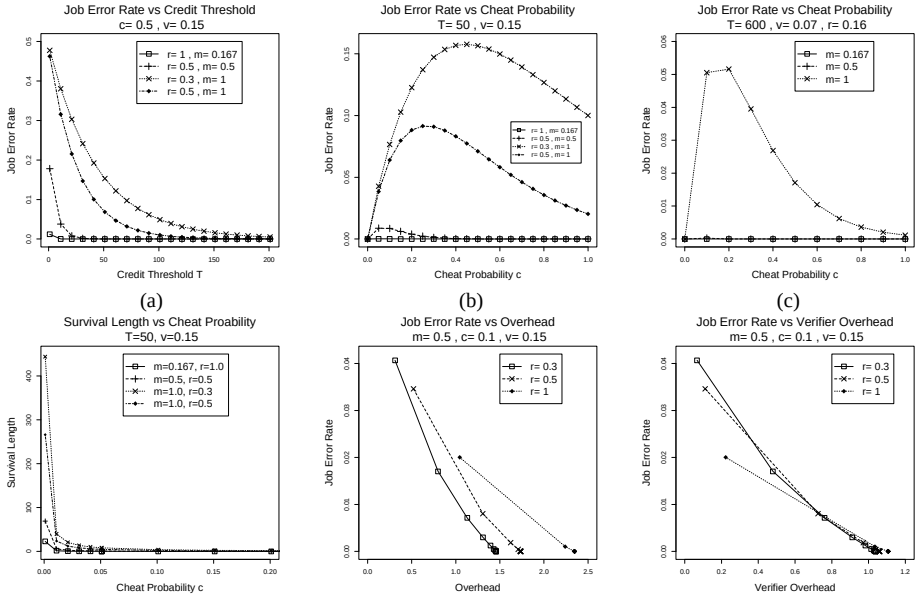


Figure 5. Simulation of CCMR Analysis Model

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 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), increasing T further would bring neither additional accuracy benefit, nor additional overhead or 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.

4.4 Communication Cost Analysis

In order to reduce the cross-cloud communication cost, we only deploy DFS nodes on the public cloud. Such a deployment avoids DFS data synchronization across clouds. In addition, it 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 task fetches input data from the DFS. Since each reducer fetches input from the mappers' local storage, and only the original reduce task writes its output to the DFS (According to Section 3.1), the only major cross-cloud communication in the reduce phase happens when a verification reduce task fetches input data from mappers on the public cloud. 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).

5. 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.

5.1 Map Phase

We measure job error rate, overhead and verifier overhead of CCMR by running a word count MapReduce job (Section 3.3) in an environment with malicious MapReduce workers. We simulate such malicious workers by implementing the adversary's strategy described in Section 4.1. 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 2 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 simulation result in Section 4.3. 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 Figure 5(e) and (f), respectively. This fact confirms our prediction in Section 4.3: 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 Figure 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

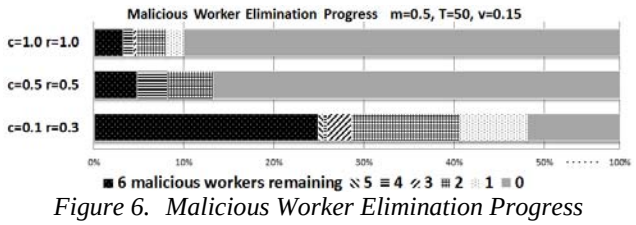


Figure 6. Malicious Worker Elimination Progress

System Setting	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 2 Accuracy and Overhead 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 3 Performance of Map Phase Integrity Check

Application	Job No.	Baseline	CCMR without request bucketing				CCMR with request bucketing			
		Exec. time (s)	Exec. time (s)	Delay (%)	Repl. Task Num.	Veri. Task Num.	Exec. time (s)	Delay (%)	Repl. Task Num.	Veri. Task Num.
Word Count	**1**	979	1398	43%	88	6	1263	29%	9	1
20 News Letters	**1**	517	983	90%	95	7	636	23%	10	1
	2	331	482	45%	108	8	465	40%	11	1
	3	210	221	5%	80	6	260	23%	8	1
	4	85	63	-25%	0	1	57	-32%	0	1
	5	161	143	-11%	0	1	138	-14%	0	1
Ave. Delay (%)*		----	46%				29%			

* Average delay does not count the direct verification job (i.e., job 4 and 5 in the 20 news letters application).

Table 4 Performance of Reduce Phase Integrity Check

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 delay, 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 3. It indicates that the extra running time compared to the base-line grows with r. When r grows from 0.3 to 1.0, the extra running time grows from 19.75% to 83.26%.

5.2 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. We set both the replication and verification bucket size B as 10 when applying request bucketing technique.

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 with our simulation in Figure 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 (The Apache Software Foundation, n.d.). For a similar reason as the map phase, we introduce neither map integrity check nor malicious nodes in this performance test.

To measure the performance gain from the request bucketing technique, we perform two sets of experiments: the one without using the request bucketing technique and the one using such technique. The experiment result is shown in Table 4. The word count job is the same job described in Section 5.1, which consists of 800 map tasks and one reduce task. We compare the running time of CCMR with baseline. On average, CCMR without request bucketing takes 1,398 seconds to finish the job. It produces 88 replication reduce tasks and six verification reduce 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%. When we apply the request bucketing to the reduce phase, the number of replication and verification tasks drops

to nine and one, respectively; the execution time is reduced to 1,263 seconds and the execution delay is reduced to 29%. We attribute the reduced execution delay to the reduction of number of replication/verification tasks.

We also run the Mahout twenty newsgroups classification example on CCMR. This application consists of five jobs. Each of the first three jobs produces more than 100,000 keys. Hence, CCMR sets the credit threshold T to 600. The last two jobs produce less than 600 keys, so CCMR directly generates a verification reduce task for each job. The total execution time under CCMR without request bucketing is 1,892 seconds. Compared to 1,304 seconds on the baseline, the execution delay is 45%. When applying request bucketing, the execution time is reduced to 1,556 seconds. The execution delay is reduced to 19%. It is interesting to notice that the CCMR execution times of the last two jobs are shorter than the baseline. This is because the master in CCMR is executed on the private cloud, which has more computation power than a micro instance on the public cloud. When evaluating the average slow down incurred by CCMR, we exclude these two special jobs.

Overall, request bucketing plays an important role in boosting the performance. Based on the execution time of word count and the first three jobs of 20 newsletters applications, the average execution delay of CCMR without request bucketing is 46%. When request bucketing is applied, the average execution delay is reduced to 29%.

6. RELATED WORK

Several existing solutions have been proposed using replication sampling, and verification techniques to address result integrity problems in other distributed environments such as P2P Systems and Grid Computing. Golle et al. (Golle & Stubblebine, 2002) propose to guarantee correctness of the distributed computation result by duplicating computations. Zhao et al. (Zhao, Lo, & Dickey, 2005) 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. (Du, Jia, Mangal, & Murugesan, 2004) 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. (Wei, Du, Yu, & Gu, 2009) 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 (Wei, Du, Yu, & Gu, 2009)) However, (2) in (Wei, Du, Yu, & Gu, 2009) 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. (Wang & Wei, 2011) 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). This paper is extended from the conference paper (Wang, Wei, & Srivatsa, 2013). Based on the original CCMR design, we improve the reduce phase performance by proposing request bucketing techniques. Our experiment result shows that the request bucketing technique reduces the average execution delay from 46% to 29%.

7. CONCLUSION

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 that CCMR can achieve low job error rate while introducing non-negligible performance overhead.

8. 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.

9. APPENDIX

Proof of Theorem 1:

S_n, the probability of a malicious mapper/reducer to survive after executing *n* original tasks/sub-tasks is the probability summation of all permutations on *n* independent events. Each event should fall into one of the below three cases.

1). The worker does not cheat. The probability in this case is (1-c);

2). The worker cheats, but the task/sub-task is not replicated. The probability in this case is c(1-r);

3). The worker cheats, and the task is replicated. However, the other worker executing the replication task/sub-task can collude with it. When the consistent results are returned to master, the verification task is not invoked. The probability in this case is crm(1-v).

Condition1	Condition 2, given condition 1 is satisfied	Condition 3, given condition 2 is satisfied	Category Number
The task/ sub-task t is assigned to a malicious worker z, which survived in the current batch	--	t is not replicated.	1
		t is replicated, but z does not cheat.	2
		t is replicated but not verified. z cheats. But the replication task/sub-task is executed by another malicious worker.	3
The task/ sub-task t is assigned to a malicious worker z, which does not survive in the current batch	t is not the last task/sub-task in the current batch(i.e., t is the one being detected)	t is not replicated.	4
		t is replicated. z does not cheat. The results are not verified.	5
		t is replicated. z cheats. The error is not detected.	6
		t is replicated. z does not cheat. The results are verified.	7
	t is the last task/sub-task in the current batch (i.e., the one being detected).	--	8
The task/ sub-task t is assigned to a benign worker z.	t is replicated.	The corresponding replication task is assigned to a malicious worker. The malicious worker cheats.	9
		The corresponding replication task returns the same result as the original one.	10
	t is not replicated	--	11

Table 5 Categories of Tasks based on Assignment Conditions.

Category Number	Probability	Workload W	Verifier overhead V
1	$P_1 = m\binom{T}{1}\cdot\frac{1}{T}\cdot S_{T-1}(1-r)$	1	0
2	$P_2 = m\binom{T}{1}\cdot\frac{1}{T}\cdot S_{T-1}r(1-c)$	2	v
3	$P_3 = m\binom{T}{1}\cdot\frac{1}{T}\cdot S_{T-1}rcm(1-v)$	2	0
4	$P_4 = m\sum_{j=0}^{1}\sum_{i=1}^{T-1}\binom{i}{1}\frac{1}{i}S_{i-1}(1-r)(rc(1-m))^j(rcmv)^{1-j}$	1+W	V
5	$P_5 = m\sum_{j=0}^{1}\sum_{i=1}^{T-1}\binom{i}{1}\frac{1}{i}S_{i-1}r(1-c)(1-v)(rc(1-m))^j(rcmv)^{1-j}$	2+W	V
6	$P_6 = m\sum_{j=0}^{1}\sum_{i=1}^{T-1}\binom{i}{1}\frac{1}{i}S_{i-1}rcm(1-v)(rc(1-m))^j(rcmv)^{1-j}$	2+W	V
7	$P_7 = m\sum_{j=0}^{1}\sum_{i=1}^{T-1}\binom{i}{1}\frac{1}{i}S_{i-1}r(1-c)v(rc(1-m))^j(rcmv)^{1-j}$	2	1
8	$P_8 = m\sum_{j=0}^{1}\sum_{i=0}^{T-1}S_i(rc(1-m))^j(rcmv)^{1-j}$	2	1
9	$P_9 = (1-m)rmc$	2	1
10	$P_{10} = (1-m)r(1-mc)$	2	v
11	$P_{11} = (1-m)(1-r)$	1	0

Table 6 Probability, Workload and Verifier Overhead of Each Category

By summing up the probability of different permutations of above three cases on n independent events, we have

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

Setting n=T, we have S_T, the probability that a malicious mapper/reducer submit a batch of task/sub-task to a master. The probability of each malicious worker can submit exactly k batches (i.e., not detected in first k batches, but detected on the (k+1)th batch) is

$$(S_T)^k \cdot (1-S_T)$$

The expected number of batches a malicious worker can submit (*survival length*) is therefore.

$$L = \sum_{k=0}^{\infty} k \cdot (S_T)^k (1-S_T) = \frac{S_T}{1-S_T} \quad (2)$$

In a batch with credit threshold T, The probability that exactly k out of T tasks return erroneous results but are not detected is as follows. It consists of (T-k) events that the worker does not cheat, and k events that the worker cheats but without detection.

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

The expected number of tasks returning erroneous results in a batch (*batch error number*) is therefore

$$E = \sum_{k=0}^{T}(k \times \Delta_k) \quad (3)$$

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

The *batch error rate* by definition is therefore

$$e = E/T \quad (4)$$

Since the assignment of tasks/sub-tasks on the public cloud is uniformly distributed on all workers. And the detected malicious workers are not added to the black list, we can assume ratio m of workers on the public cloud submit erroneous results to the master, the batch error rate for those erroneous results is e. We have total error rate in a job (*job error rate*) is

$$J = m*e + (1-m)*0 = me \quad (5)$$

In order to calculate the overhead and the verifier overhead, we divide the original tasks/sub-tasks into 11 different categories based on different conditions the task/sub-task may encounter, as shown in Table 5. We summarize the probability, workload and verifier overhead of each category in Table 6. Here we define *workload* as the number of executions each task/sub-task has to run on the public cloud, marked as *W*. It includes the original task/sub-task execution and the task/sub-task overhead. Therefore, we have $W = 1 + O$, where 1 corresponds to the original task/sub-task and O corresponds to the overhead.

Since the categories summarized in Table 5 are mutual exclusive and exhaustive, we have $\sum_{i=1}^{11} P_i = 1$. We calculate the expected workload by combining the probability and workload under each category.

$$W = P_1 + P_{11} + (1+W)P_4 + (2+W)(P_5 + P_6) + 2(P_2 + P_3 + P_7 + P_8 + P_9 + P_{10})$$

We calculate the expected verifier overhead by combining the probability and verifier overhead under each category.

$$V = vP_2 + VP_4 + VP_5 + VP_6 + P_7 + P_8 + P_9 + vP_{10}$$

By reorganizing the formula and replacing P_i with the value in Table 6, we have,

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

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

, where

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

10. REFERENCES

Balduzzi, M., Zaddach, J., Balzarotti, D., Kirda, E., & Loureiro, S. (2012). A security analysis of amazon's elastic compute cloud service. *the 27th Annual ACM Symposium on Applied Computing* (pp. 1427-1434). ACM.

Bowers, K., Juels, A., & Oprea, A. (2009). HAIL: a high-avaliability and integrity layer for cloud storage. *the 16th ACM conference on Computer and Communication Security* (pp. 187-198). ACM.

Bugiel, S., Nürnberger, S., Pöppelmann, T., Sadeghi, A., & Schneider, T. (2011). AmazonIA: when elasticity snaps back. *the 18th ACM conference on Computer and communications security* (pp. 389-400). ACM.

Dean, J., & Ghemawat, S. (2008). MapReduce: simplified data processing on large clusters. *Communications of the ACM , 51* (1), 107-113.

Du, W., Jia, J., Mangal, M., & Murugesan, M. (2004). Uncheatable grid computing. *International Conference on Distributed Computing Systems*, (pp. 4-11).

Golle, P., & Stubblebine, S. (2002). Secure distributed computing in a commercial environment. *Financial Cryptography* (pp. 289-304). Springer Berlin Heidelberg.

Popa, R., Lorch, J., Molnar, D., Wang, H., & Zhuang, L. (2011). Enabling security in cloud storage SLAs with CloudProof. *USENIX Annual Technical Conference* (pp. 355-368). USENIX.

The Apache Software Foundation. (n.d.). *Twenty news group example*. Retrieved Sep 15, 2013, from https://cwiki.apache.org/confluence/display/MAHOUT/Twenty+Newsgroups

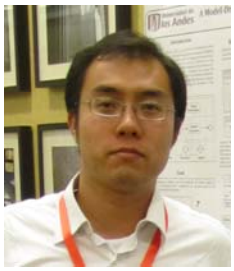
Wang, Y., & Wei, J. (2011). VIAF: verification-based integrity assurance framework for MapReduce. *the 4th IEEE International Conference on Cloud Computing* (pp. 300-307). IEEE.

Wang, Y., Wei, J., & Srivatsa, M. (2013). Result Integrity Check for MapReduce Computation on Hybrid Clouds. *the 6th IEEE International Conference on Cloud Computing*. IEEE.

Wei, W., Du, J., Yu, T., & Gu, X. (2009). SecureMR: A service integrity assurance framework for MapReduce. *Computer Security Applications Conference* (pp. 73-82). ACM.

Zhao, S., Lo, V., & Dickey, C. (2005). Result verification and trust-based scheduling in peer-to-peer grids. *the 5th IEEE International Conference on Peer-to-Peer Computing* (pp. 31-38). IEEE.

Authors



Yongzhi Wang received his BS and MS degree in School of Computer Science and Technology from Xidian University, China in 2004 and 2007, respectively. He is currently a PhD student in School of Computing and Information Sciences from Florida International University, USA. His research interests include big data, cloud security and outsourced computing security. Before starting his PhD program, he worked for SPSS and IBM as a software engineer.

Jinpeng Wei received a PhD in Computer Science from Georgia Institute of Technology, Atlanta, GA in 2009. He is currently an assistant professor at the School of Computing and Information Sciences, Florida International University, Miami, FL. His research interests include malware detection and analysis, information flow security in distributed systems, cloud computing security, and file-based race condition vulnerabilities. He is a member of the IEEE and the ACM.

Mudhakar Srivatsa is a research scientist in the Network Technologies Department at the IBM T. J. Watson Research Center. He received his Ph.D. degree in computer science from Georgia Tech. His research interests primarily include network analytics and secure information flow. He serves as a technical area leader for Secure Hybrid Networks research in International Technology Alliance in Network and Information Sciences, a research consortium formed from twenty three US and UK industrial and academic members. He also serves as a principal investigator for modeling and analyzing co-evolving networks in the Network Science Collaborative Technology Alliance.