Towards Trusted Services: Result Verification Schemes for MapReduce

Chu Huang, Sencun Zhu and Dinghao Wu
School of Information Science and Technology
Pennsylvania State University, State College, PA 16802
$\{$*cuh171,sxz16,dwu12*$\}$*@psu.edu*

***Abstract*—Recent development in Internet-scale data applications and services, combined with the proliferation of cloud computing, has created a new computing model for data intensive computing best characterized by the MapReduce paradigm. The MapReduce computing paradigm, pioneered by Google in its Internet search application, is an architectural and programming model for efficiently processing massive amount of raw unstructured data. With the availability of the open source Hadoop tools, applications built based on the MapReduce computing model are rapidly growing.**

In this work, we focus on a unique security concern on the MapReduce architecture. Given the potential security risks from lazy or malicious servers involved in a MapReduce task, we design efficient and innovative mechanisms for detecting cheating services under the MapReduce environment based on watermark injection and random sampling methods. The new detection schemes are expected to significantly reduce the cost of verification overhead. Finally, extensive analytical and experimental evaluation confirms the effectiveness of our schemes in MapReduce result verification.

***Keywords*-MapReduce, watermark injection, random sampling, result verification, trustworthy**

I. Introduction

Recently, a new wave of large-scale data processing technologies has emerged in various research and business areas, such as genomic analysis, visualization, simulation and business intelligence, etc. Due to the ever increasing demands in those fields, more and more people and organizations have come to realize the increasing needs of computational resources. MapReduce was first proposed by Google in 2004 [1]. As a compensation mechanism to make up for individual's lack of computation resources, the MapReduce framework enables huge data processing by dynamically building up the computation environment which consists of a large number of computers. Considering its benefits, MapReduce has been widely adopted by a number of large companies, such as Google, Yahoo, Amazon, Facebook and AOL [2]. An open source implementation of MapReduce called Hadoop [3], was then developed by Yahoo. The ease of application development using Hadoop further encourages wide adoption of MapReduce.

*This work was supported by NSF CAREER 0643906

Often operated in the open environment with its emphasis on Internet-wide accessibility, Mapreduce provides flexible and scalable computing services for its clients. However, this also leaves MapReduce vulnerable to third party attacks and misbehavior. Given that MapReduce clients have no control over their data once it gets fed into the distributed applications, many of them have their worries on common security issues, such as the violations of data integrity and data disclosure. Besides all those common privacy threats and security risks, MapReduce also has its own unique security deficiencies on data miscalculation while lazy or malicious servers involved in a MapReduce task. In this paper, we mainly focus on two main motivations lying behind this malicious service providing, including attacks by arbitrary purpose and attacks by strategic purpose.

To address the above challenges, we present two lightweight results verification schemes for detecting the cheating behaviors within the MapReduce framework. We primarily focus on some text retrieval related applications, which involve simple operations (e.g., count, addition) on large amount of text and graph data. Expanded from those text retrieval related applications, we believe our proposed scheme can also be effectively applied to other MapReduce based text processing applications, such as distributed grep, log data processing, speech recognition and machine translation.

II. Related Work

Security of distributed systems has been studied by many researchers, but only a few of them really focus on MapReduce. Xiao et al. [4] presented an accountable MapReduce platform which checks all working machines and detects malicious nodes in real time. By replaying the tasks executed by workers and matching output with the original results, auditors are able to generate verifiable evidence once inconsistency occurs. The Airavat system proposed by Roy et al. [5] provides strong security and privacy guarantees for sensitive data computation of MapReduce. Their work focuses on protecting privacy issues of untrusted code of data-mining and data-analysis algorithms executed on MapReduce. Wei et al. [6] proposed a secure scheme called SecureMR which aims at protecting the computation integrity issue of MapReduce. SecureMR detects misbehavior of mappers by sending same tasks to multiple mappers, and check consistency of results. Existing security frameworks for MapReduce have the same limitation: reducers and master need to be trusted. This assumption might not be practical in real world.

Several other result checking techniques have been proposed in different areas other than MapReduce. Golle and Mironov [7] proposed a Ringer scheme to protect against coalitions of lazy cheaters, which ensures client that most of the work are done correctly with high probability. Szajda et al. [8] extended this ringer scheme, and presented a probabilistic verification scheme to prevent against malicious behavior in grid computing. A generic verification scheme based on redundancy was presented by Golle and Sutbblebine [9]. They described a security framework for commercial distributed computing by simple task redundancy. Szajda et al. [10] presented another redundancy-based strategy. Their work requires fewer resources compared to Golle and Sutbblebine's, but it can achieve the same effective cheating detection rate. Another generic approach was proposed by Zhao et al. [11]. This result verification scheme is called Quiz for peer-to-peer grid computing. They inject indistinguishable tasks into a job with other normal tasks. Then the verifier checks the correctness of all quiz results.

Most of the above mentioned existing studies are based on replication techniques, with either tasks being duplicated and sent to two or more different participants for processing, or re-do partial of a task then match it with the pre-calculated result. Such redundant tasks waste resources in the system, and there is lack of an efficient way to detect colluding participants. We remove the constraints that master and/or reducers have to be trusted and provide a scheme based on watermark injection and weighted sampling techniques. Our scheme can efficiently detect cheating behavior of MapReduce computing in the context of text processing problems.

III. System Model

A. MapReduce Programming Model

The MapReduce framework consists of a single master JobTracker and more than one slave Task Trackers. Master is responsible for task assignment, scheduling, and management. A slave that contributes to the computation resources is also called worker in this model. Workers execute the tasks as directed by Master. Besides master and workers, the other important entity in MapReduce system is the distributed file system. Typically, a file in Hadoop Distributed File System (HDFS) [1], the file system underlying the Hadoop system, is gigabytes to terabytes in size. Our work will follow the Hadoop implementation of MapReduce.

As shown in Figure 1, the process of MapReduce data processing can be divided into two phases: map and reduce. In the beginning of the map phase, input data is first sliced into n splits. Then master assigns these splits to different workers for paralleled processing. Workers on the map phase are called *mappers*. In this phase, each mapper processes one split, produces an intermediate result and outputs the intermediate result in the (key, value) key-value pair format to its local disk. The intermediate results from each mapper are then partitioned into m parts by a partition function. In the reduce phase, workers read partitioned intermediate results from mappers in the preceding map phase, process them independently, and merge all intermediate results associated with the same key into a final result. Workers running the reduce function are called *reducers* correspondingly. What to follow for reducers is storing final results to their local disks. Eventually, the final result will be stored in the distributed file system.

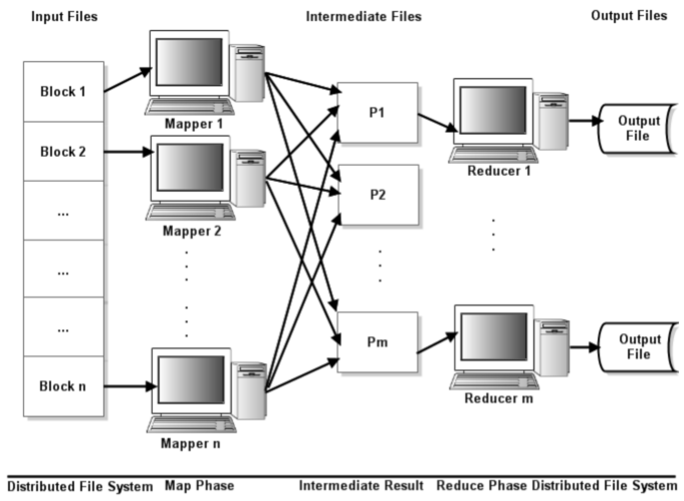

Figure 1: The MapReduce programming model

B. Model of Cheaters

In the MapReduce computation model, we consider an attack model in which the adversary is rational and economically motivated. We classify cheating behaviors using two different adversary models. We assume the input domain for MapReduce is U, and the task for MapReduce is to compute $F : f_2(f_1(x))$ for all the input domain, in which f_1 is a map function while f_2 is a reduce function.

- *Lazy Cheating Model*: In this model, the cheating worker follows the master's computation with two exceptions: 1) it will not start processing its task from the beginning of its task, or drop a task at any point before finishing the task. 2) for every $x \in U$, MapReduce uses $G : g_2(g_1(x))$ as the result instead of $F : f_2(f_1(x))$, where function G is usually much less expensive than function F. The goal of this kind of cheaters is to save computation resources and maximize their profit by performing more tasks in the same period of time.

- *Malicious Cheating Model*: In this model, the behavior of a malicious worker is either arbitrary or strategic. For any specific x, instead of returning $F(x) = (k, v)$, it would manipulate the final result and return (k', v'). In other words, the worker intentionally returns wrong results to client. Arbitrarily cheaters intentionally return wrong results for the purpose of disrupting the computation. Strategic cheaters falsify only a tiny portion of the results with a purpose (e.g., improve its website rank for search engines), and therefore is more difficult to identify.

Based on the models, we provide two secure schemes to prevent the computation provider from cheating the client by claiming that they have done the job that they actually did not, and also to protect from malicious behavior of MapReduce.

IV. Result Verification for MapReduce

A. Watermark Injection

Our watermark injection scheme is an application-specific verification scheme. The basic idea of watermark injection is to inject indistinguishable marks into normal documents. Figure 2 shows the process of this watermark injection scheme. We assume that if the marks we injected are all computed correctly, the workers have executed the tasks honestly. Otherwise, workers, one or many, have not conducted the task honestly. We outline this watermark injection scheme as follows.

- *Setup (Watermark generation)*: Before handing the data to the computation provider, the verifier pre-processes the input files by inserting a number of watermarks into the task documents.
- *Computation*: MapReduce performs computation tasks and returns final results to the client.
- *Verification*: The client verifies the correctness of the results and accepts the result if no inconsistency is found.
- *Recovery*: Then it recovers the injected documents and removes the impact of the watermark. Otherwise, the client discards the results.

Because marks are randomly injected and cannot be distinguished from other data items in the task documents, MapReduce will not be able to identify them in an easy manner. Consequently, if some task executer cheats on a task that contains injected marks, its corresponding computation result will be inconsistent with what was pre-computed. Next we use an example application, inverted index, to illustrate our proposed scheme.

1) Inverted Index: An inverted index is a data structure that maps terms to their locations. Using this approach, a search engine easily identifies the appropriate documents for terms, without scanning each document in the whole web.

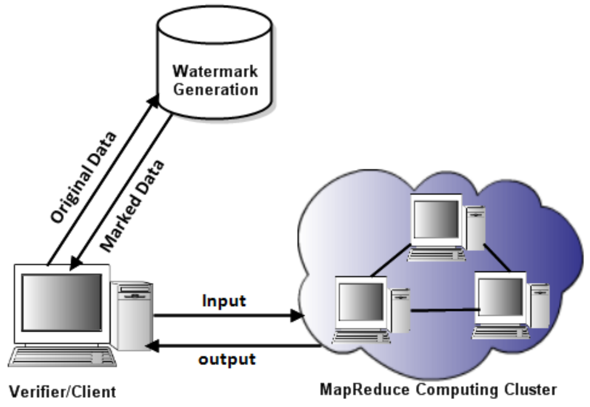

Figure 2: Watermark-based verification scheme. A watermark generation module preprocesses raw data and injects watermarks. Upon received output from MapReduce, the verifier performs verification. If the computation from MapReduce is accepted, the verifier will call a recovery module to remove the effect introduced by injecting watermarks.

Many modern search engines on the web use an inverted index scheme for data storage.

The large amount of computation may incentivize computation service provider to cheat for saving both storage and computational resources. One way to cheat is to skip the computation and return wrong (randomly picked) result from its own vocabulary. For strategic cheaters, they might only pick certain words and manipulate their corresponding locations. To detect and hence prevent cheating phenomenon, we apply the watermark injection scheme on the inverted index application.

Let $D = \{D_1, D_2, \cdots, D_k\}$ be a collection of documents being indexed. Let $d_1, d_2, \cdots, d_{k'}$ be a collection of watermark files. The verifier then injects watermarks by encoding certain words in each selected document, and records the document *id* and the corresponding words. We outline the detailed process for generating watermark as follows:

- Randomly selects k' documents $d_1, d_2, \cdots, d_{k'}$ among all the input documents before handing over to MapReduce
- For each selected documents $d_i (i = 1, 2, \cdots, k')$, the verifier conducts a simple alphabet substitution [12] on every ϵ words. A substitution table T should be predefined by the verifier. The verifier also needs to store the information about injection distance ϵ, the encoded words and the corresponding document *ID*.
- For each encoded term in document collection $d = \{d_1, d_2, \cdots, d_{k'}\}$, the verifier builds a posting list. The computation overhead for building a small inverted index I' for all encoded terms is far lower than building a complete inverted index $I(I' \in I)$.

Only the client knows which documents have been chosen and which words have been marked. The verifiable computation scheme for inverted index is performed as follows:

1. *Setup*: Before handing over the data to a computation provider, the verifier pre-processes the data and generates watermark following the detailed procedure of watermark generation above. Meanwhile, the verifier stores information about which words are encoded and constructs an index I' of encoded (watermark) words.
2. *Computation*: MapReduce performs the task of constructing an inverted index I and returns I to client.
3. *Verification*: The verifier checks the results from MapReduce by 1) examining whether all encoded terms appearing in the dictionary of I' are all included in the dictionary of I; 2) comparing the posting list of each term in I' with the results from the MapReduce system. If both steps return a matching conclusion, the verifier accepts the results. Otherwise, the verifier concludes MapReduce has cheated and discards the results.
4. *Recovery*: If the result is accepted, the verifier needs to recover these words that have been marked. In order to do so, the verifier needs to decode those words based on the same substitution table T. For each term in the index I': if its corresponding document *ID* in its posing lists of I, then doing nothing; otherwise, insert a posting list with id_x as document *ID* into index I.

2) Security Analysis: The security of the watermark injection scheme is based on the assumption that workers of MapReduce cannot determine whether the input file has been marked or not. The only way to escape detection is to correctly compute all encoded words. To determine whether a word is actually encoded or not, lazy workers of MapReduce needs to perform brute force search over all words in its dictionary (assuming it contains all words in real world). The computation overhead of this process is much heavier than that of indexing itself. If the purpose of the cheating workers were to save computation resource, they would lose the incentive to cheat. If lazy workers do not process the data, or give wrong results by random guess, the results from MapReduce would be inconsistent with the pre-computed value. Misbehavior is going to be caught with a high probability as long as they ever cheat on the computation.

To convince readers the high accuracy of the watermark injection scheme, we provide a brief theoretical derivation on a simplified scenario. Let $D = \{D_1, D_2, \cdots, D_n\}$ represent the set of documents to be sent to computation providers. We randomly select p_1 percent of totally n documents, np_1, to inject watermarks. In each document, we encode p_2 percentage of words to do the watermark injection. If we assume that each document contains roughly the same number of words, w, the total number of watermarked word would be np_1wp_2. Suppose we have m mappers and r reducers in a MapReduce system. MapReduce divides all n task documents evenly and distributes them to the m mappers. Thus each mapper gets n/m documents. The probability of this situation is

$$\Pr(\text{Detected}|\text{one mapper cheats}) = 1 - (1-p_1)^{n/m}$$

Then the probability that a mapper cheats without being detected is $(1-p_1)^{n/m}$. A similar analysis should be applied to the reducer step. The nw words are randomly assigned to r reducers and each reducer receives on average nw/r words. The probability that a word is watermarked is p_1p_2. When a reducer decides to cheat, it escapes the detection only if this reducer does not receive any watermarked words. Thus, the probability is $(1-p_1p_2)^{nw/r}$.

When there are more mappers or more reducers cheating, the detection rate is even higher. Say, x mappers and y reducers cheat, the probability of detection is

$$\Pr(\text{Detected}|x,y) = 1 - (1-p_1)^{xn/m}(1-p_1p_2)^{ynw/r} \quad (1)$$

Let us denote $(1-p_1)^{xn/m}(1-p_1p_2)^{ynw/r}$ as $f(x,y)$, describing the probability that the verification scheme fails when x mappers and y reducers have cheated. Assume each mapper and each reducer may cheat with a fixed probability, p_m and p_r, respectively. The probability that x mappers and y reducers cheat is

$$\Pr(x,y) = \frac{m!}{x!(m-x)!}(1-p_m)^{m-x}p_m^x \frac{r!}{y!(r-y)!}(1-p_r)^{r-y}p_r^y \quad (2)$$

Thus, the conditional probability of detection is

$$\Pr(\text{Detected}|\text{Cheating}) = \sum_{[0,m]}^{x}\sum_{[0,r]}^{y} Pr(x,y)Pr(\text{Detected}|x,y) \quad (3)$$

Using what we have,

$$\Pr(\text{Detected}|\text{Cheating}) = 1 - \sum_{[0,m]}^{x}\sum_{[0,r]}^{y} Pr(x,y)f(x,y) \quad (4)$$

In this summation, $x \in [0,m]$ and $y \in [0,r]$. Thus

$$\begin{aligned}\Pr(\text{Detected}|\text{Cheating}) &= 1 - (1-p_m+p_m(1-p_1)^{n/m})^m \cdot \\ &\quad (1-p_r+p_r(1-p_1p_2)^{nw/r})^r\end{aligned}$$

Let's suppose $m = 15, p_m = 0.10$ and $r = 15, p_r = 0.10$ and look at the detection rate of this verification system. Assume $n = 1500$ and $w = 1500$ and we can plot the detection rate vs. small values of p_1 and p_2 by using MatLab (as shown in Figure 3). Before proceeding, we examine the equation first. We have $n/m = 100$ and $(1-p_1)^{100} = 0.9^{100}$ that is almost 0. Similarly, we have $(1-p_1p_2)^{nw/r} \approx 0$. Thus, the equation is approximately:

$$\Pr(\text{Detected}|\text{Cheating}) = 1 - (1-p_m)^m(1-p_r)^r \quad (5)$$

This means that when each mapper or each reducer has to deal with a large number of documents or words and the system cheats, the probability that the verifier detects this cheating is independent of the percentage of the wa-

Table I: The effectiveness of the watermark injection scheme for text problems.

	MapReduce cheats	MapReduce not cheat
Results Rejected	$1-(1-p_m)^m(1-p_r)^r$	0
Results Accepted	$(1-p_m)^m(1-p_r)^r$	1

termarked files/words (as long as they are not too small). Consequently,

$$\Pr(\text{Detected}|\text{Cheating}) = 0.9576 \tag{6}$$

Table I shows the effectiveness of the watermark injection scheme for the inverted index problem. The two cells represent the probability that the verifier rejects or accepts the computation results from the MapReduce system, when the system cheats or does not cheat. As shown, the accuracy is fairly high and the Type I error rate is 0.

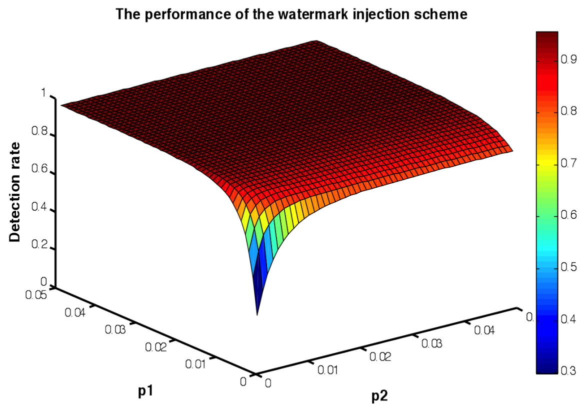

Figure 3: The theoretical performance of the watermark injection scheme for text problems.

B. Random Sampling

1) PageRank: In the real world, the number of web pages could be very large and it is still growing exponentially. According to WorldWideWebSize.com [13], the size of the World Wide Web contains about 17.5 billion indexed pages. Such a worldwide graph contains billions of nodes and several billions of links. If we assume that each URL takes 0.5KB to store, the whole Internet, containing trillions of unique URLs, will take more than 400TB space. Calculating the PageRank values frequently using the classic PageRank algorithm [14], [15] on such a big data set is computationally expensive. MapReduce provides a solution to tackle this kind of scaling challenges in a highly distributed manner.

We introduce a basic PageRank model here. The web pages and hyperlinks on the Web can be modeled as nodes and edges in a directed graph $G = (V, E)$. Let N be the total number of web pages and use $1, 2, \cdots, N$ to index these N web pages. Let $L(u)$ be the number of out-links from page u where $u = 1, \cdots, N$. The corresponding adjacency matrix is defined as follows:

$$A_{u,v} = \begin{cases} 1/L(u) & , \quad \text{if there is an edge from } u \text{ to } v; \\ 0 & , \quad \text{otherwise.} \end{cases} \tag{7}$$

Note that the value of $A_{u,v}$ can be any number between 0 and 1 in this case. The PageRank value for a page v is given as follows [14]:

$$\text{PR}(v) = \frac{1-d}{N} + d \sum_{u \in B(v)} \frac{PR(u)}{L(u)} \tag{8}$$

The damping factor [15] d in the equation is used to represent the probability that a Web surfer follows the random behavior pattern. The value of d is usually set to 0.85.

In order to retrieve a stable solution for the PageRank values of all webpages, we set the $PR(i)$ value to be $1/N$ and keep running iterations until stationary values are obtained or only small differences exist between the results from adjacent iterations. Note that in the rest of our work, we will use the term “node” and “page” interchangeably.

2) Ranking Pages with MapReduce: Suppose a client rents MapReduce to process a large set of web pages for finding a way to characterizing structure of the web graph and computing the importance of every web page using the classic PageRank algorithm. This task needs multiple map and reduce phases to accomplish. Since values needed for an iteration of a page only depend on previous page ranks of its in-links, every task sent to workers of MapReduce is independent and can be executed in parallel. This iteration will continue until convergence.

The watermark injection scheme does not work very well under the context of PageRank calculation. The reason is that in order to do the verification, we need to pre-calculate the PageRank values for these marked or injected pages. However, doing this calculation means we need to do the calculation for the whole data set. If so, using MapReduce to distribute the complex task is not meaningful anymore. We propose another approach for the problem: verification by sampling. We provide two sampling templates for cheating detection for PageRank: *naive random sampling and in-degree weighted sampling*.

Naive Random Sampling: In this naive random sampling scheme, the verifier randomly selects pages with an equal probability for each page. Using the PageRank scores returned from MapReduce, the verifier checks if the PageRank equation holds. The PageRank values of the in-nodes of the sampled nodes and the out-degree of these in-nodes need to be collected as well, according to the equation. Computation providers cannot possibly guess or make a reasonable solution that holds the equation for all the nodes in the network without actually computing the page rank. Thus, under any type of cheating models, there must be a great number of webpages on which the equation does not hold. Thus, the detection rate of this naive sampling verification scheme is almost 1.

However, there might be cases when computation providers intentionally modify the PageRank value for certain webpages for their personal sakes. The percentage of these modified pages may take only a small portion of the whole data set. Consequently, the naive sampling approach cannot guarantee a high detection rate in this scenario. Thus, we propose a weighted sampling scheme to tackle this concern.

In-degree Weighted Sampling: When selecting one page and checking if Equ.8 holds for this page, we are actually checking this page as well as all its in-nodes. If the page rank of one of its in-nodes of itself is modified, the equation will not hold and hence we have detected the cheating behavior. On the basis of this fact, when we select a page that has a high in-degree, we are actually checking a number of pages in the set. Thus, an intuitive suggestion is to sample those pages with high in-degrees as much as possible. In order to prevent the computation providers figure out our verification approach, we should add a randomness degree to the sampling. In the in-degree weighted scheme, we add a weight that is proportional to in-degree to each page and adopt a weighted sampling method. Thus, pages with higher in-degrees have a higher probability to be selected.

3) Security Analysis: When computation providers intentionally change the value of a small portion of pages, such as increasing the PageRank value of their own pages and decreasing the value of their opponent's pages, there might be only a small number of nodes for which Equation (1) does not hold. Let't look at the effectiveness of the two sampling schemes here. Assume N is the total number of pages and er is the error rate—the percentage of pages which do not satisfy Equation (8). The verifier selects M pages by one of the two sampling schemes. As mentioned, the in-degree values of these M pages affect the verification effectiveness. Let's use $f(x)$ to describe the in-degree distribution of all pages in the data set and x in the following equations refers to the in-degree value. Use c to represent the detection rate. When we only check one page whose in-degree is x, we are actually checking this node and all the x in-nodes. Thus, the detection rate is $g(x) = 1 - (1 - er)^{x+1}$. The overall detection rate should be the expectation value of $g(x)$, with respects to the distribution of x. Thus,

$$\begin{aligned} c(\text{Naive}) &= E(1 - (1 - er)^{M(1+x)}) \\ &= \int f(x)(1 - (1 - er)^{M(1+x)})dx \end{aligned}$$

According to Jamakovic and Uhlig [16], in a regulatory network, the in-degrees follow an exponential distribution. Let's assume that x follows an exponential distribution, we have, $f(x) = K\exp(-Kx), x \in [0, \inf)$.

Table II: The effectiveness of the watermark injection scheme for text problems.

	MapReduce cheats	MapReduce not cheat
Results Rejected	$1-(1-p_m)^m(1-p_r)^r$	0
Results Accepted	$(1-p_m)^m(1-p_r)^r$	1

Substituting $f(x)$ into the detection rate equation, we obtain,

$$c(\text{Naive}) = 1 - \frac{K(1-er)^M}{1-(1-er)^M er^K} \tag{9}$$

when $(1-er)^M er^K < 1$, which can be easily satisfied.

For the in-degree weighted sampling algorithm, instead of $f(x)$, we should have $(x+1)f(x)/A$ in the integration. $(x+1)A$ represents the effect of the weights and $A = \int(x+1)f(x)dx$ is only a normalization coefficient. Thus,

$$\begin{aligned} c(\text{Weighted}) &= E(1 - (1 - er)^{M(1+x)}) \\ &= \int (x+1)f(x)(1 - (1 - er)^{M(1+x)})dx/A \end{aligned}$$

For the same exponential distribution of x, we obtain,

$$c(\text{Naive}) = 1 - \frac{K(1-er)^M}{(1-(1-er)^M er^K)^2 A} \tag{10}$$

The relationship between A and K is that A=1+1/K. Let's set $K = 1/4$ (the average in-degree number is 4) and $er = 0.05$. Let's increase M from 10 to 100 and compare the two schemes under this simple scenario. Figure 4 shows the plots for the detection rates. Apparently, the weighted scheme outperforms the naive one.

For the Internet, Jamakovic and Uhlig [16] concluded that the in-degrees of the Internet follow a power law distribution $f(x) = ax^{-\gamma}, x \in [0, \inf)$. Under this circumstance, we can conduct a similar derivation and obtain the following results.

$$\begin{aligned} c(\text{Naive}) &= \int [1 - (1 - er)^{M(1+x)}]f(x)dx \\ &= 1 - (1 - er)^M a \int [(1 - er)^M]x \cdot x^{-\gamma}dx \\ c(\text{Weighted}) &= \int [1 - (1 - er)^{M(1+x)}]xf(x)/Adx \\ &= 1 - (1 - er)^M a/A \int [(1 - er)^M]x \cdot x^{1-\gamma}dx \end{aligned}$$

Computational simulations lead to similar numeric results as shown in Figure 4.

Table II shows the effectiveness of the weighted sampling scheme for the PageRank calculation problem.

V. Experimental Evaluation

To evaluate the effectiveness of our verification scheme, a prototype of watermark injection and random sampling verification schemes is implemented to test the overall method

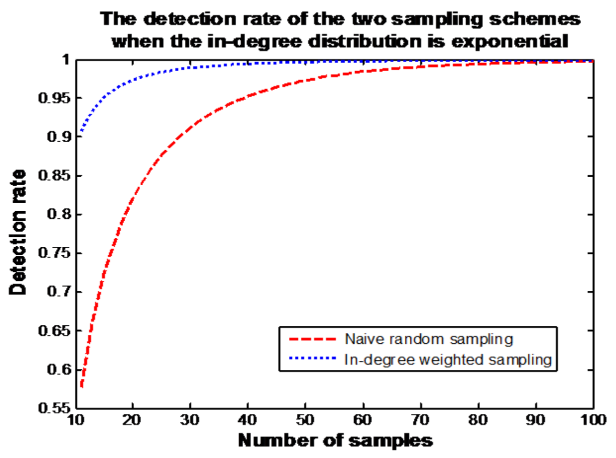

Figure 4: The theoretical performance of the two sampling schemes for PageRank.

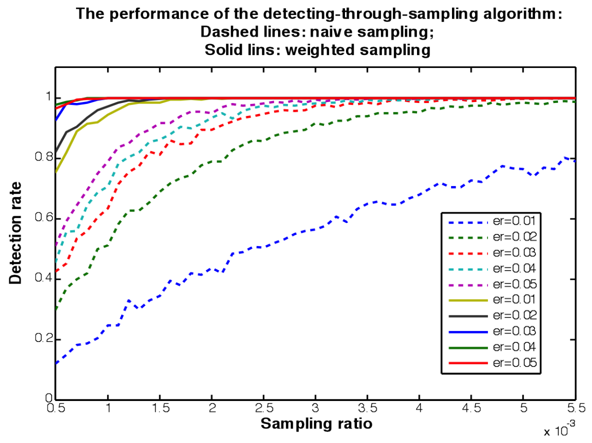

Figure 6: The performance of the two sampling schemes under computational simulations

performance. Experiments conducted in this research aim to mainly measure the detection rate under different types of cheating models.

A. *Experiment Setup*

The experiment is running on a small cloud with 10 machines. We use 9 host machines as workers that offer MapReduce services and one host as a master. All host machines run on a 3.4GHz Intel Core Quad-core i7 processor. Each virtual machine has 512MB of memory and 20GB disk and was installed Ubuntu Linux 10.04.2, Sun JDK 6 and Hadoop 0.20.2. We conduct our experiment using inverted index application and PageRank calculation based on the Google PageRank algorithm. Since this experiment is not focused on optimization of Hadoop configuration, we just use the default configuration of Hadoop.

B. *Performance Analysis*

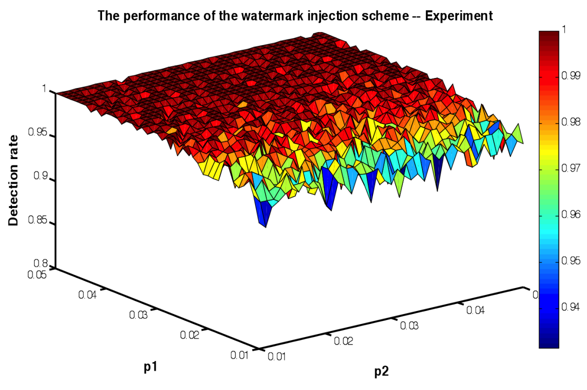

Figure 5: The performance of the watermark injection scheme from computational simulations

1) Watermark Injection: Every data point shown in Figure 5 can be viewed as a test conducted with settings of varied watermark injection proportions within documents and words. Given that 200 simulations are run for each of those tests, mean values are calculated as the data points as shown in this figure. Consistent with the conceptual modeling, p_1 in this case is the percentage of documents with injected watermarks, ranging from 1% to 5%. In the same way, p_2, also with the same range values, indicates the proportion of words within each document which are encoded with watermarks. One can interpret from the above figure that among all the 16810 results points, only less than 0.5% of them have a detection rate less than 95%. Moreover, more than half of the simulations have their detection rates larger than 99%.

Besides, when considering p_1 and p_2's effects on the overall detection rates, we further find that, compared with p_1, p_2 seems to have non-significant impact on the experiment results. It seems that detection rates do not change very much along with the variances in the percentage of watermark injected words. However, if viewing across the p_1 axis, one can notice that along with the growth of the watermark injected document numbers, detection rates of cheating behaviors begin to increase until the injection percentage reaches 3%. In other words, it shows relatively lower detection rate when the watermark injected document proportions are less than 3%. Beyond that injection percentage, detection rates begin to reach about 100%.

2) Random Sampling: In order to simulate the cheating behavior of the malicious and lazy workers, intentional miscalculations of PageRank values were injected into the final evaluation dataset at an error rate of 1% to 5%, respectively.

Figure 6 demonstrates the distributions of the experimental detection rates over the increasing number of sampling ratios. Overall, we can see that the random sampling approaches, both naive and weighted, perform nicely on the PageRank cheating detection task under the MapReduce environment.

As defined as the percentage of miscalculations, error rates ranging from 1% to 5% are represented by five different colors as shown in the figure. Dashed lines are corresponding to the naive-based random sampling approach, whereas solid lines are for the more advanced weighted-based random sampling methods. Detection rates increase dramatically with lower sampling ratios, in particular when sampling ratios are less than 0.15% in the weighted approach or less than 0.3% in the naive-based method.

Almost all of the weighted experiments reach high detection rates of over 80%, while in contrast, results of the naive based approach are not as desired as the weighted ones under small sampling sizes. Weighted samplings gradually lost its absolute superiority in detection rates as the sampling ratios increases. Except naive samplings under 1% and 2% error rates, all sampling settings tested, both naive and weighted schemes, in this study, achieve about the same detection rates when sampling more than 35 pages out of the total 10,000 ones.

VI. Discussion and Conclusion

In this work, we have proposed two verification schemes for detecting lazy and malicious behaviors that MapReduce systems may have when they are conducting specific tasks. Although we have only explained our schemes on two text retrieval related applications for easy-to-understand purpose, we would like to remark that our work can also be generalized straightforward to other text-intensive tasks, such as word count, distributed grep, log data processing, speech recognition and machine translation. In fact, given any computational task with text input, watermark injection enables MapReduce task executors to verify the results based on those watermarked items. Likewise, for computational tasks with graph input, our proposed random sampling scheme can also be effectively applied, considering the probabilistic generalizations of the general random sampling approach.

Although as mentioned above, our proposed schemes can be effectively extended to some other MapReduce text processing applications, they may not be very well generalized to numerical data-intensive computational tasks, such as knowledge discovery, pattern recognition, and more advanced statistical analysis. Also limited by the scope of this study, only lazy and malicious attacks have been considered. Although they do cover a large proportion of the possible security risks, still there are threats in specific purpose from those strategic attackers. For instance, attackers who only manipulate their proposed keywords in the inverted index application. In that case, our watermark injection method would not work as well as it did while detecting the lazy and malicious attacks. Therefore, with small watermark percentages, it would be hard to detect this kind of cheating behavior using the watermark scheme as we proposed. Considering all those aforementioned limitations, developing a more generic solution for MapReduce applications remains an open question for our future research.

References

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

[2] H. Karloff, S. Suri, and S. Vassilvitskii, "A model of computation for mapreduce," in *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms*. Society for Industrial and Applied Mathematics, 2010.

[3] Hadoop, "Apache hadoop." [Online]. Available: http://hadoop.apache.org

[4] Z. Xiao and Y. Xiao, "Accountable mapreduce in cloud computing," in *SCNC 2011.*

[5] I. Roy, S. Setty, A. Kilzer, V. Shmatikov, and E. Witchel, "Airavat: Security and privacy for mapreduce," in *Proceedings of the 7th USENIX conference on Networked systems design and implementation*. USENIX Association, 2010.

[6] W. Wei, J. Du, T. Yu, and X. Gu, "Securemr: A service integrity assurance framework for mapreduce," in *Computer Security Applications Conference, 2009.*

[7] P. Golle and I. Mironov, "Uncheatable distributed computations," *Topics in Cryptology—CT-RSA 2001.*

[8] D. Szajda, B. Lawson, and J. Owen, "Hardening functions for large scale distributed computations," *In Proceedings of IEEE Symposium on Security and Privacy*, May 2003.

[9] P. Golle and S. Stubblebine, "Secure distributed computing in a commercial environment," in *Financial Cryptography*. Springer, 2002.

[10] D. Szajda, B. Lawson, and J. Owen, "Toward an optimal redundancy strategy for distributed computations," in *Cluster Computing, 2005. IEEE International*. IEEE, 2005.

[11] S. Zhao, V. Lo, and C. Dickey, "Result verification and trust-based scheduling in peer-to-peer grids," in *Proc. Fifth IEEE Intl Conf. Peer-to-Peer Computing (P2P 05), Sept. 2005.*

[12] Wikipedia, "Substitution cipher." [Online]. Available: http://en.wikipedia.org/wiki/Substitution cipher

[13] M. de Kunder, "Daily estimated size of the world wide web." [Online]. Available: www.worldwidewebsize.com

[14] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *Computer Networks and ISDN Systems*, vol. 30, no. 1-7, pp. 107–117, 1998.

[15] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web." Stanford Digital Library Technologies Project, Tech. Rep., 1998.

[16] A. Jamakovic and S. Uhlig, "On the relationships between topological measures in real-world networks," *Networks and Heterogeneous Media*, vol. 3, no. 2, p. 345, 2008.