SecureMR: A Service Integrity Assurance Framework for MapReduce

Wei Wei, Juan Du, Ting Yu, Xiaohui Gu
Department of Computer Science, North Carolina State University
Raleigh, North Carolina, United States
{wwei5,jdu}@ncsu.edu, {gu,yu}@csc.ncsu.edu

***Abstract*—MapReduce has become increasingly popular as a powerful parallel data processing model. To deploy MapReduce as a data processing service over open systems such as service oriented architecture, cloud computing, and volunteer computing, we must provide necessary security mechanisms to protect the integrity of MapReduce data processing services. In this paper, we present SecureMR, a practical service integrity assurance framework for MapReduce. SecureMR consists of five security components, which provide a set of practical security mechanisms that not only ensure MapReduce service integrity as well as to prevent replay and Denial of Service (DoS) attacks, but also preserve the simplicity, applicability and scalability of MapReduce. We have implemented a prototype of SecureMR based on Hadoop, an open source MapReduce implementation. Our analytical study and experimental results show that SecureMR can ensure data processing service integrity while imposing low performance overhead.**

I. Introduction

MapReduce is a parallel data processing model, proposed by Google to simplify parallel data processing on large clusters [1]. Recently, many organizations have adopted the model of MapReduce, and developed their own implementations of MapReduce, such as Google MapReduce [1] and Yahoo's Hadoop [2], as well as thousands of MapReduce applications. Moreover, MapReduce has been adopted by many academic researchers for data processing in different research areas, such as high end computing [3], data intensive scientific analysis [4], large scale semantic annotation [5] and machine learning [6].

Current data processing systems using MapReduce are mainly running on clusters belonging to a single administration domain. As open systems, such as Service-Oriented Architecture (SOA) [7], [8], Could Computing [9] and Volunteer Computing [10], [11], increasingly emerge as promising platforms for cross-domain resource and service integration, MapReduce deployed over open systems will become an attractive solution for large-scale cost-effective data processing services. As a forerunner in this area, Amazon deploys MapReduce as a web service using Amazon Elastic Compute Cloud (EC2) and Amazon Simple Storage Service (Amazon S3). It provides a public data processing service for researchers, data analysts, and developers to efficiently and cost-effectively process vast amounts of data [12]. However, in open systems, besides communication security threats such as eavesdropping attacks, replay attacks, and Denial of Service (DoS) attacks, MapReduce faces a data processing service integrity issue since service providers in open systems may come from different administration domains that are not always trustworthy.

Several existing techniques such as replication (also known as double-check), sampling, and checkpoint-based verification have been proposed to address service integrity issues in different computing environments like Peer-to-Peer Systems, Grid Computing, and Volunteer Computing (e.g., [13]–[19]). Replication-based techniques mainly rely on redundant computation resources to execute duplicated individual tasks, and a master (also known as supervisor) to verify the consistency of results. Sampling techniques require indistinguishable test samples. The checkpoint-based verification focuses on sequential computations that can be broken into multiple temporal segments.

In this paper, we present SecureMR, a practical service integrity assurance framework for MapReduce. SecureMR provides a decentralized replication-based integrity verification scheme for ensuring the integrity of MapReduce in open systems. Our scheme leverages the unique properties of the MapReduce system to achieve effective and practical security protection. First, MapReduce provides natural redundant computing resources, which is amenable to replication-based techniques. Moreover, the parallel data processing of MapReduce mitigates the performance influence of executing duplicated tasks. However, in contrast to simple monolithic systems, MapReduce often consists of many distributed computing tasks processing massive data sets, which presents new challenges to adopt replication-based techniques. For example, it is impractical to replicate all distributed computing tasks for consistency verification purposes. Moreover, it is not scalable to perform centralized consistency verification over massive result data sets at a single point (e.g., the master).

To address these challenges, our scheme decentralizes the integrity verification process among different distributed computing nodes who participate in the MapReduce computation. Our major contributions are summarized as follows:

- We propose a new decentralized replication-based integrity verification scheme for running MapReduce in open systems. Our approach achieves a set of security properties such as non-repudiation and resilience to DoS attacks and replay attacks while maintaining the data processing efficiency of MapReduce.
- We have implemented a prototype of SecureMR based on

Hadoop [2], an open source implementation of MapReduce. The prototype shows that the security components in SecureMR can be easily integrated into existing MapReduce implementations.

- We conduct security analytical study and experimental evaluation of performance overhead based on the prototype. Our analytical study and experimental results show that SecureMR can ensure the service integrity while imposing low performance overhead.

The rest of the paper is organized as follows. We introduce the MapReduce data processing model in Section II. In Section III, we discuss the security vulnerabilities of running MapReduce in open systems, and state assumptions and attack models. Section IV presents the design details of SecureMR. Section V provides the analytical and experimental evaluation results. Section VI compares our work with related work. Finally, the paper concludes in Section VII.

II. Background

As a parallel data processing model, MapReduce is designed to run in distributed computing environments. Figure 1 depicts the MapReduce data processing reference model in such an environment. The data processing model of MapReduce is composed of three types of entities: a distributed file system (DFS), a master and workers. The DFS provides a distributed data storage for MapReduce. The master is responsible for job management, task scheduling and load balancing among workers. Workers are hosts who contribute computation resources to execute tasks assigned by the master. The basic data processing process in MapReduce can be divided into two phases: i) a map phase where input data are distributed to different distributed hosts for parallel processing; and ii) a reduce phase where intermediate results are aggregated together. To illustrate the two-phase data processing model, we use a typical example, WordCount [20] that counts how often words occur. The application is considered as a job of MapReduce submitted by a user to the master. The input text files of the job are stored in the DFS in the form of data blocks, each of which is usually 64MB. The job is divided into multiple map and reduce tasks. The number of map tasks depends on the number of data blocks that the input text files have. Each map task only takes one data block as its input.

During the map phase, the master assigns map tasks to workers. A worker is called a mapper when it is assigned a map task. When a mapper receives a map task assignment from the master, the mapper reads a data block from the DFS, processes it and writes its intermediate result to its local storage. The intermediate result generated by each mapper is divided into r partitions $P1, P2, ..., Pr$ using a partitioning function. The number of partitions is the same with the number of reduce tasks r. During the reduce phase, the master assigns reduce tasks to workers. A worker is called a reducer when it is assigned a reduce task. Each reduce task specifies which partition a reducer should process. After a reducer receives a reduce task, the reducer waits for notifications of map task completion events from the master. Upon notified, the reducer reads its partition from the intermediate result of each mapper who finishes its map task. For example, in Figure 1, RA reads $P1$ from MA, MB and other mappers. After the reducer reads its partition from all mappers, the reducer starts to process them, and finally each reducer outputs its result to the DFS.

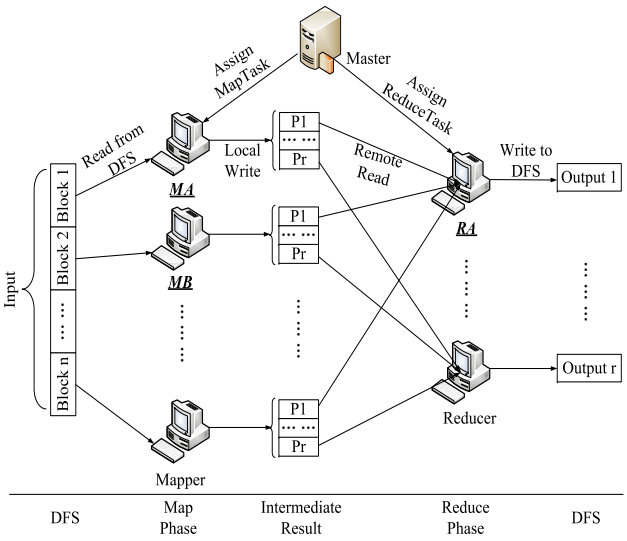


Fig. 1. The MapReduce data processing reference model.

In fact, the MapReduce data processing model supports to combine multiple map and reduce phases into a MapReduce chain to help users accomplish complex applications that cannot be done via a single Map/Reduce job. In a MapReduce chain, mappers will read the output of reducers in the preceding reduce phase, except mappers in the first map phase, which read data from the DFS. Then, the data processing enters into the map phase with no difference from the normal map phase. Similarly, reducers will read intermediate results from mappers in the preceding map phase and generate outputs to DFS or their local disks like what mappers do, which is different from a single Map/Reduce data processing model. For reducers in the middle of data processing, they may store their results in their local disks to improve the overall system performance. Eventually, the final results go into the DFS.

III. System Model

A. MapReduce in Open Systems

MapReduce can be implemented to run in either closed systems or open systems. In closed systems, all entities belong to a single trusted domain, and all data processing phases are executed within this domain. There is no interaction with other domains at all. Thus, security is not taken into consideration for MapReduce in closed systems. However, MapReduce in open systems presents two significant differences:

- The entities in MapReduce come from different domains, which are not always trusted. Furthermore, they may be compromised by attackers due to different vulnerabilities such as software bugs, and careless administration.
- The communications and data transferred among entities are through public networks. It is possible that the

communications are eavesdropped, or even tampered to launch different attacks.

Therefore, before MapReduce can be deployed and operate in open systems, several security issues need to be addressed, including authenticity, confidentiality, integrity, and availability. In this paper, we focus on protecting the service integrity for MapReduce. Since the data processing model of MapReduce includes three types of entities and two phases, to provide the service integrity protection for MapReduce, it naturally boils down to the following three steps:

1) Provide mappers with a mechanism to examine the integrity of data blocks from the DFS.
2) Provide reducers with a mechanism to verify the authenticity and correctness of the intermediate results generated by mappers.
3) Provide users with a mechanism to check if the final result produced by reducers is authentic and correct.

The first step ensures the integrity of inputs for MapReduce in open systems. The second step provides reducers with the integrity assurance for their inputs. The third step guarantees the authenticity and correctness of the final result for users. Finally, the combination of three ensures the MapReduce data processing service integrity to users. Since the first step has been addressed by existing techniques in [21]–[23], we will go through the rest of steps in the following sections.

B. Assumptions and Attack Models

MapReduce is composed of three types of entities: a DFS, a master and workers. The design of SecureMR is built on top of several assumptions that we make on these entities. First, each worker has a public/private key pair associated with a unique worker identifier. Workers can generate and verify signatures, and no worker can forge other's signatures. Second, the master is trusted and its public key is known to all, but workers are not necessarily trusted. Third, a good worker is honest and always returns the correct result for its task while a bad worker may behave arbitrarily. Fourth, the DFS for MapReduce provides data integrity protection so that each node can verify the integrity of data read from the DFS. Fifth, if a worker is good, then others cannot tamper its data (otherwise, the worker is compromised and should be considered as a bad one). Since each worker can have its own access control mechanism to protect data from being changed by unauthorized workers, the assumption is reasonable.

Based on the above assumptions, we concentrate on the analysis of malicious behavior from bad workers. In open systems, a bad worker may cheat on a task by giving a wrong result without computation [13] or tamper the intermediate result to mess up the final result. Moreover, a bad worker may launch DoS attacks against other good workers. For example, it may keep sending requests to a good worker and asking for intermediate results or it may impersonate the master to send fake task assignments to workers. Furthermore, it may initiate replay attacks against good workers by sending old task assignments to keep them busy. In addition, it may eavesdrop and tamper the messages exchanged between two entities so that the final result generated may be compromised. Here, we classify malicious attacks into the following two models:

Non-collusive malicious behavior. Workers behave independently, which means that bad workers do not necessarily agree or consult with each other when misbehaving. A typical example is that, when they return wrong results for the same input, they may return different wrong results.

Collusive malicious behavior. Workers' behavior depends on the behavior of other collusive workers. They may communicate, exchange information, and make an agreement with each other. For example, when they are assigned tasks by the master, they can know if their colluders receive tasks with the same input blocks. If so, they return the same results so that there is no inconsistency among collusive workers. By doing so, they try to avoid being detected even if they return wrong results.

IV. System Design

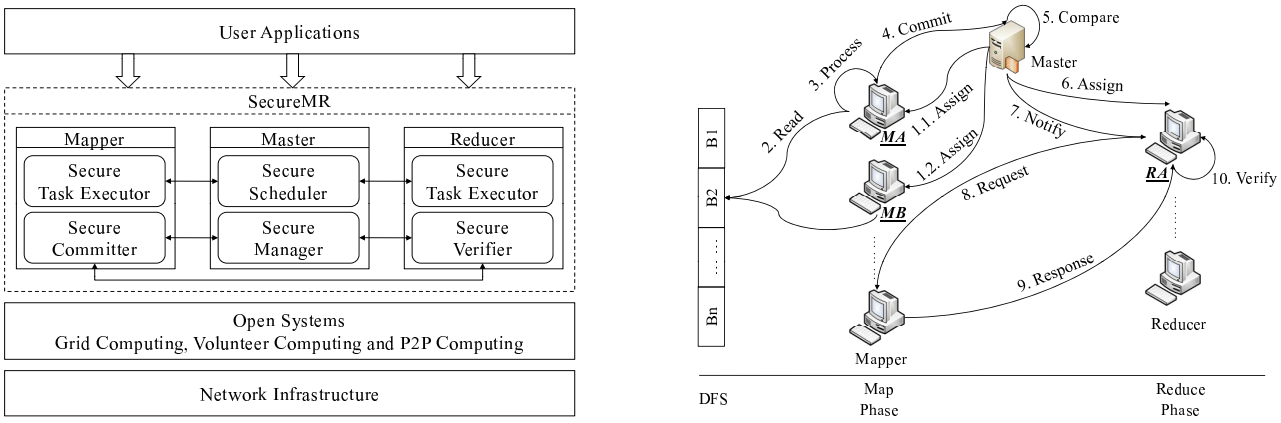
In this section, we present the detailed design of our decentralized replication-based integrity verification scheme.

A. Design Overview

SecureMR enhances the basic MapReduce framework with a set of security components, illustrated by Figure 2. To validate the integrity of map/reduce tasks, our basic idea is to replicate some map/reduce tasks and assign them to different mappers/reducers. Any inconsistent intermediate results from those mappers/reducers reveal attacks. However, due to scalability and efficiency reason, though the master is trusted in our design, consistency verification should not be carried out only by the master. Instead, in our design, this responsibility is further distributed among workers. Our design must ensure properties such as non-repudiation and resilience to DoS and replay attacks, as well as efficiency. Further, our design should preserve the existing MapReduce mechanism as much as possible so that it can be easily implemented and deployed with current MapReduce systems. We introduce the design of SecureMR from two aspects: architecture and communication.

Architecture Design. Figure 2(a) shows the architecture design of SecureMR, which comprises five security components: Secure Manager, Secure Scheduler, Secure Task Executor, Secure Committer and Secure Verifier. They provide a set of security mechanisms: task duplication, secure task assignment, DoS and replay attack protection, commitment-based consistency checking, data request authentication, and result verification.

Secure Manager and Secure Scheduler are deployed in a master mainly for task duplication, secure task assignment, and commitment-based consistency checking. Secure Task Executor is running in both mappers and reducers to prevent DoS and replay attacks that exploit fake or old task assignments. In mappers, Secure Committer takes the responsibility of generating commitments for the intermediate results of mappers and sending them to Secure Manager in the master to complete the commitment-based consistency checking. Secure



(a) SecureMR Architecture Design.

(b) SecureMR Communication Design.

Fig. 2. SecureMR Design Overview.

Verifier running in a reducer collaborates with Secure Manager to verify a mapper's intermediate result. For simplicity, we quote all components using names without Secure in the following sections, for example Manager, Scheduler, Task Executor and so on.

Communication Design. Figure 2(b) shows how the entities in SecureMR communicate with each other to provide security protection for MapReduce. Communications among them are further organized into two protocols: *Commitment* protocol and *Verification* protocol. In Figure 2(b), communications from 1 to 5 form the commitment protocol while communications from 6 to 10 form the verification protocol.

In the commitment protocol, to avoid checking the intermediate results directly (which is expensive), mappers only send commitments (which will be described in detail later) to the master, which can be used to detect inconsistency efficiently. However, this introduces another vulnerability. Mappers may send the master the right commitments but the wrong results to reducers. For this reason, we further ask reducers to check the consistency between the commitment and the result in the verification protocol. Note that this does not add much extra effort to the reducer as it has to retrieve the intermediate result for data processing anyway.

In the following two sections, we will discuss the details of communications between the five security components of SecureMR, which happen in the commitment and verification protocols.

B. Commitment Protocol

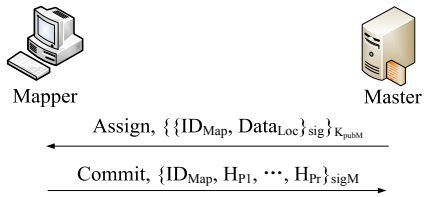


Fig. 3. The Commitment Protocol.

As mentioned in Section III-B, the master is a trusted entity. However, since the intermediate result is usually tremendous, it is impractical to require the master to check all intermediate results generated by different map tasks in different jobs, which will overload the master and lead to low system performance. Thus, instead of examining intermediate results directly, the master requires mappers to generate commitments for their intermediate results, and then check commitments [13].

1) Protocol design: Since we assume that the DFS provides data integrity protection, we do not discuss the communications between mappers and the DFS. Figure 3 shows the communications between a mapper and the master in the commitment protocol. The specific steps are described as follows.

Assign. The Scheduler in the master sends the *Assign* message to the Task Executor in a mapper to assign a map task to the mapper. Regarding task duplication, the Scheduler may assign the same map task to different mappers. For example, in Figure 2(b), MA and MB are assigned the same map task. The *Assign* message includes a monotonically increasing identity ID_{Map} of a map task and an input data block location $Data_{Loc}$, which is signed by the master and encrypted using K_{pubM}, the public key of the mapper. After the Task Executor receives the task assignment message, the Task Executor decrypts and verifies the signature of the message. Then, the Task Executor reads an input block according to $Data_{Loc}$ from the DFS. In Figure 2(b), since MA and MB receive the same task, they both read the same data block $B2$ from the DFS.

Commit. After the mapper processes the input block, the Committer of the mapper makes a commitment to the master by generating a hash value for each partition of its intermediate result and signing those hash values. We use $\{...\}sigM$ to denote a signed message of a mapper. When the Manager of the master receives the commitment, the Manager verifies the signature using the mapper's public key K_{pubM}. If the Manager has received more than one commitments for the same map task from different mappers, the Manager will compare new commitment with an old one to see if they are

consistent with each other.

Note that in this paper, we focus on expose suspicious activities. How to exactly pinpoint malicious ones is the next step and some existing techniques may be applied [24].

2) Protocol analysis: In this protocol, since the task assignment message is signed by the master and encrypted using the mapper's public key, the integrity and confidentiality of the *Assign* message is well protected. It also ensures that the mapper is the only entity that can decrypt the *Assign* message and the master is the only entity that can create it. In this case, malicious mappers cannot know task assignments of other good mappers or arbitrarily assign fake tasks to a mapper to launch DoS attacks. Furthermore, to prevent replay attacks which send old task assignments, a monotonically increasing identity ID_{Map} is associated with each map task, which is automatically generated using timestamp or sequence number by the Scheduler. The Task Executor in the mapper records the ID_{Map} for the last map task that it processed. In this way, the Task Executor can determine if a task assignment is an old one by comparing the ID_{Map} with the latest recorded ID_{Map}. Regarding the *Commit* message, the integrity of the commitment is assured since the *Commit* message is signed using the mapper's private key. Moreover, ID_{Map} is needed so that the master knows which map task this commitment is for.

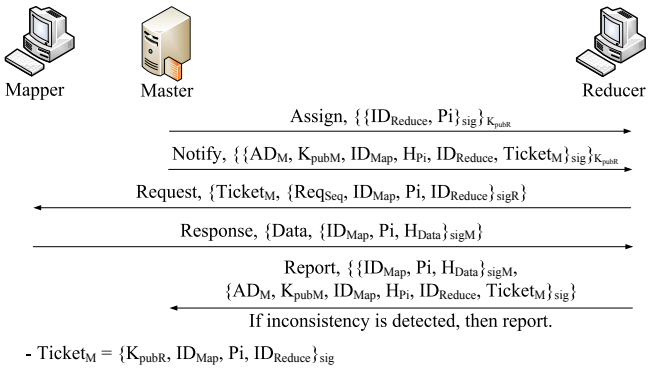


Fig. 4. The Verification Protocol.

C. Verification Protocol

In the verification protocol, reducers further help the master to verify if intermediate results generated by mappers are consistent with commitments submitted to the master. The verification protocol is built on existing MapReduce communication mechanisms. There are no additional messages introduced to MapReduce.

1) Protocol design: Figure 4 shows how the master, a mapper, and a reducer communicate with each other in the verification protocol. We illustrate each step as follows.

Assign. The master signs the *Assign* message and encrypts it using K_{pubR}, the public key of a reducer. In the message, ID_{Reduce} is a monotonically increasing identity of a reduce task, and Pi indicates the partition of intermediate results that the reducer will process. When the Task Executor in the reducer receives the task assignment, the Task Executor first verifies the integrity and authenticity of the task assignment. Then, the Verifier of the reducer will wait for notifications from the Manager.

Notify. When the Manager receives the completion event with a commitment from the Committer of a mapper, the Master sends a notification to the Verifier of each reducer, which includes the mapper's address AD_M, the mapper's public key K_{pubM}, ID_{Map}, the ticket $Ticket_M$ for the mapper signed by the master and the hash value H_{Pi} for the Pi partition committed by the Committer. The ticket $Ticket_M$ is used for data request authentication in the *Request* message.

Request. After the Verifier in a reducer gets notified, the Verifier sends a data request to the Committer of the mapper, which includes the ticket $Ticket_M$ as evidence of an authentic data request authorized by the master, the reducer's public key K_{pubR}, a sequence number Req_{Seq} and Pi which indicates which partition is requested.

Response. After the Committer verifies the authenticity of the request by verifying the ticket from the master and the reducer's signature, the mapper sends a response to the Verifier, which includes ID_{Map}, Pi, the data $Data$ and H_{Data}, the hash value of $Data$. To verify the integrity of the response, the Verifier first verifies the signature in the *Response* message, then regenerates a hash value H'_{Data} for the data, and compares H_{Data} with H'_{Data} to make sure that the data is not tampered during the *Response* communication. Finally, the Verifier compares H'_{Data} with H_{Pi} committed to the master to check if any inconsistency occurs.

Report. When the Verifier detects an inconsistency, the Verifier sends two signatures as evidence to the Manager to report the inconsistency. After the Manager receives and verifies the two signatures, the Manager can compare H_{Data} with H_{Pi} to confirm the reported inconsistency.

2) Protocol analysis: Similar to the commitment protocol, the reduce task assignment mechanism prevents both DoS and replay attacks against reducers. However, in the verification protocol, a mapper faces DoS attacks when others request data from it. To countermeasure this kind of DoS attacks, the mapper needs to authenticate data requests from reducers. The data request authentication is achieved by requiring that a reducer shows a ticket from the master. If the mapper sees a ticket at the first time, the mapper can make sure that the request must come from an authorized reducer who holds the ticket issued by the master. However, if the first attempt of data request fails somehow, attackers may get the ticket by eavesdropping the communications between the mapper and the reducer. In this case, since the mapper will record the latest request sequence number Req_{Seq} associated with a ticket, the mapper will check if this data request is an old one by comparing the two Req_{Seq} numbers when the mapper receives another data request with the same ticket. In this way, replay attacks can be defeated.

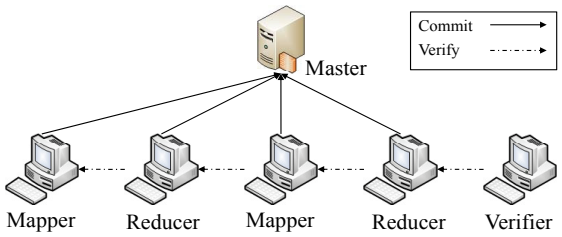


Fig. 5. SecureMR Extension for MapReduce Chain.

D. SecureMR Extension

So far, we have discussed how SecureMR provides reducers with a mechanism to verify the authenticity and correctness of the intermediate results generated by mappers. In this section, we present how SecureMR applies the replication-based verification scheme to reducers and MapReduce chain to provide users with a mechanism to check if the final result produced by reducers is authentic and correct.

Extension for Reducers. Similar to mappers, the Scheduler in the master may duplicate reduce tasks and assign them to multiple reducers. Reducers assigned the same task will read the same partition of the intermediate results from mappers. However, we observe that reducers are not configured with a Secure Committer component in current architecture described in Figure 2(a), which means they cannot make a commitment to the master. In order for reducers to make commitments, we can easily deploy a Secure Committer component for reducers. Another problem to apply the verification scheme to reducers is that there are no other entities to complete the verification protocol since reducers are in the last phase. To address this problem, we extend the MapReduce model to include an additional phase called *Verify* phase. In the verify phase, the master involves several workers with a Secure Verifier component, called verifiers to complete the verification protocol. Another alternative is to install a Secure Verifier component into MapReduce user applications and ask them to complete the verification protocol by themselves after their jobs are done.

Extension for MapReduce Chain. Similarly, the verification scheme can be applied to MapReduce chain since each map and reduce share the similar procedure of data processing. Figure 5 shows the design overview of how SecureMR applies the verification scheme to MapReduce chain. As we can see from the figure, the design is like a *Commit-Verify* chain between the master, mappers and reducers. If mappers make commitments to the master, reducers will take the role of verifiers to verify the consistency between intermediate results and commitments of mappers. If reducers make commitments to the master, mappers will take the role of verifiers to verify the consistency between outputs and commitments of reducers except the last phase, *Verify* phase. The verify phase has been discussed in the above. In order for mappers to be able to fulfill the verification protocol, the only thing that we need to do is to plug a Secure Verifier component into each mapper.

V. Analysis and Evaluation

In this section, we discuss the security properties of SecureMR, and then evaluate the performance overhead both analytically and experimentally. Note that in Section V-A and V-B, we focus on the discussion for mappers due to the similarity of the analysis between mappers and reducers.

A. Security Analysis

There are two kinds of inconsistencies for mappers in MapReduce. One is an inconsistency between results returned by different mappers that are assigned the same task. The other is an inconsistency between the commitment and the result generated by a mapper. The former can only be detected by the master in the commitment protocol and the latter can only be detected by a reducer in the verification protocol. We claim that SecureMR provides the following two properties. We also provide arguments for our statement in the following.

- **No False Alarm.** For any inconsistency detected by SecureMR, it must happen between good and bad mappers, between bad mappers or on a bad mapper. It cannot occur between good mappers or on a good mapper.
- **Non-Repudiation.** For any inconsistency that can be observed by a good reducer or the master, SecureMR can detect it and present evidence to prove it.

Arguments of No False Alarm. The assumptions in Section III-B guarantee that good mappers always produce correct and consistent results. We prove by contradiction that SecureMR provides *No False Alarm* property in terms of the two kinds of inconsistencies.

First, suppose that an inconsistency between two good mappers is detected by the master. In this case, the master must get two different sets of hash values from the commitments of two good mappers, which means that the two commitments the master received must be tampered somehow since two good mappers will not produce inconsistent results. However, if the master accepted a commitment of a mapper, the master must have confirmed the integrity and freshness of the commitment. Thus, the commitment is neither a bad commitment nor an old one. From the arguments, we can infer that there is no way to tamper a commitment of a mapper without being detected by the master. And the hypothesis implies that the master already accepted the commitments, which means it is impossible that the commitments that the master received have been tampered. Therefore, the hypothesis that an inconsistency between two good mappers is detected by the master is not true.

Second, suppose that an inconsistency between the commitment and the intermediate result of a good mapper is detected by a reducer. If the reducer is good, it can be inferred that the message received by the reducer must be tampered somehow. Since the reducer knows ID_{Map} and Pi, the reducer will not accept the message unless the reducer confirms the integrity of the message. ID_{Map} can also be the proof of the freshness of the signatures. For the same reason, it is impossible that the message has been tampered. Thus, the case that an inconsistency on a good mapper is

detected by a good reducer cannot be true. If the reducer is a bad reducer, the reducer can report an inconsistency even if there is no inconsistency. But, the verification protocol requires that the reducer present the evidence to the master, which is described in Figure 4. And the reducer cannot forge evidence without being detected by the master. Hence, the case that an inconsistency on a good mapper is detected by a bad reducer cannot be true, either. Therefore, the hypothesis that an inconsistency on a good mapper is detected by a reducer is not true.

Arguments of Non-Repudiation. We prove by contradiction that SecureMR provides *Non-Repudiation* property in terms of the two kinds of inconsistencies. Suppose that an inconsistency is observed by the master or a good reducer. Both the master and the good reducer definitely report the inconsistency since they both tell the truth. Meanwhile, the master holds the commitments of workers, which cannot be denied, and the good reducer has the signatures of mappers. They both can present the commitments or the signatures of mappers to prove the inconsistency they detect. Thus, SecureMR provides the *Non-Repudiation* property in terms of the two kinds of inconsistencies.

B. Attacker Behavior Analysis

We analyze the behavior of the following attackers under the two kinds of behavior models defined in Section III-B. When we analyze the collusive attacks, we consider the worst case that all malicious entities are colluding with one another.

- Periodical Attackers: they misbehave with a certain probability p_m. Since a naive attacker is a special case of periodical attacker with p_m equal to 1. Thus, we discuss these two kinds of attacker's behavior together.
- Strategic Attackers: with the assumption that they know the duplication strategy, they may not behave maliciously until they definitely know that they will not be caught due to the collusion, which means that all duplicates are assigned to the collusive group.

Definition V.1. *(Detection Rate) We define the detection rate, denoted D_{rate}, as the probability that the inconsistency between results caused by the misbehavior of a mapper is detected during l jobs.*

Note that due to the paper space limit, we do not discuss the inconsistency between the commitment and the result of a mapper.

Since each map task processes one block, the duplication of a map task is the same as the duplication of a block. The following discussion may use both terms, block duplication and map task duplication exchangeably. Suppose MapReduce consists of one master and n workers, and m out of n workers ($m < n$) are malicious workers. For simplicity, we assume that the input of each job has the same number of blocks b, no two blocks are the same and each worker only processes one task in one job. The percentage of blocks that will be duplicated in each job is p_b. Thus, the number of duplicated blocks is $b \cdot p_b$. SecureMR randomly chooses one block from the original b blocks to duplicate for each duplication. It uses a *naive task scheduling* algorithm, which launches all map tasks together, including duplicated map tasks. In the following, we analyze the detection rate for periodical attacker without and with collusion, and the probability that strategic attackers can misbehave in a job.

Periodical attackers without collusion. For simplicity, we assume that they return different results when they misbehave on the same input. Thus, without collusion, the detection rate of a malicious mapper is the same as the probability that the block processed by the mapper is duplicated. Therefore, the detection rate is calculated as follows:

$$D_{rate} = 1 - (1 - (1 - 1/b)^{b \cdot p_b}) \cdot p_m)^l \tag{1}$$

In Equation 1, $(1-(1-1/b)^{b \cdot p_b}) \cdot p_m$ denotes the probability that the misbehavior of the malicious mapper is detected during one job. Figure 6 shows detection rate for a naive attacker without collusion, where b is equal to 20 and l is 5, 10 and 15. Figure 7 shows detection rate for a periodical attacker with 0.5 misbehaving probability. Both of them demonstrate that as the number of tasks that a malicious mapper processes increases, high detection rate can be achieved even if the duplication rate is only 20%, which means that the chance for an attacker to cheat without being detected in the long run is very low.

Periodical attackers with collusion. With collusion, the maximum number of entities that collude with each other is m. Let $P(B_i)$ denote the probability that a block will be duplicated i times and $P(D)$ denote the probability that the inconsistency caused by the misbehavior of a malicious mapper will be detected. In this case, the detection rate is:

$$\begin{aligned} D_{rate} &= 1 - (1 - \sum_{i=0}^{b \cdot p_b} P(D|B_i) \cdot P(B_i))^l \\ &= 1 - (1 - \sum_{i=0}^{b \cdot p_b} P(D|B_i) \cdot \binom{b \cdot p_b}{i} (\frac{1}{b})^i \cdot (1 - \frac{1}{b})^{b \cdot p_b - i})^l \end{aligned} \tag{2}$$

where

$$P(D|B_i) = \begin{cases} 0 & \text{if } i = 0, \\ (1 - \binom{m-1}{i} / \binom{n-1}{i}) \cdot p_m & \text{if } i > 0 \text{ and } i < m, \\ p_m & \text{if } i >= m. \end{cases}$$

In Equation 2, the detection rate is computed using the law of total probability. The inconsistency cannot be detected only if all duplicates for the block that the malicious mapper processes are assigned to its collusive parties. $P(D|B_i)$ is the probability that the inconsistency is detected when the block that the malicious mapper processes is duplicated i times. If $i >= m$, at least one duplicate will not be assigned to its collusive parties. Figure 8 shows how the detection rate changes as the duplication rate and the percentage of malicious workers change given n, p_m, b, l equals to 50, 0.5, 20 and 15, respectively. From the figure, we observe that as long as the majority of workers are good, 90% detection rate can be achieved with 40% duplication rate.

Strategic attackers. Since the misbehavior of attackers cannot be detected, we discuss the probability $P(F)$ that the

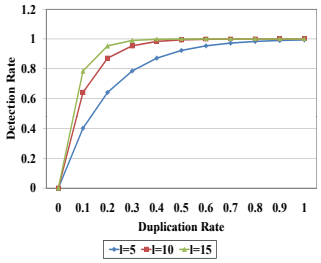


Fig. 6. Detection Rate for Non-Collusion Naive Attacker.

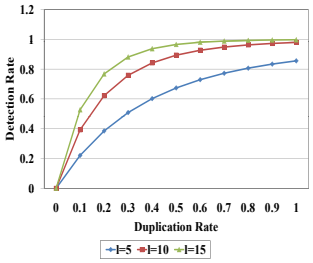


Fig. 7. Detection Rate for Non-Collusion Periodical Attacker.

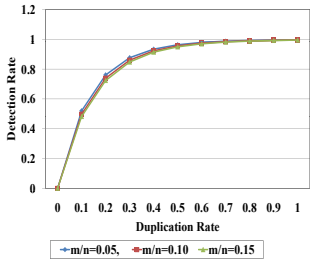


Fig. 8. Detection Rate for Collusion Periodical Attacker.

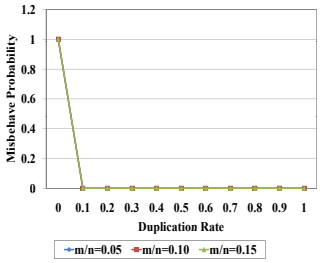


Fig. 9. Misbehaving Probability vs Duplication Rate.

intermediate result that reducers receive is tampered, which is the same as the misbehaving probability of a strategic attacker. In this case, we analyze the strategic attacker's behavior in the following two steps:

1) The master assigns b input blocks to b mappers before any duplication is made.
2) The master duplicates $b \cdot p_b$ input blocks after assignments for the original b blocks. For each duplication, the master randomly chooses one block from the original b blocks to duplicate.

Therefore, $P(F)$ can be calculated by the following formula:

$$P(F) = \sum_{i=0}^{x} P(F|E_i) \cdot P(E_i) = \sum_{i=0}^{x} P(A_i) \cdot P(M_i) \cdot P(E_i)$$
$$= \sum_{i=0}^{x} (i/b)^{b \cdot p_b} \cdot (\binom{m-i}{b \cdot p_b} / \binom{n-b}{b \cdot p_b}) \cdot \binom{m}{i} \cdot \binom{n-m}{b-i} / \binom{n}{b} \quad (3)$$

where

$$x = \begin{cases} m & \text{if } m < b, \\ b & \text{if } m >= b. \end{cases}$$

Note that E_i and $P(E_i)$ denote the event that mappers contain i collusive mappers before input block duplication and the probability that E_i happens, respectively. $P(F|E_i)$ denotes the probability that the result is tampered by some mappers when E_i occurs. $P(A_i)$ and $P(M_i)$ denote the probability that all duplicated blocks $b \cdot p_b$ belong to the set of blocks that the i collusive mappers process and the probability that all duplicated blocks $b \cdot p_b$ are assigned to the rest of m_i's collusive workers. Figure 9 shows the misbehaving probability of a strategic attacker when duplication rate and the percentage of malicious workers change, where n, b, l equals to 50, 20 and 15, respectively. The result implies that the misbehaving probability of a strategic attacker is pretty low even if the duplication rate is only 10%.

Since strategic attackers can exchange information of tasks with their collusive entities when they decide whether or not to cheat in tasks, sometimes they can misbehave without being detected. In order to address this vulnerability, we propose a *commitment-based task scheduling* algorithm. Basically, the *commitment-based task scheduling* algorithm will launch the duplicates of a task only after the task has been committed. In this case, when a strategic attacker initially processes a task, there is no way for it to know any duplication information about the task that it handles because no duplicated tasks have been assigned yet. Later when its collusive entities receive the duplicated tasks, they need to return the same results with the initial result. Otherwise, inconsistency will be produced, which can be detected by the master. Thus, the strategic attacker cannot misbehave because it is always possible that the misbehavior could be detected as long as there are duplicated tasks. However, intuitively, it delays the execution of duplicated tasks, which may bring down the performance of the system. In the following section, we will evaluate the performance overhead of SecureMR under both the *naive task scheduling* algorithm and the *commitment-based task scheduling* algorithm.

C. Experimental Evaluation

System Implementation. We have implemented a prototype of SecureMR based on one existing implementation of MapReduce, Hadoop [2]. In our prototype, we have implemented both *naive task scheduling* algorithm and *commitment-based task scheduling* algorithm mentioned in previous sections. Regarding consistency verification, we have implemented a *non-blocking* replication-based verification scheme, which means that reducers do not need to wait for all duplicates of a map task to finish and users do not need to wait for all duplicates to finish. Finally, users will be informed if an inconsistency is detected after all duplicates finish.

Experiment Setup. We run our experiments on 14 hosts provided by Virtual Computing Lab (VCL), a distributed computing system with hundreds of hosts connected through campus networks [25]. The Hadoop Distributed File System (HDFS) is also deployed in VCL. We use 11 hosts as workers that offer MapReduce services and one host as a master, and HDFS uses 13 nodes, not including the master host. We adopt the duplication strategy discussed in Section V-B. All hosts used have similar hardware and software configurations (2.66GHz Intel Intel(R) Core(TM) 2 Duo, Ubuntu Linux 8.04, Sun JDK 6 and Hadoop 0.19). All experiments are conducted by using Hadoop WordCount application [20].

Performance Analysis. First, we estimate the additional overhead introduced by SecureMR in Table I and II. Table I shows the performance overhead of SecureMR on the master, a mapper and a reducer. Table II shows the additional bytes to be transmitted on each communication between them. Note that there are no additional messages introduced. Here, T and

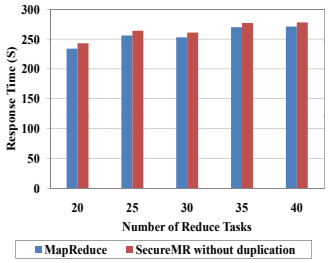


Fig. 10. Response Time vs Number of Reduce Tasks.

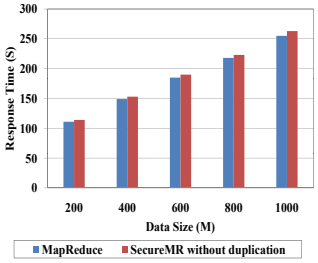


Fig. 11. Response Time vs Data Size.

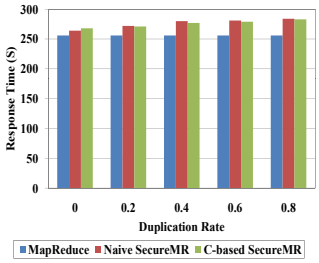


Fig. 12. Response time vs Duplication Rate.

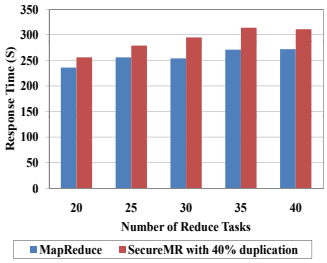


Fig. 13. Response time vs Number of Reduce Tasks.

Type	Cost Estimation	Estimated Time
Master	$4 \cdot T_{sig} + 3 \cdot T_{Epub} + T_{ver}$	$20ms$
Mapper	$2 \cdot T_{sig} + T_{Dpub} + 3 \cdot T_{ver} + r \cdot T_{hash}$	$14 + (r+1) \cdot 40ms$
Reducer	$2 \cdot T_{Dpub} + 3 \cdot T_{ver} + T_{hash}$	$51ms$

TABLE I
PERFORMANCE OVERHEAD ON ENTITIES

Type	Cost Estimation	Additional Bytes
Master-Mapper	$2 \cdot D_{sig} + r \cdot D_{hash}$	$256 + r * 20bytes$
Master-Reducer	$3 \cdot D_{sig} + D_{hash} + D_{pub}$	$532bytes$
Mapper-Reducer	$3 \cdot D_{sig} + D_{hash} + D_{pub}$	$532bytes$

TABLE II
COMMUNICATION OVERHEAD BETWEEN ENTITIES

D denote the time and data transmission cost for different secure operations such as encryption, decryption, signature, verification and hash. r is the number of reducers. The size of each partition is around 14MB. We use SHA-1 to generate hash values, and RSA to create signature or encrypt/decrypt data. The estimation shows that the cost of communication is negligible and the cost on each entity is small.

We also conduct experiments to evaluate the performance overhead caused by SecureMR. Figure 10 shows the response time versus the number of reduce tasks under two scenarios, MapReduce and SecureMR without duplication, where the number of map tasks is 60 and the data size is 1GB. The result shows that the overhead of SecureMR is below 10 seconds, which is small compared with the response time which is about 250 seconds. Figure 11 shows the response time versus the data size, where the number of map tasks is 60 and the number of reduce tasks is 25. Since the data size only affects the time to generate hash values, it shows a similar overhead in Figure 10.

Regarding the performance overhead by executing duplicated tasks, we compare the response time in three cases: MapReduce, SecureMR with naive scheduling, and SecureMR with commitment-based scheduling. Figure 12 shows the response time versus the duplication rate. Since we adopts a *non-blocking* verification mechanism, the difference between two scheduling algorithms is very small. The result shows that the time overhead increases slowly with the increase of duplication rate. Figure 13 shows the response time versus the number of reduce tasks under the two scenarios, MapReduce and SecureMR with 40% duplication rate, where the number of map tasks is 60 and the data size is 1GB. Compared with the no-duplication case in Figure 10, the performance overhead caused by executing duplicated tasks ranges from 5% to 12%.

VI. RELATED WORK

MapReduce recently has received a great amount of attention for its simple model and parallel computation capability for data intensive computation in different application and research areas. Chu et al. [6] applied MapReduce to the multicore computation for machine learning. Ekanayake et al. [4] applied MapReduce technique for two scientific analyses, High Energy Physics data analyses and Kmeans clustering. Mackey et al. [3] utilized MapReduce for High End Computing applications. Most of them focus on how to utilize MapReduce to solve issues or problems in specific application domains. Few work pays attention to the service integrity protection in MapReduce. SecureMR provides a set of practical security mechanisms to ensure MapReduce data processing service integrity.

Service integrity issues addressed in this paper also share similarity with the problem addressed in [13]–[19]. Du et al. [13] used sampling techniques to achieve efficient and viable uncheatable grid computing. Zhao et al. [14] proposed a scheme called Quiz to combat collusion for result verification. Sarmenta et al. [15] introduced majority voting, and spot-checking techniques, and presented credibility-based fault tolerance. Although several existing techniques have been proposed to address the service integrity issues in different application areas [11], [13], [26], the integrity assurance for MapReduce data processing service presents its unique challenges like massive data processing and multi-party distributed computation. SecureMR adopts a new decentralized replication-based integrity verification scheme to address these new challenges, which fully utilizes the existing architecture of MapReduce.

Regarding system security, Srivatsa and Liu proposed a suite of security guards and a resilient network design to secure content-based publish-subscribe systems [27]. PeerReview [28] system ensures that Byzantine faults observed by a correct node are eventually detected and irrefutably linked to a faulty node in a distributed messaging system. Swamynathan

et. al. proposed a scheme to improve the accuracy of reputation systems using a statistical metric to measure the reliability of a peer's reputation [29]. Different from previous works, SecureMR is based on a trustworthy master and leverages natural redundancy of map and reduce services and existing MapReduce data processing mechanisms to perform comprehensive consistency verification.

VII. Conclusion and Future Work

In this paper, we have presented SecureMR, a practical service integrity assurance framework for MapReduce. We have implemented a scalable decentralized replication-based verification scheme to protect the integrity of MapReduce data processing service. To the best of our knowledge, our work makes the first attempt to address this problem. Based on Hadoop [2], we have implemented a prototype of SecureMR, proved its security properties, evaluated the performance impact resulted from the proposed scheme, and tested it on a real distributed computing system with hundreds of hosts connected through campus networks. Our initial experimental results show that the proposed scheme can ensure data processing service integrity while imposing low performance overhead.

However, although SecureMR provides an effective way to detect misbehavior of malicious workers, it is impossible to detect any inconsistency when all duplicated tasks are processed by a collusive group. In order to counter this collusion attack, we may resort to sampling techniques. We believe that the unique properties of MapReduce may bring new opportunities and challenges to adopt such new techniques.

Acknowledgment

This work is supported by the U.S. Army Research Office under grant W911NF-08-1-0105 managed by NCSU Secure Open Systems Initiative (SOSI) and by the NSF under grant IIS-0430166. The contents of this paper do not necessarily reflect the position or the policies of the U.S. Government.

References

[1] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," in *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*. Berkeley, CA, USA: USENIX Association, 2004, pp. 10–10.

[2] "Hadoop Tutorial," *http://public.yahoo.com/gogate/hadoop-tutorial/start-tutorial.html*.

[3] G. Mackey, S. Sehrish, J. Lopez, J. Bent, S. Habib, and J. Wang, "Introducing mapreduce to high end computing," in *Petascale Data Storage Workshop Held in conjunction with SC08*, 2008.

[4] J. Ekanayake, S. Pallickara, and G. Fox, "Mapreduce for data intensive scientific analysis," in *eScience, 2008. eScience '08. IEEE Fourth International Conference on*, 2008, pp. 277–284.

[5] M. Laclavík, M. Šeleng, and L. Hluchý, "Towards large scale semantic annotation built on mapreduce architecture," in *ICCS '08: Proceedings of the 8th international conference on Computational Science, Part III*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 331–338.

[6] C. T. Chu, S. K. Kim, Y. A. Lin, Y. Yu, G. R. Bradski, A. Y. Ng, and K. Olukotun, "Map-reduce for machine learning on multicore," in *NIPS*, B. Schölkopf, J. C. Platt, and T. Hoffman, Eds. MIT Press, 2006, pp. 281–288. [Online]. Available: http://dblp.uni-trier.de/rec/bibtex/conf/nips/ChuKLYBNO06

[7] G. A. amd F. Casati, H. Kuno, and V. Machiraju, "Web Services Concepts, Architectures and Applications Series: Data-Centric Systems and Applications," *Addison-Wesley Professional*, 2002.

[8] T. Erl, "Service-Oriented Architecture (SOA): Concepts, Technology, and Design," *Prentice Hall*, 2005.

[9] "Amazon Elastic Compute Cloud," http://aws.amazon.com/ec2/.

[10] D. P. Anderson, "Boinc: a system for public-resource computing and storage," 2004, pp. 4–10. [Online]. Available: http://dx.doi.org/10.1109/GRID.2004.14

[11] "SETI@home." *http://setiathome.ssl.berkeley.edu/*.

[12] "Amazon Elastic MapReduce," http://docs.amazonwebservices.com/Elastic-MapReduce/latest/DeveloperGuide/index.html.

[13] W. Du, J. Jia, M. Mangal, and M. Murugesan, "Uncheatable grid computing," in *ICDCS '04: Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 4–11.

[14] S. Zhao, V. Lo, and C. GauthierDickey, "Result verification and trust-based scheduling in peer-to-peer grids," in *P2P '05: Proceedings of the Fifth IEEE International Conference on Peer-to-Peer Computing*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 31–38.

[15] L. F. G. Sarmenta, "Sabotage-tolerance mechanisms for volunteer computing systems," *Future Generation Computer Systems*, vol. 18, no. 4, pp. 561–572, 2002. [Online]. Available: citeseer.ist.psu.edu/sarmenta02sabotagetolerance.html

[16] C. Germain-Renaud and D. Monnier-Ragaigne, "Grid result checking," in *CF '05: Proceedings of the 2nd conference on Computing frontiers*. New York, NY, USA: ACM, 2005, pp. 87–96.

[17] P. Domingues, B. Sousa, and L. Moura Silva, "Sabotage-tolerance and trust management in desktop grid computing," *Future Gener. Comput. Syst.*, vol. 23, no. 7, pp. 904–912, 2007.

[18] P. Golle and S. Stubblebine, "Secure distributed computing in a commercial environment," in *5th International Conference Financial Cryptography (FC*. Springer-Verlag, 2001, pp. 289–304.

[19] P. Golle and I. Mironov, "Uncheatable distributed computations," in *CT-RSA 2001: Proceedings of the 2001 Conference on Topics in Cryptology*. London, UK: Springer-Verlag, 2001, pp. 425–440.

[20] "WordCount, Hadoop," http://wiki.apache.org/hadoop/WordCount.

[21] M. J. Atallah, Y. Cho, and A. Kundu, "Efficient data authentication in an environment of untrusted third-party distributors," in *ICDE '08: Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 696–704.

[22] K. Fu, M. F. Kaashoek, and D. Mazières, "Fast and secure distributed read-only file system," *ACM Trans. Comput. Syst.*, vol. 20, no. 1, pp. 1–24, 2002.

[23] P. Devanbu, M. Gertz, C. Martel, and S. G. Stubblebine, "Authentic third-party data publication," in *In Fourteenth IFIP 11.3 Conference on Database Security*, 1999, pp. 101–112.

[24] Q. Zhang, T. Yu, and P. Ning, "A framework for identifying compromised nodes in wireless sensor networks," *ACM Trans. Inf. Syst. Secur.*, vol. 11, no. 3, pp. 1–37, 2008.

[25] "Virtual Computing Lab," http://vcl.ncsu.edu/.

[26] D. Szajda, B. Lawson, and J. Owen, "Toward an optimal redundancy strategy for distributed computations," in *Cluster Computing, 2005. IEEE International*, Sept. 2005, pp. 1–11.

[27] M. Srivatsa and L. Liu, "Securing publish-subscribe overlay services with eventguard," in *CCS '05: Proceedings of the 12th ACM conference on Computer and communications security*. New York, NY, USA: ACM, 2005, pp. 289–298.

[28] A. Haeberlen, P. Kouznetsov, and P. Druschel, "Peerreview: practical accountability for distributed systems," in *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*. New York, NY, USA: ACM, 2007, pp. 175–188. [Online]. Available: http://dx.doi.org/10.1145/1294261.1294279

[29] G. Swamynathan, B. Zhao, K. Almeroth, and S. Jammalamadaka, "Towards reliable reputations for dynamic networked systems," in *Reliable Distributed Systems, 2008. SRDS '08. IEEE Symposium on*, Oct. 2008, pp. 195–204.