Toward Detecting Compromised MapReduce Workers through Log Analysis

Eunjung Yoon
Department of Computer Science and Engineering
Pennsylvania State University
University Park, USA
eyoon@cse.psu.edu

Anna Squicciarini
College of Information Sciences and Technology
Pennsylvania State University
University Park, USA
asquicciarini@ist.psu.edu

***Abstract*—MapReduce is a framework for performing data-intensive computations in parallel on commodity computers. When MapReduce is carried out in distributed settings, users maintain very little control over these computations, causing several security and privacy concerns. MapReduce activities may be subverted or compromised by malicious or cheating nodes. In this paper, we focus on the analysis and detection of attacks launched by malicious or misconfigured nodes, which may tamper with the ordinary functions of the MapReduce framework. Our goal is to investigate the extent to which integrity and correctness of computation in a MapReduce environments can be verified while introducing no modifications on the original MapReduce operations or introductions of extra operations, neither computational nor cryptographic. We identify a number of data and computation integrity checks against aggregated low-level system traces and Hadoop logs, correlated with one another to obtain insights on the operations being performed by nodes. This information is then matched against system and program invariants to effectively detect malicious activities, from lazy nodes to nodes changing input/output or completing different computations.**

I. Introduction

The MapReduce computing paradigm is an architectural and programming model for efficiently processing massive amount of raw data in a parallel and distributed manner [5]. With the recent proliferation of cloud computing services, the MapReduce framework has become a very popular approach to process Big Data in distributed environments. It provides seamless distribution of computing tasks among computing nodes, in a transparent way to programmers. Its current design allows users' data to be efficiently processed in multiple parallel tasks, with little control from the end users. Considering its benefits, MapReduce has been widely adopted by a number of large companies, such as Google, Yahoo, Amazon, Facebook and AOL. An open source implementation of MapReduce called Hadoop [11], developed by Yahoo has further encouraged wide adoption of MapReduce.

Despite the achieved flexibility and great MapReduce degree of scalable computing offered to its clients, MapReduce is vulnerable to third-party attacks and misbehavior. Some participants, be them users or nodes, can be subverters that deceptively perpetrate cybercrime or other cyber attacks [29], [30] using distributed computational resources. Further, due to the little control users have over the processes and functions carried out by the nodes on users' data, data miscalculation may easily go undetected. This is especially true when lazy or malicious servers are involved in a MapReduce task.

In this paper, we focus on the analysis and detection of attacks launched by malicious or misconfigured nodes, which may tamper with the ordinary functions of the MapReduce framework.

Our objective is to investigate the extent to which integrity and correctness of computation in a MapReduce environment can be verified using a black box approach, with no modification of the original MapReduce operations or introduction of extra operations, neither computational or cryptographic. We note that there is a significant amount of work tackling the problem of verification of remote computation. Common approaches rely on replication [12], [29], cryptographic proofs [19], [20], [2], or attestation protocols [30], [23], [28]. Most of such approaches, although effective, require modification of either the original MapReduce operation flow (e.g. [22]), or they require modification of the submitted MapReduce tasks, to include checks or computation of cryptographic attestations [3], [10]. Our challenge is instead to devise alternative solutions that do not require such intrusive modifications.

To this end, we propose a new approach to carry out semantic analysis of system calls and MapReduce logs as a novel way of detecting misuse and attacks in MapReduce frameworks. As noted in previous work [14], syscall event-streams present a rich source of statistical and semantic information for performance and diagnosis in MapReduce frameworks. In this work, we push this concept forward, and hypothesize that system calls can be leveraged to uncover specific attacks to MapReduce nodes, due to improper execution of the MapReduce framework or to incorrect or tampered execution of the MapReduce jobs tasked by clients. We therefore conduct an in-depth analysis of execution traces of parallel nodes to verify the execution pattern followed by a mapper or a reducer in a Hadoop environment. Execution trace recording through dynamic instrumentation helps identify the flow of execution of MapReduce applications.

We highlight how, given an exact trace of execution of an entire system, collected from multiple nodes, it is possible to deduce the operations that took place. Such information can even be used to demonstrate that some operation did not happen, which is a highly desirable property to ensure information assurance. Further, we identify a set of invariants that can help form a baseline behavior of both the Hadoop framework and of the applications. We correlate Hadoop logs with specific system calls, and match them against the identified invariants. We then identify whether a malicious node has subverted the functioning of the Map Reduce operations, or otherwise altered the original workflow of the computations.

The rest of the paper is organized as follows. Next section highlights related work. In Section III, we provide background information of our approach and discuss goal and assumptions taken in this work. In Section IV, we discuss our instrumentation techniques for inspecting MapReduce execution traces. Next, we present our approach toward inspection of MapReduce traces, in Section V. We continue in Section VI with discussing program invariants, followed in Section VII with our approach toward verifying the invariants using the identified invariants. In Section VII, we discuss experimental evaluation, and conclude the paper in Section VIII.

II. Related Work

There is a significant amount of work tackling the problem of verification of computation, in various computing applications, from Desktop Grids to Cloud Computing. Common approaches rely on replication [12], [29], [30], voting schemes, cryptographic proofs [19], [20], [2], or attestation protocols (e.g. [10]). Below we summarize some interesting work, carried out in the context of distributed systems, and more recently in MapReduce. Du et al. [7] used sampling techniques to achieve efficient and viable uncheatable grid computing. Zhao et al. [33] proposed a scheme to address collusion for result verification. Sarmenta et al. [24] introduced majority voting, and spot-checking techniques, to reduces the error rate linearly with the amount checks to be performed. [26] employs a replication-based scheme that allows the degree of redundancy to be adaptively adjusted based on the dynamically-calculated reputation as well as reliability of each worker.

Specific to MapReduce, and closer to our work, is the SecureMR framework [30], which adopts a new decentralized replication-based integrity verification scheme, and utilizes the existing architecture of MapReduce for replication purposes. On a similar vein, Xiao et al. [31] presented an accountable MapReduce platform which checks all working machines and detects malicious nodes in real time. Auditors are able to generate verifiable evidence once inconsistency occurs, by replicating the tasks executed by workers and matching output with the original results. Another significant contribution is the Airavat system proposed by Roy et al. [22]. Airavat is a security and privacy framework for MapReduce systems. Airavat aims to enforce differential privacy, i.e. it aims to ensure that the output of aggregate computations does not violate the privacy of individual inputs. It does so by modifying the Java Virtual Machine and the MapReduce framework by adding SELinux-like Mandatory Access Control to the DFS.

Most of the above mentioned studies are based on replication techniques, or require changing the MapReduce configuration to increase the security guarantees being offered. Tasks are either being duplicated and sent to two or more different participants for processing, or cryptographic checks are required to provide proofs of correctness of execution. Such redundant operations waste resources in the system and although accurate, are often impractical. We remove the constraints that extra operations have to be performed and investigate how careful analysis of correlated and properly parsed logs can help detect malicious activities.

In addition to work on verification and integrity of computation, there is a large body of work on detection of system errors by means of log analysis (e.g. [9], [32], [27]). These works focus on mining and statistical learning techniques applied on system logs to determine possible misconfigurations or computing issues. Xu et al [32], for example, use a two stage approach based on frequent pattern mining and distribution estimation techniques to capture the dominant patterns. The authors' empirical analysis shows high detection accuracy of system errors and also uncovers patterns of execution on a given cluster of nodes. We differ in approach and goal, as our focus is on intentional malicious actions, rather than system errors. Also note that mining techniques would not be enough to detect malice, as the client-provided program is unknown until being submitted to MapReduce framework. Thus it is almost impossible (or very impractical) to train the program for classifying normal or abnormal behavior.

III. Background and Problem Statement

A. MapReduce Framework

MapReduce is a functional programming paradigm [6]. It enables parallel programming of large data efficiently using multiple nodes. Its programming model is built upon a distributed file system (DFS) which provides distributed storage (see Figure 1). Programmers specify two functions: *Map* and *Reduce*. The Map function receives a key/value pair as input and generates intermediate key/value pairs to be further processed. The Reduce function merges all the intermediate key/value pairs associated with the same (intermediate) key and then generates final output. On a cloud computing setting, these functions are orchestrated by the Master node. The Master acts as the coordinator responsible for task scheduling, job management, etc. The *mappers*, and *reducers* nodes, carry out the functions. Next, we provide a brief review of this model in the context of Hadoop, since it is the reference architecture used in our study.

1) Hadoop: The Hadoop framework is a popular example of the MapReduce paradigm. Hadoop is based on a master/slave model. The master node runs the JobTracker, TaskTracker, NameNode and DataNode, whereas a slave node can run the TaskTracker and DataNode. The JobTracker is responsible for distributing MapReduce tasks to worker/slave nodes and keeping track of them. The TaskTracker is responsible for running the job accepted from the JobTracker on the node. The TaskTracker spawns a new JVM for each task received and then monitors the progress of this spawned process, capturing its output and exit codes.

One TaskTracker runs on each slave node and TaskTracker spawns multiple child JVMs to handle many map or reduce tasks in parallel on each slave node.

2) Hadoop Distributed Filesystem: By default, Hadoop comes with the Hadoop Distributed Filesystem (HDFS), which is a distributed, scalable filesystem designed to scale to petabytes of data while running on top of the underlying filesystem of the operating system. User applications access

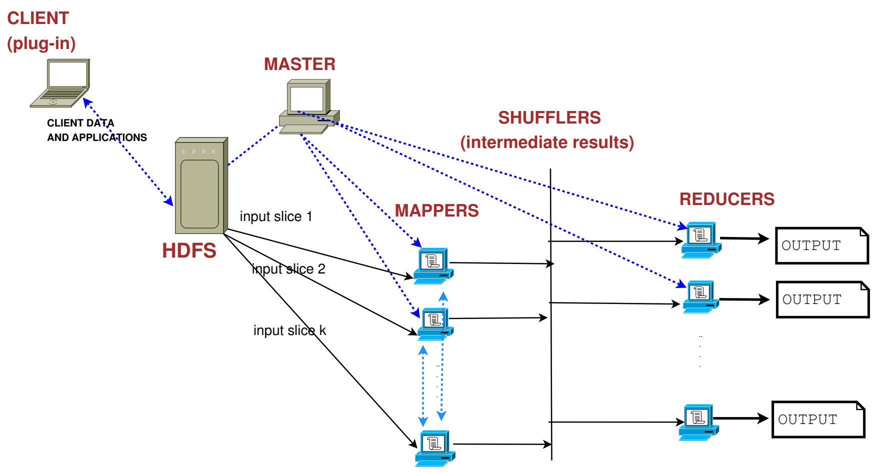

Fig. 1: Overview of the main MapReduce framework

the file system using the HDFS client. HDFS is rack-aware, meaning it keeps track of where the data resides in a network by associating with the data set the name of its rack. This allows Hadoop to efficiently schedule tasks to those nodes that contain data (or which are nearest to the data) in order to optimize bandwidth utilization.

NameNode and DataNode are part of HDFS. The NameNode is responsible for maintaining the directory structure of the filesystem (i.e., metadata) and to track where the file data is kept in the Hadoop cluster (i.e., DataNode). The NameNode is a single point of failure in HDFS; if it fails, the whole filesystem will come down (support for a secondary NameNode is present). It does not, however, store any data itself. The DataNode is where data is actually stored. An optimal cluster has multiple DataNodes that store data across multiple locations for increased reliability.

B. Goal

Our main goal is to investigate the extent to which it is possible to detect any cheating or malicious behavior of worker nodes by monitoring execution behavior at runtime on the nodes. Cheating nodes are nodes which skip part or all the requested computation to save computational resources. Malicious nodes are nodes which deliberately subvert the computation in order to obtain some selfish gain [18]. For instance, they may execute additional computational tasks to charge the client, or they may attempt to obtain the input or output data of the user.

Note that we *do not* aim to provide a data-level program verification technique, as it would be impractical. We also do not aim to zoom in on unknown attacks that involve hidden or compromised system calls; for instance, mimicry attacks which try to masquerade as normal behavior by modifying detectable system call sequences are beyond the scope of this work. Further, an attacker can create the wrong result without causing any system failure, so we do not aim to detect faulty nodes. Faulty behavioral patterns have been studied in previous work [32], [27], and are beyond the scope of our work.

C. Assumptions

We rely on the following assumptions.

1) MapReduce framework is not tampered with. In particular, the Hadoop system, in terms of scheduling, assigning jobs (map and reduce jobs) to workers, and collecting the results has not been compromised. We assume workers compute the same MapReduce tasks at the same time on portions of the same input data.
2) Worker nodes have similar hardware configuration. This assumption is consistent with common real-world applications, as most instances of MapReduce clusters are designed to have the same technical specifications. We tried making trace comparisons among workers to be fair and comparable in this work.
3) Master node and HDFS are trusted. In the context of Hadoop MapReduce, the master node is a single point of failure and it manages scheduling, monitoring (i.e., MapReduce JobTracker) and the file system namespace and access control to files by clients (i.e., HDFS NameNode). Accordingly, we assume that Hadoop's logging facility is trusted. Further, system call logs at the nodes are reliable. We note that this assumption is reasonable, as several secure logging systems (e.g., SELiux-based and VM-based [13], [8]) have been deployed and may be used in MapReduce environments.
4) A majority of the MapReduce nodes exhibit honest behavior.
5) The Map function executed by the nodes is deterministic: that is to say, given the same input, MapReduce yields the same output.

D. Approach Overview

We build execution profiles of underlying application during the map and reduce stage on each slave node through the aggregated logs and identify deviations from common patterns of the execution of the behavior.

We adopt a black-box, low-level, dynamic instrumentation approach for collecting system-call level traces, which presumes no modification on Hadoop framework, and correlate the system call logs with Hadoop logs. Correlating Hadoop's logs and low-level system traces enables us to obtain a comprehensive view of execution flow and data flow. Aggregated system traces collected through dynamic instrumentation are parsed to extract the execution flow of the program in a fine grained manner, and in search of patterns indicative of suspicious execution behavior. Here, by suspicious execution behavior we mean behavior that is either (a) in violation with the operations computing nodes should perform for the MapReduce task they are in charge of or that (b) successfully subverts MapReduce operations or the overall Hadoop execution flow.

Our intuition is as follows: a MapReduce job consists of multiple copies of Map and Reduce tasks, each running the same client application, operating on different portions of the data set. We expect that the Map tasks exhibit similar behavior with other Map tasks even from different architecture, and corresponding Reduce tasks. Precisely, note that all MapReduce jobs follow the same flow of execution: (1) Map tasks are assigned, and begin by reading input data from DataNodes; (2) upon completion of Map task, the Map output is shuffled to Reducers. Eventually, (3) the job terminates after Reducers write their outputs back to the HDFS. Since each slave node executes a subset of the global set of Map and Reduce tasks, this temporal ordering is reflected on the slave nodes as well. As recent work on fault analysis in cloud computing and MapReduce applications has demonstrated [14], [27], it is reasonable to expect that slaves nodes will encounter similar workloads, and that the workloads will be mapped into consistently similar traces of execution. The analysis of these traces, along with information about the MapReduce configuration, the input data and the system setup, can reveal important insights on the ongoing activities at the nodes. More importantly, by correlating these traces with expected invariants on the programs execution and their workflow, we can unveil ongoing malicious activities, that affect confidentiality, integrity and availability of client's data and computations.

To verify the intuition that we can detect a large number of malicious activities with proper instrumentation techniques and in-depth analysis, we leverage dynamic instrumentation tools to monitor the execution flow of the MapReduce by checking a large set of actions and events, of the following types:

- Program integrity: We check whether execution of the map/reduce function follows the expected workflow, determined using known invariants.
- Input operations: We check whether file split inputs of the map function are from the correct HDFS addresses/locations.
- Output operations: We verify whether the output files of the reduce tasks are mapped into the expected correct HDFS addresses/locations.
- I/O activities at the MapReduce function level: We trace and analyze sets of system call related to file I/O activity such as `read()` and `write()` system calls within each map or reduce function, and semantically correlate this information with Hadoop logs and configurations.

IV. Inspection of MapReduce Execution Traces

We collect traces of running TaskTracker on each slave node and store the aggregated traces in HDFS as log files. Precisely, we leverage Hadoop logs for the high-level analysis

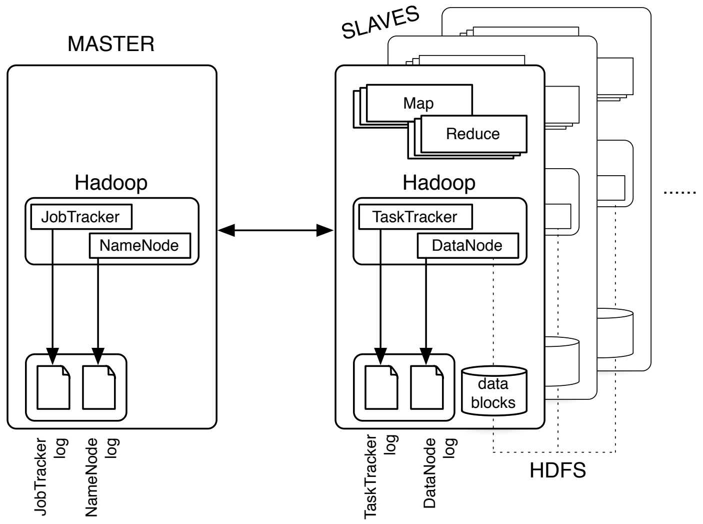

Fig. 2: Hadoop Logs used for the log analysis

of the interaction between the HDFS and Map/Reduce tasks. Specifically, Hadoop uses Apache `Log4j` facility to generate logs (*white-box approach*); logs are generated from the daemons of JobTracker, TaskTracker, NameNode and Datanode, which include configuration logs, statistics, standard error, standard output, and internal diagnostic information. Since each TaskTracker executes a subset of the Map and Reduce tasks, each TaskTracker observes a sampling of the global distribution of completion of tasks.

Conversely, we observe the execution behavior of the Map and Reduce functions through analysis of Map and Reduce JVM-generated system call logs. System call logs are generated by dynamic instrumentation (i.e., DTrace/strace) from the daemons of TaskTracker, (NameNode, and DataNode). We note that system call logs are particularly important to extract the program execution behavior of the Map and/or Reduce tasks executed on top of TaskTracker's spawned JVM processes.

Our analysis of these logs is based on some important insights about the information available through Hadoop logs and system call logs:

- Hadoop Logs have several unique identifiers of JobTracker, TaskTracker, NameNode, and DataNode.
  `JobID` (e.g., `job_200707121733_0003`): JobTracker and TaskTracker logs include the JobID which represents the unique identifier for the job. JobID represents the JobTracker identifier and the job number.
  `BlockID` (e.g., `blk_<id_1>`): DataNode log includes the BlockID which represents the HDFS blocks that consists of raw bytes of the data file.
  `TaskID` (e.g., `task_200707121733_0003 _m_000005`): TaskID represents a Map or Reduce task. TaskID is JobID along with the number of attempts and either 'm' (i.e., map task) or 'r' (i.e., reduce task), and the task number (i.e., fifth map task)
- System call traces provide finer grained execution logs than Hadoop logs. System call logs are grouped by process ID (i.e, PID) that represents the unique

identifier of each child JVM process of TaskTracker.

- The logs in distributed systems tend to be interleaved. We use a timestamp information to extract the correct order of execution. We take into account the drift time, which is due to possible discrepancies between clock times on the machines, by inspecting system events within the same time frame and synchronizing the timestamps. The information from Hadoop logs and system calls is matched against the order of the task invocations. For example, Hadoop creates an on-the-fly script file named `taskjvm.sh` to launch new task JVMs so we extract related log entries from Hadoop's TaskTracker log and system calls (i.e., `open taskjvm.sh`) for synchronizing the timestamp of the two different log files.

Log Parsing: The original logs collected in distributed environment is typically repetitive, unstructured and semantically hard to interpret, so we parse the log in order to get the desired information [16]. Precisely, to extract meaningful information from Hadoop logs and system call logs, we use the unique identifiers described above to create custom regular expressions that allow to match to parse particular log entries. To correlate Hadoop logs and system call logs, we extract `JobID` and `MapID/ReduceID` (i.e., `attemptID`) from the `taskjvm.sh` related log entry of Hadoop TaskTracker logs and matching log entry (i.e., system call events on `taskjvm.sh` with corresponding JobID, MapID/ReduceID, and PID) from system call logs. We also analyze syscall parameters such as the file descriptor (i.e.,filename and pathname) that the system call accesses for exploring I/O operations and the dependency in Map/Reduce tasks.

V. Integrity Checks

Next, we discuss our detailed approach toward detecting anomalies in the data and processing of MapReduce computations. We describe how we can extract patterns of communication among nodes and leverage access patterns to uncover data being misplaced or wrongly accessed.

A. Monitoring communication between HDFS client and the NameNode and the DataNode

The HDFS client (i.e., Map or Reduce task) accesses the HDFS service via RPC (Remote Procedure Call). In order for a client to first submit the data to HDFS, the client connects to the NameNode. The client's request flows in the form of Hadoop RPC requests over a TCP connection. Once the connection has been established, data blocks are transferred from the client to DataNodes. We analyze communication between the client and the NameNode and the DataNode by extracting network related events from Hadoop and system call logs. The Hadoop and system call logs are correlated with the extracted IP addresses, port numbers, and socket connection related system calls for tracing the network connection between the HDFS client and the HDFS service so that we can identify any unauthorized connection from the HDFS client to the NameNode or DataNode. Any suspicious network connection attempts such as wrong IP addresses of DataNoes or port numbers, a number of unsuccessful socket connection system calls, etc can be a good indication of malicious behavior. Inconsistency between correlated information and each logs' information is also an indicator of anomaly.

B. HDFS access patterns

HDFS is a distributed file system that has been created with the purpose of storing large data for large scale data intensive applications. We use collected Hadoop's logs to extract HDFS client's access patterns to HDFS along with the location of data blocks as a means for validating the integrity of input data. Hadoop's NameNode and DataNode logs provide the information of data block ID and DataNode location where the block is stored, which is accessed by the HDFS client. Namenode manages lists of files, list of blocks in each file, list of blocks per Datanode, and other metadata. DataNode stores blocks of data in its local file system and stores metadata for each block.

We specifically extract the location (i.e., IP address or hostname) of DataNode and the data blockID being accessed by the HDFS client. We also extract the information about the operations on the HDFS such as `HDFS_READ` and `HDFS_WRITE`. This information is useful for validating input data to Map task - if the input data loaded to Map task is originated from the correct DataNode location not from attacker. The BlockID extracted from the log is also used for validating the integrity of input data to Map task. Examples of Hadoop NameNode and DataNode trace logs are reported in Figure 3.

C. Snooping IO between HDFS and MapReduce by using DTrace/Strace

By analyzing HDFS access patterns, we can identify malicious accesses to HDFS, such as writing arbitrary data blocks to DataNodes or unauthorized access to the data blocks. We specifically check for the following HDFS actions:

- Storing client-provided data file as blocks in a set of DataNodes
- Loading data blocks into Map task from DataNodes that hosts replicas of the blocks.

We analyze the system call traces collected from DataNode to identify the data transfer activity between the HDFS client and DataNode. For example, `accept()` system call gives us information about the client establishing a connection with the DataNode. `open()` system call with the argument of blockID reveals which data block is being accessed. HDFS data transfers are carried out using TCP/IP sockets directly, thus the system call `sendfile()` with arguments of file descriptors enables us to check if the DataNode sends the correct data block to the corresponding HDFS client. Below is an example of system call traces on DataNode.

```
accept() = out_fd;
open("pathname(blockID)", O_RDONLY) = in_fd;
sendfile(out_fd,in_fd,offset,bytes) = bytes;
```

D. Execution traces of Map and Reduce function

We collect the dynamic execution traces of Map and Reduce tasks by DTracing TaskTracker's child JVM processes both in method level and system-call level on each slave

*** Hadoop NameNode log:**

```
STATE* Network toplogy has 1 racks and 2 datanodes

BLOCK* registerDatanode: node registration from ($DataNode):50010
              storage  DS-624241665-192.168.1.14-50010-1382597957423
BLOCK* allocateBlock:  $path  blk_5905677021831100640_2283
BLOCK* addStoredBock: blockMap updated: 127.0.0.1:50010
              is added to  blk_5905677021831100640_2283 size 33890
```

*** Hadoop DataNode log:**

```
DatanodeRegistration( (DataNode):50010
 storageID= DS-624241665-192.168.1.14-50010-1382597957423, infoPort=50075, ipcPort=50020)In
  DataNode.run, data = FSDatasetdirpath=' ($HDFS-Path)/dfs/data/current'

Receiving  blk_5905677021831100640_2283 src: ($HDFS-Client-IP):55950 dest: ($DataNode):50010

src:  ($DataNode):50010, dest:  ($HDFS-Client-IP):56002, bytes: 34158, op: HDFS_READ, cliID:
DFSClient_NONMAPREDUCE_1010168535_38, offset: 0, srvID:  DS-624241665-192.168.1.14-50010-1382597957423,
blockid:  blk_5905677021831100640_2283, duration: 671000
```

Fig. 3: Examples of Hadoop NameNode and DataNode logs

node. We extract semantic information of program execution from the trace, such as what classes have been loaded in Map/Reduce task and what kind of operations are performed on system components. The semantic analysis of system traces provides fine grained information about the control flow and data flow of the Map/Reduce task. Using system-call level logs, we can also identify dependency and causual relationships between different components in Map/Reduce tasks. The causual relationship indicates the execution logic expressed in the source code of the user application.

VI. MapReduce Invariants

In order to effectively use the traces described in previous section, we correlate them against some anticipated or trustworthy behavior of the MapReduce nodes. Given that we do not assume ground truth is given in terms of data output or correctness of traces, we resort to analyze this information against some expected invariants. Program invariants always hold in system logs under different inputs and workloads [15]. Hence, they naturally serve as checkpoints for validating program execution. Our approach is therefore to consider a set of program invariants, and analyze log sequences to compare them with an expected workflow. An anomaly can therefore be detected by checking if the new log fails to satisfy the anticipated invariant conditions. The invariants in Hadoop framework that we observed include: presence of Hadoop and Java libraries, reading correct configuration files, writing intermediate datablocks and fetching intermediate blocks by reduce nodes.

Hadoop libraries are common to all Hadoop applications. Since Hadoop libraries are not dependent on the applications, the events related to loading Hadoop libraries at runtime on worker nodes should be commonly observed without any deviation. Further, we can check if necessary Hadoop libraries have been loaded for the Map and/or Reduce tasks by analyzing system call logs.

The mapper function transforms input key-value pairs to intermediate key-value pairs and the reducer function takes a number of pairs sharing the same key and produces the final result of the computation. In detail, input data in HDFS are split and fed to the mapper function as key-value pairs. The key-value pairs produced by the mapper function are then sorted and combined by the function named shuffleSort which aggregates the values having the same key and sends the key and the list of the values to the reducer. The final result of the reducer is stored in HDFS.

In addition to simple operation and log events checks, we can check if the execution pattern follows this expected workflow through analysis of HDFS access patterns before the mapper execution and after the reducer execution, and the I/O events within the mapper and reducer task. Precisely, Hadoop configuration provides the location of intermediate output being stored by the mappers and system call parameters reveal the location where the mapper wrote intermediate output. These two locations should always match, regardless of the data size or the specific Map application.

For example, when checking whether HDFS has correctly interacted with MapReduce, the following events should be observed.

- Client writes data file to HDFS (splitted and stored as blocks in DataNodes)
- Mapper loads the data chunk from DataNodes
- Reducer writes the final output to HDFS unless the output is written to another Mapper

The order of the above steps is important in Hadoop workflow and should not be changed.

Further, for more detailed understanding of the behavior of the actual functions, the following invariants at the Map and Reduce function level can be observed.

- *Input/Output:* Given with the same input data, Map function always emits the same output data, unless the Map function has been compromised.
- *Method invocations:* The methods should be invoked in the same order for sequential tasks where the execution flow is the same.
- *Execution Order:* multiple invariants can be considered. For instance, loading input data block to Map function from HDFS is always followed by writing the intermediate output to the local disk/HDFS. Further, loading the intermediate output to Reduce task is always followed by writing the output to the local disk/HDFS.

In terms of the number of system calls in Map/Reduce function, the number of called and returned system calls (e.g., open(), close()) should be the same within the mapper and the reducer. Checking this number is a knowingly effective way to detect possible anomalies such as *file descriptor attacks*[25].

VII. MATCHING INVARIANTS AGAINST INTEGRITY CHECKS

Our verification of MapReduce execution is based on the analysis of correlated Hadoop log and system call log collected at runtime. Apache Hadoop and HDFS is built in Java language. Thus, we verify the methods of the mapper and the reducer implemented in Java. In this section, we discuss how to verify MapReduce computations based on observed invariants discussed in previous section.

A. Verification of Input Data Integrity

In order to check for possible malicious activities, the first step is to verify the input data being used by the nodes. This is determined by tracing the interaction between Map/Reduce task and HDFS. Figure 2, inspired by [27], summarizes the main logs instrumented for our analysis.

Hadoop's JobTracker log provides information about input data size and the number of input splits for a given task, which matches the number of mappers. The filename of input data that a specific map task and its path in HDFS can be extracted from Hadoop's TaskTracker log. In particular, we can extract the JobID, TaskID, MapID, and the location of the nodes that Map and/or Reduce tasks have been assigned and also information about the input data that is used for mappers. Using this information, we can correlate JobTracker, TaskTracker log, and instrumented system I/O event log to extract the semantic and behavioral relations between HDFS and Map/Reduce tasks.

One simple verification that can be done on the input data is to check the size of input data from the JobTracker log and HDFS. The JobTracker provides us *Input size for job* that should match with the size of input file stored in HDFS. For example, the size of input data in the example of a WordCount application (described in the next sections), is a a text file of 1573078 bytes. The size information can be extracted from JobTracker log and HDFS. Block-level operations on HDFS (e.g., `HDFS_READ, HDFS_WRITE`) can be observed from the DataNode log. Finer grained information such as BlockID, the location of blocks, and Map task's access patterns to the blocks can be extracted from system I/O event traces obtained through dynamic instrumentation on DataNode.

The intermediate and final results produced from mappers and reducers can be similarly verified through the correlated log analysis. We observe HDFS access behavior with the information about data block used for MapReduce task and its location from the correlated log. The intermediate output is generally stored in `'$JobID/$attempID/output'` folder. Hadoop logs and system call logs also provide the information about the data size. We can extract the size of MapReduce output data from TaskTracker log (e.g., reported output size for `$attemptID` was 450916), which is same as the sum of the number of bytes that the Reducer successfully has written to HDFS (i.e., return value of `write()` system calls), extracted from System call log on TaskTracker instance.

B. Verification of Computation Integrity

Although valid input data (i.e, original data that a user submitted) has been used by a mapper, a compromised node can lead to run cheating or malicious code for Map and/or Reduce tasks, resulting in incorrect computation results. To verify the integrity of Map/Reduce computation, we inspect mapper's execution behavior by analyzing Hadoop TaskTracker log as well as system call logs collected from each worker node at runtime. Hadoop logs are useful to check that the programs are properly instantiated and configured, but cannot provide information about how and if Map/Reduce tasks are actually executed correctly. Hence, we dynamically instrument mappers and reducers at runtime and collect specific system call traces. As system call tracing can incur high overhead, we collect specific I/O related system calls only such as `read()` and `write()`. The system call traces enable us to detect any abnormal execution behavior of Map/Reduce task without any a priori knowledge on client's application (i.e., black-box approach) by comparing traces from peer workers to detect any abnormal execution behavior. This approach is more reasonable when outsourcing the data and the workload into the public Cloud.

By comparing system calls sequences across nodes, our aim is to detect significant discrepancies in the system call statistics between honest worker nodes and cheating worker nodes or different pattern on the execution flow of Map function for malicious worker node. For example, a cheating node that tries to save computational resources may show system call patterns of significantly lower number of `write()` system calls than honest worker nodes, since most expensive computation involves writing operations.

Our findings, discussed next, demonstrate that the statistics related to traced system calls are very useful for detecting cheating behavior in Map/Reduce computation. However, raw statistics are not always enough for detecting malicious behavior, as even malicious Map/Reduce functions could invoke similar number of a system calls with a honest code while it could change the program semantics. We also need to consider the program semantics and the execution flow from the log analysis. We discuss this idea further in the context of our case study, on Section VIII.

C. Verifications on Hadoop Configuration

We investigated how the manipulation of Hadoop configuration/settings could affect the MapReduce execution behavior and its workflow. An attacker can modify Hadoop configuration with the intention of subverting the computation of MapReduce or corrupting computation[1]. One possible attack using insufficient configuration is *Symlink attack* [17]. Attacker identifies the target application by determining whether there is sufficient check before writing data to a file and creating symlinks to files in different directories. System call traces show different patterns with manipulated configuration settings. For example, TaskTracker's IP address and port number settings can be changed to connect to attacker's node by manipulating the value of *"mapreduce.tasktracker.report.address"* in `mapred-default.xml`. Also, the local directory where MapReduce stores intermediate data files can be changed to the value of *"mapreduce.tasktracker.local.dir"* in `mapred-default.xml` to the attacker's specified directory to be able to manipulate the intermediate output. Mappers read the configuration file before writing their output to the specified location so an attacker may can modify the specification about the location before Mappers read it. In both cases, system call traces can provide evidence that the configuration has been changed with the information about original and changed IP address, port, and the local directory.

VIII. Experiments

We used Hadoop version 1.2.1 in a Ubuntu based cluster. We aggregate the system traces from each slave node and store the logs (e.g. Hadoop syslogs and JobTracker and TaskTracker daemon logs) in HDFS. As mentioned previous sections, logs were collected upon execution of Map and Reduce functions. We used Perl scripts to read, correlate, and parse the logs by matching regular expressions. For Hadooop logs we collected JobTracker, TaskTracker, NameNode, and DataNode log. Finally, for Dtrace log we collected system call traces and traces of I/O snooping between HDFS and Map/Reduce tasks.

We chose WordCount for our experiments. WordCount is a well-known Hadoop application that counts the occurrences of words in a file. The map and reduce methods of WordCount application are shown in Figure 4. In our experiments, we looked for patterns indicative of cheating and malicious behavior, as follows.

1) Cheating Worker: We audited the MapRedue applications being executed on Hadoop framework and created a cheating node that tries to skip some computations by altering the execution flow (i.e., control flow of loop in WordCount example). We traced the child JVMs of TaskTracker which runs Mapper/Reduce function.

2) Malicious Worker: Malicious workers may attack MapReduce framework in many ways. Given our lack of assumptions about known correct traces, detecting malicious behavior only using log analysis is very challenging. Even one compromised mapper can lead to incorrect final result of the Map/Reduce job as one compromised mapper can create wrong intermediate output which will affect to the final result.

[1]A honest user can also make change the configuration in MapReduce [21], but typically this would not affect the overall execution of MapReduce functions

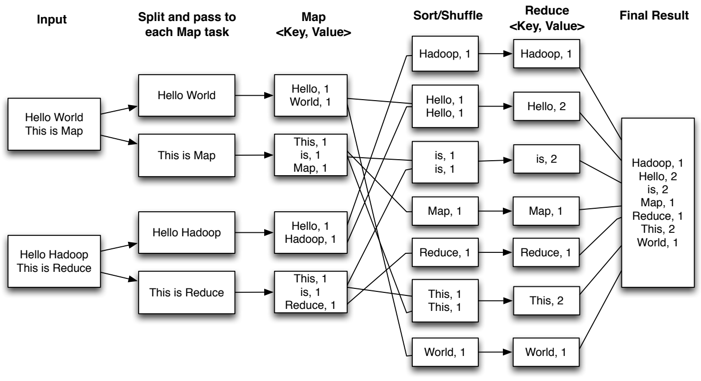

Fig. 4: WordCount MapReduce process

Here, we discuss a few scenarios in which the attacker subverts ordinary Map/Reduce functions. For these experiments, we are particularly interested in exploring whether relevant system calls and arguments can provide sufficient semantic information from the system call traces to detect a malicious activity. In this vein, we studied the following attacks. For each attack, we discuss our approach for detection.

- *Detecting malicious JVM activity such as suspicious JVM spawning.* We can extract JobID, MapID, and JVM ID from Hadoop log and System call log of nodes, and check whether they match or if one node has different values from all the other nodes. Precisely, this attack is effectively detected by looking at the system call log collected from TaskTracker child processes and Hadoop TaskbTracker log. The logs show detailed information of all spawned JVMs for Map/Reduce tasks.
- *Detecting malicious/suspicious JAR or Class files loaded.* The attacker can create malicious JAR (Java class files archives) files can be used for exploiting JRE (Java Runtime Environment) vulnerabilities. System call traces show that all the classes loaded for performing Map/Reduce tasks. The name and the location of the JAVA class files can be extracted from the logs.
- *Detecting misplaced intermediate result generated by Mappers.* This attack can be uncovered by checking the location where the intermediate output files are stored from the system call log. We detect if the mapper writes intermediate output to a different location than its configured location. Also, we can detect if a Reducer fetches intermediate output from a different location than the location where the data is stored.
- *Detecting malicious modification on Hadoop configurations.* The attacker may manipulate certain Hadoop configuration files which include `core-site.xml, mapred-site.xml, and hdfs-site.xml` in the middle of MapReduce computation. We can detect any configuration change on a worker node such as data block location (`'dfs.datanode.data.dir'` parameter setting for DataNode) by correlating Hadoop logs and system

call logs.

A. Results

Below we report the results of cheating and malicious worker case in the WordCount application. Our setup includes six mappers and one cheating or malicious mapper among them. We used eBooks obtained from the Project Gutenberg collection[1] as input data and worker nodes were only executing the WordCount program.

In the cheating scenario, Hadoop TaskTracker log indicates that the size of Map output of the cheating node is significantly different than output of the honest nodes. Specifically, from Hadoop TaskTracker log we report the size of map output of cheating mapper to be 6 bytes, whereas the size of map output of honest mapper is 409313 bytes.

However, these statistics do not provide any details about execution behavior of Mapper. The analysis of system calls from the system call logs instead provides additional insights. System call log indicates large difference in the number of `write()` system calls in Map function. The number of `write()` system calls of cheating mapper is much less than that of honest mappers (which are the remaining nodes), while the number of `read()` system calls are similar. We excluded logging related `write()` system calls in Map function from the log as most of `write()` system calls in WordCount example are related to Hadoop's logging. In particular, a cheating node does not show any spill files recorded from the Map phase, while honest mappers show a number of records spilled during Map phase before starting Reduce phase.

These differences are because either the cheating Mapper does not enter the loop (i.e., while statement), or it skips the loop many times, which loop involves in generating a record for each occurring word and its count. In repeated runs of this experiment, we report the following average differences.

- n_o of `write()` system calls of honest mapper: 622 out of 42270 read/write system calls
- n_o of `write()` system calls of cheating mapper: 189 out of 44514 read/write system calls

In the malicious scenario, since raw statistics are not sufficient for detection, we correlate Hadoop and System call logs, to check for possible anomalies or inconsistencies with other honest nodes. We note that the expected workflow of a generic MapReduce application is as follows:

1) Worker nodes read Hadoop configuration files
2) Worker nodes read necessary JAR files such as `classes.jar, hadoop-core-1.2.1.jar` (Hadoop Common package) and other logging related libraries and Java libraries.
3) Workers read and write `taskjvm.sh` and read JVM launching related libraries for launching new JVM
4) Workers read Hadoop configuration files
5) Mappers read Mapper class (e.g., `Map.class`)
6) Mappers read input data blocks from HDFS node
7) Mappers write intermediate output such as `file.out, file.out.index` to the local directory
8) Reducers load Reducer class (e.g., Reduce.class)
9) Reducers fetch Map outputs.
10) Reducers write final output to HDFS

Starting from this workflow, we consider three malicious events: a case where the workflow is altered when the attacker launches its own JVM to perform some malicious activity. Second, we tested a case where a map function modification occurs to tamper the intermediate results (change of source code to corrupt the correctness of the counting operation) by increasing the number of actual word counts. Finally, we tested a case where input data is not authentic but it is accessed from a secondary location.

Our analysis revealed that in the first case, when the attacker tried to launch the attacker's own JVM, the system call log collected from TaskTracker child processes and Hadoop TaskTracker log provide the expected discrepancies in the JobID, MapID, and JVM ID. The logs show all spawned JVMs for Map/Reduce tasks, in particular, we chose `taskjvm.sh` file from the log for the indication of new JVM launch. Typically, one JVM runs one task unless JVMs are configured to be reused in the job configuration. With this information, we are able to detect the maliciously spawned JVM if it shows suspicious JobID or Map/ReduceID associated with the operations.

In the second case, we found that while the workflow analysis is not detailed enough to reveal any anomaly, the differences in system calls statistics is significant enough to denote ongoing unexpected activities. Precisely, in this case, all nodes observe about 70-80% of their total system calls to be of `write()` system calls, whereas the suspicious node has only 20% `write()` system calls.

In the final case, after loading Mapper Class correctly, one node will not access HDFS nodes. Instead, it reads data blocks from its own local directory which is not configured by Hadoop framework. The file descriptor of `read()` system call which provides details about the file such as filename and the path, reveals the location of the datablock (i.e., path of the file). This shows the attacker's configured location, which is different from the information extracted from other worker nodes that access DataNodes. For the system call logs, we can compare file descriptors or the pathname as Dtrace can log the pathname extracted from the file descriptor.

IX. Discussion and Conclusion

In this paper, we presented a study of system logs, as a way to detect possible malicious or cheating worker nodes. Despite our minimal assumptions on knowledge of the computations being assessed (i.e. no known output or replicated input), we identified several important indicators that can lead to quick detection of malicious ongoing activities.

We believe that these findings can lead toward strong and seamless solutions for the detection of cheating or malicious worker nodes in MapReduce framework. Nevertheless, this work is still at its infancy. Currently, our main limitation lies in the little confidence over small modifications of the client's input that may tamper with the final output. While this is possibly addressable through existing replication and quiz-based methods [12], [29], we are still to investigate how to further process system logs for better accuracy of detection. One approach we are interested in exploring is based on oblivious hashing [4]. That is, we may obliviously hash portion of the applications' source code, and determine which system level calls hashing may incur into, to verify the hash, and therefore verify the integrity of the function, directly from the logs themselves.

Further, there are some assumptions we would like to relax, in our future work. we assume system logs are trustworthy, which may be not always realistic in case of nodes which configuration is corrupted. Further, we are currently able to zoom in on nodes performing malicious activities, but it is not possible to identify which portion of the output is corrupt as a result. Finally, although outside the scope of the current paper, we are yet to investigate how the approach would fare in case multiple nodes are compromised. While we know that statistical comparison of system calls would fail to provide meaningful insights, we also have identified other non-comparative checks that may be effective. The extent to which these are sufficient for any type of assessment is subject of future work.

Acknowledgements The work from Squicciarini and Yoon was partially supported by National Science Foundation under award No. 1250319.

References

[1] Project Gutenberg. http://www.gutenberg.org/wiki/Main_Page.

[2] M. Blanton, J. M. Atallah, B. K. Frikken, and Q. M. Malluhi. Secure and efficient outsourcing of sequence comparisons. In *ESORICS*, pages 505–522. Springer, 2012.

[3] Benjamin Braun, Ariel J Feldman, Zuocheng Ren, Srinath Setty, Andrew J Blumberg, and Michael Walfish. Verifying computations with state.

[4] Yuqun Chen, Ramarathnam Venkatesan, Matthew Cary, Ruoming Pang, Saurabh Sinha, and Mariusz H Jakubowski. Oblivious hashing: A stealthy software integrity verification primitive. In *Information Hiding*, pages 400–414. Springer, 2003.

[5] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[6] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[7] Wenliang Du, Mummoorthy Murugesan, and Jing Jia. Uncheatable grid computing. In *Algorithms and theory of computation handbook*, pages 30–30. Chapman & Hall/CRC, 2010.

[8] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. *SIGOPS Oper. Syst. Rev.*, 36(SI):211–224, December 2002.

[9] Qiang Fu, Jian-Guang Lou, Yi Wang, and Jiang Li. Execution anomaly detection in distributed systems through unstructured log analysis. In *Data Mining, 2009. ICDM'09. Ninth IEEE International Conference on*, pages 149–158. IEEE, 2009.

[10] Liang Gu, Xuhua Ding, Robert Huijie Deng, Bing Xie, and Hong Mei. Remote attestation on program execution. In *Proceedings of the 3rd ACM Workshop on Scalable Trusted Computing*, STC '08, pages 11–20, 2008.

[11] Hadoop Apache. http://hadoop.apache.org, 2013.

[12] Chu Huang, Sencun Zhu, and Dinghao Wu. Towards trusted services: Result verification schemes for mapreduce. In *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*, pages 41–48. IEEE, 2012.

[13] Trent Jaeger, Reiner Sailer, and Xiaolan Zhang. Analyzing integrity protection in the selinux example policy. In *Proceedings of the 12th conference on USENIX Security Symposium-Volume 12*, pages 5–5. USENIX Association, 2003.

[14] Nikhil Khadke, Michael P Kasick, Soila P Kavulya, Jiaqi Tan, and Priya Narasimhan. Transparent system call based performance debugging for cloud computing. In *Usenix Workshop on Managing Systems Automatically and Dynamically*, 2012.

[15] Ivo Krka, Yuriy Brun, Daniel Popescu, Joshua Garcia, and Nenad Medvidovic. Using dynamic execution traces and program invariants to enhance behavioral model inference. In *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, volume 2, pages 179–182. IEEE, 2010.

[16] Jian-Guang Lou, Qiang Fu, Yi Wang, and Jiang Li. Mining dependency in distributed systems through unstructured logs analysis. *ACM SIGOPS Operating Systems Review*, 44(1):91–96, 2010.

[17] Mitre. Symlink attack. http://capec.mitre.org/data/definitions/132.html.

[18] Fabian Monrose, Peter Wyckoff, and Aviel D Rubin. Distributed execution with remote audit. In *Network and Distributed System Security (NDSS) Symposium*, 1999.

[19] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 238–252, Washington, DC, USA, 2013. IEEE Computer Society.

[20] M.O. Rabin, R.A. Servedio, and C. Thorpe. Highly efficient secrecy-preserving proofs of correctness of computations and applications. In *Proc. of 22nd Annual Symposium on Logic in Computer Science*, pages 63–76. ACM, 2007.

[21] Ariel Rabkin and Randy Howard Katz. How hadoop clusters break. *IEEE Software*, 30(4):88–94, 2013.

[22] I. Roy, S. T. V. Setty, A. Kilzer, V. Shmatikov, and E. Witchel. Airavat: security and privacy for mapreduce. In *Proc. of the 7th USENIX conference on Networked systems design and implementation*, NSDI'10, pages 20–20, Berkeley, CA, USA, 2010. USENIX Association.

[23] Anbang Ruan and Andrew Martin. Tmr: Towards a trusted mapreduce infrastructure. In *Services (SERVICES), 2012 IEEE Eighth World Congress on*, pages 141–148. IEEE, 2012.

[24] Luis F. G. Sarmenta. Sabotage-tolerance mechanisms for volunteer computing systems. *Future Generation Computer Systems*, 18:561–572, 2002.

[25] Benjamin Schwarz, Hao Chen, David Wagner, Jeremy Lin, Wei Tu, Geoff Morrison, and Jacob West. Model checking an entire linux distribution for security violations. In *ACSAC*, pages 13–22. IEEE Computer Society.

[26] Jason Sonnek, Abhishek Chandra, and Jon B Weissman. Adaptive reputation-based scheduling on unreliable distributed infrastructures. *Parallel and Distributed Systems, IEEE Transactions on*, 18(11):1551–1564, 2007.

[27] Jiaqi Tan, Xinghao Pan, Eugene Marinelli, Soila Kavulya, Rajeev Gandhi, and Priya Narasimhan. Kahuna: Problem diagnosis for mapreduce-based cloud computing environments. In *Network Operations and Management Symposium (NOMS), 2010 IEEE*, pages 112–119. IEEE, 2010.

[28] V. Vu, S. Setty, A. J. Blumberg, and M. Walfish. A hybrid architecture for interactive verifiable computation. In *Proc. of IEEE Symposium on Security and Privacy*, 2013.

[29] Yongzhi Wang and Jinpeng Wei. Viaf: Verification-based integrity assurance framework for mapreduce. In *IEEE International Conference on Cloud Computing (CLOUD)*, pages 300–307. IEEE, 2011.

[30] W. Wei, J. Du, T. Yu, and X. Gu. Securemr: A service integrity assurance framework for mapreduce. In *Proc. of Computer Security Applications Conference, ACSAC*, pages 73–82, 2009.

[31] Zhifeng Xiao and Yang Xiao. Accountable mapreduce in cloud computing. In *Computer Communications Workshops (INFOCOM WKSHPS), 2011 IEEE Conference on*, pages 1082–1087. IEEE, 2011.

[32] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael Jordan. Online system problem detection by mining patterns of console logs. In *Data Mining, 2009. ICDM'09. Ninth IEEE International Conference on*, pages 588–597. IEEE, 2009.

[33] Shanyu Zhao, Virginia Lo, and CG Dickey. Result verification and trust-based scheduling in peer-to-peer grids. In *Peer-to-Peer Computing, 2005. P2P 2005. Fifth IEEE International Conference on*, pages 31–38. IEEE, 2005.