Policy Driven Node Selection in MapReduce

Anna C. Squicciarini[1](✉), Dan Lin[2], Smitha Sundareswaran[1], and Jingwei Li[3]

[1] Pennsylvania State University, State College, USA
asquicciarini@ist.psu.edu, sus263@psu.edu

[2] Missouri University of Science and Technology, Rolla, USA
lindan@mst.edu

[3] Nankai University, Tianjin, People's Republic of China
lijw1987@gmail.com

Abstract. The MapReduce framework has been widely adopted for processing Big Data in the cloud. While efficient, MapReduce offers very complicated (if any) means for users to request nodes that satisfy certain security and privacy requirements to process their data.

In this paper, we propose a novel approach to seamlessly integrate node selection control to the MapReduce framework for increasing data security. We define a succinct yet expressive policy language for MapReduce environments, according to which users can specify their security and privacy concerns over their data. Then, we propose corresponding data preprocessing techniques and node verification protocols to achieve strong policy enforcement. Our experimental study demonstrates that, compared to the traditional MapReduce framework, our policy control mechanism allows to achieve data privacy without introducing significant overhead.

Keywords: MapReduce · Node selection · Access control

1 Introduction

The MapReduce computing paradigm is an architectural and programming model that utilizes a large number of worker nodes in parallel to efficiently process massive amount of raw unstructured data [1,3,9]. Initial constructions of MapReduce only ran in a single trusted data center. With the proliferation of the cloud computing, MapReduce has now become a popular means, and typically uses the worker nodes residing in untrusted public cloud to process Big Data [13,22]. For instance, in the Cisco Nexus 1000V InterCloud, not only are the virtual machines' environment heterogeneous, but the actual physical hosts are also geographically distributed, and offer different degrees of trust and security.

The fact that MapReduce may utilize un-trusted nodes for processing data raises concerns to data owners who wish to use MapReduce tasks on sensitive information. For example, with the explosion of patient data after the adoption of electronic health record, health care organizations are currently outsourcing data analytics tasks to the cloud, such as counting the occurrences of common

J. Tian et al. (Eds.): SecureComm 2014, Part I, LNICST 152, pp. 55–72, 2015.
DOI: 10.1007/978-3-319-23829-6_5

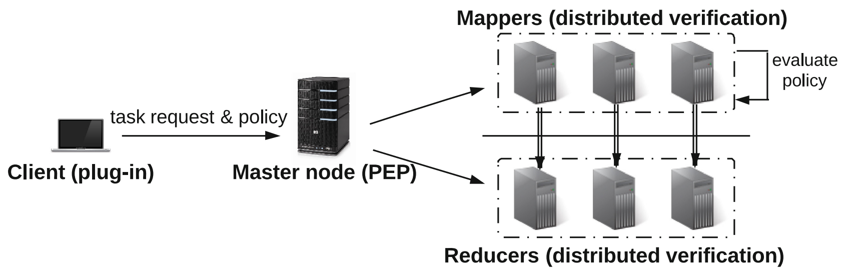

Mappers (distributed verification)
evaluate policy
task request & policy
Client (plug-in)
Master node (PEP)
Reducers (distributed verification)

Fig. 1. Overview of the main ACEM framework

diseases at different age ranges using the MapReduce. Some health records that belong to young adults may be privacy sensitive according to the HIPAA law, and hence the health care organizations may require such sensitive records to be processed only by the cloud servers (worker nodes) with cryptographic capabilities and also located in USA. Unfortunately, this kind of requirements on the worker nodes cannot be achieved in any existing MapReduce implementations, without tedious manual configuration. It is also worth noting that although methods such as homomorphic encryption [5,18] or outsourced private computation [4] can protect the data by processing in the encrypted domain, these approaches are typically computationally expensive and are only feasible for limited applications [7].

To overcome the above challenge, we propose a novel access control enforcement mechanism, called ACEM (**A**ccess **C**ontrol **E**nforcement in **M**apreduce), which automatically selects and verifies worker nodes in the MapReduce according to data owners' access control policies. In particular, we first propose a MapReduce Policy Language (MPL), that is tailored according to the characteristics of MapReduce environments (e.g., the properties of worker nodes in the MapReduce), facilitating data owners to specify their privacy and security concerns regarding worker nodes that handle their data. For example, using MPL, a data owner can explicitly specify that his/her sensitive data can only be processed by worker nodes located in USA. Since a data owner may have different access control policies regarding different portions of his/her data, we further propose a data-policy binding algorithm that automatically partitions the user's data based on access control policies and binds each data partition with the respective policy. Then, we design an efficient collaborative verification protocol to select qualifying worker nodes for each data partition. Our proposed approach is elegantly interleaved with existing MapReduce scheduling process without affecting the core MapReduce architecture. We have implemented a prototype of our proposed ACEM mechanism as an extension to the Azure's iterative MapReduce-Daytona [3,15], and our experimental results demonstrate both effectiveness and efficiency of our approach.

The rest of the paper is organized as follows. Section 2 gives an overview of MapReduce and discusses security and privacy issues in MapReduce. Section 3 presents an overview of the proposed framework. Section 4 defines a policy language for MapReduce. Section 5 introduces the data preprocessing algorithms. Section 6 describes the policy-based node selection in MapReduce. Section 7 reports experimental results. Finally, Sect. 8 concludes the paper.

2 Related Works

In this section, we first give an overview of the MapReduce, and then discuss security and privacy issues in MapReduce.

2.1 Background of MapReduce

MapReduce is a functional programming paradigm. 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. 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. In a cloud computing setting, these functions are orchestrated by the Master, and carried out by the mappers, and reducers. The Master acts as the coordinator responsible for task scheduling, job management, etc. A Master's module (typically the data partitioner) splits input data into a set of M blocks, which will be read by M mappers through DFS I/O. The execution of map and reduce tasks are automatically distributed across all the nodes in the cluster. The *map* function takes as input one of the M blocks, which is defined as a key-value pair, and produces a set of intermediate key-value pairs. The intermediate result is sorted by the keys so that all pairs with the same key will be grouped together (the shuffle phase). If the memory size is limited, the locations of the intermediate results are sent to the Master who notifies the reducers to prepare to receive the intermediate results as their input. Reducers then use Remote Procedure Call (RPC) to read data from mappers and execute user defined reduce functions, in which the key pairs with the same key will be reduced in some way, depending on the user defined reduce function. Finally, the output will be written to DFS.

2.2 Security and Privacy in MapReduce

There is growing interest in security of MapReduce [2,5,12,14,16,18,19,24,25]. The Sedic framework [25], is the closest effort to ours. Sedic aims to partition the data according to the inputs sensitivity level. If a data piece is sensitive, it is sent to a sensitive mapper. For reducer computations, Sedic modifies the reducer routines by checking whether they contain certain loop dependent variables: if so, the partition of sensitive and non-sensitive data is affected, otherwise data from sensitive mappers would be pushed to non sensitive reducers. Sedic achieves this goals by modifying how the data is read: normally, the entire data is read using a single pointer, while with Sedic only a block of data is read using a given pointer. As we discuss in Sect. 5.1, ACEM also includes algorithms for data partitioning, in addition to checking that the workers satisfy user-specified conditions before they are allowed to process the data.

Also closely related is the Airavat [19] project. Airavat is a secure and private 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 achieves this by modifying the Java Virtual Machine and the MapReduce framework and adding SELinux-like Mandatory Access Control to the DFS. It is worth noting that, not only does the methodology of Airavat differ from that of ACEM, but the end goals of the frameworks are also different: While Airavat tries to prevent the processing of the data against untrusted code, ACEM tries to prevent the processing of the data against untrusted nodes.

Another related work, which, similar to our work, relies on distributed verification (see Sect. 6.1) is the SecureMR framework [24]. The framework is intended to be a practical service integrity assurance framework for MapReduce. It allows mappers to examine the integrity of data blocks from the DFS; verify the authenticity and correctness of the mappers' results; and allows users to check the authenticity and correctness of the reducers' final results.

Finally, this paper is loosely related to the body of work focusing on cloud computing integrity of computation [4,16,23]. For example Moca's [16] proposal deals with distributed results checking for MapReduce. The work relies on the distributed voting method to check the correctness of the results produced by MapReduce. This work is complementary to ours in that while it relies on a distributed approach, it verifies the correctness of the computation, rather than whether a user's requirements of the nodes are satisfied.

3 An Overview of ACEM Mechanism

We propose an ACEM (Access Control Enforcement in Mapreduce) framework that considers security and privacy issues of worker nodes in MapReduce. The ACEM framework enables data owners to impose security requirements on their sensitive data and then selects worker nodes that satisfy the users' security requirements to perform the MapReduce functions on their data. Figure 1 illustrates the main components of the ACEM framework and their interactions. There are three entities involved in ACEM.

- Clients equipped with the policy specification plug-in is able to submit computation task as well as a set of policies. These policies express the constraints against properties of the workers computing or managing clients' data.
- Master node taking with a policy enforcement point (PEP) module is responsible for scheduling the MapReduce tasks and coordinate distributed evaluation of users' policies.
- Each worker node is installed a policy evaluation/verification module, such that the properties of this worker node could be evaluated to assess their

eligibilities to access potion of user data, while the other nodes in synch with the master node could act as verifiers to verify the correctness of the policy evaluations.

We assume that at least the master node is always under the control of the cloud service provider, and therefore can be fully trusted. We adopt the semi-honest adversary model for worker nodes in that workers in the cloud are expected to follow the ACEM protocols but may explore the information they processed.

The execution of ACEM-MapReduce programs is similar to MapReduce programs, the difference being that nodes processing MapReduce tasks on a given client input are selected according to clients' policies, while preserving the original execution flow. Recall the example of outsourcing patient records mentioned in the introduction, wherein the health care organization would request that sensitive patient records should be handled only by the worker nodes located in the USA and with cryptographic capabilities. In this case, the master node, upon receiving the policy, completes two preliminary steps: (1) it pre-processes the input data to partition them according to the user's policies, and (2) it triggers the collaborative property verification protocols, to identify nodes capable to carry out the required MapReduce tasks that also meet the users' policy requirements (e.g. it verifies which nodes are in the USA and whether they have cryptographic capabilities). Upon verification of the policy's satisfiability, the data distributed by the master becomes available to the eligible nodes, starting the process at the mappers. As the mappers complete their tasks, the control is back to the master. The master, upon shuffling the data in accordance with the application's logic and MapReduce routines, will assign the intermediate data to worker nodes that satisfy policies for reduce tasks. To keep track of the input data and its related policies, the data may be tainted after the processing as they go through intermediate stages.

4 MapReduce Policy Language

The first challenge in achieving access control in MapReduce is to formally specify data owners' various security and privacy requirements on their data items such that only the policy-compliant worker nodes are allowed to access these items. Although traditional policy language such as XACML is high expressive, it would introduce high degree of complexity (for both configuration and enforcement of policies), escaping from for our purposes. To tackle this challenge, we propose a more succinct yet still expressive policy language, called MapReduce Policy Language (MPL). Compared to traditional policy language, MPL enjoys two unique features: (1) MPL policies can be quickly composed by removing unnecessary components in traditional policy languages. (2) MPL policies can be evaluated within a tractable time. In what follows, we firstly give the definition of MPL, and then describe how to evaluate MPL.

4.1 MPL Definition

Before introducing MapReduce Policy Language (MPL), we first provide the definition of a condition language that is used in MPL.

Definition 1 (Condition Language). *Suppose $\mathcal{U} = \{u_1, u_2, \dots, u_n\}$ is the attribute universe, and dom_{u_i} is the domain of each attribute $u_i \in \mathcal{U}$. Let $\theta_i \subseteq \{<, \leq, =, \neq, >, \geq, \subset, \subseteq, \supseteq, \supset\}$ denote the operation set defined for attribute u_i in its domain dom_i. Then, we can recursively define the condition language $\mathcal{L}_{\mathcal{U}}$ on $\mathcal{U}$ as follows:*

- *For any attribute u_i, value $v \in dom_{u_i}$ and operation $\theta \in \theta_i$, the atomic condition $\langle u_i \theta v\rangle$ belongs to $\mathcal{L}_{\mathcal{U}}$.*
- *For any condition $c_i, c_j \in \mathcal{L}_{\mathcal{U}}$, the composite conditions $c_i \wedge c_j$ and $c_i \vee c_j$ belong to $\mathcal{L}_{\mathcal{U}}$, where "$\wedge$" and "$\vee$" respectively denotes "AND" and "OR" operation.*

As an example, we consider a two attribute-universe $\{u_1, u_2\}$. Suppose u_1 has domain $dom_{u_1} = [1, 3]$ and the operation set $\{<, >, \leq, \geq, =, \neq\}$ defined on dom_{u_1}; u_2 has domain $dom_{u_2} = \{1, 2, 3\}$ and the operation set $\{\subset, \subseteq, \supseteq, \supset, =, \neq\}$ defined on dom_{u_2}. Then, we can have atomic conditions: $\langle u_1 < 1\rangle, \langle u_1 < 2\rangle, \langle u_1 < 3\rangle, \langle u_1 > 1\rangle, \dots$ for u_1 and $\langle u_2 \subset \{1\}\rangle, \langle u_2 \subset \{1, 2\}\rangle, \dots$ for u_2, which belong to $\mathcal{L}_{\{u_1,u_2\}}$. Moreover, any AND/OR-composition (e.g., $\langle u_1 < 3\rangle \wedge \langle u_2 \subseteq \{1, 2, 3\}\rangle$) of the conditions in $\mathcal{L}_c$ still belongs to $\mathcal{L}_c$. It is worth noting that, $\mathcal{L}_c$ has infinite number of conditions and most of them (e.g., $\langle u_1 < 3\rangle \wedge \langle u_2 \subseteq \{1, 2, 3\}\rangle \wedge \langle u_1 > 3\rangle$) are permanently not satisfied. Of course, we only account for the *significant* subset of $\mathcal{L}_c$, in which the conditions could be satisfied.

Unlike arbitrary Boolean expressions, the Boolean expressions in $\mathcal{L}_c$ can be solved in a polynomial time, which is important for ensuring the efficiency of policy evaluation when integrating ACEM system into MapReduce. In terms of expressiveness, $\mathcal{L}_c$ covers all cases except the condition that involves direct comparison of multiple attributes (e.g., $u_i > u_j$). We argue that such comparison of multiple attributes rarely occurs in the MapReduce data processing since attributes associated with a data item (or a worker node) are different from one another and usually not comparable, e.g., we do not compare attributes "age" with "location" in a person's medical record.

Next, we explain our proposed policy language MPL. In MPL, both users' data items and worker nodes are represented as a set of *attribute-value* pairs. Specifically, a user's dataset is a collection of data items, i.e., $\mathcal{D} = \{data_1, data_2$ ..., $data_n\}$, and each data item $data_i$ $(1 \leq i \leq n)$ is in the form of $data_i$=$\{(u_1, v_1)$ $\dots (u_s, v_s)\}$, where $u_j (j = 1, \dots, s)$ is an attribute name and v_j $(j = 1, \dots, s)$ is the corresponding attribute value. Similarly, a worker node *node* is represented by a set of property-value pairs, i.e., *node*=$\{(w_1, v_1)$, ..., $(w_t, v_t)\}$, where w_i is property name and v_i is the corresponding value. For example, we consider a health care organization (HCO) outsource computing task to cloud and have a set of attributes for data items like *data*=$\{(age, 26)$, $(gender, male)$, $(country, USA)$, $(diagnos, HIV)$, $(date, 10/2013)\}$, which means a 26-year old male born in USA was diagnosed HIV in Oct 2013. Correspondingly, a worker node's properties may look like: $node{=}\{(location, USA), (AES, enabled)\}$ which means the worker node is located in USA and has cryptographic capability.

Based on the attribute expression on both user's data and worker node, we can then define MPL. Informally, MPL is a set of policies, each of which specifies the requirements that a worker node should satisfy to access a certain data item. The formal definition of MPL is as follows.

Definition 2 (MapReduce Policy Language). *Suppose $\mathcal{U}$ and $\mathcal{W}$ are respectively the universe of data items' attributes and worker nodes' properties. A policy $\mathcal{P}_i$ in MPL is a set of rules $\mathcal{P}_i = \{R_1, R_2, \ldots, R_k\}$, and each rule includes two components.*

- `Target` *is a condition in the condition language $\mathcal{L}_\mathcal{U}$ on $\mathcal{U}$, describing which data item is to be accessed in this rule.*
- `Cond` *is a condition in the condition language $\mathcal{L}_\mathcal{W}$ on $\mathcal{W}$, specifying the security and privacy requirements that a worker node should satisfy to access the data item.*

To be more clear, let us re-consider the previous HCO example. Suppose that the task outsourced to the cloud by HCO is to count the number of diseases occurring at each age range. Since some of the patient records are privacy sensitive, such as patient records belong to young adults (age$\leq$ 14) or patients who have severe diseases (e.g., HIV), HCO may require the sensitive records to be handled by cloud servers (worker nodes) that are located in USA with cryptographic capability, while other non-sensitive records just need to be processed by the servers located in USA. Such requirements can be specified in a MPL policy as follows:

$$\begin{aligned}
\mathcal{P}_{HCO} = \{ & \\
& R_1 : \texttt{Target}\langle (age \leq 14) \vee (diagnose = HIV)\rangle, \\
& \quad\ \texttt{Cond}\langle (location = USA) \wedge (AES = enabled)\rangle \\
& R_2 : \texttt{Target}\langle (age > 14) \wedge (diagnose \neq HIV)\rangle, \\
& \quad\ \texttt{Cond}\langle (location = USA)]\rangle \\
& \}
\end{aligned}$$

4.2 MPL Evaluation

In this section, we discuss how an access request is evaluated against a policy. First, we define an access request from a worker node as follows.

Definition 3. *An access request $\mathcal{Q}_{node}$ is in the form $\mathcal{Q}_{node} = (data, node)$, which means a worker node node requests to access a data item data.*

Given an access request $\mathcal{Q}_{node}$ from a worker node, a rule in a policy will output a decision value belonging to $\{\texttt{Permit}, \texttt{Deny}, \texttt{NotApplicable}\}$ as defined in Definition 4.

Definition 4 (Rule Evaluation). *Given an access request* $\mathcal{Q}_{node} = (data, node)$ *and a rule* $R = (\texttt{Target}, \texttt{Cond})$*, the effect* $\mathsf{E}(R(\mathcal{Q}_{node}))$ *of the rule* R *on* $\mathcal{Q}_{node}$ *is defined as follows.*

– $\mathsf{E}(R(\mathcal{Q}_{node})) = \texttt{Permit}$*, if* $R.\texttt{Target}$ *is satisfied by the data item data and* $R.\texttt{Target}$ *is satisfied by node.*
– $\mathsf{E}(R(\mathcal{Q}_{node})) = \texttt{Deny}$*, if* $R.\texttt{Target}$ *is satisfied by data but* $R.\texttt{Cond}$ *is not satisfied by* $\texttt{attr}_{node}$*.*
– $\mathsf{E}(R(\mathcal{Q}_{node})) = \texttt{NotApplicable}$*, if* $R.\texttt{Target}$ *is not satisfied by data.*

Since one policy may contain multiple rules and each rule may return different effects regarding the same request, we adopt the *first-one-applicable* rule combining algorithm to resolve any possible policy conflict in a simple and efficient manner. The first-one-applicable rule combining algorithm can speed up the policy evaluation process since the evaluation stops once one applicable rule is identified.

Definition 5 (First-One-Applicable). *Suppose* $R_1, R_2, \ldots, R_n$ *is a set of rules in a policy* $\mathcal{P}$ *and* $\mathcal{Q}$ *is an access request. The evaluation* $\mathsf{Eval}(\mathcal{P}(\mathcal{Q}))$ *of the policy* $\mathcal{P}$ *on* $\mathcal{Q}$ *is defined as follows.*

– $\mathsf{E}(\mathcal{P}(\mathcal{Q})) = \texttt{Permit}$*, if the first rule in* $\mathcal{P}$ *that is applicable to* $\mathcal{Q}$ *yields* $\texttt{Permit}$*.*
– $\mathsf{E}(\mathcal{P}(\mathcal{Q})) = \texttt{Deny}$*, if the first rule in* $\mathcal{P}$ *that is applicable to* $\mathcal{Q}$ *yields* $\texttt{Deny}$*.*
– $\mathsf{E}(\mathcal{P}(\mathcal{Q})) = \texttt{NotApplicable}$*, if none of the rules in* $\mathcal{P}$ *is applicable to* $\mathcal{Q}$*.*

5 Policy-Based Binding

A data owner may have fine-grained security and privacy requirements on various portions of their data (e.g., sensitive data and non-sensitive data), leading to multiple access control rules in the corresponding access control policy. In order to ensure that each portion of data is protected by the respective policy before being processed, we propose a simple approach to assign the data with the access control rules that apply to it. Our approach involves two tasks: (i) data partitioning; and (ii) data tainting.

5.1 Policy-Based Data Partitioning

The policy-based data partitioning aims to partition a data owner's data items into subsets according to the access policy imposed on them. After the partitioning process, we will obtain multiple equal-sized data buckets. Each data bucket will contain one or more groups of data items, and each group of data items is associated with the same access rule. These data buckets will then be treated as input files to MapReduce for further data processing. In what follows, we present the detailed algorithm for policy-based data partitioning.

Suppose that a user submits a set of data items $\mathcal{D} = \{data_i\}$ along with a policy $\mathcal{P} = \{R_1, ..., R_n\}$ to be enforced. Algorithm 1 shows the data partitioning algorithm on $\mathcal{D}$ in terms of $\mathcal{P}$. Initially, the master node creates an empty bucket with fixed capacity for each rule in $\mathcal{P}$ (lines 2 to 5 in Algorithm 1). The size of the bucket is pre-defined according to a scheduling algorithm followed by the master node.

The master node then starts scanning the data items. Each data item will be evaluated against the rules in $\mathcal{P}$. According to the "first-one-applicable" rule combining algorithm, if R_i is the first rule that is applicable to *data*, i.e., *data* satisfies the target component in R_i, the master node will insert *data* into the bucket $\texttt{bucket}_i$ and stop checking the remaining rules. In the case that the bucket of the first applicable rule R_i is full, the master node will add one more bucket to the first identified applicable rule and assign *data* to it. If none of the remaining rules applicable to d_i, d_i will be inserted to a separate bucket marked as "FreeBucket". Data items in this FreeBucket can be assigned to any worker nodes. At the end, up to $n+1$ data partitions will be generated, where each partition may be associated with multiple buckets. An example of the data partitioning is given below.

Reconsider the policy $\mathcal{P}_{HCO}$ in Sect. 4.1 and the following data items:

$data_1$=$\{(age, 26), (gender, male), (country, USA), (diagnos, HIV), (date, 10/2013)\}$
$data_2$=$\{(age, 22), (gender, female), (country, USA), (diagnos, flu), (date, 11/2013)\}$
$data_3$=$\{(age, 56), (gender, male), (country, USA), (diagnos, diabeties), (date, 8/2013)\}$
$data_4$=$\{(age, 10), (gender, male), (country, USA), (diagnos, flu), (date, 10/2013)\}$

After data partitioning, two buckets will be generated with respect to the two rules in $\mathcal{P}_{HCO}$. Since $data_1$ and $data_4$ satisfy $\mathcal{P}_{HCO}$.R1.Target while $data_2$ and $data_3$ satisfy $\mathcal{P}_{HCO}$.R2.Target, $bucket_1 = \{data_1, data_4\}$ and $bucket_2 = \{data_2, data_3\}$. FreeBucket is not needed in this case.

5.2 Data Tainting

In some MapReduce applications that involve multiple rounds of map and reduce phases, the output data may no longer possess the same set of attributes as the original input, which causes difficulty in determining the proper access policies on the intermediate results. For example, suppose a user has a spatial policy $\mathcal{P} = \{(\texttt{Target}\langle length > 10\rangle, \texttt{Cond}\langle\ (location = \text{“}US\ WEST\text{”}) \wedge (crypto = \text{“}3DES\text{”})\rangle\}$ on the initial input file. After the first round of computation, we could obtain an area of rooms as the output which typically does not have the same type or unit compared to the input. In this case, we cannot easily determine whether the policy target still applies to the data (now an area) for the next round of processing.

To address this issue, we adopt data tainting techniques to the data being protected. The underlying idea is to taint the data so that output data items are protected in the same way as the input data, i.e., under the protection of the same policy rule. In order to track the relationship between the input and output data, we let the master node apply the taint [8,17] to the input data before assigning the mapping tasks. Tainting results in a modification of the input data type to add a new property to the data. In the above example, tainting consists of

Algorithm 1. Data Partitioning Algorithm

```
procedure DataPartition($\mathcal{D}$, $\mathcal{P}$)
    for $i \leftarrow 1$ to $n$ do
        create an empty bucket bucket$_i$
    end for
    for each item $data \in \mathcal{D}$ do
        for $i \leftarrow 1$ to $n$ do
            if $data$ satisfies $R_i$.Target then
                if bucket$_i$ is not full then
                    insert $data$ into bucket$_i$
                    break
                else
                    add one more bucket appended with bucket$_i$ and put $data$ in it
                    break
                end if
            end if
            if $i$ equals $n$ then
                if freebucket is not created then
                    create a new freebucket
                end if
                insert $data$ into freebucket
            end if
        end for
    end for
end procedure
```

modifying the input length (usually defined as `int` or `float`) to an object. The object includes a data portion with the original `integer` or `float`, along with a Boolean portion called *tainted* showing whether the object is tainted or not, and a string portion called *taint* which is used to set a particular taint value. After the map round, mappers may also apply or re-apply the taint in either of the following two cases: (1) when the input to the mapper is tainted, or (2) when the user inserts, deletes or revises existing policies. Implementation details about data tainting will be provided in Sect. 7.

6 Policy Evaluation and Enforcement in MapReduce

In this section, we first present the overall algorithm for collaborative policy evaluation in MapReduce, and then make specific to two important issues in the collaborative verification protocol, i.e., (1) how to determine the number of nodes needed for verification and (2) how to conduct a single property verification at a verifying node.

6.1 Collaborative Policy Evaluation Protocol

In order to verify whether the properties of worker nodes in charge of computing satisfy the conditions imposed in the respective policy, a straightforward method is to let the trusted master node verify the worker's properties and perform the policy evaluation. However, this method suffers from several shortcomings: on the one side, it introduces overhead computation at the master node, which would become the bottleneck of the entire system and negatively impact the distributed nature of MapReduce; on the other side, it is also hard for master node to keep track of all the worker nodes' properties up to date [3,9].

To overcome these issues, we propose a collaborative property verification protocol to facilitate the policy evaluation at the master node. The underlying intuition in the collaborative property verification is to maximize computing resource utilization and use ordinary worker nodes, instead of the master node in straightforward method, to carry out verification of other worker nodes' properties. In our proposed protocol, at each round any worker node's properties is verified by multiple peers randomly selected; and any peer is able to verify a randomly selected set of properties, not only speeding up the verification process but also reducing the probability of worker nodes' collusion. The number of nodes to use as verifiers is chosen carefully according to the probabilistic scheme discussed in the next section, to define a combination of verifier nodes which are redundant enough to ensure low risk of collusion.

Our proposed collaborative verification protocol works as follows. Suppose that a client submits a policy $\mathcal{P} = \{R_1, R_2, \ldots, R_n\}$ along with the data $\mathcal{D}$, and the master node has partitioned the data into buckets: $\texttt{bucket}_1, \ldots, \texttt{bucket}_n$ and $\texttt{freebucket}$, respectively associated with the rule $R_1, R_2, \ldots, R_n$ and non-compliant policy, as described in Sect. 5.1.

Initially, the master node scans the condition components of all the rules and extracts a set of worker node properties $\texttt{attr}_R$ that need to be verified. For instance, consider the rules exemplified in Sect. 4.1, $\texttt{attr}_{R_1} = \{location, AES\}$ with respect to $R_1 : \texttt{Target}\langle(age \leq 14) \vee (diagnose = HIV)\rangle, \texttt{Cond}\langle(location = USA) \wedge (AES = enabled)\rangle$, while $\texttt{attr}_{R_2} = \{location\}$ for the rule $R_2 : \texttt{Target}$ $\langle(age > 14) \wedge (diagnose \neq HIV)\rangle$, $\texttt{Cond}\langle(location = USA)]\rangle$.

Next, the master node invoke the collaborative verification protocol to verify the extracted properties. Suppose that the condition component of R.cond is written in the conjunctive form $c_1 \wedge \ldots \wedge c_k$ where c_i $(1 \leq i \leq k)$ is a disjunctive form (i.e., $c_i = c_{i_1} \vee c_{i_2} \ldots$). To reduce the risk of possible corruption of the verifier nodes, our proposed verification protocol aims to verify each disjunctive sub-clause $c_i$by at least t peer worker nodes. Accordingly, the master node computes hash $S = \mathsf{Hash}(w_1 || \ldots || w_{|\texttt{attr}_R|})$ and conducts a two-layer secret sharing on S according to $R.\texttt{Cond}$. Specifically, S is firstly broken into k first layer shares (denoted as $s_1, s_2, \ldots, s_k$) through (k, k)-secret sharing, and then for the first layer share s_i, the master node further breaks it into $|c_i|r$ sub-shares (denoted as $s_{i,1}, s_{i,2}, \ldots, s_{i,|c_i|r}$) through $(t, |c_i|r)$-secret sharing, where $|c_i|$ denotes the number of properties in c_i and r is a system parameter restricting the number of verifying nodes. Then master node assigns the verification tasks (e.g. location verification, security property verification) to selected verifying nodes (the total number of verifying nodes is $r\sum_{i=1}^{k} |c_i|$), and each verification task includes verifying a particular property against the corresponding condition. The verification task is distributed and assigned to each verifying node along with a sub-share $s_{i,j}$ for $i = 1, 2, \ldots, k$ and $j = 1, 2, \ldots, |c_i|r$.

Besides assigning verification tasks, the master node needs to inform the selected worker nodes where to verify their properties. To this end, a verification direction message of the form

$$(\{(vlist_i, w_i)\}, rnd, \mathsf{Enc}_S(data), \mathsf{Sig}(\mathsf{Hash}(\{(vlist_i, w_i)\}||rnd||data))$$

is required to be delivered to each selected worker node, where $(vlist_i, w_i)$ is the pair of verifying node list and its assigned property to be verified, $\mathsf{Enc}_S(data)$ is the encrypted data item using key S (or the address where the encrypted data is located), rnd is a random sequential number for preventing replay attack and $\mathsf{Sig}(\mathsf{Hash}(\{(vnode_i, w_i)\}||rnd||data)$ is a signed hash of all the message content to ensure authentication and integrity of the entire message.

Upon receiving a verification direction message, the worker node (say $wnode_i$) sends t claims for each property w_i to the corresponding verifiers in $vlist_i$ to be verified. The message to be sent to verifying nodes includes the verifying property w_i, the corresponding claim c_i, a random nonce non and a hash of the content for guaranteeing message integrity. For example, if three verifiers are in charge of property $location$ verification and the threshold t is set 2, the worker node randomly picks verifying nodes, and respectively sends a request message $(location, c_{location}, non, \mathsf{Hash}(location||c_{location}||non))$ to two of them. Upon receiving the request message, verifying node $vnode_i$ and the worker node $wnode_i$ engage in a property-specific verification protocol. As the protocol is successfully completed, the share for $vnode_i$ is released to $wnode_i$.

Upon completing t successful verifications for each property in sub-clause c_i, the worker node is able to obtain $t|c_i|$ shares to reconstruct the first layer share s_i. Notice that since c_i is a disjunctive sub-clause, the worker node only needs t sub-shares for reconstruction (even some of them originate from the verification of different properties). The rest of sub-shares could be used for verifying the correctness of reconstruction, i.e., check whether all obtained the sub-shares are from a single secret. In a similar way, the other first layer shares can be obtained, and the master could further access the data item by reconstructing S and decrypting $\mathsf{Dec}_S(data)$.

6.2 Number of Verifiers for Collusion Control

Let $c_1 \wedge \ldots \wedge c_k$ denote the condition component in a rule to be evaluated against a worker node, where c_i $(1 \leq i \leq k)$ is a disjunctive form. Since some peer worker nodes may be corrupted and may not send back the requested secret share in time, we estimate the minimum number (denoted as n) of nodes needed for a worker node to successfully compute the rule effect from received verification results, at a probability larger than a given threshold ρ. Specifically, $n = \sum_{i=1}^{k} |c_i|r$, where $|c_i|$ denotes the number of worker properties in c_i, and r denotes the number of verifying nodes needed for each property to guarantee the desired verification successful rate ρ.

Suppose that *prob* is the probability of a verifying node being corrupted. The probability $prob_S$ of receiving secret shares from non-corrupted nodes could be computed as follows.

$$prob_S = \prod_{i=1}^{k} \binom{r|c_i|}{t} prob^{r|c_i|-t}(1-prob)^t \tag{1}$$

$$= (1-prob)^t \prod_{i=1}^{k} \binom{r|c_i|}{t} prob^{r|c_i|-t} \tag{2}$$

Equation (2) can be understood as follows. For each disjunctive sub-clause c_i in $R.\texttt{Cond}$, There are $r|c_i|$ verifying nodes having been assigned. Since each disjunctive sub-clause c_i needs to be verified at least t times, there are $\binom{r|c_i|}{t}$ different ways to choose t from $r|c_i|$ nodes. The number of combinations is then multiplied with the probability for t nodes not being corrupted, i.e., $prob^{n-t}(1-prob)^t$ to get the probability of successfully reconstructing the share s_i for c_i. Finally, the probabilities of k disjunctive sub-clauses are multiplied together to compute the final probability $prob_S$.

The corruption probability *prob* could be obtained from statistic data while t is a system parameter with respect to the user desired reliability level. Given known values of t and *prob*, we can compute the minimum value of r and hence the minimum value of n by resolving the following inequality $prob_S \geq \rho$.

6.3 Property Specific Verification

Verifying the properties associated with any worker (see Sect. 6.1 of the verification protocol), entails some property-specific verification protocols. We now present two examples of two possible types of such protocols.

Location Verification. Location specific verification protocol includes two main steps. First, the verifying node ascertains that the input and output locations specified by the worker node match its actual input and output locations, and checks the locations of the virtual machine hosts to perform computations satisfy the location requirement specified by the user. Second, the verifying node continues to check whether the directories specified for the input and output as well as the computation assemblies indeed exist. The latter location verification protocol is treated by our system as a security verification task, and is similar to file access security verification, i.e., the verifier tries to either store or access a document from the specified directory

To estimate a node's location with reasonable accuracy, the verifier can test and analyze the round trip time (RTT) of a message sent from the worker node to estimate its source, following an approach similar to the mulitlateration scheme used for distance verification in mobile ad-hoc networks [6]. Specifically, in the MapReduce environment, the verifying node could have multiple sub-nodes from different locations working as sub-verifiers, and know the maximum, minimum and average number of hops from its own location to the geographical locations wherein the sub-verifiers are located. Each sub-verifier requests the worker node to echo a message within a given number of hops or a specified time time interval. With the knowledge of sub-verifiers' locations, the verifying node can then use the minimum and maximum time/number of hops collected by the sub-verifiers to estimate the worker's location. Notice that the number of hops and the time constraint requested by each verifier should be varied, such that the worker node could not know the location being requested beforehand.

Security Capabilities. The restrictions on security capabilities are expressed to identify whether a node is capable of providing basic security functions. One such example is the support of file level access control, or encryption/decryption. Additional properties also include database access control, private calculations, secure storage, etc. By specifying one or more of these security properties in a policy, clients could gain security guarantees on the MapReduce computation. Intuitively, these security properties specified by client in policy could be numerous, and the corresponding verification protocol may change accordingly. In what follows, we briefly discuss the verification of encryption/decryption support for instance.

The encryption/decryption support could be verified using the cryptographic algorithm verification program provided by NIST (National Institute of Standards and Technology) [21]. Specifically, the verification program maintains a list of implementations of various algorithms such as the AES, DED, Triple-DES. For each of the algorithms, the program also has a set of tests built to verify different modes of operation of these algorithms, with different key sizes. When the start of verification, the program requests configuration information, and then provides the worker node with some test data (i.e. the key, some plaintext, and an initialization vector if applicable) to be processed. The results are finally sent back to the verifying node and validated to identify whether the worker node's implementation of the algorithm is indeed standard-compliant.

7 Deployment and Evaluation

In this section we discuss our proof-of-concept implementation, followed by a discussion of the results of our experimental analysis on the proposed protocols.

7.1 Deployment Overview

We implemented the proposed framework on top of Microsoft Azure framework. For our deployment, we used the Daytona as our MapReduce runtime, and selected the West US affinity group to create and co-host the host service. In our testbed, we allocated a varying number of VMs per core, starting from 1 and scaling up to 20. We deployed 5 of these projects, to utilize a total of about 5 cores. The sample application of k-means is updated with our modules using Visual Studio 2012.

We extended Daytona's modules to integrate the functionalities offered by the proposed ACEM framework. The core modules of ACEM are Policy Enforcement Point (PEP) and Policy Decision Points (PDP), which are respectively in charge of the enforcement and evaluation of policies.

– PEP includes data pre-processing, tainting and evaluation, and is typically deployed at the master node. (1) Pre-processing is a method integrated into the `IDataPartitioner` class, such that the data loaded from the input file could be treated in a single batch into string arrays for pre-processing. (2) Taint is allowed to be part of the `Controller` and `IMapper` respectively deployed at both worker nodes and master node, and to carefully taint the data in different ways based on the proprietary nature of the applications, to avoid *bleaching*. (3) Evaluation of worker nodes' properties.
– PDP implements the collaborative verification modules including data partitioning, verifiers' selection and secret key generation, all of which reside with the classes `Controller` and `IDataPartitioner` at the master node. In our current prototype, the `Controller` calculates the minimum number of verifiers required [11] according to Eq. (6.2), with an assumption that the probability that any given verifier is corrupted is 0.1, and uses a random 256-bit length *nonce* to generate the requisite number of key share. The communication protocols between the worker nodes and the verifying nodes reside at both the `IMapper` and `IReducer` classes. Every node hosts the methods required to carry out property verification.

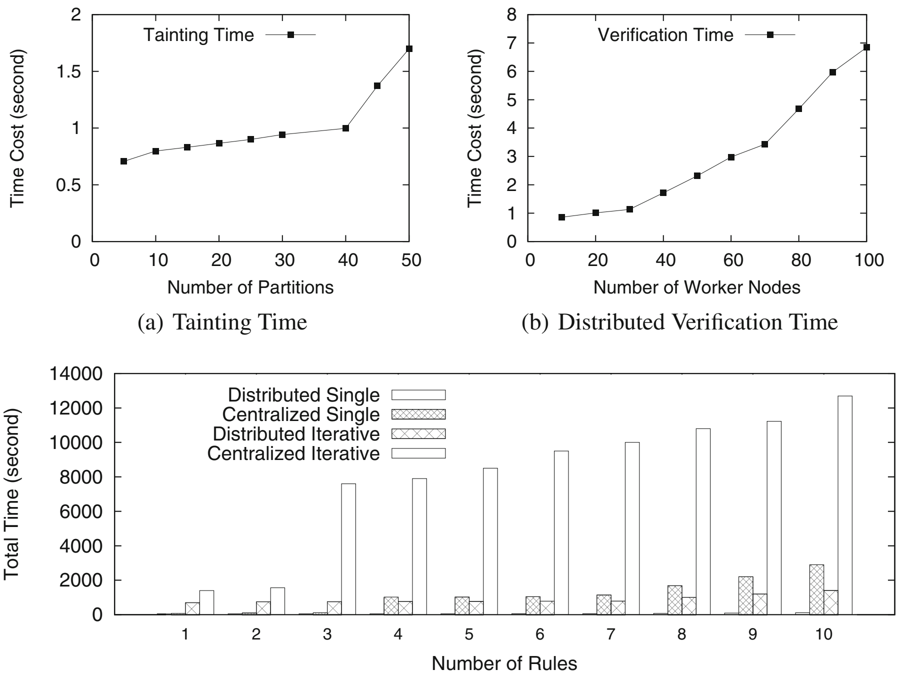
7.2 Experimental Evaluation

Since the data partitioning only needs to be done once off-line for all kinds of analysis tasks, in the following, we report the runtime overhead caused by tainting and collaborative policy evaluation.

The first set of experiments aims to measure the overhead introduced by tainting. Tainting is executed by the master node before mapping phase, and the mappers have to apply the taint again once the mapping is completed. Since the master and worker nodes have the similar configurations, the measurement for both tasks can be done at any node. We simply chose a worker node at random for these measurements. The results of this evaluation are reported in Fig. 2(a). As shown in the figure, it is not surprising to see that the time taken for tainting increases with the number of the data partitions. This is because the more data partitions, the more data items need to be tainted.

Next, we evaluate the efficiency of the collaborative policy evaluation protocol introduced in Sect. 6. Figure 2(b) shows the time from the property extraction to the key generation by executing the two-layer secret sharing; and then Fig. 2(c) shows the total time for executing a k-means clustering task that involves the actual verification of a worker node's properties including nodes' capabilities, location, files access, support for cryptographic protocols (i.e. AES, DES, 3DES). The detailed explanation of the results are the following.

As shown in Fig. 2(b), the time taken for the keys to be obtained by the verifying worker nodes increases linearly with the total number of worker nodes.



(a) Tainting Time (b) Distributed Verification Time

(c) Total Execution Time

Fig. 2. Processing Time

Specifically, in the experiments, up to 36 nodes act as verifiers for 99 workers. When a fewer number of worker nodes are used for a job, the number of verifiers is accordingly reduced to maintain a ratio of worker nodes to verifiers as about 2.75 : 1, to ensure that the probability of receiving a key share from an uncorrupted verifier is higher than 0.5. Therefore, the more worker nodes to be verified, the more key shares need to be generated and hence the time increases.

Figure 2(c) shows the comparison of our proposed collaborative policy evaluation approach (dented as "distributed") against a centralized policy evaluation approach (denoted as "centralized" in the figure). Both one-round K-means clustering and iterative (10-round) K-means clustering are considered. It is clearly shown that our distributed approach is several orders of magnitude faster than the centralized approach. This is because the centralized approach requires the master node to conduct all the property verifications and the master node becomes the performance bottleneck. In addition, we would like to mention that our approach incurs very little overhead to the original K-means algorithm. For instance, the runtime for the original k-means algorithm on 1000 data points averages at around 40 seconds for one iteration. After introducing our approach for privacy protection, the runtime is only 45 seconds (about 10 % overhead).

8 Conclusion

In this paper, we proposed a novel access control mechanism for node selection and data processing in MapReduce, i.e., the ACEM (Access Control Enforcement in Mapreduce). ACEM provides data owners with strong controls on the worker nodes managing their potentially sensitive data. Bay restricting access to nodes with desirable properties, simultaneously not burdening users with complex configuration tasks, data owners can gain confidence on the trustworthiness of the computation.

Needless to say, our solution tackles only a small problem in the complex space of secure and customized computation in the cloud settings, and has some limitations itself. For instance, even if successfully verified, there is no guarantee that if some functional properties are tested (like cryptographic support) the worker will actually behave as expected. Further, since properties are verified by at least t nodes, the nodes can cheat the verification process if enough of them collude with each other. In future, we will strengthen our current approach by ensuring the verifiers selected do not consist of any loops [10]. Alternatively, we may employ incentivized supervision schemes (e.g. [20]).

Acknowledgement. Portion of the work from Dr. Squicciarini was funded under the auspices of National Science Foundation, Grant #1250319. Portion of the work from Dan Lin was funded by the National Science Foundation (NSF-CNS-1250327 and NSF-DGE-1433659).

References

1. Amazon: Amazon EMR with the mapr distribution for Hadoop (2009). http://aws.amazon.com/elasticmapreduce/mapr/
2. Ananthanarayanan, G., Kandula, S., Greenberg, A.G., Stoica, I., Lu, Y., Saha, B., Harris, E.: Reining in the outliers in map-reduce clusters using mantri. In: OSDI 2010 Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation, vol. 10, p. 24 (2010)
3. Barga, R.: Project Daytona: Iterative mapreduce on Windows Azure (2011)
4. Blanton, M., Atallah, M.J., Frikken, K.B., Malluhi, Q.: Secure and efficient outsourcing of sequence comparisons. In: Foresti, S., Yung, M., Martinelli, F. (eds.) ESORICS 2012. LNCS, vol. 7459, pp. 505–522. Springer, Heidelberg (2012)
5. Brenner, M., Wiebelitz, J., von Voigt, G., Smith, M.: Secret program execution in the cloud applying homomorphic encryption. In: Proceedings of the 5th IEEE International Conference on Digital Ecosystems and Technologies Conference (DEST), pp. 114–119 (31 May–3 June 2011)
6. Capkun, S., Hamdi, M., Hubaux, J.P.: Gps-free positioning in mobile ad-hoc networks. In: Proceedings of the 34th Annual Hawaii International Conference on System Sciences, p. 10. IEEE (2001)
7. Chen, X., Li, J., Ma, J., Tang, Q., Lou, W.: New algorithms for secure outsourcing of modular exponentiations. In: Foresti, S., Yung, M., Martinelli, F. (eds.) ESORICS 2012. LNCS, vol. 7459, pp. 541–556. Springer, Heidelberg (2012)

8. Dalton, M., Kannan, H., Kozyrakis, C.: Raksha: a flexible information flow architecture for software security. In: ACM SIGARCH Computer Architecture News, vol. 35, pp. 482–493. ACM (2007)
9. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008). http://doi.acm.org/10.1145/1327452.1327492
10. Dutta, D., Goel, A., Govindan, R., Zhang, H.: The design of a distributed rating scheme for peer-to-peer systems. In: Workshop on Economics of Peer-to-Peer Systems, vol. 264, pp. 214–223 (2003)
11. Hazewinkel, M.: Lagrange Interpolation Formula. Encyclopedia of Mathematics. Springer, Berlin (2001)
12. Kagal, L., Finin, T., Joshi, A.: Moving from security to distributed trust in ubiquitous computing environments. IEEE Comput. **34**(12), 154–157 (2001)
13. Lordan, F., et al.: Servicess: an interoperable programming framework for the cloud. J. Grid Comput. **12**(1), 1–25 (2013)
14. McSherry, F.D.: Privacy integrated queries: an extensible platform for privacy-preserving data analysis. In: Proceedings of the 2009 ACM SIGMOD International Conference on Management of data, pp. 19–30. ACM (2009)
15. Microsoft: Windows azure (2010). http://www.windowsazure.com/en-us/
16. Moca, M., Silaghi, G., Fedak, G.: Distributed results checking for mapreduce in volunteer computing. In: 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), pp. 1847–1854 (2011)
17. Myers, A.C.: Jflow: practical mostly-static information flow control. In: Proceedings of the 26th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 228–241. ACM (1999)
18. Naehrig, M., Lauter, K., Vaikuntanathan, V.: Can homomorphic encryption be practical? In: Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop, pp. 113–124. ACM (2011). http://doi.acm.org/10.1145/2046660.2046682
19. Roy, I., Setty, S.T.V., Kilzer, A., Shmatikov, V., Witchel, E.: Airavat: security and privacy for mapreduce. In: Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation, NSDI 2010, p. 20. USENIX Association, Berkeley (2010). http://dl.acm.org/citation.cfm?id=1855711.1855731
20. Saroiu, S., Gummadi, K.P., Gribble, S.D.: Measurement study of peer-to-peer file sharing systems. In: Electronic Imaging 2002, pp. 156–170 (2001)
21. National Institute of Standards and Technology: Cryptographic module validation program management (2013). http://csrc.nist.gov/groups/STM/cmvp/index.html
22. Vizard, M.: Hybrid cloud computing faces multiple challenges (2013). http://www.cioinsight.com/it-strategy/cloud-virtualization/hybrid-cloud-comp
23. Vu, V., Setty, S., Blumberg, A.J., Walfish, M.: A hybrid architecture for interactive verifiable computation. In: Proceedings of the IEEE Symposium on Security and Privacy (2013)
24. Wei, W., Du, J., Yu, T., Gu, X.: Securemr: a service integrity assurance framework for mapreduce. In: Proceedings of the Computer Security Applications Conference, ACSAC, pp. 73–82 (2009)
25. Zhang, K., Zhou, X., Chen, Y., Wang, X., Ruan, Y.: Sedic: privacy-aware data intensive computing on hybrid clouds. In: Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, pp. 515–526. ACM (2011)