Towards A Cross-Domain MapReduce Framework

Thuy D. Nguyen, Mark A. Gondree, Jean Khosalim, Cynthia E. Irvine
Department of Computer Science, Naval Postgraduate School
Monterey, California 93943
{tdnguyen, mgondree, jkhosali, irvine}@nps.edu

***Abstract*—The Apache™ Hadoop® framework provides parallel processing and distributed data storage capabilities that data analytics applications can utilize to process massive sets of raw data. These Big Data applications typically run as a set of MapReduce jobs to take advantage of Hadoop's ease of service deployment and large-scale parallelism. Yet, Hadoop has not been adapted for multilevel secure (MLS) environments where data of different security classifications co-exist.**

To solve this problem, we have used the Security Enhanced Linux (SELinux) Linux kernel extension in a prototype cross-domain Hadoop on which multiple instances of Hadoop applications run at different sensitivity levels. Their accesses to Hadoop resources are constrained by the underlying MLS policy enforcement mechanism. A benefit of our prototype is its extension of the Hadoop Distributed File System to provide a cross-domain read-down capability for Hadoop applications without requiring complex Hadoop server components to be trustworthy.

Keywords: MapReduce, Hadoop, cross-domain services, multilevel security.

I. INTRODUCTION

The US Department of Defense (DoD) and US Intelligence Community (IC) recognize they have a Big Data problem. High volumes of streaming data are ingested from the tactical edge, originating from a variety of sensors [1]. The National Geospatial-Intelligence Agency anticipates collecting on the order of four petabytes, annually [2]. Mission analytics may further require archival data to compare with current intelligence data.

Agencies are embracing a data-centric model that empowers analysts to query data anywhere in the cloud, based on need-to-know. The IC envisions an agile, shared cloud architecture, following this paradigm [3]. The NSA reportedly operates three private clouds, already: a utility cloud, a storage cloud and a data cloud; the latter uses versions of Hadoop and MapReduce to manage intelligence analytics [4]. The Naval Tactical Cloud (NTC) employs distributed cloud-based data services to provide timely access to mission-relevant intelligence and operational data under advanced Anti-Access/Area Denial conditions. The NTC architecture leverages an open-source software stack featuring HDFS, Hadoop MapReduce, ZooKeeper and Accumulo [5].

The ability of these products to appropriately handle data of multiple classifications is dubious. Researchers have already discovered commercial cloud products where information flows violating the isolation requirements for multi-tenancy both leaked cryptographic keys and exposed private data [6][7].

The prototype described here is part of our larger investigation into security issues for cloud computing with Big Data from sources of different sensitivities. We describe our initial experiments using a modified MapReduce platform to perform Big Data analytics across security domains in an MLS environment, leveraging a novel architecture supported by an underlying secure platform. This MLS-aware *cross-domain Hadoop* (CD-Hadoop) prototype is implemented using Security Enhanced Linux[1] (SELinux) [8] configured to enforce MLS policy following the Bell-LaPadula confidentiality policy model [9]. SELinux mediates access to information of different sensitivity levels based on hierarchical and non-hierarchical security labels of the subjects and objects; Hadoop itself is not involved in MLS policy enforcement. The problem of inadvertent contamination of low information by inept or malicious users is beyond the scope of this work.

The remaining sections describe Hadoop [10], the concept of operations, the system architecture, and the implementation of a CD-Hadoop prototype. We conclude by discussing performance tests used to evaluate the overhead incurred while processing read-down operations.

II. BACKGROUND

Apache Hadoop is an open-source implementation that is based on the Google File System [11] and the MapReduce parallel computational model developed by Google [12]. The Hadoop Distributed File System (HDFS) consists[2] of three components: the NameNode, Secondary NameNode and DataNode. Two components comprise the Hadoop MapReduce engine: the JobTracker and the TaskTracker. In a Hadoop cluster, there are one NameNode, one Secondary NameNode, one JobTracker, and multiple DataNodes and TaskTrackers.

To run a MapReduce job, the client first copies the job's input data, configuration file and Map and Reduce functions onto the Hadoop file system as HDFS files. Each file is divided into multiple HDFS blocks, stored on the DataNodes. The client then submits the job to the JobTracker, which creates a set of Map and Reduce tasks for the job. The JobTracker delegates these tasks to different TaskTrackers and monitors the progress of all jobs. Each TaskTracker executes the tasks assigned to it by the JobTracker and regularly informs the JobTracker about the status of all outstanding tasks and when it is ready to run a new task. Next, we describe the NameNode

The views expressed in this material are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

[1] In particular, Fedora 13 with Security Enhanced Linux enabled was used.

[2] Description reflects Hadoop v0.20.2, used in our prototype.

and DataNode components in more detail, as our design significantly impacts those components.

A. NameNode

A NameNode is a daemon that manages the HDFS file system namespace and coordinates file access requests from clients. An HDFS file consists of blocks that are replicated and stored on different DataNodes. The block size and replication factor are configurable; defaults are 64MB and 1x, respectively. The NameNode decides where the blocks are to be replicated and informs the corresponding DataNode of its selection.

The primary HDFS namespace data structure (*fsImage*) contains the metadata associated with individual files, e.g., file properties and the locations where each block and its replicas are stored. The NameNode also uses a transaction log (*edits*) to keep track of changes to the HDFS metadata. Both data structures are stored as files in the NameNode's local file system. During start-up, the NameNode reads both files, creates a new *fsImage* file in volatile memory, applies the changes indicated by the *edits* log, clears *edits*, and persists both back to disk. During runtime, whenever the *edits* structure is updated, it is flushed to disk.

B. DataNode

A DataNode is a daemon that provides the block storage functionality for the cluster. The DataNode stores blocks as files in its local file system. After getting information about the blocks associated with a particular HDFS file from the NameNode, a client sends data requests to the DataNodes that are directly responsible for those particular blocks. The DataNode sends periodic messages to the NameNode, informing it of its status.

III. CONCEPT OF OPERATIONS

Hadoop enforces an application-level discretionary access control policy using permission bits similar to UNIX file access controls. Hadoop also maintains user sessions based on the user login IDs. However, with respect to mandatory access control, Hadoop lacks the ability for an authenticated user to negotiate a session at a specific sensitivity level, which would be used to determine the resources that user can access under MLS policy. In our CD-Hadoop prototype, the user session level is implicitly established by the sensitivity level of the network interface and TCP/IP port from which the request is received.

The Hadoop file system is structured as a hierarchical tree of directories and files, with an interface similar to the traditional UNIX file system. In a traditional Hadoop cluster, there is only one file system and its root is at "/". In our MLS-enhanced cluster, there are multiple file systems, one per sensitivity level.

Each of these file systems is managed by an HDFS instance that runs at that level. The root directory of a file system at a particular level is expressed as `/<level>` (e.g., `/unclass`, `/sec-level0`). The `<level>` value is a user-defined string that is administratively associated with one SELinux sensitivity level. To be backward compatible with existing applications, the CD-Hadoop prototype treats the traditional root directory (`/`) as the file system root at the user's session level. For example, a client running at SECRET can access files stored under the SECRET root directory as either `/secret/<filename>` or, simply, as `/<filename>`.

A user can access HDFS file objects using tools provided with the Hadoop distribution (e.g., FS Shell) and with HDFS-aware applications that use the HDFS API. A user can read and write file objects at their session level, but can only read file objects at lower levels, i.e., the user can read any objects whose level is dominated by their session level. Writes are only permissible at the user's current session level.

IV. SYSTEM OVERVIEW

Software is considered *MLS-aware* if it executes without privileges in an MLS environment, and yet takes advantage of that environment to provide useful functionality [13]. For example, on a system enforcing a mandatory security policy as modeled by Bell and LaPadula, when an application executes at a particular sensitivity level, it can read from resources labeled at the same or lower levels but can only write to resources labeled at the same level or higher. If the application is modified to reflect the underlying mandatory policy—e.g., to return the level of the data or make decisions based on the level of the data—we say that the application is MLS-aware.

The Hadoop code base is large and very complex. Thus, it is prudent to minimize the code changes required to make Hadoop MLS-aware. The Hadoop MapReduce engine enables parallel data processing while HDFS provides distributed data storage. Although the MapReduce server processes keep their internal data structures on the local file system, the data used by the Map and Reduce application tasks are kept in the HDFS. Hence, this project focuses on making HDFS MLS-aware, a step towards a secure Hadoop platform suitable for use in MLS environments. In our proof-of-concept design, HDFS server processes running at their particular sensitivity levels are cognizant of the file system namespaces at lower security levels and can access those file objects as the system's security policy permits. To use this design with real data, each physical node in the Hadoop cluster must be hosted on a trusted platform that mediates the node's access to the local file system (where local files are labeled) according to an MLS policy.

Before discussing our implementation of the CD-Hadoop prototype, we describe the high-level functional requirements for the system and its information flow design. The CD-Hadoop system must satisfy the following requirements:

- Allow users to modify data only at their session level;
- Allow users to observe data at their session level and at lower sensitivity levels;
- Support a backward compatible HDFS API, to allow existing applications (which do not require read-down support) to run unmodified;
- Defer MLS policy enforcement to the underlying trusted computing base (TCB);
- Minimize the introduction of trusted processes, which would extend the TCB boundary. In particular, avoid a

trusted proxy that can communicate with all HDFS server processes running at different levels;

- Minimize changes to the existing HDFS software. This is motivated by our desire to both minimize code and simplify upgrading to new Hadoop releases.

Given these requirements, an information flow design for the CD-Hadoop system was developed (see Fig. 1). Instances of the Hadoop server processes run at each sensitivity level on the same physical node. The underlying TCB (not the Hadoop servers) enforces domain separation, information flow control, and mandatory access controls. Our SELinux-based prototype uses sixteen sensitivity levels and up to 1,024 categories.

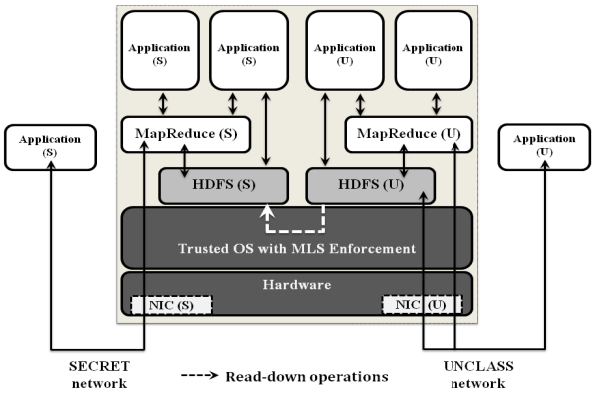

Figure 1. MLS-aware Hadoop Information Flow

Information flow between an application (both local and remote) and the Hadoop server processes is constrained by the MLS policy. Specifically, the application can only communicate with Hadoop processes running at the same level as the application's session level. When an application running at SECRET requires read-access to an HDFS file at UNCLASS, it must request the HDFS server running at SECRET to perform this operation on its behalf. Although the HDFS server is not trusted with respect to MAC, it is trusted to perform its other security functions correctly. Details of the read-down implementation are discussed in the next section.

V. Design and Implementation

This section describes the design and read-down mechanisms of the prototype CD-Hadoop system.

A. Prototype Design

In a CD-Hadoop cluster, there is one physical NameNode node and multiple physical DataNode nodes. All per-level NameNode instances run on the same physical NameNode node, whereas DataNode instances are distributed across different physical nodes. A notional heterogeneous MLS-enhanced cluster, capable of handling data labeled at three different levels, is illustrated in Fig. 2.

There are two types of DataNode instances. A *primary* DataNode instance is the main handler of the HDFS blocks, i.e., it is the owner of the files used to store the blocks in the local file system. A *surrogate* DataNode instance running at the user session level is responsible for handling read-down requests on behalf of a primary DataNode instance running at the sensitivity level of the desired blocks. This is necessary because a client cannot communicate directly with a primary DataNode instance if the client's session level does not dominate the sensitivity level of the primary DataNode instance, i.e., no write-down. The number of primary and surrogate DataNode instances running on a physical node is defined administratively via the HDFS configuration files.

In its original design, when a client requests a file, the NameNode daemon will direct the client to retrieve the associated blocks from the DataNodes responsible for those blocks. This process becomes more complicated with the introduction of read-down support. To handle a read-down request, a NameNode instance running at the client's sensitivity level must obtain the metadata of the requested file and the storage locations of the associated blocks from the NameNode instance running at the requested (lower) sensitivity level. Similarly, when a client contacts a DataNode instance to request a block at a lower level (as directed by the NameNode), the DataNode instance must locate and read the file used to store the requested block in its local file system. The level of this local file is the same as the level of the requested block.

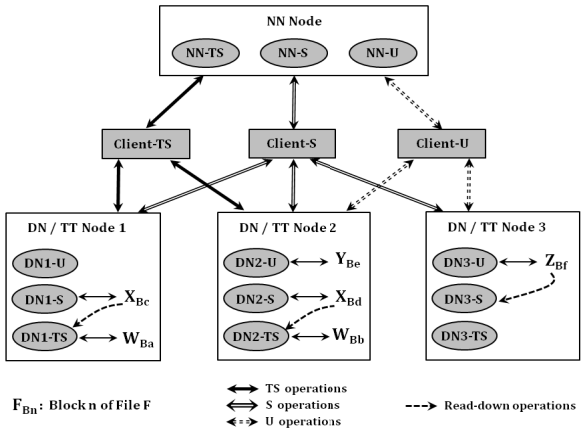

Figure 2. Example of read-down operations

In the example scenarios depicted in Fig. 2, X and W are multi-block files with blocks stored on different nodes, while Y and Z are single-block files stored on separate nodes. When Client-TS requests file W, a file at its level, the NameNode at that level (NN-TS) directs it to retrieve the blocks from the corresponding TS DataNode instances (DN1-TS, DN2-TS). In contrast, when Client-TS requests read access to file X at a lower level (S), it is instructed to contact the co-located TS DataNode instances (DN1-TS, DN2-TS), which are the surrogate DataNodes for the blocks associated with the file X (on DN1-S, DN2-S). This is because the client's session level is higher than the levels of the blocks, and the MLS policy enforcement does not allow write-down or read-up operations. Upon receiving the requests, the TS surrogate nodes for X (DN1-TS, DN2-TS) will perform read-down operations (shown as dashed lines in Fig. 2) and return the results to Client-TS. Similarly, node DN3-S handles read-down requests for file Z labeled at U. No other scenarios in Fig. 2 involve read-downs and are straight-forward, e.g., Client-S accesses file X.

B. Prototype Implementation

The read-down mechanisms introduced in the CD-Hadoop prototype mostly impact the NameNode and DataNode logic and not the JobTracker and TaskTracker logic, as those components only interact with the NameNode and DataNode as HDFS clients.

1) Changes to NameNode

In its original design, whenever a change is made to the file system (e.g., when a directory is created), the NameNode daemon updates the *fsImage* and *blockMap* data structures that it uses for file system management. The *fsImage* database keeps track of the current file hierarchy, while the *blockMap* database records the DataNode where each block is stored. Both databases are kept in the NameNode's private memory and are not visible to other NameNode instances.

Our design introduces two additional data structures, the *Cache-fsImage* and the *Cache-blockMap*, to allow a high sensitivity NameNode instance to look up file metadata and block storage information associated with a file at a lower level. Each NameNode instance manages its own *Cache-fsImage* and *Cache-blockMap* on the local RAM disk. For every write request, the NameNode instance servicing the request updates pertinent information in the *fsImage* and *blockMap* databases, and then copies the entire content of these databases to the *Cache-fsImage* and *Cache-blockMap* structures.

Other NameNode instances consult the *Cache-fsImage* and *Cache-blockMap* structures to handle read-down requests, if their sensitivity levels strictly dominate the sensitivity levels of these objects. These other NameNodes use the file metadata obtained from the *Cache-fsImage* to check for file permission and block allocation, and the blocks-to-DataNodes mapping information from the *Cache-blockMap* to inform the requesting client where blocks are located. The client contacts surrogate DataNodes to perform the actual read-down data transfer.

Concurrent access to the *Cache-fsImage* and *Cache-blockMap* by different processes is synchronized using a lock-free multiple-reader, single-writer integrity mechanism—a high reader gets a valid view of data at a lower level without using locks. Our read-and-retry data consistency mechanism has been designed so that no new covert channels are introduced.

2) Changes to DataNode

Each block is stored as two files—a block file and a metadata file—on either the local file system or a remote file system, e.g., an NFS volume. In its original design, the DataNode daemon maintains a blocks-to-files map to keep track of the file used to store each block on the local file system. Created during initialization, the *volumeMap* is updated at run-time whenever a new block is allocated to the DataNode or an existing block becomes inaccessible, e.g., the block is deleted or the file storing the block is corrupted.

When a client asks to read an HDFS file at a lower level, the NameNode instance running at the client's session level directs the client to contact the file's surrogate DataNode, co-located with the primary DataNode actually handling the file's blocks. To return a requested block at a lower level, a surrogate DataNode instance must have access to the *volumeMap* maintained by the primary DataNode instance, to locate the (non-HDFS) files associated with the requested block. Since the *volumeMap* is kept in each DataNode's private memory, a *Cache-volumeMap* is used to capture the content of the *volumeMap* on the system's RAM disk, so that all surrogate DataNodes whose levels dominate that of the *Cache-volumeMap* can read it while handling read-down requests.

The DataNode instances use the same read-and-retry synchronization mechanism utilized by the NameNode to access the shared *Cache-volumeMap*.

C. Implementation Discussion

Given our objectives to minimize changes to the Hadoop code base and avoid introducing trusted processes, a number of design choices were made for the prototype implementation.

1) Extended Block ID to Distinguish Levels

To ensure uniqueness, the NameNode daemon generates a 64-bit pseudo-random number (via the Java pseudo-random number generator) for the Block ID of each new block. In the CD-Hadoop prototype, the Block ID is extended to include the security label associated with the block, to partition the namespace across the Hadoop instances. The Extended Block ID includes a new 4-byte identifier (Level ID), describing the sensitivity level of each block. An alternative design is to introduce a new process to manage a pool of Block IDs for the entire cluster. In this alternative design, instead of generating the Block IDs, the NameNode would obtain them from the new process; however, adding a new process would complicate the overall design, and the additional inter-process communication may further decrease the performance of the NameNode.

With a Level ID in the block identifier, the DataNode can determine independently whether to use its own *volumeMap* or a *Cache-volumeMap* at some lower level to look for the file data. Without the Level ID, the DataNode must search, with a consequent performance impact, its *volumeMap* and the *Cache-volumeMap*s at all lower levels until the required data is found.

2) Scalability across Sensitivity Levels

Regarding block storage, the original HDFS design scales linearly, as DataNodes work independently and the number of DataNodes in a cluster can grow over time. The management of the file namespace and block information, however, introduces a performance bottleneck, since there is only one NameNode in the cluster and the NameNode keeps the *fsImage* and *blockMap* databases entirely in memory. In addition, the NameNode is also highly susceptible to resource exhaustion.

The recently introduced HDFS Federation architecture [14] addresses NameNode scalability by keeping block information across multiple NameNodes. Although this approach partitions the Hadoop file system into multiple namespaces, the memory exhaustion problem still exists since the NameNode continues to keep the *fsImage* and *blockMap* databases in memory.

In the MLS-enhanced environment, the memory exhaustion problem on the NameNode is exacerbated since there are multiple instances of the NameNode server process on a physical node, requiring additional memory for the *Cache-fsImage* and *Cache-blockMap* structures. Using a separate

physical node as a Cache Manager to maintain these databases may ameliorate this problem. With multiple single-level instances of the Cache Manager on the new node, one per sensitivity level, each Cache Manager instance would provide services that a NameNode instance at the same level could use to store and retrieve these databases.

Using the sensitivity level of the receiving NIC as the requestor's session level, the current prototype can only handle simple policies with a small number of sensitivity levels—a typical server platform can support up to 16 NICs. This limits the prototype's ability to scale up to a larger number of levels. A more flexible system design such as the Monterey Security Architecture (MYSEA) could be leveraged to overcome this limitation. MYSEA supports an MLS LAN interface on which users can negotiate sessions at different levels [15][16]. Similar to MYSEA, using one MLS-NIC would allow the prototype to support more complex policies.

TABLE 1. MODIFICATION COMPLEXITY (SLOC)

	SLOC		Delta	% Change
	Original Hadoop	MLS-aware Hadoop		
NameNode	14373	15974	1890	13.15%
DataNode	6914	7399	692	10.01%
Other modules	68328	68890	732	1.07%
Total HDFS changes	89615	92263	3314	3.70%

3) *Implementation Complexity*

Source lines of code (SLOC) can be quickly calculated and thus is commonly used as an intuitive metric to estimate the complexity of software and development cost in terms of program size. The Count Lines of Code (CLOC) tool [17] was utilized to compare the SLOC of the original Hadoop HDFS code with the MLS-aware HDFS code (see Table 1).

CLOC can calculate differences in blank, comment, and source lines in a given file, directory, or archive. The SLOC values shown in Table 1 summarize the number of source lines that were added, removed, modified or unchanged. The delta value is the sum of the addition, removal, and modification of source lines. The percent change value reflects the overall increase in the SLOC between the original and MLS-aware Hadoop, demonstrating that the prototype appears to meet our requirement to minimize changes to the existing software (under the assumption that <10% overall change is acceptable).

VI. PERFORMANCE EVALUATION

This section describes the performance of our MLS-aware Hadoop prototype on a virtualized test environment.

A. Benchmark Configuration

The test cluster consists of eleven nodes distributed across two racks. Each node in the cluster is a virtual machine (VM) hosted on VMware ESXi 5.0.0. Rack-1 contains three server blades (each, a Dell PowerEdge R710 system, with 8 CPUs x 2.925 GHz with hyper-threading active, 48GB of memory and Gigabit Ethernet); Rack-2 contains a single server blade (a Dell PowerEdge R610 system, with 8 CPUS x 2.26 GHz with hyper-threading active, 24GB of memory and Gigabit Ethernet). The racks are connected with multi-port Gigabit Ethernet switches.

The Hadoop distribution includes a number of performance benchmarking tools that run as MapReduce jobs. The following tools were used to gauge the performance of the current prototype: (a) NNBench, a NameNode stress test; (b) TestDFSIO, an HDFS I/O performance test that reads and writes files in parallel; and (c) TeraSort, a combination of three test applications designed to sort large amounts of data. Mahout's *Recommender* example program [18] was used to measure performance, representing a real-world, popular MapReduce application.

Each test scenario was run on both the original Hadoop and the CD-Hadoop. The JobTracker's web interface was used to collect job statistics. The HDFS directory used by each test program was removed before starting each test case.

B. Benchmark Results

The NNBench program stresses the NameNode by creating zero-length files, thus forcing the NameNode to repeatedly update its databases. Under this test, the performance of our prototype degrades almost exponentially as the number of files increases. This performance behavior is caused by the overhead of caching the *fsImage* and *blockMap* every time a file is created/modified. Note that NNBench is designed to strain the NameNode with excessive file system operations with zero data, and is not representative of a normal load.

The TestDFSIO program is designed to measure the I/O performance of HDFS in a normal context. The test consists of writing and reading three datasets of different sizes: 1GB, 10GB and 20GB. The results indicate that for the 1GB and 10GB test cases, the prototype performance is marginally slower for both read and write tests. However for the 20GB test case, the writing overhead is much higher while the reading overhead is slightly lower (but still within the margin of error). These write operations take longer because there are more blocks associated with the 20GB file, so it takes more time to flush the entire *blockMap* to the RAM disk.

The TeraSort test suite was executed three times with different data sizes: 1GB, 10GB and 20GB. For each trial, its three test utilities are invoked in the same order: (TeraGen → TeraSort → TeraValidate). Without read-down operations, the TeraSort test results are roughly comparable to the performance of the original Hadoop. This indicates that the cost of accessing HDFS files has a minimal effect on the overall performance of a typical MapReduce job. However, the performance impact grows exponentially for read-down requests as the data size increases. The additional degradation was caused by the overhead of reading the cache databases, which the NameNode and DataNodes must do to handle a read-down request.

The Mahout Recommender test scenario used a 1MB dataset obtained from GroupLens Research [19], which consisted of 1 million ratings of 4000 movies by 6000 users.

This test consisted of running the Recommender program five times, in three different configurations: Hadoop, CD-Hadoop without read-down, and CD-Hadoop with read-down. Unlike the test programs discussed previously, which use only one MapReduce job, the Recommender test uses ten MapReduce jobs. The averages of the five runs are presented in Fig. 3. The overhead of CD-Hadoop with no read-down, compared to the original Hadoop, is approximately 3.25%; the additional read-down overhead is about 0.08%, a total overhead of 3.34%.

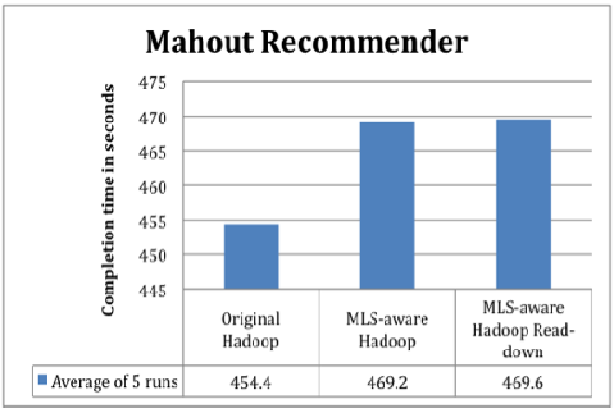

Figure 3. Average of Mahout Benchmark Results

The test data used contained one million entries, but its file size was relatively small relative to typical Big Data applications (about 10MB). Tests with larger datasets may allow more meaningful reasoning about the performance cost of the prototype with multi-job applications like Mahout.

VII. Related Work

Roy *et al.* propose an approach that supports MapReduce computations on sensitive data, while preserving the privacy of the data providers [20]. Airavat runs on SELinux but uses SELinux's default *targeted policy*—a relaxed *type enforcement* policy that only constrains selected (targeted) subjects. Airavat modifies HDFS to enforce SELinux-like mandatory access control, expanding its reference monitor and TCB boundary to include complex application code. This is in contrast with our approach, which depends solely on SELinux's MLS policy enforcement to control information leaks through system resources.

VIII. Conclusion

The CD-Hadoop prototype demonstrates the feasibility and practicality of using Hadoop in an MLS environment. Multiple instances of Hadoop servers can run at different sensitivity levels while their accesses to Hadoop resources are constrained by the underlying trusted OS. Our design does not introduce any trusted processes outside the pre-existing TCB boundary and only affects the HDFS servers. The current implementation is a first step toward a highly secure MapReduce platform that can be used in high-risk MLS environments. Preliminary testing shows notable performance degradation under the pathological workload of the NameNode stress test, but relatively modest performance overhead with small, realistic access patterns. We believe a platform that can access Big Data at multiple classifications—without increasing the classification of data through duplication, necessarily incurring complications due to poly-instantiation—can facilitate better utilization of data for mission-oriented intelligence analysis.

References

[1] D. Meiron, S. Cazares, P. Dimotakis, F. Dyson, D. Eardley, S. Keller-McNulty, D. Long, F. Perkins, W. Press, R. Schwitters, C. Stubbs, J. Tonry, and P. Weinberger, "Data analysis challenges," JASON, Tech. Rep. JSR-08-142, Dec. 2008.

[2] P. Buxbaum, "GEOINT's big data challenge," *Geospatial Intelligence Forum: The Magazine of the National Intelligence Community*, Vol. 10, Issue 3, pp. 4-7, 2012. http://goo.gl/sbYch. Accessed: March 2013.

[3] Intelligence and National Security Alliance, "IC ITE: Doing in common what is commonly done," Whitepaper, Feb. 2013. http://goo.gl/uTIWR. Accessed: March 2013.

[4] J. Nicholar Hoover, "NSA CIO pursues Intelligence-sharing architecture," *InformationWeek Government*, April 21, 2011. http://goo.gl/HMXbY. Accessed: March 2013.

[5] B. Junker, "Navy Tactical Cloud S&T Objectives," presented at the 6th Annual C5ISR Government and Industry Partnership Summit, 2012. http://goo.gl/yMMCE. Accessed: March 2013.

[6] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds," in *Proc. of the 16th ACM Conference on Computer and Communications Security*, 2009.

[7] S. Chen,; R. Wang, X. Wang, and K. Zhang. "Side-Channel Leaks in Web Applications: A Reality Today, a Challenge Tomorrow," in *Proc. of the IEEE Symposium on Security and Privacy*, pp. 191-206, 2010.

[8] "The SELinux Notebook — The Foundations," 2nd Edition. http://goo.gl/9wJkQ. Accessed: March 2013.

[9] D. Bell and L. La Padula. "Secure Computer Systems: Unified Exposition and Multics Interpretation," Electronic Systems Division, USAF. ESD-TR-75-306, MTR-2997 Rev.1. Hanscom AFB, MA. 1976.

[10] Apache Hadoop. http://hadoop.apache.org/. Accessed: Sept. 2012.

[11] S. Ghemawat, H. Gobioff and S-T. Leung, "The Google file system," in *Proc. of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP '03)*, New York, NY, 2003.

[12] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *Proc. of the Sixth Symposium on Operating System Design and Implementation (OSDI '04)*, San Francisco, CA, 2004.

[13] C. E. Irvine, T. Acheson, and M. F. Thompson, "Building trust into a multilevel file system," in *Proc. 13th National Computer Security Conference*, Washington, DC, pp. 450–459, 1990.

[14] "Apache Hadoop: HDFS Federation." http://goo.gl/lhxzz. Accessed: March 2013.

[15] C. E. Irvine, T. D. Nguyen, D. J. Shifflett, T. E. Levin, J. Khosalim, C. Prince, P. C. Clark, and M. Gondree, "MYSEA: the Monterey Security Architecture," in *Proc. of the 2009 ACM Workshop on Scalable Trusted Computing (ACM CCS STC '09)*, Chicago, IL, pp. 39–48, 2009.

[16] T. D. Nguyen, M. Gondree, D. J. Shifflett, J. Khosalim, T. E. Levin, C. E. Irvine, "A Cloud-Oriented Cross-Domain Security Architecture," in *Proc. of the 2010 Military Communication Conference (MILCOM 2010)*, San Jose, CA, pp. 1702–1707, 2010.

[17] Count Lines of Code (CLOC) project. http://cloc.sourceforge.net/. Accessed: March 2013.

[18] Apache Mahout. http://mahout.apache.org/. Accessed: March 2013.

[19] GroupLens Research project, MovieLens Data Sets. http://goo.gl/MydMo. Accessed: March 2013.

[20] I. Roy, S. T.V. Setty, A. Kilzer, V. Shmatikov and E. Witchel, "Airavat: Security and Privacy for MapReduce," in Proceedings of the 7th Usenix Symposium on Networked Systems Design and Implementation (NSDI 2010), San Jose, CA, April 2010.