Vigiles: Fine-grained Access Control for MapReduce Systems

Huseyin Ulusoy, Murat Kantarcioglu, Erman Pattuk, Kevin Hamlen
The University of Texas at Dallas, Richardson, Texas, USA
{huseyin.ulusoy, muratk, erman.pattuk, hamlen}@utdallas.edu

***Abstract*—Security concerns surrounding the rise of Big Data systems have stimulated myriad new Big Data security models and implementations over the past few years. A significant disadvantage shared by most of these implementations is that they customize the underlying system source code to enforce new policies, making the customizations difficult to maintain as these layers evolve over time (e.g., over version updates).**

This paper demonstrates how a broad class of safety policies, including fine-grained access control policies at the level of key-value data pairs rather than files, can be elegantly enforced on MapReduce clouds with minimal overhead and without any change to the system or OS implementations. The approach realizes policy enforcement as a middleware layer that rewrites the cloud's front-end API with reference monitors. After rewriting, the jobs run on input data authorized by fine-grained access control policies, allowing them to be safely executed without additional system-level controls. Detailed empirical studies show that this more modular approach exhibits just 1% overhead compared to a less modular implementation that customizes MapReduce directly to enforce the same policies.

I. Introduction

The last few years have witnessed a meteoric rise in the volume of digital data generated and collected worldwide. Many organizations, ranging from large-scale Internet companies to government agencies, are interested in storing, processing and mining this *Big Data* for competition, productivity, and consumer surplus. Most traditional data management systems, including typical relational databases, do not adequately scale to the higher velocity and greater variety demanded by Big Data. Academic and industrial researchers have therefore devoted considerable effort toward more effectively storing, generating, and processing Big Data. *Big Data systems*, which boast highly parallel and distributed data processing atop commodity hardware, have emerged as a popular choice due to their easy deployment and attractive business model. Apache Hadoop project has become one of the most widely used Big Data systems, due in part to its adoption of Google's elegant MapReduce computing model [1].

Unfortunately, despite extensive tool support for Big Data processing on Hadoop architectures (e.g., Hive, Pig Latin, Hbase, etc.), security and privacy enforcement has suffered less development. Most of the tools developed atop Hadoop lack even simple authentication and access control mechanisms. This dearth of well-developed security mechanisms in Big Data systems has emerged as an important hindrance for widespread adoption, and has prompted some industry experts to ask, "Does NoSQL Mean No Security?"[1]. Recent work has sought to address this need by customizing the implementation of the cloud, virtual machine (VM), or operating system (OS) to include extra, system-level access controls. For example, *Apache Accumulo* allows multi-level access control at the cell level in a key-value store.

Yet, none of the aforementioned systems address the *fine-grained access control* (FGAC) challenges for all types of data (viz., structured, unstructured, semi-structured) in the generic MapReduce model even though each of them addresses different security/privacy issues. Almost all relational database management systems support FGAC due to the security and privacy requirements of the many industries that use them, ranging from health care to finance. For example, Oracle Virtual Private Database [2] automatically modifies the submitted SQL queries to enable FGAC. Industry experts have already observed that FGAC is a must-have addition to MapReduce systems as the model becomes more widely adopted [3].

To address this necessity, we developed Vigiles[2], a FGAC enforcement mechanism for MapReduce systems. Vigiles implements FGAC as a middleware layer that automatically rewrites the cloud's front-end API by augmenting them with *reference monitors* (RMs).The cloud-resident RMs filter the outputs of data accesses, preventing unauthorized access to policy-prohibited key-values at runtime. Policies are expressed as user-specific, computable predicates over records. For example, to enforce a policy that prohibits user u from accessing sensitive data, an administrator can define a predicate $p_u(d)$ that grants access if and only if record d is not sensitive. Vigiles then enforces p_u over all jobs submitted by u, resulting in a MapReduce environment that self-censors its object accesses to only those objects consisting of records that satisfy p_u. In general, predicates of this form are known to be capable of enforcing a large class of important safety policies, including access control policies [4], [5].

To our knowledge, Vigiles is the first system to provide FGAC for MapReduce without requiring any modification to MapReduce system source code. By automatically in-lining the enforcement programming within key-value access APIs prior to reaching the cloud kernel or OS layers, the enforcement implementation remains completely separate, making it much easier to maintain. To demonstrate the feasibility of the

[1]http://www.darkreading.com/database/does-nosql-mean-no-security/232400214

[2]The Vigiles Urbani were the firefighters and police of Ancient Rome.

proposed approach, we have implemented our system atop Apache Hadoop without changing any Hadoop source code. Our contributions can be summarized as follows:

- We show how user-defined FGAC predicates can be realized as RMs for efficient policy enforcement for wide range of data models (e.g., structured and unstructured).
- We provide detailed empirical studies indicating that our solution exhibits just 1% overhead compared to a non-modular implementation that changes the MapReduce internals to support FGAC.
- Our solution is extensible to other MapReduce implementations, and can therefore be seen as a general strategy for scalably enforcing FGAC on MapReduce.

The remainder of this paper is organized as follows: §II discusses related studies, §III summarizes background knowledge, and §IV details our assumptions. The FGAC problem in the context of the MapReduce model is formally defined in §V. §VI presents the architectural details of Vigiles. Our empirical results are evaluated in §VII. Finally, §VIII concludes the paper.

II. Related Work

FGAC in Relational Databases: In 1974, the access control system *INGRES*, which modifies the queries by conjugating safety conditions to the WHERE clauses before being processed, was introduced by Stonebraker and Wong [6]. Later on, Virtual Private Database (VPD) [2] has been included as a FGAC component in Oracle DBMS since Oracle8i. VPD allows to specify predicates as strings appended to the WHERE clause of the queries. In 2004, Rizvi et al. [7] generalized the query modification approach as *Truman model*, where each user's view of database can be inconsistent with additional information derived from external sources. They addressed this issue by rewriting queries using only the authorized views. Alternatively, LeFevre et al. [8] introduced *table semantics* model, where the queries remain the same while the tables are effectively modified by injecting dynamically created views. Agrawal et al. [9] proposed to use grant commands of DBMS to provide cell level access control. In 2007, Chaudhuri et al. [10] proposed to use predicated grants by considering other features (e.g., aggregate authorization, user groups, authorization groups, etc.). Rosenthal and Sciore [11] extended this approach for management of predicated grants as well. However, these approaches are not applicable to MapReduce model due to the lack of structured query languages in MapReduce systems, where the above approaches leverages the features specific to relational databases (e.g., *grant* commands, SQL, etc.).

Security Applications For MapReduce: *Apache Accumulo* is a distributed key-value store based on Google's *BigTable* [12] design and built on top of *Apache Hadoop*. It improves the BigTable design in the form of cell-based mandatory and attribute-based access control capabilities and a server-side programming mechanism that can modify key-value pairs in the data management process. Similarly, BigSecret [13] enables secure querying of cell-level encrypted data in HBase. SecureMR [14] provides a decentralized replication-based integrity verification scheme for MapReduce job execution. Airavat [15] employs SElinux to achieve multi-level access control and guarantees MapReduce computation results to satisfy *differential privacy*. However, none of the aforementioned systems address the FGAC challenges for all types of data (viz., structured, unstructured, semi-structured data) in the generic MapReduce model even though each of them addresses different security/privacy issues. To our knowledge, this is the first work that provides FGAC capabilities to MapReduce model without changing the underlying MapReduce system.

III. Background

This section provides the background information on *MapReduce* and *aspect-oriented programming (AOP)*, which are two core technologies that Vigiles is based on.

A. MapReduce

MapReduce is a programming model and associated implementations for processing and storing large data sets [1]. The model enables large cluster of commodity machines to be employed in parallel while reducing computation costs. Simply, the MapReduce model can be expressed in 5-step parallel computations. (1) *Pre-process:* The input data is pre-processed to form a valid format for the subsequent steps. For example in Hadoop, RecordReader classes read the input from Hadoop file system (HDFS) and produce key-value pairs. (2) *Map:* The input key-value pairs are processed by producing intermediate key-value pairs. (3) *Shuffle:* The intermediate pairs are transfered to the reduce functions by assigning the pairs having the same key to the same reduce functions. (4) *Reduce:* The intermediate pairs are processed by forming smaller set of pairs. (5) *Post-process:* The outputs of reduce functions are combined and written to HDFS.

B. Aspect-Oriented Programming

Aspect-oriented programming (AOP) is a programming paradigm for addressing *cross-cutting concerns*—concerns whose implementations must typically be scattered over many modules in a traditional programming language. AOP allows such implementations to be consolidated as *aspects*, which consist of (1) *pointcuts*—expressions that identify *join points* (sites throughout the rest of the code) that are relevant to the cross-cutting concern—and (2) *advice* consisting of code that modifies each join point to implement the concern. Prior work has recognized that such aspects are an elegant means of expressing RMs [16]–[20]. In such contexts, the pointcuts identify security-relevant program operations, and the advice introduces guard code that secures each such operation.

AspectJ is an AOP extension for the Java programming language. Its pointcut language allows the aspect-writer to advise any method call, object instantiation, or variable access in programs to which the aspect is applied. Aspects can observe the control flow of the program, and can even change it by modifying the input and return values of methods. The advice can be injected into Java programs at the source level, or into raw Java bytecode programs that is separately compiled.

IV. Threat Model

Vigiles treats the submitted MapReduce jobs as untrusted because a user may have malicious intentions, and may try to compromise the system by injecting malicious code into MapReduce jobs. The underlying MapReduce and OS are trusted with the following setup: Vigiles and the MapReduce system need to be installed on a *hardened OS*, where all communication ports are closed, except the ones used by Vigiles. This is because if the users directly access the MapReduce system, they can retrieve the entire data file without any FGAC restrictions. For example, one can read the files from HDFS by using command line. In addition, Vigiles requires that the programming language, in which the MapReduce system is implemented, provides AOP support within the language or as an external library, because AOP is employed to enforce the security policies in Vigiles.

Hadoop system provides **only one legitimate method to access the input data** for submitted MapReduce jobs which is through **RecordReader** interface. However, the jobs include arbitrary code and data, where one can easily place a malicious code to circumvent RecordReader. We have performed two attacks to show the feasibility of this vulnerability. In the first attack, we have placed a code inside a MapReduce job that accesses to raw input data (or any data stored in the corresponding data node) by means of Java I/O API. In the second attack, the malicious code opened a socket as a *backdoor* to communicate with our malicious server. Both attacks show that the MapReduce jobs must be confined to prevent these security breaches and policy violations. There are numerous studies [21]–[23] in the literature to circumscribe untrusted codes and programs. These studies can be employed to immunize Hadoop against such attacks by potentially malicious MapReduce jobs.

MapReduce systems are generally used to extract useful information from big datasets. In this model, data providers upload their datasets into the MapReduce system, and the end-users run their jobs to extract information. Unlike relational databases, only *read* and *append* data permissions are required because uploading data requires *append* permission and running jobs requires *read* permission. Vigiles only provides FGAC for read permission, and do not consider append permission. One justification is that read access permission is more likely to be used by the end-users while trying to extract information from the data. On the other hand, append access permission is more likely to be used by data providers. FGAC for append and deletion will be addressed in our future work.

V. Problem Definition

The current access control model of MapReduce systems are at file-level. However, authorizing the whole files is not desirable in the MapReduce model because of three main reasons: (1) The size of files can be very large. (2) The system can be used by many users having different intentions and security clearances. (3) The files can contain data from different domains and sensitivity specifications. To this end, we propose the *fine-grained access control predicates (FCAC predicates)*. In this model, each user can access the files after the user specific predicates are applied to the files. The predicates independently run the *access control filters (ACFs)* on the individual records. ACFs perform an action $\alpha \in$\{*reject, grant, modify*\} according to the specified security policy. *Reject* action refuses access by returning nothing, *grant* action accepts access by returning the original record, and *modify* action changes the original record by returning a modified version (cf. Eqn. 1). Especially the modify action enables the MapReduce model to work with diverge and sensitive data, since it enables filtering out unnecessary parts of records and sanitizing sensitive parts.

More formally, let $\mathcal{M} = (\mathcal{S}, \mathcal{O}, \mathbb{P})$ denote FGAC predicate model, where $\mathcal{S} = \{s_1, \ldots, s_k\}$ denote the set of subjects, $\mathcal{O} = \{o_1, \ldots, o_n\}$ denote the set of objects, and $\mathbb{P} = \{p_1, \ldots, p_l\}$ denote the set of predicates. Moreover, let $\mathbb{F} = \{f_1, \ldots, f_l\}$ denote the set of ACFs. Without loss of generality, we assume that each object $o = \{d_1, \ldots, d_m\}$ is composed of finite number of atomic data records $d_i \in \mathcal{D}$ that cannot be split into smaller pieces without losing semantics. Then, the predicates and ACFs can be expressed as follows:

$$\begin{aligned} &\forall d \in o, \quad f : \mathcal{D} \rightarrow \{\emptyset \cup \mathcal{D} \cup \{0,1\}^*\} \\ &\forall o \in \mathcal{O} \quad p : \mathcal{O} \rightarrow \{\emptyset \cup \mathcal{D} \cup \{0,1\}^*\}^{|o|} \\ &\qquad\qquad p(o) = \{f(d) \mid f(d) \neq \emptyset \wedge d \in o\} \end{aligned} \tag{1}$$

VI. System Architecture

Vigiles is an application firewall that provides FGAC capabilities to the MapReduce systems. It employs a *middleware* architecture that lays between the untrusted end-users and underlying OS/MapReduce system by authorizing all data accesses. Fig. 1 shows the overview of the Vigiles system, in which thick, black-dashed, red-dashed and black arrows indicate the actions of end-users, admins, Vigiles and MapReduce system, respectively.

Vigiles authenticates the users by using the same user IDs and passwords of host OS. In the system, there are two types of users: the *end-users* and *admins*. The end-users have no responsibilities specific to Vigiles. They write MapReduce jobs as usual and send it to Vigiles along with the required parameters, such as input/output names and job specific variables. Other than the interface provided by Vigiles, they have no communication with the underlying MapReduce system nor OS. Their submitted MapReduce jobs have also limited view of the input data (e.g., *authorized view* in [7]) because of the FGAC predicates. On the other hand, the admins are responsible from the input files in HDFS and configuration of ACFs. They load data into Vigiles system and setup the configuration of ACFs so as to activate FGAC predicates, $\mathcal{M} = (\mathcal{S}, \mathcal{O}, \mathbb{P})$. Furthermore, they also load the libraries, which are used in the ACFs for the first time. Unlike end-users, they can access and configure the MapReduce system and OS.

Vigiles encapsulates the OS/MapReduce system such that the users can only communicate with the underlying systems through Vigiles. It accepts authenticated network communication from the users. If the communication is established by an admin, the OS's terminal is returned as interface. Otherwise,

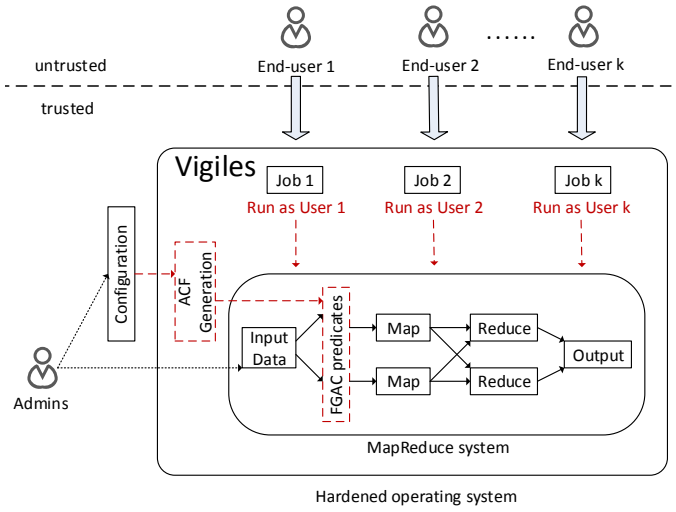


Fig. 1. Vigiles System Overview

a special interface is returned, where the end-user can submit MapReduce jobs and related parameters, such as input/output paths, variables. Since the current implementation of Vigiles system uses the Java Security [23] to confine the MapReduce jobs in a secure environment, Vigiles only accepts managed Java bytecode programs and conservatively rejects those that contain native code or that link to type-unsafe libraries. We are planning to support native libraries in the future by integrating Vigiles with the Robusta system [22]. When a job is accepted, Vigiles runs the job on behalf of the user, and by so doing leverage the current file level access control policy of MapReduce system. Thus, the outputs of end-users are protected by the current access control mechanism. Moreover, the ACFs are generated by means of the given configuration, and injected into MapReduce system (see §VI-A and §VI-C for details).

Vigiles can use any multi-user MapReduce and OS as follows: (1) The employed MapReduce system is consolidated with the FGAC predicates. (2) All outside communications of the underlying OS are prevented, except the ones initiated by Vigiles. (3) The MapReduce jobs are confined by a sandbox technique [21]–[23] so as to prevent unauthorized data accesses.

A. ACF Generation

Suppose that an unstructured text data, containing sensitive entries, is stored in a MapReduce system, and an ACF is going to be designed to sanitize the sensitive entries. Since the data is unstructured, the sensitive entries need to be located for sanitization. An intuitive approach is first to decompose the text into words, then find the indexes of sensitive words, and sanitize them one by one. Inspired by the above example, Vigiles generates the ACFs by combining following three phases:

1) Decompose: This phase aims to produce a list of small processable tokens from its input. The input key-value pairs and the output of other phases (i.e., *fetch* and *action*) can be used as input. It fragments the input into a list of small tokens by means of a given algorithm and the meta-data of input if exists, and outputs the produced list. Tokenization of text data, parsing an HTML code, and decompression of images are three examples of decompose phase.

2) Fetch: This phase aims to detect the indexes of targeted tokens. A list of tokens, which is result of a decompose phase or another fetch phase, can be used as input. It finds the indexes by employing a search algorithm and the meta-data of input if exists, and outputs both the list and indexes. Regular expression based text search and fetching columns on Google's BigTable are two examples of fetch phase.

3) Action: This phase aims to apply ACF specific action to the indexed tokens in a given list. A list of tokens and indexes are its input. It applies the action to each indexed token. According to given configuration, it may output the filtered list to another decompose phase, or merge the list, and output using one of three options: (1) Nothing, (2) the original key-value pair, and (3) the modified key-value pair. Sanitization of sensitive tokens and reduction of a list to indexed tokens are two examples of action phase.

Vigiles automatically generates the ACFs by means of given configuration. The admins organize the configuration by clearly stating each phase. Vigiles provides many fundamental *decomposition*, *fetching* and *action* algorithms by default, but new algorithms can also be loaded if necessary. In order to add new algorithms to any of the aforementioned phases, the admins need to load the required libraries to the Vigiles system and set their properties in the configuration. A sample ACF configuration is provided in §VI-B.

```
<ACF ID="sanitization">
      <decompose ID="tokenize">
            <method>text.tokenize</method>
            <input>
                  <source>value</source>
                  <type>text</type>
            </input>
            <arg>'|'</arg>
      </decompose>
      <fetch ID="search">
            <method>text.regex_search</method>
            <input>
                  <source>tokenize</source>
                  <type>text</type>
            </input>
            <arg>'\\d{3}-\\d{3}-\\d{4}'</arg>
      </fetch>
      <action ID="replace">
            <method>string.replace</method>
            <input>
                  <source>search</source>
                  <type>text</type>
            </input>
            <arg>'*'</arg>
            <merge>'true'</merge>
      </action>
</ACF>
```

Fig. 2. Sample configuration file

B. Sample ACF Configuration

Fig. 2 shows a sample configuration for an ACF, where the phone numbers in 'ddd-ddd-dddd' format are sanitized in text data. The decompose phase takes *value* as input, and use "text.tokenize" function to tokenize the value into words by using '|' as separator. Then, the fetch phase uses the output of decompose phase as input and searches the words matching given regular expression by using "text.regex_search" function. Finally, the action phase uses the output of decompose phase

as input. It sanitizes the indexed words, merges the list and emits the modified value.

C. ACF Injection

Vigiles depends on our novel ACF injection technique to enforce the security policies. This technique enables enforcing the FGAC predicates to the processing key-value pairs in a complete manner by incurring minimal overhead to the performance. In this section, we elucidate the details of this technique. Since the current implementation of Vigiles is based on *Apache Hadoop* and *AspectJ* [24], we will explain the details of the injection technique through them.

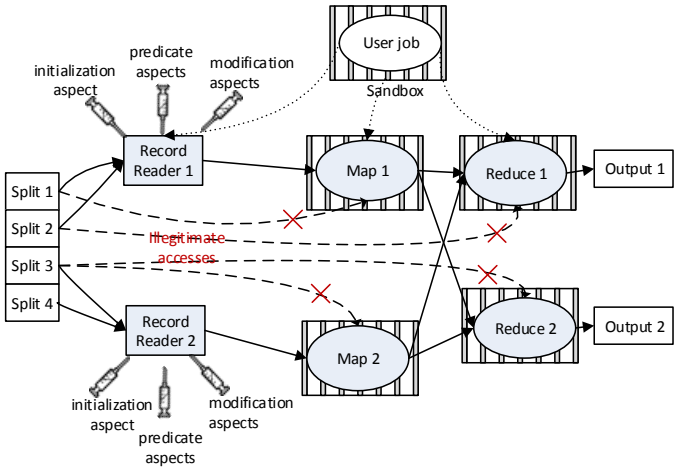


Fig. 3. Injection of ACFs and access restriction aspects

When a job is submitted to a MapReduce system, its input data has to be transformed into formatted key-value pairs by **RecordReader** classes before passing to the *map functions*. We leverage this obligation to enforce our FGAC predicates to the submitted jobs due to following reasons: (1) Prior to this transformation, the raw input data is an enigma for the jobs. (2) This transformation is the only legitimate way to access data for the MapReduce jobs. Therefore, applying FGAC predicates during this transformation guarantees that the only data accessed by jobs is *the authorized views* [7] created by FGAC predicates. Fig. 3 shows the overview of ACF injection. Note that these processes are similar for other MapReduce systems and AOPs, too.

```
1: Input : A data split
2: Output: <key,value> pairs for mapper
3: initialize()
4: while nextKeyValue()=true do
5:     key ← getCurrentKey()
6:     value ← getCurrentValue()
7:     sendToMap(key,value)
8: end while
```

Fig. 4. Input preprocessing in Hadoop

In Hadoop environment whenever a class, whether existing in the system or created by an end-user, is used for key-value transformation, **it has to implement RecordReader interface** to be accepted by the system. Because, Hadoop system uses RecordReader's methods to create key-value pairs to be processed by Map functions. The algorithm in Fig. 4 shows how the RecordReader interface is called by Hadoop. Four methods of RecordReader are important for our injection technique: *initialize()*, *nextKeyValue()* and *getCurrentKey()/getCurrentValue()*. The initialize() method is called once when a RecordReader object is initialized. The nextKeyValue() method is called to read an individual record and create a key-value pair correspondingly. The getCurrentKey()/getCurrentValue() methods are called to get current key-value pairs.

```
1: Input : A data split and configuration
2: Output: <key,value> pairs for mapper
3: initialize()
4: filepath ← determineInput()
5: user ← determineUser()
6: ACF ← generateACF(user, filepath, configuration)
7: while nextKeyValue()=true do
8:     key ← getCurrentKey()
9:     value ← getCurrentValue()
10:    if ACF_predicate(key,value)=allow then
11:        sendToMap(ACF_modify(key),ACF_modify(value))
12:    end if
13: end while
```

Fig. 5. Input preprocessing after ACF injection in Hadoop

We determined these four methods as our injection points (i.e., our pointcuts) in *Hadoop version 1.1.2*. Three types of aspects are injected into these pointcuts: (1) *initialization aspect* is injected to *initialize()* method; (2) *predicate aspect* is injected to *nextKeyValue()* method; and (3) *modification aspects* are injected to *getCurrentKey()/getCurrentValue()* methods. The algorithm in Fig. 5 shows how the injected aspects augment the execution flow of RecordReader methods. The *initialization aspect* runs once at the beginning of each MapReduce job when initialize() is called. Firstly, it determines the input file(s) and the owner of MapReduce job ($o \in \mathcal{O}$ and $s \in \mathcal{S}$ in the FGAC predicates $\mathcal{M} = (\mathcal{S}, \mathcal{O}, \mathbb{P})$, respectively). Then, it generates the ACF by using the file(s), user ID and the given configuration (If any access control model, such as role-based access control, is employed by Vigiles, it is handled here as well). The generated ACF is attached to the job to be employed in the other aspects later. Moreover, the auxiliary data structures are constructed in the initialization aspect if they are used by the ACF. For example, a hash map for keyword whitelisting can be constructed by reading keywords from a file. The *predicate aspect* runs whenever *nextKeyValue()* method is called. It checks the key and/or value, and either grants access by returning the original key-value or rejects access by returning nothing. The *modification aspects* run whenever *getCurrentKey()/getCurrentValue()* are called. They modify the returned key-value pairs by performing *modify* action. Since the predicate aspect always runs before modification aspects, *reject* and *grant* actions are performed before *modify* action.

Therefore, the admins need to consider it when preparing ACF configurations. Note that in order to use other version of Hadoop or another MapReduce system, only these four pointcuts need to be modified. The other parts of Vigiles, including ACFs and configurations, can be used without any modification. This increases the modularity of our system.

Security Discussion: As previously mentioned, Vigiles only accepts managed Java bytecode based MapReduce jobs. This restriction enables Vigiles to adopt a secure sandbox technique (i.e., Java Sandbox [23]) so as to prevent a broad class of security breaches discussed in §IV. Thus, unorthodox ways to access data are prevented in Hadoop—a MapReduce job can only access data in HDFS through RecordReader interface which is consolidated by FGAC predicates. In other words, whenever a MapReduce job is accepted and run by Vigiles system, the given ACFs are enforced on the input data in a correct and complete manner.

Moreover, Vigiles improves the performance of access control system by using *optimized operations*—pushing all operations to initialization aspect as much as possible. Because, unlike other operations that are run per record, the initialization aspect run once for each job.

VII. Evaluation

We evaluated the efficiency and scalability of Vigiles via a series of experiments. We first explain the details of experiment setup, data generation, sample ACFs and the MapReduce jobs. Then, we present the empirical results.

A. Setup

We conducted our experiments on a cluster containing 14 nodes. Each node consists of a Pentium IV processor with 290GB-320GB disk space and 4GB of main memory. The cluster is setup using Hadoop 1.1.2 and AspectJ 1.7.3. To enable AOP in Hadoop, the AspectJ JARs and compiled aspects are placed into *lib folder* of each node in the cluster.

B. Data Generation

We have randomly generated five input files, formated as *compressed sequence* in HDFS, to use in our experiments. The files are composed of 10M, 20M, 30M, 40M, and 50M key-value records where they allocate 48GB, 96GB, 144GB, 192GB, and 240GB space, respectively. The size of each data record is approximately 11KB, and the records are generated by using two types of data. The first part is organized as *relational table*. A medical dataset is simulated by using the personal information of patients, such as name, address, age, doctor's name, diagnosis, etc. To this end, 1000 different male and female first and surnames, 32 different treatment groups, and 100 diagnosis types are used by uniform randomly selecting. The other columns based on numbers (i.e. age, phone and ssn numbers) are uniformly distributed within their domain ranges. The second part of the records contains an unstructured text data that represents the medical history of patients written by doctors. For this part, we used 10 different real life medical histories. The key part of each record is also labeled with a set of security classifications. In addition, HDFS is set to use replication factor 3 by achieving approximately 60% load rate.

C. ACFs

We have generated five ACFs, two predicate, two modification and a combination of the first four, for experiments. The first one, *key ACF*, uses the security classification labels in the key of each pair to filter the unauthorized pairs. The second one, *relational ACF*, filters the records based on the doctor name column. The third one, *sanitization ACF* detailed in §VI-B, uses a regular expression to sanitize phone numbers in the relational part. The fourth one, *redaction ACF*, reads a set of medicine names and transforms the medical history of patients into list of medicines existing in the medical histories. The fifth ACF is the combination of first four, where the application order is key, relational, sanitization and redaction ACFs.

The ACFs are first generated by means of given configuration and called from the aspects injected into Hadoop as a part of the Vigiles system (namely *Vigiles implementation*). Then, the same ACFs are implemented in Hadoop source code (namely *integrated implementation*) so as to compare the performance of these two approaches. To this end, all built-in RecordReader classes of Hadoop are enhanced with ACFs, where four pointcut methods of these classes are overridden. Note that the integrated implementation cannot provide the same security guarantees as Vigiles implementation does because the MapReduce jobs can contain a custom RecordReader class, which would bypass the ACFs in the integrated implementation.

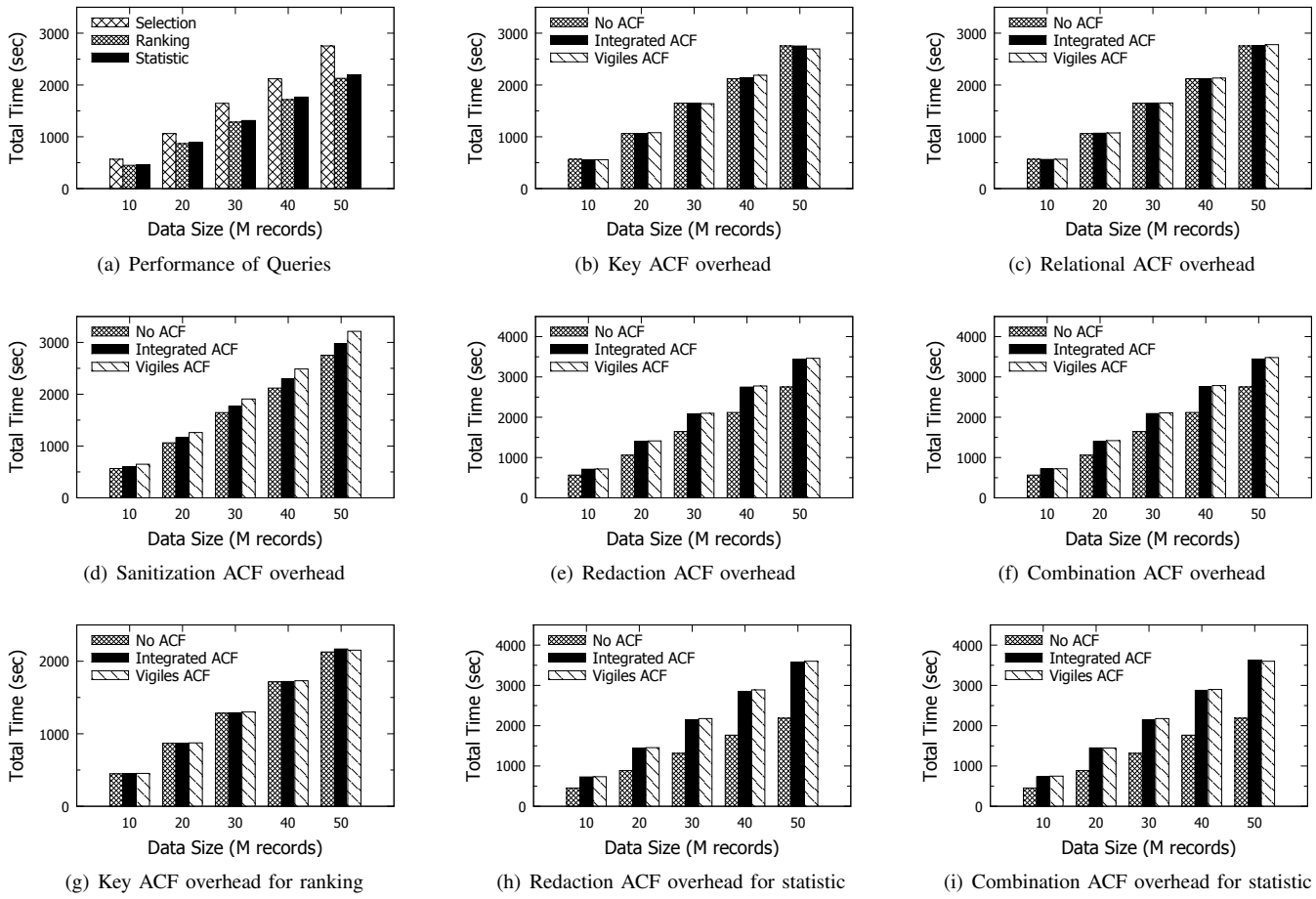
By using AspectJ compiler version 1.7.3, the aspects are compiled independently of the Hadoop and jobs source code. Then, the aspects are weaved into *hadoop-core-1.1.2.jar*, where the RecordReader methods are called. While the generated ACFs are running in the aspects, we observed some performance issues especially for *lazy copies*. We believe this is due to poor optimization of AspectJ compiler as analyzed in [25]. To address this issue, we optimized the functions used in three phases of ACFs by preventing unnecessary data copy.

D. Queries

We implemented three MapReduce jobs for our experiments. The first job is a *selection* query that selects the records by patient name. The second job is *ranking* query that sorts the records by the ascending ordered list of doctors having the most patients. The third job is a *statistic* query that calculates the average age of patients with heart disease.

E. Results

We firstly ran three MapReduce jobs on each data set described in §VII-B so as to measure the performance without ACFs (termed *raw performance*). Fig. 6(a) shows the performance of the queries. The ranking and statistic queries run faster than the selection query because their mappers emit less data to reducers. During all experiments, the total running time of the queries is used as our primary metric. Moreover, in a typical scenario, the ACFs are expected to reduce the data amount shown to the MapReduce jobs. To run the MapReduce



(a) Performance of Queries (b) Key ACF overhead (c) Relational ACF overhead

(d) Sanitization ACF overhead (e) Redaction ACF overhead (f) Combination ACF overhead

(g) Key ACF overhead for ranking (h) Redaction ACF overhead for statistic (i) Combination ACF overhead for statistic

Fig. 6. Comparison of three setup: No ACF, Integrated ACF and Vigiles ACF

jobs on the same amount of data with the raw performance experiment, we setup the ACFs not to filter out the data while performing all required operations. For example, the key ACF checks the security labels but do not reduce the data amount.

The main purpose of experiments is to evaluate the performance of (1) the generated ACFs, and (2) the injection technique. To this end, we have measured the overhead of ACFs and injection technique by comparing Vigiles implementation with *raw performance*—no safety policy is enforced, and *integrated implementation*—safety policies are integrated into Hadoop. Thus, three queries are run on five datasets when (1) no ACF is active, (2) the integrated ACFs are active, and (3) Vigiles ACFs (weaved by AspectJ) are active.

	Selection		Ranking		Statistic	
	Vigiles	**Integ.**	**Vigiles**	**Integ.**	**Vigiles**	**Integ.**
Key	0.23%	-0.15%	1.02%	0.54%	1.29%	0.38%
Rel.	0.63%	0.13%	1.14%	0.97%	1.00%	0.53%
San.	16.68%	7.81%	22.01%	11.27%	21.11%	10.55%
Red.	28.8%	27.61%	64.98%	63.72%	64.00%	62.15%
Com.	29.49%	28.32%	66.25%	64.96%	64.55%	63.98%

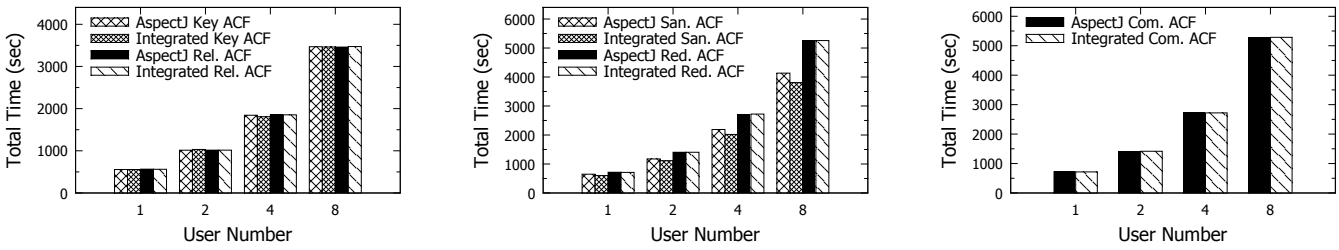
TABLE I
THE OVERHEADS OF VIGILES AND INTEGRATED IMPLEMENTATION ACFS

The overhead of ACFs: Fig. 6(b), 6(c), 6(d), 6(e) and 6(f) show the running time of selection query. The overhead of predicate ACFs, key and relational, is almost negligible (respectively 0.23% and 0.63% on average). On the other hand, the modification ACFs, sanitation and reduction, have 16.68% and 28.8% overheads due to costly functions used in fetch phases (i.e., regular expression search and whitelisting via a hashmap). Tab. I shows the overhead of ACFs for each query type. The overhead difference between different queries is due to the difference of queries' running times (see Fig. 6(a)).

	Selection	**Ranking**	**Statistic**
Label ACF	0.33%	0.48%	0.90%
Value ACF	0.48%	0.16%	0.49%
Sanitization ACF	7.58%	8.80%	8.71%
View Creation ACF	0.92%	0.78%	1.14%
Combination ACF	0.90%	0.79%	0.7%

TABLE II
THE OVERHEADS OF ASPECTJ INJECTION

The overhead of ACF injection: The overhead of injection technique is less than 1% in all ACFs expect the sanitization ACF, where 8.36% overhead is observed on average. We believe the regular expression based search algorithm underperforms because of the relatively poor optimization of AspectJ compiler (see. [25] for the detailed performance analysis of AspectJ). Tab. II shows the overhead of injection technique for each query type. Fig. 6(g), Fig. 6(h) and Fig. 6(i) show the running time of ranking and statistic queries for the key, redaction and combination ACFs. We observe the similar overheads of ACFs



(a) Key and Relational ACFs overhead (b) Sanitation and Redaction ACFs overhead (c) Combination ACF overhead

Fig. 7. The overhead of injection for multiple users

and injection technique with selection query.

The overhead for multiple users: To compare the performance of ACFs and injection technique for multiple MapReduce jobs, we performed another set of experiments. In these experiments, the selection query is simultaneously run by multiple users on the dataset containing 10M records when Vigiles and integrated implementation of ACFs are assigned to the MapReduce jobs. To run the jobs simultaneously, the fair scheduler, developed by Zaharia et al. [26], is employed in Hadoop. The fair scheduler is set to preemptive mode to evenly assign resources to jobs. We begin with 1 user and exponentially increase the number of users up to 8. The graphs in Fig. 7 show the performance of two approaches when the selection query is run. The average performance differences are 0.14%, 0.56%, 0.02% and 0.05% for the key, relational and redaction ACFs. The performance of integrated implementation is slightly better than the performance of Vigiles implementation, where the difference decreases when the number of users increases due to higher running time of queries. On the other hand, the performance difference of the sanitization ACF is 7.15%. As discussed in previous experiments, the sanitization ACF suffers from the poor compiler optimization of AspectJ.

VIII. Conclusion

To our knowledge, Vigiles is the first system that provides a critical security component for MapReduce, FGAC, without modifying the source code of MapReduce system. It realizes a modular policy enforcement by rewriting the front-end API of MapReduce system with RMs. Our empirical results indicate Vigiles exhibits just 1% overhead compared to the implementation that modifies Hadoop's source code.

IX. Acknowledgements

This work was partially supported by Air Force Office of Scientific Research FA9550-12-1-0082, National Institutes of Health Grants 1R0-1LM009989 and 1R01HG006844, National Science Foundation (NSF) Grants Career-CNS-0845803, CNS-0964350, CNS-1016343, CNS-1111529, CNS-1228198 and Army Research Office Grant W911NF-12-1-0558

References

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

[2] K. Browder and M. A. Davidson, "The virtual private database in oracle9ir2," *Oracle Technical White Paper*, vol. 500, 2002.

[3] C. Mohan, "History repeats itself: sensible and nonsensql aspects of the nosql hoopla," in *EDBT*. ACM, 2013, pp. 11–16.

[4] F. B. Schneider, "Enforceable security policies," *TISSEC*, vol. 3, no. 1, pp. 30–50, 2000.

[5] K. W. Hamlen, G. Morrisett, and F. B. Schneider, "Computability classes for enforcement mechanisms," *ACM Trans. Programming Languages and Systems*, vol. 28, no. 1, pp. 175–205, 2006.

[6] M. Stonebraker and E. Wong, "Access control in a relational data base management system by query modification," in *Proceedings of the 1974 Annual Conference - Volume 1*, ser. ACM '74, 1974, pp. 180–186.

[7] S. Rizvi, A. Mendelzon, S. Sudarshan, and P. Roy, "Extending query rewriting techniques for fine-grained access control," in *SIGMOD*, 2004.

[8] K. LeFevre, R. Agrawal, V. Ercegovac, R. Ramakrishnan, Y. Xu, and D. DeWitt, "Limiting disclosure in hippocratic databases," in *VLDB*. VLDB Endowment, 2004, pp. 108–119.

[9] R. Agrawal, P. Bird, T. Grandison, J. Kiernan, S. Logan, and W. Rjaibi, "Extending relational database systems to automatically enforce privacy policies," in *ICDE*. IEEE, 2005, pp. 1013–1022.

[10] S. Chaudhuri, T. Dutta, and S. Sudarshan, "Fine grained authorization through predicated grants," in *ICDE*. IEEE, 2007, pp. 1174–1183.

[11] A. Rosenthal and E. Sciore, "Extending sqls grant operation to limit privileges," in *Data and Application Security*. Springer, 2002.

[12] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *in TOCS*, vol. 26, no. 2, p. 4, 2008.

[13] E. Pattuk, M. Kantarcioglu, V. Khadilkar, H. Ulusoy, and S. Mehrotra, "Bigsecret: A secure data management framework for key-value stores," in *IEEE CLOUD*, 2013.

[14] W. Wei, J. Du, T. Yu, and X. Gu, "Securemr: A service integrity assurance framework for mapreduce," in *ACSAC*. IEEE, 2009, pp. 73–82.

[15] I. Roy, S. T. V. Setty, A. Kilzer, V. Shmatikov, and E. Witchel, "Airavat: Security and privacy for mapreduce," in *USENIX*, 2010, pp. 20–20.

[16] F. Chen and G. Roşu, "Java-MOP: A Monitoring Oriented Programming environment for Java," in *TACAS*, 2005, pp. 546–550.

[17] D. S. Dantas and D. Walker, "Harmless advice," in *POPL*, 2006.

[18] K. W. Hamlen and M. Jones, "Aspect-oriented in-lined reference monitors," in *PLAS*, 2008, pp. 11–20.

[19] M. Jones and K. W. Hamlen, "Enforcing IRM security policies: Two case studies," in *ISI*, 2009, pp. 214–216.

[20] K. W. Hamlen, M. M. Jones, and M. Sridhar, "Aspect-oriented runtime monitor certification," in *TACAS*, 2012, pp. 126–140.

[21] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer, "A secure environment for untrusted helper applications: Confining the wily hacker," in *Proceedings of the 1996 USENIX Security Symposium*, 1996.

[22] M. Sun, G. Tan, J. Siefers, B. Zeng, and G. Morrisett, "Bringing java's wild native world under control," *ACM Trans. Inf. Syst. Secur.*, 2013.

[23] L. Gong, M. Mueller, and H. Prafullch, "Going beyond the sandbox: An overview of the new security architecture in the java development kit 1.2," in *USENIX*, 1997, pp. 103–112.

[24] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. marc Loingtier, and J. Irwin, "Aspect-oriented programming," in *ECOOP*. SpringerVerlag, 1997.

[25] E. Hilsdale and J. Hugunin, "Advice weaving in aspectj," in *Aspect-oriented software development*. ACM, 2004, pp. 26–35.

[26] M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Job scheduling for multi-user mapreduce clusters," *EECS Department, UC Berkeley, Tech. Rep. USB/EECS-2009-55*, 2009.