Enhancing MongoDB with Purpose-Based Access Control

Pietro Colombo and Elena Ferrari, *Fellow, IEEE*

Abstract—Privacy has become a key requirement for data management systems. Nevertheless, NoSQL datastores, namely highly scalable non relational database management systems, which often support data management of Internet scale applications,still do not provide support for privacy policies enforcement. With this work, we begin to address this issue, by proposing an approach for the integration of purpose based policy enforcement capabilities into MongoDB, a popular NoSQL datastore. Our contribution consists of the enhancement of the MongoDB role based access control model with privacy concepts and related enforcement monitor. The proposed monitor is easily integrable into any MongoDB deployment through simple configurations. Experimental results show that our monitor enforces purpose-based access control with low overhead.

Index Terms—Purpose-based access control, NoSQL datastores, MongoDB

1 Introduction

NoSQL datastores are emerging non relational databases committed to provide high horizontal scalability for database operations over clusters of servers [4]. These platforms are getting increasing attention by companies and organizations for the ease and efficiency of handling high volumes of heterogeneous and even unstructured data.

Although NoSQL datastores can handle high volumes of personal and sensitive information, up to now the majority of these systems provide poor privacy and security protection (e.g., see [19]). Initial research contributions started to address these issues, but they have mainly targeted security aspects (see e.g., [16]). To the best of our knowledge, we are not aware of any work targeting privacy-aware access control for NoSQL systems, but we believe that, similar to what has been proposed for relational DBMSs by Agrawal et al. [1], privacy-aware access control is an urgency for NoSQL datastores as well [10]. However, different from relational databases, where all existent systems refer to the same data model and query language, NoSQL datastores operate with various languages and data models. This variety makes the definition of a general approach to the integration of privacy-aware access control into NoSQL datastores a very ambitious goal. We believe that a stepwise approach is necessary to define such a general solution. As such, in this paper, we start focusing on: 1) a single datastore, and 2) selected components of privacy policies. More precisely, we approach the problem by focusing on MongoDB[1] which, according to the DB-Engines Ranking[2] ranks, by far, as the most popular NoSQL datastore.

1. http://www.mongodb.org/
2. http://db-engines.com/en/ranking

- *The authors are with the Dipartimento di Scienze Teoriche e Applicate Università degli Studi dell'Insubria, Varese, Italy. E-mail: {pietro.colombo, elena.ferrari}@uninsubria.it.*

Manuscript received 23 June 2015; revised 28 Sept. 2015; accepted 29 Oct. 2015. Date of publication 4 Nov. 2015; date of current version 10 Nov. 2017. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TDSC.2015.2497680

MongoDB uses a document-oriented data model [4]. Data are modeled as documents, namely records, possibly with a hierarchical structure, and grouped into heterogeneous collections that are stored into a database.

We analyzed several privacy-aware access control models proposed for relational DBMSs (see Section 8) to identify the characteristics of privacy policies to be supported. In all the analyzed models: 1) privacy policies require fine-grained specification and enforcement mechanisms, as different data owners can have different privacy requirements on their data; and 2) the compliance of the purposes for which data should be accessed with those for which they are stored is considered as the key required condition to grant the access. Although privacy policies can involve actions, conditions, and obligations (e.g., see [8], [9]), the concept of purpose is thus the essence of any privacy policy [3]. As such, fine grained purpose-based policies have been selected as the target policy type for our proposal.

MongoDB integrates a role-based access control (RBAC) [11] model which supports user and role management, and enforces access control at collection level. However, no support is provided for purpose-based policies. As such, in this work we extend MongoDB RBAC with the support for purpose-based policy specification and enforcement at document level. More precisely, we refine the granularity level at which the MongoDB RBAC model operates, integrating the required support for purpose related concepts. On top of this enhanced model we have developed an efficient enforcement monitor, called MongoDB enforcement monitor (Mem), which has been designed to operate in any MongoDB deployment.

Within the client/server architecture of a MongoDB deployment, a MongoDB server front-end interacts, through message exchange, with multiple MongoDB clients. Mem operates as a proxy in between a MongoDB server and its clients, monitoring and possibly altering the flow of messages that are exchanged by the counterparts.

Access control is enforced by means of MongoDB message rewriting. More precisely, either Mem simply forwards

the intercepted message to the respective destination, or injects additional messages that encode commands or queries. In case the intercepted message encodes a query, Mem rewrites it in such a way that it can only access documents for which the specified policies are satisfied. The integration of Mem into a MongoDB deployment is straightforward and only requires a simple configuration. No programming activity is required to system administrators. Additionally, Mem has been designed to operate with any MongoDB driver and different MongoDB versions. First experiments conducted on a MongoDB dataset of realistic size have shown a low Mem enforcement overhead which has never compromised query usability.

The remainder of this paper is organized as follows. Section 2 introduces MongoDB background knowledge. Section 3 provides an overview of the main functional and non functional requirements related to the development of Mem. Section 4 presents the enhanced access control model, whereas Section 5 introduces Mem and its correctness properties. Section 6 presents the experimental evaluation of Mem overhead. Section 7 provides a short discussion related to non functional properties of Mem, whereas Section 8 surveys related work. Section 9 concludes the paper. Finally, the paper includes two appendixes: Appendix A, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TDSC.2015.2497680, presents formal correctness proofs, whereas Appendix B, available in the online supplementary material, provides details related to experiments set up.

2 Background

In this section, we introduce selected MongoDB basic aspects. According to the MongoDB document-oriented data model, a *document* is a record of pairs $\langle field : value \rangle$, where *field* is the identifier of a document property, whereas *value* is the representation of the property value which is defined as instance of a primitive type, an array, a document or an array of documents. Documents are stored within *collections* which can group documents characterized by different sets of fields. A *database* is a set of collections.

MongoDB allows user to execute queries on collections, which range from simple selections to complex aggregations. More precisely, *find* queries allow selecting documents of a collection that satisfy given selection criteria; *count* queries derive the number of documents in a collection that match a condition; *distinct* queries retrieve the distinct values of a key field among the documents that satisfy selection criteria; *group* queries aggregate documents in a collection by keys, executing simple aggregation functions (e.g., sum or avg); *aggregate* queries perform aggregation using a sequence of stage-based manipulations, called aggregation pipeline; finally, *mapReduce* queries allow the execution of MapReduce jobs over a collection of documents.

MongoDB operates according to a client-server architecture. The MongoDB server front-end decides whether to establish a connection and, in this case, it allows clients to send administration commands and queries. The server back-end can be composed of a single node, or multiple nodes distributed over a cluster. Cluster-based solutions allow handling large volumes of data ensuring, at the same time, good performance and availability. The client interacts with the server in a way that is independent from the adopted server side architecture. Several MongoDB drivers[3] allow custom client applications implemented with different programming languages to interact with the MongoDB server. The client-server interaction is based on the MongoDB Wire protocol. MongoDB drivers and the MongoDB shell convert queries and commands, possibly defined with different programming languages, to MongoDB Wire messages. Similarly, responses to client requests are issued by the server in the form of Wire protocol messages. Wire is a TCP/IP socket-based protocol. The communication protocol uses 9 different types of messages (see Fig. 1). Based on their senders, messages are classified into *client requests* and *server responses*.

Message type	Op_code	Category
OP_REPLY	1	server response
OP_MSG	1000	client request
OP_UPDATE	2001	client request
OP_INSERT	2002	client request
RESERVED	2003	-
OP_QUERY	2004	client request
OP_GET_MORE	2005	client request
OP_DELETE	2006	client request
OP_KILL_CURSORS	2007	client request

Fig. 1. MongoDB wire messages.

Clients are allowed to send all types of messages belonging to the *client request* category, whereas servers only send *OP_REPLY* messages and exclusively in reply to client requests of type *OP_QUERY* and *OP_GET_MORE*.

Wire messages are composed of several fields. All messages include the message *header*, a record that provides meta information related to the message content, organized into the fields *messageLength*, which specifies the length of the message (expressed as the number of bytes); *requestID*, which specifies the message identifier; *responseTo*, which is used by messages of type *OP_REPLY* to refer the request for which the message is the response; and *Op_code*, which specifies the message type (e.g., *OP_QUERY*, *OP_REPLY*).

Although all Wire messages are analyzed by Mem, the only messages possibly involved in rewriting activities are those of type *OP_QUERY* and *OP_REPLY*. For space limitations, we only report here a short overview of the fields of these two types of messages (see Fig. 2).[4]

OP_QUERY messages carry queries and administrative commands. Among the *OP_QUERY* fields involved in the rewriting approach, *fullCollectionName* specifies the data resource accessed by the query, whereas the fields *query* and *returnFieldsSelector* encode the selection and projection criteria, respectively. In contrast,*OP_REPLY* messages carry server responses. The response data are stored into the field *documents*, which stores a subset of the documents that compose the query result set, *numberReturned* specifies the cardinality of the set *documents*, whereas *cursorID* and *startingFrom* are used to identify which portion of the query result set is referred to by the current message.

3. See http://docs.mongodb.org/ecosystem/ for an updated list

4. The interested reader can refer to http://docs.mongodb.org/meta-driver for a thorough description of the other messages.

	Field	Description
OP_QUERY	header	message header
	flags	query options
	fullCollectionName	db and collection name
	numberToSkip	# documents to skip
	numberToReturn	# documents to return
	query	query object
	returnFieldsSelector	projected fields
OP_REPLY	header	message header
	responseFlags	query options
	cursorID	cursor identifier
	startingFrom	starting position wrt cursor
	numberReturned	# documents in the
	documents	result set

Fig. 2. Structure of wire messages.

3 Mem Requirements

The RBAC model of MongoDB does not operate at a granularity level which is suited to privacy policy specification and enforcement. Collections represent the finest granularity level at which RBAC is enforced, however, this granularity is not sufficient in quite common application scenarios where documents refer to multiple subjects. If we do a comparison with relational DBMSs, this is similar to regulating the access at table level only. In other words, either a user is authorized to access a whole table or it cannot access any tuple. Several work (e.g., [3]) recognize that fine grained access control [2] is required to support privacy policies. For instance, let us consider a scenario where the documents stored within a MongoDB collection are the emails exchanged by employees of an organization. Since employees could specify different privacy preferences for their emails, a finer granularity level than collection is required to regulate the access. The schemaless data model of MongoDB further aggravates the inappropriate granularity at which MongoDB RBAC operates. Indeed, documents can have different structure, the sensitivity of the stored data can vary according to the structure of the documents, and it may be required to regulate the access based on the presence of given fields within the documents. For instance, emails can have zero to multiple attachments, and different policies could be defined based on the presence of these elements.

We aim at addressing these MongoDB RBAC shortcomings through the definition of Mem, a monitor supporting the efficient enforcement of purpose and role based access control at document level within MongoDB platforms. In the remainder of this section, we provide an overview of the manifold requirements behind Mem definition.

It is worth noting that, although mechanisms to enforce purposed-based fine-grained access control have been proposed for relational DBMSs [3], these cannot be directly applied to MongoDB. The identification of alternative solutions appears as one of the most challenging goals of this work. Indeed, SQL query rewriting (e.g., see [2]), which underneath the techniques proposed for RDBMSs, requires the knowledge of the schema of the protected tables, the availability of a unique query language, and the ease and efficient implementation of join operations used for some enforcement techniques (e.g. [7]). None of these elements is available for MongoDB, which is based on a schemaless data model, and supports clients operating with different APIs and programming languages. Moreover, although join operations can be implemented by means of complex stepwise MapReduce jobs, low performances make them unusable for enforcement purposes. The hypothetical implementation of query rewriting with approaches similar to those proposed for RDBMSs also requires to support queries that include arbitrary Javascript code. Within these scripts the accessed document fields can be referred in a variety of ways, such as using local variables which can be declared and used in different parts of the code. This makes the identification of the accessed fields a complex engineering task, and the rewriting of the queries even a more complex and error prone activity. Based on all these considerations we believe that: 1) query rewriting strategies proposed for RDBMSs cannot be directly and efficiently implemented within MongoDB, and 2) novel approaches need to be defined for MongoDB to fulfil the identified requirements.

A further key requirement is that the enhanced access control features have to be defined on top of those natively provided by MongoDB. In other words, we believe that Mem does not have to integrate an ad-hoc access control model defined from scratch, and MongoDB RBAC has to be complemented rather then disused. Thus, it is required to investigate how the MongoDB RBAC model can be used as the basis for the definition of the Mem's enhanced access control features. The reasons for this development choice are manifold. First of all, some access control models proposed for RDBMSs (e.g., [3], [7]) show the effective combination of purpose and role based access control. In addition, it seems more convenient to use the already existent MongoDB RBAC, rather than defining from scratch a Mem's RBAC component. This choice allows reusing authentication and authorization mechanisms provided by MongoDB, and makes the use of Mem easier within existing MongoDB deployments.

Several non functional requirements specify additional constraints to Mem definition. Mem has to be *general* enough to operate within deployments where MongoDB clients developed with any of the supported programming languages interact with a MongoDB server through proper drivers. Moreover, Mem has to be *robust* enough to support the definition of new drivers. We believe that the achievement of these requirements needs an approach that focuses on the analysis and rewriting of the messages exchanged between MongoDB clients and servers, abstracting from the languages of the multiple existing clients. This novel approach promises to handle the complexity of query rewriting, which, for the previously listed reasons, we believe could be hardly handled with more traditional approaches based on the analysis of queries source code. Mem has to be *efficient*. The enforcement of purpose-based policies at document level should not affect MongoDB usability, and the overhead due to policy enforcement has to be low. Mem has to be easily *integrable* into existing MongoDB deployments, minimizing the activities required to the system administrators, which should be charged with limited configuration tasks, but no programming activity. The choice of building Mem on top of MongoDB RBAC promises to simplify the upgrade of MongoDB deployments. Indeed, role and user management can be propagated without modifications from the original system to the upgraded system. Purpose-based access control is the only aspect which needs to be added.

The development of Mem considering all above mentioned functional and non functional requirements appears as a challenging task. In the next sections, after introducing the fundamental conceptual elements characterizing Mem's purpose-based access control model (Section 4), we describe the development strategies we have followed to define Mem (Section 5).

4 The Access Control Model

The MongoDB RBAC model is characterized by the concepts of *privilege*, *data resource*, *action*, *role*, and *user*. It regulates the access to document collections on the basis of the privileges granted to roles. We enhance this basic scheme introducing fine grained purpose-based access control at document level.

A *privilege* models a set of actions that access a data resource. The referred resource can be a collection, a set of collections, a database, or a set of databases, whereas actions are a subset of the predefined MongoDB data management operations (e.g., *find*, *update*, *remove*, and *insert*), administrative, and review functions (e.g., add/remove a user/role, show users/roles).

A *role* models a set of privileges to be assigned to users. Roles definition is aligned with the principles of the proposed NIST standard for RBAC [11]. Roles are hierarchically organized and a role can extend other roles inheriting their privileges. The set of privileges that are granted to a role r is the union of the privileges specified for r and those specified for each role from which r descends. MongoDB includes a set of predefined roles and allows administrators to introduce custom roles by means of administrative functions.

Finally, element *user* models a stakeholder that interacts with MongoDB requiring the execution of data manipulation operations, administrative, or review functions. When a role is assigned to a user, the user receives the authorization to execute all the actions specified by the privileges associated with the role on the specified data resources.

We have enhanced the MongoDB RBAC model with additional conceptual elements, instrumental to support purpose-based access control. The key element of the enhanced model is the concept of *purpose*, which is used to specify the reasons for which documents can be accessed, and to declare the reasons for which users aim at accessing them. The union of the purposes considered for an application scenario forms the scenario purpose set *Ps*. The purposes for which a document can be accessed are denoted as *intended purposes*. The intended purpose set specified for a document d is a set *Ip* that groups the identifiers of the purpose elements of *Ps* which specify the reasons for which the access to d is allowed.

In order to access data through the execution of actions, users have to specify the *access purpose*, i.e., the reason for which they aim to perform the access. An access purpose is specified by referring to an element of *Ps* by means of the element identifier, denoted as *pid*.

Users are only allowed to execute queries for access purposes for which they have a proper authorization. Purpose authorizations are granted to users as well as to roles.

Concept	Description
Privilege *	A set of actions authorized to the access of a data resource
Data resource *	A collection / a set of collections / a database / a set of databases
Action *	A MongoDB read/write operation
Role *	A set of privileges to be assigned to users
User *	A stakeholder requiring actions execution
Access purpose$^+$	The reason for which queries access data
Intended purposes $^+$	The reasons for which data can be accessed
Purpose authorization$^+$	The set of purposes authorized for a user/role
Connection $^+$	A communication channel through which an authenticated user interacts with MongoDB
Session $^+$	A period during which queries are invoked for an access purpose on a connection

Fig. 3. Native concepts of MongoDB RBAC (*) and new concepts of the enhanced purpose based access control model ($^+$).

Definition 1 (Purpose authorization). *A purpose authorization pa is a tuple* $\langle id, tp, Aps \rangle$ *where id specifies the identifier of a role or user, tp indicates if id refers to a user or a role, and* $Aps = \{pid \mid \exists p \in Ps \wedge p.pid = pid\}$ *is the set of identifiers of access purposes in Ps authorized for id.*

The union of the purpose authorizations defined for an application scenario forms the purpose authorization set *As*.

Users interact with MongoDB through communication channels referred to as connections. In order to issue any request through a connection c, a user must have successfully executed his/her authentication on c.

Definition 2 (Connection). *A connection c is a tuple* $\langle id, uid, adb \rangle$*, where id is the connection identifier, uid is the identifier of the user that has opened the connection, and adb is the name of the database on which the user is authenticated.*

The concept of session is used to define an interval during which a user, who is authenticated on a connection c, issues command execution requests through c for a given access purpose. A user u can activate a session on a connection c for an access purpose p iff u has been authenticated on c and p is among the access purposes authorized for u. If a session s has been activated on c for the purpose p and the user authenticated on c requests to activate a second session s' for purpose p', s' substitutes s as active session of c. At most one session at a time can be active for a connection.

Definition 3 (Session). *A session s is a pair* $\langle cid, pid \rangle$*, where cid is the identifier of the connection within which the session is activated, whereas pid is the identifier of the purpose element specified as access purpose of s.*

A summary of the elements of MongoDB RBAC and the enhanced access control model is shown in Fig. 3.

MongoDB provides a rich set of administrative functions. Let us denote with *Us* and *Rs* the sets of users and roles that have been defined for an application domain. User and role management functions allow manipulating *Us* and *Rs*, respectively. For instance dedicated functions allow one to add/remove a user/role and to add/remove the privileges that are granted to a role. The interested reader can refer to the MongoDB documentation for a complete overview of the available functions.

5 Mem

In this section, we discuss the design and implementation of Mem, a software module which implements the enhanced access control model presented in Section 4.

Mem has been designed with the goal to operate with any driver and different MongoDB versions, and to make straightforward its integration into a MongoDB deployment. These requirements lead us to define the enforcement monitor as a specialized MongoDB Wire protocol interpreter. More precisely, Mem has been developed as a Java multi-thread socket server which operates as a proxy in between a MongoDB server and its clients, intercepting, analyzing, and possibly rewriting the MongoDB Wire messages that are exchanged by the clients and the server. Although MongoDB source code and API continuously evolve, we believe that, apart from extensions, the definition of the Wire protocol will be preserved to allow existing clients to operate with new server versions. Mem does not substitute or obfuscate the MongoDB access control features, rather, it relies on the security mechanisms natively provided by MongoDB, complementing and refining them.

For space restrictions, we do not discuss here implementation technicalities, we only focus on selected aspects related to system configuration and development of enforcement mechanisms.

5.1 MongoDB Server Configuration

Mem relies on two basic configuration activities that must be performed on the MongoDB server: 1) the enhancement of *admin*, namely the native MongoDB database for administrative data, and 2) the specification of intended purposes for the documents to be protected.

The enhanced access control model introduces the concepts of *purpose* and *purpose authorization* (see Section 4), which are not natively handled by MongoDB. Similar to users and roles information, purposes and purpose authorizations must be constantly accessible within application scenarios, and thus we define them as documents of the collections *purposeSet* and *authorizationSet*, which are added to the *admin* database.

Purpose documents are structured as records composed of the fields *id* and *code*, where *id* specifies the name of the purpose element, whereas *code* specifies the position of the element within the purpose set, based on a given ordering criterion. The structure of the documents in *authorizationSet* maps the purpose authorization tuple (see Def. 1). To minimize enforcement overhead, the set of authorized purposes *Aps* is defined as a numeric field encoding a binary mask. Each bit of the field *Aps* encodes a purpose element, and the bit is set to 1, if the corresponding purpose is authorized, it is set to 0, otherwise. The field *code* of *purpose* documents gives the position of the corresponding bit within the binary mask. Update privileges to the collections *purposeSet* and *authorizationSet* are natively granted to the role *root* and can be optionally granted by users covering *root* to other administrative roles.

The set of intended purposes specified for a document is encoded as a boolean array *ipa* of size corresponding to the maximum allowed size of the purpose set.[5] *ipa[x]* is set to *true*, if *x* corresponds to the value of field *code* of a *purposeSet* document which models a purpose that composes *ip*, it is set to *false*, otherwise.

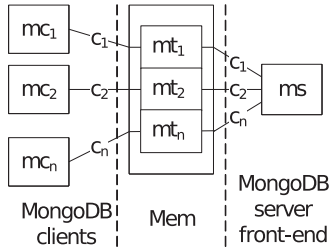


Fig. 4. Mem application scenario.

5.2 Enforcement Mechanisms

The enhanced model requires the definition of two main control mechanisms: 1) *session activation control*, which allows the activation of a session on a connection if the access purpose specified by the requiring user for the session is a valid purpose and is authorized to the user; and 2) *purpose-based access control at document level*, which regulates the execution of queries issued by a user on a valid connection. The target application scenario is characterized by n MongoDB clients mc_1 ..mc_n, possibly operating with different drivers, interacting with ms, a MongoDB server front-end. $mc_1..mc_n$ are configured specifying as target host the one on which Mem is deployed, whereas Mem is configured to interact with ms. The scenario is shown in Fig. 4.

Mem continuously listens for connection requests issued by potential clients. When a client mc_i sends a connection request, Mem opens a communication channel with mc_i and requests ms to open a connection through which all Wire messages related to the communication between mc_i and ms will flow. The client-server interaction is thus achieved by means of a macro connection c_i composed of two branches: mc_i to/from Mem and Mem to/from ms. Once the connection c_i has been established, Mem activates a dedicated monitoring thread mt_i in charge of analyzing all the messages that transit through c_i and enforcing purpose-based access control.

Once a message m is intercepted, mt_i first checks m's type. *OP_REPLY* and *OP_QUERY* messages (cfr Section 2) are processed by functions *analyzeResponse* and *analyzeRequest*, respectively, whereas messages of other types are directly forwarded. Listing 1 shows the pseudo-code of *analyzeResponse* and *analyzeRequest*.

In what follows, we denote with *u* a user who has been authenticated on a database *db* through a connection c_i that links the MongoDB client mc_i used by *u* with the MongoDB server ms hosting *db*. In addition, we denote with mt_i the Mem thread operating on c_i.

5.2.1 Session activation

A session can only be activated if the specified access purpose is valid and among the purposes authorized for the requiring user. Let *getAap* be a function that derives the

5. We believe that it is reasonable to assume that it should not include more that few tens of elements.

union of the access purposes authorized for all the roles assigned to u and the access purposes (if any) directly authorized to u. Precondition to activate a session on a connection c on request of an authenticated user u specifying p as access purpose is that $\exists p \in Ps \wedge p \in getAap(u)$.

Listing 1. Mem's Core Analysis Functions

```
1  function analyzeResponse(OP_REPLY m){
2   id=getResponseTo(m);
3   if(id==getRequestId(proxyReq)){
4    switch(typeOfCmd(proxyReq)){
5     case authenticate:
6      if(isSuccessfulAuthentication(m)){
7       authReply=m; pars=null;
8       sendProxyMsg(connectionStatus,pars,m);
9      } else{forward(m)} break;
10    case connectionStatus:
11     pars=getUsersInfoPars(m);
12     sendProxyReq(usersInfo,pars,m); break;
13    case usersInfo:
14     pars=getRolesInfoPars(m);
15     sendProxyReq(rolesInfo,pars,m); break;
16    case rolesInfo:
17     pars=getFindAuthPars(m);
18     sendProxyReq(findAuth,pars,m); break;
19    case findAuthorizations:
20     pm=deriveAuthorization(m);
21     sendProxyReply(authReply); break;
22    case findPurpose:
23     isValid=checkValidity(m)
24     if(isValid) sAP=ap;
25     sendProxyReply(deriveActReply(isValid));
26     break;
27   }
28  }else{forward(m);}}
29
30  function analyzeRequest(OP_QUERY m){
31   switch(typeOfCmd(m)){
32    case authenticate:
33     proxyReq=m; break;
34    case setParameter:
35     pars=getFindPurposePars(m)
36     sendProxyMsg(findPurpose,pars,m); break;
37    case query: rewriteQuery(m);
38   } forward(m);
39  }
```

Mem implements session activation control in two steps: 1) profiling the requesting user by deriving the user authorized access purposes, and 2) controlling the validity of the access purpose specified for the session to be activated wrt the purpose set computed during step 1.

User profiling. In order to derive the purposes that are authorized for the user requesting to activate a session, Mem has first to identify the user and collect information related to the covered roles. Data related to a user u authenticated within a connection c are bound to the state information of c, and are derivable querying the MongoDB server on the connection status. User's data are available starting from the authentication time for the whole connection lifetime, or until another authentication is executed within the same connection. As such, Mem activates the profiling process as soon as the user asks to be authenticated.

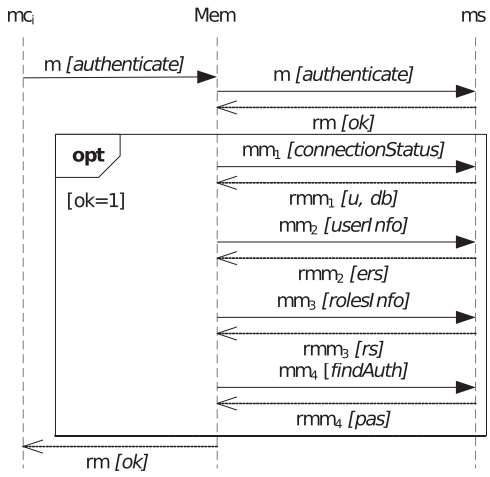


Fig. 5. Messages exchanged for user profiling.

Message exchange related to the profiling process is illustrated, at a high level of abstraction, in Fig. 5. Labels between brackets associated with each transmitted message informally describes the content of the message. Right arrows represent messages of type *OP_QUERY*, whereas left arrows refer to messages of type *OP_REPLY*. The optional frame *opt* groups messages that are only exchanged if the condition between brackets is satisfied.

Let m be a message that encodes the *authenticate* command which has been sent to the server ms by a client mc_i through the connection c_i monitored by the Mem thread mt_i. When Mem intercepts m, function *analyzeRequest* of mt_i is invoked (m is of type *OP_QUERY*) specifying m as parameter (cfr line 30 of Listing 1). If the field *fullCollectionName* of m (see Fig. 2) specifies *$cmd* as collection identifier and the field *query* includes the sub-field *authenticate*, function *typeOfCmd* recognizes m as an authentication request (cfr line 31). In this case a copy of m is stored within *proxyReq* (a global variable of mt_i), and m is forwarded to *ms* (cfr lines 33 and 38). The server can either successfully authenticate the requesting user, or it can invalidate the request. The server response is issued back to Mem through c_i, encoded as an *OP_REPLY* message denoted *rm*.

When mt_i intercepts *rm*, function *analyzeResponse* is invoked. *analyzeResponse* compares the field *requestID* of *rm* with the field *responseTo* of the previously stored message *proxyReq* (cfr lines 2-3), recognizing that *rm* is the server response to the previous authentication request. Function *isSuccessfulAuthentication* is then invoked to analyze the content of the field *documents* of *rm*. If the field *ok* of the included document is set to *0*, the authentication fails, and *rm* is forwarded to the client. If the field *ok* is set to *1*, the server has successfully authenticated the requester user. In this case, mt_i stores a copy of *rm* within the global variable *authReply* (cfr line 7), but it does not forward *rm* to mc_i as before. As a result mc_i is blocked waiting for the server response to its authentication request and it is temporarily unable to issue other commands to the server. mt_i exploits this state to inject into c_i several commands aimed at profiling the authenticated user. All the commands are encoded

as *OP_QUERY* messages and sent to *ms* by executing function *sendProxyReq*. Similar to the previous case, to allow the identification of future server responses, *sendProxyReq* keeps track of a copy of the issued message within *proxyReq*. More precisely, *sendProxyReq* is first invoked to issue message mm_1, which encodes a *connectionStatus* command finalized to derive information related to the active connection (see line 8). *ms* responds with a *OP_REPLY* message denoted rmm_1. When mt_i intercepts rmm_1, *analyzeResponse* is invoked. After verifying that rmm_1 encodes the server response to the *connectionStatus* request (cfr lines 2-4), function *getUserInfoPars* is invoked to extract from the field *documents* of rmm_1 the identifiers of the authenticated user *u* and of the target database *db* on which u is authenticated (cfr line 11). Such data are specified as input parameters of the command *usersInfo*, which allows deriving the roles explicitly granted to *u*. *sendProxyReq* is therefore invoked to send an *OP_QUERY* message, denoted mm_2, to *ms*, which encodes the *userInfo* command (cfr line 12). The server replies with an *OP_REPLY* message denoted rmm_2. Following the same strategy used for previous messages, once intercepted, rmm_2 is analyzed by *analyzeResponse*. Function *getRolesInfoPars* extracts from rmm_2 the set of roles *ers* that have been explicitly granted to *u* (cfr line 14). These can extend other built-in or custom roles, as such to derive the whole set of roles assigned to *u* the command *rolesInfo* is issued (cfr line 15), which is denoted as mm_3. The whole set of roles granted to *u*, referred to as *rs*, is extracted from rmm_3, namely, the server response to mm_3 (cfr line 17). In contrast, purpose authorizations granted to *u* are derived by executing a *find* query on the collection *authorizationSet* of the database *admin*, specifying as selection criteria that *documents* include either the field *user* set to *u* or the field *role* set to one of the elements of *rs*. *sendProxyReq* is invoked to issue such a *find* query, which is encoded as an *OP_QUERY* message denoted mm_4 (cfr line 18). The *OP_REPLY* message rmm_4 returned by *ms* includes the list of purpose authorizations that are granted to *u*. This is computed as the union of all the elements referred to by component *Aps* of the purpose authorizations granted to *u*. Due to the binary encoding of *Aps* (see Section 5.1) this set is straightforwardly calculated by function *deriveAuthorization* executing the or-bitwise on the field *Aps* of the *authorizationSet* documents included in rmm_4 (cfr line 20). The resulting value is a binary mask *pm*, referred to as *purpose authorization mask*, which encodes the set of authorized access purposes. Finally, the previously stored copy of message *rm*, which keeps track of the successful user authentication, is issued back to mc_i, executing function *sendProxyReply* (cfr line 21).

Access purpose validity check: a session can only be activated by a user on the same connection that has been used for the authentication. Based on the profiling mechanisms presented in Section 5.2.1, if a user is authenticated it has also been profiled, and a purpose mask, which encodes all the authorized purposes, has been derived for him/her.

Within Mem, session activation requests are specified by means of the command *setParameter*, providing as input parameter the field *accessPurpose* which is initialized to the identifier of the access purpose that should be activated for the session.[6] Assuming that *u* aims at activating *s* specifying *p* as access purpose, he/she will specify the command:

db.runCommand("setParameter":1, "accessPurpose":"p")[7]

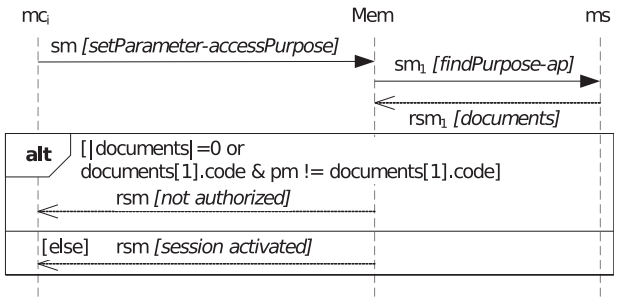


Fig. 6. Interaction for session validity check.

Fig. 6 shows the messages exchanged to check session validity. The main involved activities and details related to the interaction are presented in what follows.

Once the session activation command is entered, mc_i encodes it as an *OP_QUERY* message, denoted as *sm*, and transmits *sm* on c_i. *sm* is intercepted by mt_i and analyzed by function *analyzeRequest* (cfr line 30 of Listing 1). When *typeOfCmd* is invoked, this function recognizes that *sm* encodes a session activation command (cfr line 31). Indeed, the (sub) fields *setParameter* and *accessPurpose* are included within the field *query* of *sm* (see Fig. 2 for the structure of *OP_QUERY* messages). Function *getFindPurposePars* is then invoked to extract from the field *query* of *sm* the value of *accessPurpose* (cfr line 35). Function *sendProxyReq* is then invoked to request the execution of a predefined *find* query on the database *admin*, which selects all *purposeSet* documents whose field *name* is set to *ap*. *sendProxyReq* encodes the query execution request as an *OP_QUERY* message denoted sm_1, and sends sm_1 to ms (cfr line 35). The server answers with an *OP_REPLY* message denoted rsm_1 which is intercepted by mt_i. The analysis of rsm_1 is achieved by function *analyzeResponse*, which verifies that rsm_1 is the reply to message sm_1 (cfr lines 1-4), and invokes function *checkValidity* to analyze the content of rsm_1 (cfr line 23). If no element is included in the field *documents* of rsm_1, *p* is not a valid element and thus the session cannot be activated. Otherwise, if a document *d* is included in *documents*, *checkValidity* verifies whether the purpose referred to by *d* is among the purposes authorized to *u*. This is achieved executing the *and-bitwise* of the field *code* of *d* and the purpose mask *pm* derived at profiling time. More precisely, if *code& pm==code*, the element referred to by *ap* is among the authorized purposes. In this case, the session is activated keeping track of the specified access purpose *p*, which is stored within the global variable *sAP* of mt_i (cfr line 24). Finally, function *deriveActReply* is invoked to derive an *OP_REPLY* message, denoted *rsm*, which encodes the response to the activation request encoded by *sm*, whereas function *sendProxyReply* is executed to send *rsm* to mc_i (cfr line 25).

6. *setParameter* is a predefined MongoDB function used to configure target data store properties. We extend the set of *setParameter* target properties with the new property *accessPurpose*. When *setParameter* is invoked specifying *accessPurpose* as target property, the command is only processed by Mem and it is not forwarded to the server.

7. The used syntax is the one supported by the MongoDB shell. Different syntaxes can be used with other clients.

5.2.2 Purpose-Based Access Control

Let q be a query that accesses a collection *cl* of a database *db*, which is submitted for execution by u through c_i. If based on MongoDB RBAC, q execution is allowed, the access performed by q is then regulated based on the session that has been possibly activated by u on c_i and the intended purposes possibly specified for *cl* documents. More precisely, if no session is active on c_i when q execution is requested, q is only authorized to access documents for which no intended purpose has been specified. Conversely, if a session s is active on c_i at q invocation time, q can only access documents whose intended purposes comply with the access purpose p specified for s, or documents for which no intended purpose has been specified.

In order to introduce the approach that implements the above mentioned checks, by abstracting from syntactical aspects related to queries, we model a MongoDB query as follows.[8]

Definition 4 (MongoDB Query). *A MongoDB query q is a tuple ⟨db: text, cl: text, tp: {f,c,d,g,a,m}[9], bd: object⟩, where cl is a collection of a database db accessed by q[10], tp specifies the type of operation performed by q, and bd represents the query body.*

The structure of the query body depends on the achieved operation. It can range from a straightforward expression (e.g., *collection.find()*) to a complex structure with several components and functions (e.g., *mapReduce* queries).

Example 1. Let us consider a query q defined for the collection *messages* of a database *emails* which stores the emails exchanged by the employee of a company. Let q be a *find* query that selects the emails sent by the email account of Alice Smith. Based on Definition 4, q is modeled as ⟨*db: emails, cl: messages, tp: f, bd: db.messages.find({headers.from:' alice.smith@company.it'})*⟩.

All query operations allow one to specify *selection criteria* that limit the documents of the target collection that are actually accessed during query execution. Abstracting from structural details of query bodies, component *bd* of a query q includes, among others, the specification of the *selection criteria* defined for q, namely a predicate built on top of field identifiers which is used as a document selection filter. Let us model the selection criteria predicate defined for a query q as the sub-component *cd* of *bd*, referring to such a component with notation *q.bd.cd*. For instance, in Example 1, component *cd* of q is the expression between curly braces specified as input parameter of *find*.

Purpose-based access control is achieved by rewriting q as a query q', accessing only purpose compliant data. The rewriting targets the enhancement of the selection criteria *cd* specified for the submitted query q operating at the level of the Wire message m that encodes q. Each *OP_QUERY* message m issued by u through c_i, which according to the analysis performed by function *analyzeRequest* (cfr lines 30 and 31 of Listing 1) results to encode a query q, is processed by function *rewriteQuery* (cfr line 37). *rewriteQuery* rewrites m in such a way that the selection criteria specified within the field *query*, which encodes component *cd* of *q.bd*, also integrates purpose-based access control checks. Afterwords, the message is forwarded to m_S (cfr. line 38).

Listing 2 shows the pseudocode of *rewriteQuery* where first the complementary selection criteria *ac* are derived, and then integrated into the considered message.

Listing 2. Function *rewriteQuery*

```
1  function rewriteQuery(OP_QUERY m){
2   if (sAP!=null){
3     x=sAP.code;
4     ac={$or:[{ip: {$exists:false}},{ip.x:
      true}]};
5   }
6   else{ ac={ip: {$exists:false}}; }
7   switch(typeOfOp(m)){
8     case find: case count: case distinct:
9     case mapreduce: integrateAC(m,ac,'query');
10    break;
11    case group: integrateAC(m,ac,'cond');
12    break;
13    case aggregate: integrateAC(m,ac,'$match');
14  }
15 }
```

Different derivation strategies are applied for the cases in which a session / no session is active when q is invoked.

Active session. Let us first consider a scenario where q is submitted for execution within a session s which has been activated for the access purpose p. Required condition to allow the access to a document d by q is the compliance of p with the intended purposes *Ip* possibly specified for d. If $Ip \neq \emptyset$, the condition is satisfied iff p is among the purpose elements of *Ip*.

Definition 5 (Purpose compliance). *Let p be an access purpose and let Ip be an intended purpose set such that $p \in Ps \wedge \forall p \in Ip \rightarrow p \in Ps$. p complies with Ip iff $p \in Ip$.*

In contrast, if no intended purpose is specified for d (i.e., $Ip = \emptyset$), the condition is satisfied regardless p.

As a consequence, if no selection criterion is specified for q, $q'.bd.cd$ has to specify that $(Ip \neq \emptyset \wedge p \in Ip) \vee (Ip = \emptyset)$, otherwise $q.bd.cd \wedge (Ip \neq \emptyset \wedge p \in Ip) \vee (Ip = \emptyset)$. The implementation follows the above-mentioned checks. Since m has been sent within a session s, the global variable *sAP* of mt_i stores the access purpose activated for s (cfr. line 24 of Listing 1 and Section 5.2.1). Thus, in this case *sAP!=null*. The complementary selection criteria *ac* are defined in such a way to specify a condition that checks if the access purpose p activated for s and stored within *sAP*, complies with the intended purposes, if any, specified within the fields *ip* of the documents accessed by q. The complementary selection criteria *ac* is encoded as the JSON object {*$or*: [{*ip*: {*$exists* : *false*}}, {*ip.x*:*true*}]}, [11] where x specifies the value of

8. For a detailed presentation of queries, the interested readers can refer to http://docs.mongodb.org/manual/

9. The enumerated values correspond to the initial letters of the operation names (see Section 2).

10. MongoDB queries access only one collection.

11. For the sake of brevity the pseudo code in Listing 2 shows the JSON representation of *ac* but it does not show the object instantiation and setting.

component *code* of the purpose element *p* (cfr. lines 3-4 of Listing 2). *ac* is satisfied if either no field *ip* has been specified for *d*, or the *x-th* element of the array *ip* is set to *true*.

No active session. If no session is active when *q* is submitted for execution, the access is only granted to documents for which no intended purpose has been specified. In such a case, if no selection criterion is specified for *q*, $q'.bd.cd$ has to be set to $Ip = \emptyset$, otherwise to $q.bd.cd \wedge Ip = \emptyset$.

The implementation is straightforward. Since no session has been activated, *sAP* is *null*. *ac* is defined in such a way that queries can only access documents for which no purpose-based policy has been specified. Thus, *ac* specifies that {*ip*: {*$exists* : *false*}} (cfr. line 6).

Once the enhanced selection criteria has been derived, it needs to be integrated into the query. The integration strategy depends on the operation modeled by *m*. Function *typeOfOp* (cfr. line 7) infers the operation type by analyzing the possible presence of the sub-field *count*/ *distinct*/ *group*/ *aggregate*/ *mapreduce* within field *query* of *m*. The integration is achieved by function *integrateAC*, which is invoked specifying the target (sub) field of *m* where *ac* has to be integrated. If *m* encodes an operation of type *find*, *distinct*, *count*, or *mapreduce*, *integrateAC* conjuncts *ac* with the selection criteria specified within the field *query* (cfr. line 9 of Listing 2). In this case the JSON object that models *ac* is appended as sub-field of *query* within *m*. Similarly, if *m* encodes an operation of type *group*, *integrateAC* conjuncts *ac* with the selection criteria specified within the sub-field *cond* (cfr. line 11), whereas in case *cond* is not included among the sub-fields of *query*, *cond* is added to *query* and then it is set to *ac*. Finally, if *m* models an operation of type *aggregate*, but the *pipeline* does not specify *$match* as first operator, *integrateAC* adds the field *$match* set to *ac* at the head of *pipeline*. Otherwise, *integrateAC* conjuncts *ac* with the selection criteria specified within the sub-field *$match* (cfr. line 13).

Example 2. Let us suppose that query *q* presented in Example 1 is submitted for execution within a session activated for purpose *p1*, whose *code* field is *5*. The *OP_QUERY* message *m* that encodes *q* is intercepted and analyzed by Mem. Function *analyzeRequest*, after recognizing *m* as a query, invokes *rewriteQuery* (cfr. lines 31 and 37 of Listing 1). Since a session has been activated, which specifies *p1* as access purpose, the global variable *sAP* is set to *p1*. Thus, the first branch of the conditional instruction is executed to derive *ac* (cfr. lines 3-4 of Listing 2). *typeOfCmd* recognizes that *m* encodes a *find* operation, thus *integrateAC* is invoked to integrate *ac* into the field *query* of *m* (cfr. lines 7-9 of Listing 2). As a result, the rewritten query is *db.messages.find({headers.from:'alice.smith@company.it',$or: [{ip:{$exists: false}}, {ip.5: true}]})*.

5.3 Enforcement Correctness and Completeness

The correctness and completeness of the enforcement mechanisms implemented by Mem is stated by the theorems introduced below. Proofs are presented in Appendix A, available in the online supplementary material. The theorems refer to a scenario composed of a MongoDB server front-end, several MongoDB server nodes and MongoDB clients, and one server hosting Mem. The system is configured in such a way that: 1) the server front-end along with the rest of the MongoDB cluster are deployed in a trusted LAN, 2) all MongoDB clients are deployed on external networks, and 3) Mem is deployed on a DMZ proxy[12] operating as unique interface of the LAN with the external networks. Firewalls placed at the internal and external interfaces of the DMZ proxy block all communication ports but the ones used for MongoDB connections. The designed deployment forbids components of the external networks to open a communication channel with components of the LAN, which is not mediated by the DMZ proxy. In addition, when Mem is active, all MongoDB connections are mediated by Mem, thus no clients can directly connect with any server node. These assumptions, which can be easily implemented by system administrators, comply with MongoDB security recommendations,[13] according to which, to limit network exposure, MongoDB should run in a trusted network environment and the interfaces on which MongoDB instances listen for incoming connections should be limited.

Let us consider an instance of the above mentioned scenario where a MongoDB server *ms* hosts a database *db* which stores the collection *cl*. Let *u* be a user authenticated on *db* via a connection *c* monitored by the Mem thread *mt*.

Let us start considering enforcement mechanisms related to session activation. Let *Ps* and *Au* be the purpose set, and purpose authorization set of the considered scenario, respectively, and let *p* be an element of *Ps*.

Theorem 1 (Correctness of session activation). *A session s specifying p as access purpose is activated by mt, upon request by u, iff there exists a purpose authorization pa granted to u which includes p among the authorized purposes.*

Let us consider now query rewriting, assuming that *u* has activated a session *s* on *c* for the access purpose *p* which, according to Theorem 1, implies that *p* is authorized for *u*. Let *q* be a MongoDB query that accesses *cl*, which is submitted for execution by *u* within *s*. Let q' be the rewritten version of *q* generated by Mem, which is submitted for execution to *ms*,[14] and let Ad' be the set of *cl* documents accessed by q', which will be used to derive q' result set.

Theorem 2 (Correctness of query rewriting). *For each document d accessed by q', namely, $\forall d \in Ad'$, either no intended purpose has been specified for d, or the intended purpose set Ip specified for d complies with p.*

If Mem is enabled, when *q* is invoked, Mem rewrites *q* as q'. Therefore, *ms* can only execute *q* in a deployment where Mem has been disabled and the DMZ proxy hosting Mem has been configured for MongoDB port forwarding, which allows external MongoDB clients to establish a connection *c* with *ms*. It is worth noting that, even though Mem is disabled, due to MongoDB RBAC, *q* can only be executed if *u* covers a role authorized to access *cl*.

12. https://tools.ietf.org/html/rfc2647
13. http://docs.mongodb.org/manual/
14. Based on MongoDB RBAC, q' execution is only allowed if *u* covers a role *r* ($r \in Rs$) with read access privileges on *cl*.

Let us now suppose that Mem is disabled and *ms* executes the original query q. Let Ad be the set of all the documents of cl that are accessed during q execution. Query rewriting completeness is stated by the following theorem.

Theorem 3 (Completeness of query rewriting). *There does not exist a document d accessed by q ($d \in Ad$), whose intended purposes Ip comply with p, and such that d is not accessed by q' ($d \notin Ad'$).*

Correctness and completeness of query rewriting can also be similarly proved for a scenario where no session is active at query invocation time.

6 Experimental Results

In this section, we assess the impact of Mem on MongoDB query execution time, analyzing several MongoDB queries that access a target dataset on the basis of multiple MongoDB configurations. The target dataset is the Enron corpus [15], a well known benchmark for data mining techniques. In the remainder of this section after shortly introducing Enron and the queries used for the experiments, we discuss the criteria we have used to specify the intended purposes that regulate the access to Enron emails. Afterwords, we discuss the experiments.

6.1 The Enron Corpus

The Enron corpus is a large public dataset of email messages that includes over 500K emails exchanged by employees of the Enron corporation. Enron corpus characteristics make this dataset ideal to be imported and analyzed with MongoDB. Emails can be easily represented with the document based data model of MongoDB as JSON objects. The document based model also allows the straightforward modeling of composite fields, like the email header, by means of composite objects. In addition, the easy to use but powerful MongoDB analysis features (e.g., MapReduce and the aggregation framework), allow the definition of advanced analytics forms. For our experiments we use a collection of 12 queries based on those proposed by Russel [21]. Although achieving the same type of analytics proposed in [21], in some cases we provide a different query implementation as we are interested in measuring Mem overhead with all types of query operations. As such, we consider queries that: count documents, select a single document from a collection, select distinct documents in a collection, use basic grouping operations, the aggregation framework, and MapReduce. Fig. 7 summarizes the queries that have been considered for the experiments.

The version of Enron corpus that has been used is derived from the MongoDB dump available at http://mongodb-enron-email.s3-website-us-east-1.amazonaws.com. The imported MongoDB database, denoted *enron* in what follows, is composed of a single collection *messages* which includes documents characterized by fields that specify intrinsic structural properties of an email (e.g., the email body and receiver).

To execute our experiments, it was first necessary to specify the intended purposes that regulate the access to Enron emails. Our aim is to evaluate Mem enforcement overhead varying the characteristics of the specified

Id	Query description	Type
q1	Count messages issued in a date range	count
q2	Select a single message to look at	find
q3	Select emails sent in a specific time interval and sort them by date	find
q4	List all distinct receivers	distinct
q5	List all distinct senders	distinct
q6	Derive senders in common with receivers	aggregate
q7	Derive senders who did not receive any email	aggregate
q8	Derive senders who has also received an email (To/Cc/Bcc)	aggregate
q9	Derive senders of emails who were Enron employees	aggregate
q10	Count the number of messages received by given email addresses	group
q11	Derive the recipient lists of each sender	aggregate
q12	Derive the number of emails received by a user grouping them by message sender	MapReduce

Fig. 7. Analyzed queries.

intended purposes. In particular, we are interested to analyze in which measure the *selectivity* of intended purposes affects the overhead, where by selectivity we mean the percentage of documents of a collection accessed by a query which are filtered out from the result because of purpose-based access control. To this aim, let us first formalize the selectivity concept, and then let us analyze how we can tune the selectivity. Let Ps be a purpose set composed of n purposes p_1,..p_n, let C be the collection *messages* of *enron*, and let us assume that an intended purpose set Ip defined over Ps has been specified for each document d in C.

Definition 6 (Selectivity). *Let p be the access purpose specified for a user activated session s. The selectivity of the intended purpose sets specified for documents in C evaluated wrt p, denoted as s_C^p, is the percentage of documents of C whose intended purpose set Ip does not comply with p. More precisely, $s_C^p = |\{d|d \in C \wedge p \notin d.Ip\}|/|C|$, where with $||$ we denote the cardinality of the set.*

If $s_C^p = 0$, the intended purposes specified for the documents in C evaluated for compliance wrt the access purpose p have no pruning effect. Conversely, if $s_C^p = 1$, the specified intended purposes prohibit the access to any document of C regardless of the queries executed for access purpose p.

In order to evaluate enforcement overhead wrt the intended purposes selectivity, intended purposes are defined in such a way that they can ensure given selectivity values for given access purposes. We consider 6 target selectivity values in the range [0,1] each to be reached wrt an access purpose p_i, where $1 \leq i \leq 6$. Let $apS = \{p_1, p_2, p_3, p_4, p_5, p_6\}$ be the set of access purposes wrt which the intended purpose set to be defined for C documents shall reach the target selectivity measures 1, 0.8, 0.6, 0.4, 0.2 and 0, respectively. We aim at specifying intended purposes for documents in C in such a way that $s_C^{p_1} = 1$, $s_C^{p_2} = 0.8$, $s_C^{p_3} = 0.6$, $s_C^{p_4} = 0.4$, $s_C^{p_5} = 0.2$ and $s_C^{p_6} = 0$. Moreover, intended purpose sets have to be specified in such a way that, the set of documents accessed by q within a session must be a subset of those accessed within any other session with lower selectivity. This ensures a fair comparison of the execution time of q in sessions with different selectivities. More precisely, let q be a query accessing C, d a document

of C that satisfies q's original selection criteria, and s_1 and s_2 two sessions respectively specifying p_1, $p_2 \in Ps$ as access purposes, and within which q is executed. The intended purposes for C documents must be specified in such a way that given the target selectivity measures $s_C^{p_1}$ and $s_C^{p_2}$ such that $s_C^{p_1} < s_C^{p_2}$, if d is accessed by q within s_1 it is also accessed within s_2. This ensures that selection criteria always have the same impact on query execution time with the decrease of the selectivity.

The process used to specify the intended purposes for Enron messages, which implements the above mentioned criteria, is presented in Appendix B, available in the online supplementary material.

6.2 Experiments

For our experiments, we have used both a MongoDB 2.6.3 single node configuration and a cluster-based configuration characterized by five nodes. In both the scenarios, each node is a Linux Debian 64-bit PC equipped with a 64-bit 1.6GHz CPU and 2 GB of RAM. In the cluster-based configuration we have enabled data sharding on the collection *messages*, specifying field *_id* as sharding key. Due to the enabled auto-balancing, at experiments running time, each node includes around 1/5 of the documents in the target collection.

For each set-up, we consider seven different execution scenarios within which we execute in sequence the 12 MongoDB queries presented in Fig. 7. Experiments were run 10 times. More precisely, we first consider a scenario where Mem is not active and the MongoDB client directly interacts with the MongoDB server front-end. This case is introduced to measure the execution time of queries when no enforcement mechanism is enabled. We then consider a running scenario for each policy selectivity measure based on which we have generated the intended purposes for the documents in *messages* (see Section 6.1).

The MongoDB client that we have used for our experiments is a MongoDB shell hosted by a PC with characteristics similar to the server nodes. Experiments are run executing on the MongoDB client a Javascript function that activates a session for a considered target access purpose and sequentially invokes the execution of the queries within such session. This task is repeated several times by varying the access purpose referred to by the session. More precisely, we consider the access purposes p_1, p_2, p_3, p_4, p_5, and p_6 wrt which the intended purposes specified for *messages* documents provide selectivity 0, 0.2, 0.4, 0.6, 0.8 and 1, respectively (see Section 6.1).

Based on our experiments the time spent by Mem to perform the rewriting of a MongoDB Wire message is around 2 ms. Since the rewriting overhead is small, constant, and only affects Wire OP_QUERY messages, i.e., query execution requests, we consider it as negligible with respect to the overall query execution time. The execution time is calculated as the difference between the time at which the MongoDB client issues the query execution request and the time at which it receives a response. The measured execution time includes Mem enforcement activities and network communication latencies between the MongoDB client and Mem and between Mem and the MongoDB server.

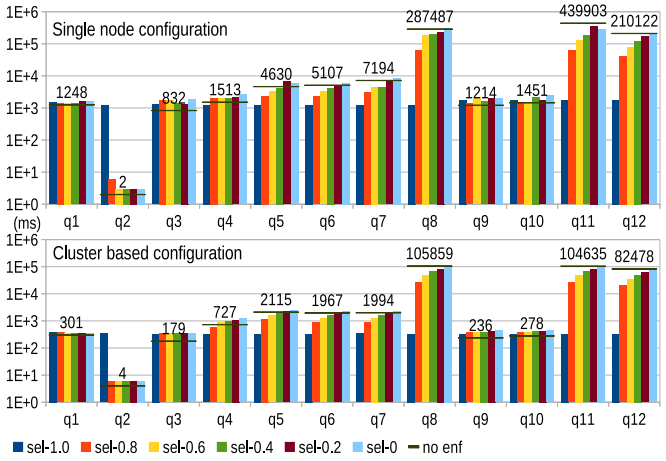


Fig. 8. Queries execution time.

Figs. 8 shows the execution time in ms for queries q1-q12 in Fig. 7, for the seven distinct scenarios, varying selectivity, on the single node and cluster-based configuration, respectively. Coloured bars show the execution time in scenarios with given intended purposes selectivity, whereas a black horizontal line indicates the query execution time in the case where no enforcement mechanism is enabled. Hereafter, scenarios with access control enforcement are denoted with the notation *sel-x*, where x specifies the intended purpose selectivity (e.g., the scenario with selectivity 0.2 is denoted sel-0.2), whereas the scenario where no enforcement mechanisms is activated is denoted with *no-enf*.

By comparing the results for the two configurations, it appears that, although with different absolute measures, the execution time trend is similar for all the queries in both configurations, and in the cluster-based configuration the execution is generally quicker. An exception is query q2, characterized by the quickest execution time among the considered queries (only few documents are accessed). In this case, inter nodes communication latency, which for other queries is negligible, causes a significant (in percentage) increase of the execution time. As a consequence, the cluster-based executions of q2 are longer then those on single node.

Considering the results achieved within sel-1.0 for both the configurations, the execution time of all the queries appears the same. This because all the enhanced queries need to exhaustively scan the collection looking for documents whose intended purposes comply with the target access purpose. Since with selectivity 1.0 no document in *messages* satisfies the compliance, the computation of all queries terminates immediately after the purpose compliance check, thus blocking any other processing activity. The measured times show that when the selectivity decreases, the execution time usually increases. Indeed, the selectivity alters the number of documents that are processed. The effect of purpose compliance checks is amplified by the selection criteria specified for each query. Therefore, in the remainder of this section, we analyze the enforcement overhead focusing on the characteristics of the queries. Since, as previously mentioned, with selectivity 1.0 the execution time is the same for all queries, for our analysis we consider scenarios with selectivity varying from 0.8 to 0.

Queries q1 and q9 simply counts the documents that satisfy given selection criteria. Although using different

operations (q1 is a count query whereas q9 is a group query) q1 and q9 show similar results. Within the scenario *sel-0.8-sel-0* selection criteria are enhanced with purpose compliance checks, which introduce some overhead wrt *no-enf*. Since the counting is executed on document sets of similar size, and the query result set only includes the counting information, the execution time is similar in all the scenarios.

Query q2 selects a single document of *messages*, which satisfies given selection criteria. In this case, the enforcement overhead is minimum and similar in each scenario as the query has to access a very small number of documents.

Query q3 is a selection query to which a count query is enqueued. In MongoDB selection queries are characterized by a stepwise execution. Clients access query results set by means of partial views of customizable size. If the data composing the view are not sufficient, the client has to issue a request to the server which continues the execution and provides a new chunk. A possible way to estimate the overall execution time is calculating the sum of the time used for serving each client request. However, due to the limited size of a client view (the default size includes 20 items), thousands requests are required for each query (*messages* includes around 550 K documents), each of which requires few milliseconds. A good percentage of the overall measured time would refer to message transmission operations. However, possible network latencies, multiplied by thousands transmissions, would compromise the accuracy of the measuring. Therefore, we have decided to enqueue a count query to the selection query. In this way, the counting is executed only after the whole result set is derived. Thus, a single client request is sufficient for the execution of the whole query, reducing any possible undeterministic delay due to client-server communications. This allows us to derive a more precise measure of the time used by the server to perform the selection. Our results show that, except for small variations, the execution time is similar in all the scenarios, with an average enforcement overhead of 793 ms for the single node and 166 ms for the cluster base configuration. This overhead is not negligible (+88,0 percent and +93,4 percent, respectively) but, at the same time, does not compromise the usability. Similar behavior have also been observed for query q10, which performs a counting using a group operation. However, in this case the overhead is significantly smaller.

Queries q4 and q5 perform distinct selections and return an array of documents. The execution time depends on the number of documents that are effectively transmitted. As for each enforcement scenario, all the documents are preventively checked for purpose compliance. For all scenarios, q5 execution is longer than q4, since the receiver field of a message can include multiple elements and therefore an higher computational effort is required for the selection of distinct elements. However, due to policy selectivity, the overhead of q5 is lower than q4. Indeed, the processing is applied only for selected elements.

Queries q6, q7, q8 and q11 use the aggregation framework, whereas q12 the MapReduce framework. Different operation pipelines and MapReduce operations are defined for the considered queries. However, most of these queries share as common point the computational complexity of the performed operations. The execution time depends on the number of documents that are effectively processed, decreasing with the selectivity increase. The query execution with access control enforcement is significantly lower than the execution time of the same queries with no enforcement.

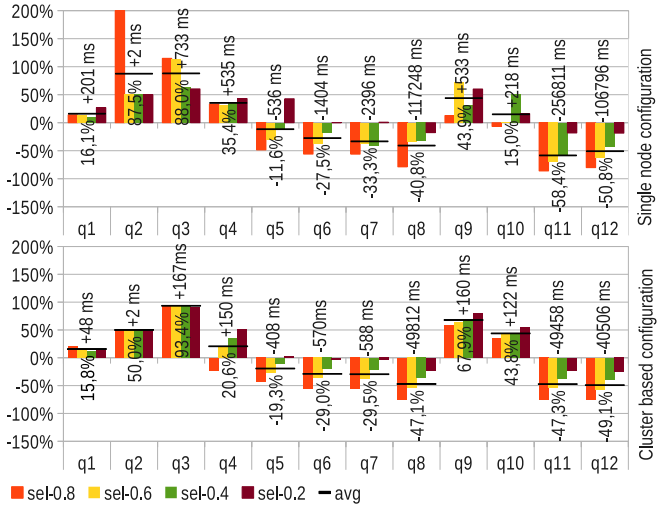


Fig. 9. Enforcement overhead—in percentage.

Fig. 9 shows the enforcement overhead of queries in each configuration when varying the policy selectivity, and the average overhead of each query. The considered selectivity range (i.e., [0.2..0.8]) takes into account policy with a non-null but non-exhaustive filtering effect, which are reasonably expected to represent the typical use case.

Although the percentage overhead for queries with short execution time (i.e., q2, q3, q9, q10) is not negligible, the absolute delay value is always contained, and in no case compromises usability. Conversely, enforcement mechanisms even allow to significantly cut the execution time of complex queries (e.g., q8, q11, q12).

7 Discussion

The analysis illustrated in Section 5.3 shows the correctness of purpose-based access control enforcement performed by Mem, which represents the key functional requirement specified for Mem development.

Mem also fulfils all non functional requirements discussed in Section 3. The choice of implementing Mem as Wire protocol interpreter makes this enforcement monitor completely independent from the programming languages and APIs of the MongoDB clients and allows using it with MongoDB clients operating with any MongoDB driver. Mem can even be used within MongoDB deployments where multiple clients, each operating with different drivers, interact with the same server.

Mem operates independently from the intrinsic characteristics of the MongoDB clients with which it communicates. Thus, it is expected to be reliable enough to support the interaction with clients operating with new MongoDB drivers. Indeed, MongoDB drivers convert client requests expressed in any proprietary format to Wire messages. In turn, Wire messages encode server responses into messages with a format that is interpretable by the clients. Mem only operates with Wire messages flowing through MongoDB client-server connections. Therefore, since the development

of new drivers does not affect the definition of MongoDB Wire protocol, Mem is expected to be robust enough to support the interaction with MongoDB clients operating with new drivers. Moreover, the experimental results illustrated in Section 6.2 show Mem efficiency in a variety of different scenarios. In no case Mem has compromised query usability, and the measured enforcement has always been contained.

Finally, the choice of developing Mem as a proxy makes it easy integrable into any MongoDB deployment. No programming activity is required to the administrators. In addition, we believe that the configuration activities presented in Section 5.1 have a reasonably low complexity. The integration can be achieved in scenarios where MongoDB is deployed from scratch, as well as in scenarios where MongoDB datastores are already deployed, using the same configuration process.

The main ideas we have presented in this paper of the approach to the integration of purpose-based access control into MongoDB can also be applied to other document-oriented NoSQL datastores. The identified requirements are achievable through the development of enforcement monitors which, similar to Mem, act as proxies, rely on the native access control models of the host platforms, and perform query rewriting. As far as policy specification is concerned, the same guidelines proposed for Mem can be applied to other NoSQL datastores as well. Policies should be specified with the query language of the considered platforms, encoding the intended purposes in a proper format, and they should be bound to documents by means of dedicated fields integrated into the documents. As a matter of fact, although differences can exist, which are related to the supported languages and data types that can be used for policy specification, all document-oriented NoSQL datastores support the concepts of document and field.

A more challenging aspect is related to the definition of enforcement mechanisms. Although the final goal is to rewrite queries in such a way to integrate purpose compliance checks, the implementation strategies of these mechanisms depend on the characteristics of the considered platforms. More precisely, we believe that the main differences among the various document-oriented platforms that may affect the implementation of the monitors are related to: 1) the data model, 2) the access control model, 3) the supported query types, and 4) the query language. For instance, if we consider CouchDB (http://couchdb.apache.org/) as an alternative platform, differences exist for all the above mentioned points. Indeed, different from MongoDB, documents are directly stored into databases, and the access to the whole database content is granted to users registered for such database. Queries are defined with Curl and the client-server communication is achieved via HTTP. Essentially, the enforcement monitor should be a HTTP proxy that rewrites queries integrating purpose compliance checks. Although the mechanisms are similar to those proposed for Mem, the different languages, data model and access control model require a definition compatible with these elements. As a consequence, we believe that the enhancement of the access control features of other document oriented datastores is achievable by means of similar mechanisms, but the intrinsic characteristics of each platform require dedicated implementation choices.

8 Related Work

Companies and organizations that store large amounts of data are increasingly using NoSQL datastores [17], as these systems ensure high levels of scalability, flexibility, and performance [4]. However, NoSQL datastores suffer from several privacy and security vulnerabilities (see e.g., [19]).

As far as access control is concerned, the main weakness is that fine grained authorizations are not supported. Kulkarni [16] proposes an access control model for key value systems, which allows the specification of policies at the level of keyspace, column family, rows and columns. Two implementations have been defined. In the former, access control is integrated modifying the source code of Cassandra, whereas in the latter, access control is enforced by an external software module that interacts with the datastore. Vormetric Data Security Platform has been recently released,[15] which provides encrypted data storage and cryptographically enforced access control within MongoDB systems. Different from these proposals, Mem targets the integration of purpose-based access control into MongoDB.

Shermin [22] proposes a context aware RBAC model for NoSQL databases. Although enforcement mechanisms for the proposed model have been defined, it is not analyzed how the model can be implemented and integrated into a target NosQL platform.

Fine grained access control (FGAC) for MapReduce systems has been recently considered. In some cases, Hadoop-based datastores have been specifically designed to integrate FGAC. For instance, Accumulo[16] is a key value data store based on BigTable [6], which implements cell based access control. Sentry[17] is a system allowing the enforcement of role based fine grained access control within an Hive system deployed on top of a Hadoop cluster. Finally, Vigiles [24] and GuardMR [23] target the direct integration of FGAC into Hadoop, regulating the data records that are processed by the submitted MapReduce jobs. FGAC for MapReduce systems is enforced with techniques alternatives to those used for Mem, as, different from queries, the rewriting of MapReduce jobs can hardly be automatically achieved. For instance, in [23], [24] bytecode weaving is used for filtering the data records.

Recent work focuses on additional weaknesses of NoSQL datastores such as confidentiality of data storing (e.g., [12]), but we are not aware of other proposals for the effective integration of privacy aware access control enforcement into NoSQL systems.

Privacy awareness of systems has been thoroughly investigated in the recent period in the context of traditional (i.e., non NoSQL) data management systems. For instance, Privacy by design (PbD) [5] basically consists of developing software applications according to privacy aware principles. Agrawal et al. [1] have been the first to discuss the development of "Hippocratic" DBMSs, namely relational DBMSs where privacy policies, regulate the execution of SQL queries. Following the principles introduced in [1], several proposals, such as [3], [14], [13], [18], [20], [7] and [8]

15. http://www.vormetric.com/data-security-solutions/applications/big-data-security
16. https://accumulo.apache.org/
17. https://sentry.incubator.apache.org/

integrate purpose-based access control into relational DBMSs. More precisely, Byun and Li [3] present a purpose and role based access control model for relational DBMSs, where access control enforcement is achieved by means of SQL query rewriting. Kabir et al. extend [3] by first proposing a model variant, denoted conditional purpose-based access control (CPBAC), which introduces the concept of conditional purpose, and then by extending CPBAC with role management [13]. Ni et al. [18] and Peng et al. [20] propose and extension of RBAC with purpose-based access control, the former [18] with obligation management, the latter with mechanisms for the dynamic association of access purposes to user queries based on system and user attributes. In [7] we presented a framework that automatically generates enforcement monitors for purpose and role based privacy policies, and integrates them into relational DBMSs. In [8], we extend the framework in [7] to the support of obligations. Mem shares some intents with these proposals, but differently from them, it targets a NoSQL datastore, with related data model and query language, and enforces purpose-based access control by injecting commands and rewriting queries at communication protocol level.

9 Conclusions

This paper presented an effective solution to the integration of fine-grained purpose-based access control into MongoDB. The RBAC model of MongoDB has been extended with purpose concepts and related enforcement mechanisms to regulate the access at document level on the basis of purpose and role based policies. An enforcement monitor, called Mem, has been designed to implement the proposed enhanced model. Mem operates as a proxy between MongoDB clients and a MongoDB server, and enforces access control by monitoring and possibly manipulating the flow of exchanged messages. Our experiments have shown low enforcement overhead.

Our work is still progressing for refining the access control granularity from the document to the field level. Furthermore, we plan to generalize the presented approach to the support for multiple NoSQL datastores. Our long term goal is the definition of a framework for the integration of purpose-based access control into several NoSQL platforms.

References

[1] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu, "Hippocratic databases," in *Proc. 28th Int. Conf. Very Large Data Bases*, 2002, pp. 143–154.

[2] K. Browder and M. A. Davidson, "The virtual private database in oracle 9i R2," Oracle Tech. White Paper, Jan. 2002, http://www.cgisecurity.com/database/oracle/pdf/VPD9ir2twp.pdf

[3] J. Byun and N. Li, "Purpose based access control for privacy protection in relational database systems," *Int. Very Large Data Bases*, vol. 17, no. 4, pp. 603–619, 2008.

[4] R. Cattell, "Scalable SQL and NoSQL data stores," *SIGMOD Rec.*, vol. 39, no. 4, pp. 12–27, May 2011.

[5] A. Cavoukian, "Privacy by design: Leadership, methods, and results," in *European Data Protection: Coming of Age*, S. Gutwirth, R. Leenes, P. de Hert, and Y. Poullet, Eds. New York, NY, USA: Springer, 2013.

[6] 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," *ACM Trans. Comput. Syst.*, vol. 26, no. 2, p. 4, 2008.

[7] P. Colombo and E. Ferrari, "Enforcement of purpose based access control within relational database management systems," *IEEE Trans. Knowl. Data Eng.*, vol. 26, no. 11, pp. 2703–2716, Nov. 2014.

[8] P. Colombo and E. Ferrari, "Enforcing obligations within relational database management systems," *IEEE Trans. Dependable Secure Comput.*, vol. 11, no. 4, pp. 318–331, Jul./Aug. 2014.

[9] P. Colombo and E. Ferrari, "Efficient enforcement of action-aware purpose-based access control within relational database management systems," *IEEE Trans. Knowl. Data Eng.*, vol. 27, no. 8, pp. 2134–2147, Aug. 2015.

[10] P. Colombo and E. Ferrari, "Privacy aware access control for big data: A research roadmap," *Big Data Res.*, Sep. 2015, [Online]. Available: http://dx.doi.org/10.1016/j.bdr.2015.08.001. In press.

[11] D. Ferraiolo, R. Sandhu, S. Gavrila, D. Kuhn, and R. Chandramouli, "Proposed NIST standard for role-based access control," *ACM Trans. Inf. Syst. Security*, vol. 4, no. 3, pp. 224–274, 2001.

[12] Y. Guo, L. Zhang, F. Lin, and X. Li, "A solution for privacy-preserving data manipulation and query on NoSQL database," *J. Comput.*, vol. 8, no. 6, pp. 1427–1432, 2013.

[13] M. Kabir, H. Wang, and E. Bertino, "A role-involved conditional purpose-based access control model," in *E-Government, E-Services and Global Processes*,vol. 334, *IFIP Advances in Information and Communication Technology*, M. Janssen, W. Lamersdorf, J. Pries-Heje, and M. Rosemann, Eds.Berlin, Germany: Springer, 2010, pp. 167–180.

[14] M. E. Kabir and H. Wang, "Conditional purpose based access control model for privacy protection," in *Proc. 20th Australasian Conf. Australasian Database*, 2009, pp. 135–142.

[15] B. Klimt and Y. Yang, "The Enron corpus: A new dataset for email classification research," in *Proc. Eur. Conf. Mach. Learn.*, 2004, pp. 217–226.

[16] D. Kulkarni, "A fine-grained access control model for key-value systems," in *Proc. 3rd ACM Conf. Data Appl. Security Privacy*, 2013, pp. 161–164.

[17] N. Leavitt, "Will NoSQL databases live up to their promise?" *Computer*, vol. 43, no. 2, pp. 12–14, Feb. 2010.

[18] Q. Ni, E. Bertino, J. Lobo, C. Brodie, C. Karat, J. Karat, and A. Trombetta, "Privacy-aware role-based access control," *ACM Trans. Inf. Syst. Security*, vol. 13, no. 3, p. 24, 2010.

[19] L. Okman, N. Gal-Oz, Y. Gonen, E. Gudes, and J. Abramov, "Security issues in NoSQL databases," in *Proc. IEEE 10th Int. Conf. Trust, Security Privacy Comput. Commun.*, 2011, pp. 541–547.

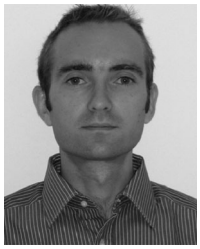
[20] H. Peng, J. Gu, and X. Ye, "Dynamic purpose-based access control," in *Proc. Int. Symp. Parallel Distrib. Process. Appl.* 2008, pp. 695–700.

[21] M. A. Russell, *Mining the Social Web: Data Mining Facebook, Twitter, LinkedIn, Google+, GitHub, and More*. Sebastopol, CA, USA: O'Reilly Media Inc., 2013.

[22] M. Shermin. (2013). An access control model for NoSQL databases. Master's thesis, Univ. Western Ontario, London, ON, Canada [Online]. Available: http://ir.lib.uwo.ca/etd/1797

[23] H. Ulusoy, P. Colombo, E. Ferrari, M. Kantarcioglu, and E. Pattuk, "GuardMR: Fine-grained security policy enforcement for MapReduce systems," in*Proc. 10th ACM Symp. Inf., Comput. Commun. Security*, 2015, pp. 285–296.

[24] H. Ulusoy, M. Kantarcioglu, K. Hamlen, and E. Pattuk, "Vigiles: Fine-grained access control for MapReduce systems," in *Proc. IEEE Int. Congress Big Data*, 2014, pp. 40–47.



Pietro Colombo received the PhD degree in computer science in 2009 from the University of Insubria, Italy. He currently works as research associate within the STRICT SociaLab investigating the definition of privacy-aware data management systems. His research interests are mainly related to data privacy and model driven engineering.

Elena Ferrari is a full professor of computer science at the University of Insubria, Italy, and scientific director of the K&SM Research Center. Her research activities are related to access control, privacy, and trust. In 2009, she received the IEEE Computer Society's Technical Achievement Award for "outstanding and innovative contributions to secure data management". She received a Google Award in 2010, and an IBM Faculty Award in 2014. She is a fellow of the IEEE and an ACM Distinguished Scientist.