



HHS Public Access

Author manuscript

IEEE Trans Emerg Top Comput. Author manuscript; available in PMC 2024 January 01.

Published in final edited form as:

IEEE Trans Emerg Top Comput. 2023 ; 11(1): 208–223. doi:10.1109/tetc.2022.3193577.

Enabling Attribute-based Access Control in NoSQL Databases

Eeshan Gupta,

Shamik Sural

Department of Computer Science and Engineering, Indian Institute of Technology, Kharagpur, India.

Jaideep Vaidya,

Vijayalakshmi Atluri

Rutgers University, Newark, USA.

Abstract

NoSQL databases are being increasingly used for efficient management of high volumes of unstructured data in applications like information retrieval, natural language processing, social computing, etc. However, unlike traditional databases, data protection measures such as access control for these databases are still in their infancy, which could lead to significant vulnerabilities and security/privacy issues as their adoption increases. Attribute-based Access Control (ABAC), which provides a flexible and dynamic solution to access control, can be effective for mediating accesses in typical usage scenarios for NoSQL databases. In this paper, we propose a novel methodology for enabling ABAC in NoSQL databases. Specifically we consider MongoDB, which is one of the most popular NoSQL databases in use today. We present an approach to both specify ABAC access control policies and to enforce them when an actual access request has been made. MongoDB Wire Protocol is used for extracting and processing appropriate information from the requests. We also present a method for supporting dynamic access decisions using environmental attributes and handling of ad-hoc access requests through digitally signed user attributes. Results from an extensive set of experiments on the Enron corpus as well as on synthetically generated data demonstrate the scalability of our approach. Finally, we provide details of our implementation on MongoDB and share a Github repository so that any organization can download and deploy the same for enabling ABAC in their own MongoDB installations.

Index Terms—

Attribute-based Access Control; NoSQL datastores; MongoDB

1 Introduction

NoSQL databases are non-relational database management systems that support a custom and flexible schema based on the user requirements, and are known to scale well horizontally [1]. Since the data model can be tailored to suit specific use cases, they often

provide superior performance and are considered more developer friendly. Hence, NoSQL databases are becoming a preferred choice for handling organizational data, especially unstructured data [2]. However, these databases still lack some of the data protection and privacy measures, such as access control, that are present in traditional relational databases [3]. With the recent extensive proliferation of NoSQL databases, developing support for such measures is of extreme importance. In a database system, access control mechanisms determine the level of access each user can have on the various data elements. Most of the NoSQL databases, including the more popular ones like MongoDB, only allow basic methods for authorization management like Role-based Access Control (RBAC) over a coarse granularity [4].

While Role-based Access Control [5] is one of the most popular access control mechanisms currently in use, it does not naturally permit fine-grained access control or taking dynamic access decisions based on environmental conditions, thus making it unsuitable for emerging computing applications. Instead, Attribute-based Access Control (ABAC) [6] [7] has been widely recognized as the next-generation access control model, and is rapidly gaining in popularity. ABAC systems can control access to users dynamically using the concept of attributes, which are name-value pairs describing the states of the users, objects and the environment. By this means, ABAC allows modeling of complex policies and is quite flexible. It can even provide access to external ad-hoc users by verifying their attributes instead of identities.

Currently, NoSQL databases do not have native support for ABAC although they could really benefit from the integration. There has been some prior work considering access control in NoSQL databases and proposing different enhancements (A detailed survey is presented in Section 6). In a recent work [33], we introduced an approach that enhances MongoDB's RBAC system with ABAC using a role mining algorithm which converts ABAC policies into RBAC roles. However, it is limited in its functionality such as the handling of environmental attributes and in providing dynamic access to ad-hoc users. Besides, it does not natively support ABAC other than acting as a means for policy specification using attributes and enforcement through roles.

In this paper, we tackle the problem of providing support for fine grained ABAC into NoSQL holistically, allowing administrators to restrict permissions on individual fields within a document. We propose a novel approach based on an intercepting proxy server, which uses the MongoDB Wire Protocol [8], and communicates with the database server on behalf of the clients. Attributes and policies are modeled using JSON for flexibility and ease of use. Using message rewriting at the proxy layer, we enforce the ABAC policies and also apply it at a fine level of granularity. The ABAC proxy can determine whether to provide access to external ad-hoc users as well by relying on the concept of Central Attribute Authorities [9] to authenticate the attributes presented by such users. Further, the time and location of a request are considered as factors in determining whether the access should be granted or not. The time attribute is used to control access on the basis of the current time, given certain validity intervals and is based on the notion of periodic events as previously defined for Temporal RBAC (TRBAC) [10]. On the other hand, the location attribute is used to control access on the basis of IP address of the origin of the request.

Thus, our approach has the functionality of dynamically controlling access to external users by verifying their attributes on the fly and integrating support for environmental attributes like time and location of access.

We demonstrate a MongoDB implementation with multiple use cases and measure its performance over both real and synthetic data. Comparison is made with a baseline MongoDB server. Detailed experimental evaluation shows that our approach imposes minimal overhead even if the number of attributes and rules in the ABAC policy increases considerably. Furthermore, all MongoDB queries are fully supported since any interaction between the server and the clients takes place through the Wire Protocol as defined in the official MongoDB documentation [13]. While the implementation details are specific to MongoDB, our approach is quite generic in nature and can be extended to other NoSQL databases besides MongoDB.

The rest of the paper is organized as follows. Section 2 provides a brief overview of the basic concepts used in the rest of the work. Section 3 describes the proposed approach in detail including the system architecture, attribute handling, environmental condition evaluation, policy description and the algorithm for making ABAC access decisions. It also discusses extending the approach to other NoSQL databases. Section 4 presents the implementation details and demonstrates several user flows through the ABAC layer. In Section 5, we study the performance of the proposed approach using an extensive set of real and synthetically generated data. Section 6 surveys existing literature in this domain and finally, Section 7 concludes the paper along with future directions for research.

2 Preliminaries

We first present the various types of NoSQL databases and the kinds of access control currently supported. We then discuss MongoDB and the Wire protocol, which is extensively used in our work. Finally, we provide an overview of the Binary JSON format.

2.1 NoSQL Databases

NoSQL databases are often quite different from each other, and are primarily categorized [12] on the basis of the data model supported, as follows:

Document Databases: Data stored in JSON-like documents with nested fields and values. Example: MongoDB

Key-Value Datastores: Highly efficient and much simpler databases with each item containing only keys and values. Example: DynamoDB, redis

Wide-column Stores: Data stored in tables with dynamic columns instead of a specified schema like in SQL. Example: Cassandra, HBase

Graph Databases: Data stored in nodes and edges. Example: Neo4j

These databases are specialized for specific use-cases and hence provide much better performance than traditional relational databases for those use cases. They are also designed

to be highly scalable and always available, often by aiming towards weaker levels of consistency guarantees.

2.2 Access Control in NoSQL Databases

Access control mechanisms allow an organization to restrict access to their data based on the characteristics of users and objects. Most NoSQL databases have some form of access control built into them. In this sub-section, we discuss the form of access control available in the more commonly used NoSQL databases.

MongoDB, the most popular NoSQL database according to the DB-Engines ranking [2], supports Role-based Access Control (RBAC) at the coarse granularity of MongoDB Collections, each of which is a group of documents within the database. There is a set of pre-defined roles and the administrators can define custom roles as well using the `db.createRole` command [13]. Cassandra also supports RBAC, with the ability to create roles using the Cassandra Query Language (CQL). The permissions can be granted or denied at the granularity of a table within a keyspace, with no support for granting access to specific columns or rows [14]. Amazon's DynamoDB supports access control based on the AWS Identity and Access Management (IAM) service [15], which allows IAM administrators to assign certain permissions to IAM users and user groups. It also has a fine-grained access control system, with an option to allow access only to certain items and attributes within a table. Redis supports access control, with the ability to restrict access to specific commands and keys [16]. However, open-source Redis does not support roles and hence, permissions need to be granted for each user individually. Redis Enterprise Cloud service, on the other hand, supports RBAC as well. Neo4j is a graph-based database that supports RBAC, with the ability to assign permissions to specific nodes and relationships in the graph as well. [17].

2.3 Attribute-Based Access Control

Attribute-based Access Control (ABAC) is an authorization model that allows granting permissions to users for performing various operations on objects based on the user and object attributes as well as environmental conditions [7]. These attributes together capture the state of the system at any point of time. Thus, ABAC is a dynamic model that can accommodate complex access control requirements in an organization. It supports a significant amount of flexibility even under extremely complex settings by leveraging consistently defined attributes. Many of the other access control models such as RBAC can be considered to be special cases of ABAC with appropriately defined attribute sets. Formally, an ABAC system is comprised of the following components:

- *Users (U)*: Represents a set of authorized users/subjects. u_i for $1 \leq i \leq |U|$ denotes each member of this set.
- *Objects (O)*: Represents a set of resources to be protected. Each member of this set is denoted as o_i for $1 \leq i \leq |O|$.
- *Environment (E)*: Represents a set of environment, independent of users and objects. Each member of this set is denoted as e_i for $1 \leq i \leq |E|$.

- *UA*: Represents a set of user attribute names. Members of this set are represented as ua_i , for $1 \leq i \in |UA|$. Each ua_i is associated with a set of possible values it can acquire.
- *OA*: Represents a set of object attribute names. Members of these sets are represented as oa_i , for $1 \leq i \in |OA|$. Each oa_i is associated with a set of possible values it can acquire.
- *EA*: Represents a set of environment attribute names. Members of these sets are represented as ea_i , for $1 \leq i \in |EA|$. Each ea_i is associated with a set of possible values it can acquire.
- *OP*: A set consisting of all possible permissions / operations on objects allowed in a system. Each member of *OP* is represented as op_i , for $1 \leq i \in |OP|$.
- U_C : Represents a set of all possible user attribute conditions. Each $uc \in U_C$ is an equality of the form $ua = c$, where $ua \in UA$ and c is a constant.
- O_C : Represents a set of all possible object attribute conditions. Each $oc \in O_C$ is an equality of the form $oa = c$, where $oa \in OA$ and c is a constant.
- E_C : Represents a set of all possible environment attribute conditions. Each $ec \in E_C$ is an equality of the form $ea = c$, where $ea \in EA$ and c is a constant.
- A user attribute relation, an object attribute relation and an environment attribute relation can now be defined to build a many to many mapping of users, objects and environment to user attribute conditions, object attribute conditions and environment attribute conditions, respectively.
- Π : Represents a set of access rules called the Policy of the ABAC system. Each rule in the ABAC policy, denoted as r_i , for $1 \leq i \in |\Pi|$, is comprised of a set of attribute conditions uc , oc and ec , and an operation op .

In the rest of the paper, we use these notations while referring to the ABAC components.

2.4 MongoDB and the Wire Protocol

MongoDB is a document-based NoSQL database in which data is stored in the form of BSON (binary-JSON) documents [18] through nested fields and values. Developers can map their classes into the format of the MongoDB document, enabling faster development and efficient programming. It is also scalable with high availability, making MongoDB arguably the most popular NoSQL database [2].

The interaction between a client and a server in MongoDB databases is based on the MongoDB Wire Protocol [8]. The queries and commands defined in any programming language or even the MongoDB shell are converted by MongoDB drivers to Wire Messages. The responses to the client requests are also issued as corresponding Wire Messages. The Wire Protocol uses the TCP protocol through sockets. The protocol for communication uses nine different types of pre-defined messages that are classified into server responses and client requests.

Each message has a standard header, followed by the request-specific body. The header contains the following:

1. `messageLength`: length of the message
2. `requestID`: to uniquely identify the message
3. `responseTo`: to associate query responses with the originator
4. `opCode`: the `opCodes` supported are -
 - `OP_REPLY` - reply to an `OP_QUERY` or `OP_GET_MORE` request by the client. The `cursorID` and a list of documents are returned with this message.
 - `OP_UPDATE` - apply updates to a document. The collection name, query to select which document, specifications of the update required to be performed and some additional flags are set by these messages
 - `OP_INSERT` - insert a single or multiple documents. The collection name and a list of documents are set by these messages.
 - `RESERVED` - for backward compatibility.
 - `OP_QUERY` - query a database for documents. The collection name, number of matching documents and the query to select which documents to be sent are set by these messages.
 - `OP_GET_MORE` - similar to an `OP_QUERY` message, but it establishes a cursor at the server if there are more documents than the limit.
 - `OP_DELETE` - delete documents from a collection. The collection name and the query to select which documents are to be deleted can be set by these messages.
 - `OP_KILL_CURSORS` - notify the server that a client is done with the cursor.
 - `OP_MSG` - send a message according to the specification defined in MongoDB 3.6 which absorbs the functionality provided by the other `opCodes`.

2.5 Binary JSON (BSON)

BSON [18] is binary in structure instead of a text-form used in JSON, hence allowing it to be parsed more quickly since it encodes the length and information about type. In addition to all that is supported by JSON, BSON also has support for types such as dates and binary data, which is essential for providing many MongoDB features.

Using BSON as the data storage format allows MongoDB to allow the most powerful form of querying and indexing available in the industry, in addition to it being one of the most usable formats in the web. Some of these features include the ability to efficiently query, manipulate and index objects even if the documents are nested many layers deep.

3 Supporting ABAC in MongoDB

To have a native implementation of ABAC in NoSQL databases such as MongoDB or Cassandra, it would be necessary to intercept each message sent to the database using a proxy. We first present the system architecture and discuss the configuration of the intercepting proxy server including the specifics of the connection and authentication mechanisms. We then describe how the user attributes are obtained for both internal as well as external users for any incoming request, as well as how object attributes are obtained. Next we elaborate on the two environmental attributes (time and location of access) supported natively by our model. Finally we describe both the ABAC policy specification steps as well as the algorithm used for making access decisions.

3.1 System Architecture

Attribute-Based Access Control is implemented natively into a database server using a proxy, which is considered to be a trusted extension of the database server. Hence, any access to MongoDB documents by the proxy server for determining a user's authorized access is not treated as a violation of access control. Access to authorized documents by the client is only accomplished after a detailed authentication procedure. Note that this is different from the access control achieved by databases such as Oracle Virtual Private Databases (VPDs)¹. Unlike VPDs, here access control is not attached to the database objects since NoSQL databases do not have fixed table schemas. Instead, it is accomplished by a proxy server, which is implemented as a trusted extension.

The proxy intercepts all the messages intended for the database server and the full ABAC functionality is built into this proxy. It is able to parse these messages using the TCP protocol used by the database. The default access control mechanism of the database is disabled preventing all direct requests from reaching the database server while granting root access to the proxy server. By intercepting each message, we are able to build a custom authorization layer either before or after the calls to the actual database are made, thereby implementing native ABAC. The following section discusses the methodology with a special focus on MongoDB.

3.1.1 Proxy Server—A proxy resides before a server and acts as a relay between the server and its clients. The proxy can function transparently by directly forwarding messages both ways without modifying either the request or the response. These are often used for monitoring, duplication, authentication, load balancing and validation. On the other hand, it can also function as an intercepting proxy, modifying the request or the response to provide or constrain some functions. Such types of proxies are extremely powerful tools, and often used for extending protocols, encryption and transforming data without modifying the server.

For working with multiple clients, the proxy server typically needs to be multi-threaded, which can be accomplished using threads or by leveraging an event-driven architecture with the reactor pattern. The reactor design pattern handles service requests delivered

¹. https://docs.oracle.com/cd/B28359_01/network.111/b28531/vpd.htm

simultaneously to a server by multiple clients. The idea is to de-multiplex the requests, and dispatch them to the server's handlers in a synchronous manner. Such an approach provides the ability for the developers to completely separate the application logic from the implementation and also enhance modularity. Thus, these proxy designs provide good stability and scalability even under highly demanding workloads.

3.1.2 Alternative Architecture Designs—A detailed analysis of the working principle of MongoDB revealed that there could be two possible ways in which the authorization layer can be architected. The first approach is to enforce access control before forwarding a request message from the client to the MongoDB server. The other alternative is to first forward a request directly to the server, receive its response and then apply the policies for deciding whether to forward this response back to the client. While modification (insert, update, delete) requests can only be handled using the first approach, queries can be addressed using either. There are relative merits and demerits of the two techniques as described below.

Policy Enforcement before Forwarding Request: In this use case, the proxy server intercepts every request of the client. From the corresponding MongoDB wire message, it extracts user details, type of request, name of the database, name of the collection as well as other data as described in the previous section. Based on these and the corresponding retrieved attributes, an appropriate access decision is made (described in Section 3.6). If access is granted by the ABAC policy, the wire message gets forwarded to the MongoDB server, and the corresponding reply is sent back to the client. However, if access is denied, the *UnauthorizedError* message is propagated back. The detailed steps are shown in the sequence diagram corresponding to this use case in Figure 1.

This approach has the advantage that only valid requests reach the MongoDB server, thus conserving bandwidth and resources that would otherwise might have been wasted there. Hence, it is imperative that update requests follow this protocol for preserving data integrity. However, for retrieval queries, such an approach limits the granularity at which access control can be applied. Since there is no way of knowing the fields in a collection before retrieval from NoSQL databases, the finest possible granularity of access control achievable is that of a collection.

Policy Enforcement after Receiving Reply: In this case, the proxy server intercepts a request from the client, and immediately forwards it to the MongoDB server as is. The server sends a reply back to the proxy server after fetching the appropriate documents. The proxy server then applies the ABAC rules on the attributes received and makes a decision.

Since the documents have already been retrieved at the time of enforcing access control, a finer granularity over the documents can be applied. Now it is possible to achieve granularity as fine as specifying the fields within a document of a collection. This makes the access control system inherently more powerful. However, it can be further improved by employing a binding strategy where policy application is subject to conditions that consider field values dynamically to allow regulating access to a specific document within a collection similar to that accomplished by Oracle VPD. This is an ongoing topic of research (see [32]). The

detailed steps for policy enforcement after receiving the reply are shown in the sequence diagram in Figure 2.

Considering the advantages and shortcomings of the two approaches, for retrieval queries, it makes most sense to combine the two and apply a part of access control first before forwarding the request message to the MongoDB server and then after receiving the reply message from it. All update requests such as insert, update and remove must have access control applied before forwarding the request, which is essential for correctness. However, for queries, it is possible to follow a combined approach. An initial coarse grain decision is taken at the proxy server before forwarding the request to the MongoDB server, thus saving on network and server resource usages. This would only take into consideration the attributes related to the document, and not the underlying fields. After receiving the documents as a reply message from the MongoDB server, access control is applied at the granularity of fields. Such an approach balances both performance and effectiveness of the access control system. Note that the entire MongoDB database is not duplicated in the proxy server. It only maintains a list of the attributes of the MongoDB objects in order to reach access decisions.

Connection and Authentication: A connection request from a MongoDB client consists of a request identifier, a user identifier and an encoded password string. This request is intercepted by the proxy server and forwarded to the MongoDB server, where the user is authenticated and the corresponding reply message is sent back to the proxy. The authentication mechanism is identical for both architectures, since it is essential for the user to be authenticated before sending any further requests to the MongoDB server. Once authorized, the message is parsed to obtain the user credentials and a new session is created for the user in the proxy server. Next, the user attributes are retrieved. A session is defined as the duration for which the user is authenticated on the server and can execute commands and requests with appropriate access control.

```
[
  {
    "Alice": {
      "position": "Manager",
      "region": "India",
      "team": "Ads"
    }
  },
  {
    "Bob": {
      "position": "Developer",
      "region": "USA",
      "team": "Backend",
      "wfh": "yes"
    }
  }
]
```

Listing 1.

Sample user attribute JSON stored locally at the proxy server

A connection request can either be from the `mongo` shell or through a connection string using a variety of language APIs described by MongoDB, which includes drivers for C++, Java, Python, Ruby, Node.JS and several others. All of these requests are converted by MongoDB drivers and transmitted in the form of MongoDB Wire Messages, which are intercepted by the proxy server, making the connection to the MongoDB server on behalf of the client and creating a session in memory.

3.2 Obtaining User Attributes (UA)

Once a session is created with the user identifier, the next step is to obtain the attributes of the user and apply appropriate ABAC authorization rules. This process should be as seamless as possible, handling not only internal registered users but also external users based on verified attributes without any manual intervention either to register them or for entering their attributes. Based on the type of the user, there are two distinct sequence of steps needed for fetching their attributes, as described below.

3.2.1 Attributes of Internal Users—This is the default scenario which is triggered when a user's attributes are already available locally at the proxy server either because these were previously entered by the administrator or because the system remembers them (i.e., has them cached) from an earlier visit by the same user. In the system, the user attributes are stored in the form of a JSON file, which consists of a JSON array, with each array entry corresponding to a particular user's attributes. A sample user attribute JSON for two users is shown in Listing 1. In this example, Alice has values for three attributes while Bob has four. Note that, each entry of the array has the user identifier as the key and the value as the nested JSON of their user attributes.

3.2.2 User Attributes Certified by a Central Attribute Authority—The concept of Central Attribute Authority (CAA) was introduced as a framework in [9] for supporting seamless sharing of user attributes across organizations, thus enabling ABAC for ad-hoc users. This alleviates the need for the system administrators of a particular organization to first create user accounts of external users and feed attribute information before giving them access to its resources. If an external user can prove her attribute values while making request for accessing a resource, the ABAC rules can decide whether the desired access can be granted.

In this approach, the CAA would digitally sign a user's attributes, thus certifying her credentials. The user can then attempt to access a resource in MongoDB, where the ABAC proxy cannot apriori recognize the user attributes in its local user attribute JSON. Instead, the user presents her digitally signed user attribute credentials issued by the CAA along with the connection request. The entire request is encrypted by a symmetric key, which is typically a session key previously exchanged between the MongoDB instance and the user's client. The proxy server can then verify the certificate using the widely known public key of the CAA. If found acceptable, the attribute values presented by the user are considered to be valid and these can now be used to determine whether access should be granted as per the ABAC policy maintained by the proxy server.

The user attributes may be stored locally in the proxy server cache for the currently active session. However, although the attributes of users tend to be relatively stable, still they may change after an initial connection is set up and before making an access request. To support the principle of complete mediation, instead of caching, every access request from a user should be evaluated against the current policy and the applicable user attributes while retrieving an object. This would ensure that these have not been changed in a way that affects the user's access to the object.

Thus, in addition to internal users, dynamic and ad-hoc access to external users can be provided by verifying attributes using the CAA without any need for intervention by the system administrator. This is done by validating the digital signature accompanying a request to confirm that the user attributes were indeed verified by the CAA. Unknown users, however, cannot directly submit their own attributes since those cannot be verified. However, in future, the applicability of the concept of Verifiable Credentials² can be explored for relaxing the need for a central attribute authority.

3.3 Obtaining Object Attributes (OA)

Unlike user attributes, object attributes do not typically change, and are often pre-determined based on the type of the object created. Hence, while creating the objects, most or all attributes can be automatically set by the proxy server. Advanced users can manually add, remove or modify the attributes associated with each object as well.

Similar to the user attributes of internal users, object attributes are also stored at the proxy server, with the attribute values being auto-generated upon creation of a new object, with an option to modify them later if needed. In addition, the proxy server requires an initialization step as well, in which it retrieves the attributes of all the existing objects in the MongoDB database. Hence, during query processing, whenever a new object attribute request arrives, the latest object attributes of the system are always available, and fetched instantly.

```
[
  {
    "inventory": {
      "item": "card",
      "qty": 15
    }
  }
]
```

². <https://www.w3.org/TR/vc-data-model/>

Listing 2.

Sample object attribute JSON stored locally at the proxy server. Each entry corresponds to the attributes for a particular object.

The JSON file consists of a JSON array, with each array entry corresponding to a particular object's attributes. A sample object attribute JSON is shown in Listing 2. Each entry of the array has the object name as the key (*inventory* in this example) and the values as a nested JSON of their object attributes.

3.4 Handling of Environmental Attributes (EA)

We next show how environmental attributes are effectively supported in the proposed ABAC proxy layer for MongoDB. It may be noted that, although environmental attributes add a lot of power to ABAC and differentiate it from other access control models, there is little work in the existing ABAC literature on handling these attributes. We support two types of environmental attributes, namely time and location, using which MongoDB access control policies can be specified. Each of these attributes is discussed below.

3.4.1 Time—Time is a necessary environmental attribute and can have a variety of uses. For instance, an organization may want its resources to be modified only during office hours on weekdays to prevent accidental or intentionally incorrect changes. This can be enforced by setting time-related periodic constraints in the ABAC policy. Likewise, a university might want to freeze the grades for a course after a certain date. Such a requirement can be handled by setting a validity duration in the policy.

As defined in Temporal RBAC (TRBAC) [10], temporal constraints for access control can be captured using periodic events of the form (*I, P*):

I: specifies the time interval during which a periodic event is valid and can be used to capture the validity duration of a rule.

P: represents a periodic expression, which consists of multiple periodic intervals, each of which has a starting point and the duration for which it repeats. An example of a periodic expression described in natural language is *Every Sunday of January in All Years*.

We have designed and implemented a `PeriodicEvent` class for describing the periodic events. The current date and time is captured and first checked for validity. Once validity is ascertained, comparison is done against all the available periodic expressions to verify if the current date and time falls within a valid period. For defining a periodic expression, the policy administrator has complete flexibility in terms of the time of the day chosen and the date or the day of the week. In addition, we also provide support for some special keywords like `weekdays` and `office-hours`, which capture the corresponding periodic expressions automatically. The `weekdays` keyword would add Monday to Friday of each week into the periodic expression, whereas the `office-hours` keyword would add Monday to Friday of each week from 8 am to 5 pm into the periodic expression. Multiple keywords can also be used together which are treated as a conjunction. More than one periodic expression may be defined in the environmental condition as a disjunction. Combining both *I*, the time interval

and P , the periodic expressions, dynamic ABAC policies can be specified for protecting data stored in MongoDB.

3.4.2 Location—Location is another useful environmental attribute that gives the MongoDB security administrators even more power while designing their ABAC policies. For example, a university might want to restrict modification of grades stored in MongoDB from outside their campus LAN. Hence, they need to set a policy that allows access only from an IP in the private block, e.g., those of the form $172.16.*.*$.

In the proposed proxy server design for mediating access to MongoDB, the IP address of a user is captured along with the connection request. The location attribute component of the ABAC policy is enforced using regular expressions to denote multiple ranges of IP addresses that are allowed. The source IP address of the incoming request is checked against these regular expressions. Access is granted if any of the ranges match. The corresponding regular expression for the example provided above would be $172\backslash.16\backslash.[0-9]*\backslash.[0-9]*$.

3.5 Defining ABAC Policies

For setting up an ABAC policy Π , the permissions to be granted are structured as the object name, followed by the permissions available on that object. For the permissions such as `find` and `update`, specific field names can also be mentioned upon which the permissions should be granted. These fields can be nested JSON objects as well. If no particular field name is mentioned, permission is granted over the entire Collection. For operations such as `insert` and `delete`, the granularity of a field is not of much relevance.

The ABAC policy is defined in JSON due to its high readability and flexibility. A sample ABAC policy JSON is shown in Listing 3. Each element of the JSON array is a rule, where each rule consists of the user attributes, object attributes and environment attributes to be eligible for certain permissions on a set of objects.

3.6 Making ABAC Access Decisions

The query first reaches the proxy server, where all the attributes are obtained and then the response is fetched from the MongoDB server according to the request. The ABAC access decision layer is applied either before or after this step and through enforcement of the defined ABAC policy, the user is granted access to the intended resource or the request is denied. While a variety of algorithms and data structures like PolTree [19] could have been chosen, we use a simple approach. Any access request, say *req*, is evaluated sequentially against all the rules in the ABAC policy specified in the proxy layer. While ensuring correctness, some optimizations are made to ensure scalability.

```

[
  {
    "user_attributes":{
      "position":"Manager", "region":"India"
    },
    "object_attributes":{
      "region":"India", "readonly":"no"
    },
    "environment":{
      "time":["weekends", "night"], "
      location":["127\\.\\.[0-9]+\\.\\.[0-9]
      +\\.\\.[0-9]+"
    },
    "permissions":{
      "inventory":[
        {
          "find":["item", "qty"]
        }
      ]
    }
  },
  {
    "user_attributes":{
      "position":"Developer", "region":"USA"
    },
    "object_attributes":{
      "region":"USA"
    },
    "environment":{
      "time":["office-hours"]
    },
    "permissions":{
      "inventory":["insert"], "profiles":["
      find", "insert"]
    }
  }
]

```

Listing 3.

Sample ABAC policy JSON defined by the system administrator

Algorithm 1 provides the details. In the algorithm, UA refers to the attributes of the user the access decision is being made for and OA refers to the attributes of the object being considered. The components $time$ and ip together constitute the environmental attributes, with $time$ being the time at which the access request is made and ip being the IP address from which the request has originated. As mentioned in Sub-section 2.3, $op \in OP$ refers to the desired operation on the object involved in the access request, whereas Π refers to the ABAC policy of the organization. doc is the document retrieved from the database server, which is used to enforce access control for queries at a fine granularity.

For each $rule$ in Π , the request req is first matched against all the user and object attributes of that $rule$ in the first block of the algorithm from Lines 5 to 12. If the req attributes do not

match the attribute requirements of this *rule*, it cannot be used to grant the access and hence, we *continue* with the next *rule*. However, if satisfied, a *PeriodicEvent* object is created from the *time* requirements in the *rule*, and is evaluated against the *time* component of *req*. Similarly, the *location* is also fetched and evaluated. If the environmental attributes of the *req* match those required by the rule, we proceed further, without having to *continue* to the next rule. This comprises the second block, from Lines 13 to 23 of the algorithm. Note that, policy specifications can be further enriched by considering hierarchical relationships among attributes and handling subsets and supersets during enforcement as described in [28].

If all the attributes have been matched, we retrieve the permissions *prms* provided by this *rule* and check if the operation (*op*) requested by *req* can be granted by it. Note that, the evaluation is different for query and non-query requests. This is because for query requests, the document *doc* needs to be retrieved first, in order to apply access control at the granularity of fields as described in Section 3.1.2. For other requests, it is achieved by retrieving the fields specified in the request. This step is described in the final block of the algorithm, from lines 24 to 32. Applying fine-grained access control to complex operations such as aggregate and Map-Reduce is an ongoing topic of research (see [20] and [21]) and is beyond the scope of this work.

If access is denied, the proxy server creates an OP_REPLY message with the following body and sends it to the user.

```
{{"$err"=>"UnauthorizedError: not authorized
to execute command #{@recent[:header][:
opCode]} on collection #{@recent[:
collection]} of database #{@recent[:
database]}", "code"=>13}}
```

The algorithm has a time complexity of $O(|\Pi| \times (|UA| + |OA| + |EA|))$. Instead of sequentially evaluating the rules, other data structures for policy representation and evaluation can also be used. This includes recently proposed techniques like PolTree [19], which are unique tree-based data structures for storing ABAC policies and require less number of comparisons to make an access decision. For example, in the n-ary version of PolTree named as N-PolTree, the number of comparisons required for resolving an access request is of the order of $O(|A|)$, where *A* is the set of all attributes.

3.7 Extending to other NoSQL Databases

The approach proposed in this paper can be extended to several other NoSQL databases as well. This can be achieved by parsing the messages for those databases according to their standards. For instance, a native protocol is used by Cassandra to establish communication between a database server and its clients.³ The message formats exchanged over TCP are appropriately defined in such NoSQL databases, which are broadly classified as requests and responses. Each message also has a header that identifies the type of message using its opcode. Parsing this protocol, our approach can easily be deployed in Cassandra using a similar proxy server. Likewise, Redis uses the RESP (REdis Serialization Protocol),

³. https://cassandra.apache.org/_/native_protocol.html

which serializes different queries and establishes communication between the server and the clients.⁴ Since RESP is also a TCP based protocol, by intercepting and parsing the messages, the proposed approach can be extended to Redis as well.

Algorithm 1 Making access decisions in ABAC

```

1: procedure GRANTACCESS(req,  $\Pi$ , doc)
2:   allow  $\leftarrow$  False
3:   for rule in  $\Pi$  do
4:     satisfies  $\leftarrow$  True
5:     Matching User and Object Attributes:
6:     for (key, value) in rule['user_attributes'] do
7:       if req.UA[key]  $\neq$  value then
8:         satisfies  $\leftarrow$  False
9:     for (key, value) in rule['object_attributes'] do
10:      if req.OA[key]  $\neq$  value then
11:        satisfies  $\leftarrow$  False
12:     if not satisfies then
13:       continue
14:     Obtaining and Evaluating Environmental Attributes:
15:     PeriodicEvent  $\leftarrow$  rule['environment']['time']
16:     if not req.time  $\subseteq$  PeriodicEvent then
17:       satisfies  $\leftarrow$  False
18:     ip_match  $\leftarrow$  False
19:     for ip_regex in rule['env']['location'] do
20:       if ip_regex matches req.ip then
21:         ip_match  $\leftarrow$  True
22:     if not ip_match then
23:       satisfies  $\leftarrow$  False
24:     if not satisfies then
25:       continue
26:     Evaluating Request Against Matched Rule From  $\Pi$ :
27:     if req.op  $\in$  rule['perms']['object'] then
28:       if req.op = 'Find' then
29:         if req.op.fields  $\subseteq$  doc.fields then
30:           allow  $\leftarrow$  True
31:           break
32:         else
33:           if req.op.fields  $\subseteq$  req.fields then
34:             allow  $\leftarrow$  True
35:             break
36:     return allow

```

4 Implementation

This section presents the implementation and deployment details and also demonstrates some major user flows.

4.1 Prototype Implementation

We have developed a complete system enforcing ABAC supporting all the methods described in the last section using Ruby. The proxy server is designed with the event-driven I/O employing the reactor design pattern for handling multiple clients at once, while separating the access control from the network proxy implementation. Ruby's `em-proxy`

⁴ <https://redis.io/topics/protocol>

[22] tool is used for the same. The performance overhead in the proxy implementation of `em-proxy` is within 3–5% in terms of latency.

While obtaining user attributes verified by the CAA, Ruby's `OpenSSL` gem is used to decrypt the packet containing the encoded user attribute JSON and the X.509 digital certificate. The cipher used is AES in the CBC mode with a 256-bit key encryption. The X.509 certificate then obtained is also verified using the `OpenSSL` gem's X.509 implementation. The encoded JSON containing the user attributes is then decoded using the `Base64` decoder to obtain the correct set of user attributes, which are then added to the local cache for future use during the session.

For defining the environmental attributes using time, the `PeriodicEvents` class is implemented in Ruby using the `Ruby Date` class. The time of day verification is implemented using the `TimeOfDay` class in the `tod` gem. The location IP regular expressions are defined and verified using Ruby's `Regexp` class. All the attributes and policies are defined using JSON and parsed using Ruby's `json` gem. The proxy server is running on port 29017, whereas the MongoDB server is running on port 27017.

The MongoDB Wire Messages are sent over the network as TCP packets, with most of the body being encoded in BSON format. When a message is intercepted by the proxy server, first the `opCode` field is checked from the message, which identifies what kind of message it is. The types of messages with the corresponding `opCodes` were described in Section 2.4. According to the message type, it is further parsed and any BSON documents which can be the queries, insert-documents, update specifications, etc., are deserialized using Ruby's `BSON` gem. These are converted to a Ruby `Hash`, which is similar to a dictionary in other programming languages. The corresponding flags are also decoded according to the specification mentioned in the protocol. For instance, a message with the `opCode` `OP_QUERY`, the flags mean the following:

- 0 - reserved
- 1 - `TailableCursor` - cursor not closed upon retrieval of last data
- 2 - `SlaveOk` - allow queries from the replica slave
- 3 - `OplogReplay` - replay a replica's `oplog`
- 4 - `NoCursorTimeout`
- 5 - `AwaitData` - block rather than not returning any data if at the end of the data
- 6 - `Exhaust` - stream the data in multiple `OP_GET_MORE` packages
- 7 - `Partial` - get incomplete results even if some shards are crashed
- 8 – 31 – reserved

The query also has information such as the `fullCollectionName`, which contains the database and the collection name, the number of documents to skip (`numberToSkip`), the maximum number of documents to return in the first reply message (`numberToReturn`), establishing a cursor for more documents and the `query` BSON document representing the

query using the elements prescribed by MongoDB. The information about parsing other type of messages is provided in the MongoDB Wire Protocol documentation and parsers are designed for each of them in Ruby.

In addition to the `opCode` `OP_QUERY` described above, methods are also implemented for building messages of other `opCodes` such as `OP_REPLY`, `OP_INSERT`, etc., as described in Section 2.4. This involves serializing a `Hash` into a BSON document and building up the TCP packet following the specifications provided in the MongoDB Wire Protocol. Thus, we are able to fully parse and construct the MongoDB Wire Messages following these protocols.

4.2 Demonstration and Example Outputs

This section demonstrates a typical usage of the proposed system, with the user attributes, object attributes and ABAC policy JSONs as described in Sections 3.2, 3.3 and 3.5, respectively, and the access decisions made using the algorithm in Section 3.6. The system also accommodates two environmental attributes, namely, time and location as described in Section 3.4. It is scalable and can run on multiple nodes at once. Some possible user flows of the system are demonstrated below.

Access granted for internal user: Alice has her attributes in the user attribute JSON stored locally in the ABAC server. She makes a request for performing the `find` operation on collection `inventory` as shown in Figure 3. The first rule in the policy defined in Section 3.5 grants her that access, provided she sends the access request on a weekend night, from an appropriate location. Hence, the access is allowed and the MongoDB ABAC system logs are generated as shown in Figure 4.

Access denied for internal user: Like Alice, Bob also has his attributes in the user attribute JSON in the ABAC server. He makes a request for performing the `find` operation on the collection `inventory` as shown in Figure 5. However, no rule in the policy grants him that access at the given environmental conditions, hence denying his request as shown in the MongoDB ABAC system logs of Figure 6.

Access denied for external user: Kate is a user external to the organization and has never made any request to the proxy server yet. Her attributes are first decrypted using the symmetric session key provided by her. The attributes present in the decrypted file are verified using the digital certificate present in it. No rule in the policy grants her the access for performing `find` operation on collection `inventory` as shown in Figure 7. Hence, she is denied the requested access. Verification of her digital certificate issued by the Central Attribute Authority and the subsequent access decision are shown in the MongoDB ABAC system log of Figure 8.

4.3 Deployment

We now discuss the steps involved in deploying the proposed ABAC proxy layer for MongoDB using a sample deployment source code from our GitHub repository [34]. The software is designed as a Ruby gem, which when built and deployed, functions as a proxy

tool and intercepts all the connections to the MongoDB database from the clients. The implementation requires the em-proxy tool that is derived from the mongo-proxy gem [35].

Once the Ruby gem is installed, instantiating the MongoABAC class, adding front and back callbacks, defining the ports for MongoDB and the proxy server, and executing the code is sufficient for installing the ABAC proxy server on any running MongoDB server. This proxy server will now intercept all the communication from any number of MongoDB clients, thus enforcing the ABAC policy.

Our GitHub repository 'mongo-abac' [34] contains the complete source code for the Ruby gem, which can be deployed by any interested organization/user in their MongoDB server instances. It has been distributed with the MIT License for free-use.

5 Empirical Evaluation

We carried out an extensive experimental evaluation to study the performance of our proposed architecture. In this section, results of these experiments are presented in terms of the time taken to process each query and enforce the ABAC policy on the MongoDB database. The system was implemented in Ruby on a 2.2GHz Intel i7 machine having 16 GB of RAM. While there is no pre-processing time for setting up the policy, since each query gets intercepted by the proxy server and subsequently processed for making an appropriate access decision, there is some query overhead. However, we show that this overhead is not very significant relative to the baseline query time on a MongoDB server with no access control enforced. The processing time depends asymptotically only on the number of rules, the number of attributes and the number of fields in each rule.

Two sets of experiments were conducted. First, we use the Enron data set [11], which is a publicly available large corpus previously used also by other researchers for similar studies. We then carry out additional studies with synthetically created data sets as well. This was done to evaluate additional variations in data that are not present in the Enron corpus. In the following two sub-sections we present our results on these two data sets.

5.1 Results on Enron Data set

The Enron Dataset is a large, publicly available dataset of emails from Enron Corporation. It can be represented conveniently as JSON objects since there are many composite fields that make up an email such as the email header, body, etc. We carry out a number of experiments on this dataset using a variety of MongoDB queries such as `find`, `count`, `aggregate`, `insert`, etc., and also consider different scenarios as described next. For each scenario, the query execution time was measured. The results were obtained for variation in the number of rules (R) as well as user attributes (UA), while keeping the other parameters, namely, number of users, number of objects, etc., as constant.

Our first query was to count the number of emails sent within a particular time period. The access control policy was appropriately set up to grant permission for the `count` operation on the collection `messages`. The results are shown in Figure 9. In the figure, we vary the number of rules from 10 to 500 and observe the execution time keeping the number of

user attributes fixed for each run. We then repeat the experiments for other values of the number of user attributes. The number of user attributes chosen were 5, 10 and 20. We also plot the baseline in the figure. It is observed that the overhead introduced by the ABAC layer is not appreciably high. For 100 rules, which is a reasonable number, the additional overhead is less than 100 milliseconds. Even with 500 rules and 20 attributes, the overall query time is about half a second on the platform specified earlier in this section, which is quite reasonable.

The settings for the second query are similar to those of the first. The main difference is that instead of count, the goal is to actually fetch all the emails sent by a particular employee. The number of fetched documents was 160. The results along with the baseline are plotted in Figure 10. It is observed that the overhead for the *find* query is close to that for the *count* query.

As we can see, for both the above queries, the ABAC policy allowed access and hence, the MongoDB server was hit for query execution. The next query is to insert a new email into the collection. The ABAC policy was configured in a way that access is denied to the user for *insert* operation on the collection *messages*. The results are plotted in Figure 11. In this case, since the access is being denied from the proxy server itself and the MongoDB server is never contacted, the execution time in most of the cases is much less than the time taken by the command to execute directly on the MongoDB server. This is one of the benefits obtained by following the architecture proposed in Section 3.

Overall, it is observed that, matching expectations, the overhead introduced by the number of rules and attributes does not depend on the type of the query, user or the object.

5.2 Results on Synthetically Generated Data Sets

Since the Enron corpus is a real dataset, it has data distributions likely to be encountered in real life when our architecture is deployed. However, it does not have some of the variations that can affect the performance of the ABAC layer added to MongoDB. In order to accomplish this, we carried out another set of experiments using synthetically generated datasets that exhibit the desired variations. In our experiments with these datasets, while keeping the number of rules and attributes fixed at 50 and 10, respectively, we observe the variation in execution time with other factors.

In the first such experiment, we study the variation in execution time while changing the number of fields in the document that need to be checked in order to make the appropriate access decisions. We run this experiment for *find* and *update* requests and compare their performance. For this experiment, the baseline against which the performance is being compared is when the granularity of access control is that of a collection. The results are plotted in Figure 12. It is seen that the performance overhead of checking fields is not significant. This is because the fields only need to be checked for a particular operation on an object, once all the attributes of the request have already matched with the corresponding attributes in the rule.

As described in Section 3, there are two possible architectures that can be considered for query operations. The proxy server can enforce ABAC either before or after sending the request to the MongoDB server. In the final experiment, we compare execution time of the same query for these two settings. The number of user attributes is fixed at 10. The results are plotted in Figure 13. It is observed that the proposed system is quite efficient. When an access is denied, the enforcement of ABAC before forwarding the messages from the proxy server leads to a major improvement in performance, often even beating out the execution time taken by the MongoDB server without any access control. The query overhead time does not vary with the number of users in the database, thus ensuring scalability, since the number of rules and attributes is always much less compared to the number of potential users in a large organization.

6 Related Work

NoSQL databases [1], due to their high degree of flexibility, scalability and availability have become quite popular in recent times [2]. However, as mentioned in Section 2.2, a detailed analysis reveals that most of the NoSQL databases have limited support for data protection methods, specifically access control.

The National Institute of Standards and Technology (NIST) came out with a special publication (No. 800–162) in 2014 [7], in which they standardized the definition of attribute based access control components [6]. The standard also discusses the use of ABAC within organizations to improve information sharing and also between organizations, while enforcing control over the access of that information. They also describe two standards for ABAC called Extensible Access Control Markup Language (XACML) and Next Generation Access Control (NGAC) [24]. A unified ABAC model which could be configured to cover DAC, MAC and RBAC was proposed in [25]. The current state-of-the-art for handling privacy and security challenges such as access control, authentication, etc., in several NoSQL databases was studied by Sicari et al. [36].

There has been limited research on enhancing NoSQL databases with ABAC. Colombo and Ferrari define a general approach to enforce ABAC in NoSQL databases [26]. They rely on a unifying query language for NoSQL called SQL++ [27] and develop a query rewriting step to be able to enforce heterogeneous types of ABAC policy rules upto the cell level of granularity. Although promising, there has not been wide adaptation of SQL++ as a unifying query language for NoSQL platforms, and most NoSQL databases operate with their own language and data models. The approach proposed by us in this paper is significantly different since we aim to provide support for existing NoSQL databases using their own language models by intercepting and parsing messages using a proxy server to enforce ABAC. They also worked on evaluating the impact of access control policies on schemaless data, and proposed a data model agnostic approach to point out contents which are authorized and those that are unauthorized in a protected data resource.

In [28], a query-modification model is proposed for the implementation of ABAC in big data, relational and some NoSQL databases. However, query rewriting techniques cannot be adapted directly and efficiently to NoSQL databases such as MongoDB since they are

schema-less and have support for complex operations such as aggregate and map-reduce jobs. Batra et al. [29] propose an algorithm for converting ABAC attributes and policies into RBAC roles, thus providing a method for systems with support for RBAC to extend into ABAC. However, still this cannot realize the full power of ABAC. Gupta et al. [23] present a fine-grained ABAC model called HeABAC catering specifically to the multi-tenant Hadoop ecosystem.

There have been some research efforts at enhancing access control in NoSQL databases with other techniques as well. For example, Colombo and Ferrari [4] present an organized literature review and also study future trends in access control for big data and NoSQL databases. In another work [30], they propose an approach for the integration of purpose based access control and policy enforcement in MongoDB using a monitor called Mem that intercepts the Wire Messages and applies a layer of access control in Mem. A related though different work [20] discusses about refining the granularity of access control in NoSQL Document-Oriented datastores like MongoDB, addresses the issues with such a framework, and reasons why existing fine-grained access control solutions (such as [31]) do not work for NoSQL databases. They then propose several strategies to develop such a framework. In [32], an enforcement monitor called ConfinedMem is proposed, which extends the access control policies with refined granularity up to the field level, with support for context-based policies as well, aiming towards virtual private databases.

In [37], a key-value based access control mechanism is implemented for key-value stores like Cassandra at column family levels. Mrgado et al. [38] propose a security model for use in graph-based NoSQL databases such as Neo4j based on metadata, whereas [39] is based on the eXtensible Access Control Markup Language (XACML), which is adapted for graph-structured data. A recent work by Colombo and Ferrari [40] systematically evaluates the effect of various access control policies on the performance of NoSQL databases. As is apparent, none of the existing work provides support for attribute-based access control for NoSQL databases. Instead, they either focus on other forms of access control or survey the existing state of access control, with an emphasis on big data and NoSQL databases.

In a recently published poster paper [33], we leveraged MongoDB's existing role-based access control, and extended it with attributes by converting the ABAC policies into MongoDB roles. While the idea is interesting, it has several limitations including its inability to support environmental attributes and supporting access to ad-hoc users. The approach proposed in the current paper is fundamentally different, as we enforce ABAC in NoSQL databases natively by intercepting the messages sent by the clients through a proxy server. We also present novel methods for handling environmental attributes like time and location, controlling access to external users by verifying their attributes certified by a CAA, and applying ABAC on a much finer granularity. Thus the richness lies both in the architecture of the proposed system as well as in its functionality.

As discussed above, access control for NoSQL databases is an active topic of research that would be of great utility to the developers as well as security administrators. So far, there has so been little research done towards incorporating ABAC into NoSQL databases. Hence, it is

of utmost importance to develop and implement practical solutions as presented in this paper which can work at scale.

7 CONCLUSIONS

In this article, we have presented a novel approach for integrating a full-fledged support for attribute-based access control at a fine granularity into NoSQL databases. Towards this end, we provide a complete end-to-end implementation for MongoDB. We define techniques for enforcing access to external ad-hoc users without any need for intervention by the system security administrators. This is achieved through the use of digitally certified attributes presented along with an access request. In addition, our system has native support for capturing environmental attributes such as time and location. Extensive experiments with the Enron corpus as well as synthetically generated datasets validate correctness of the approach and demonstrate its usability.

Going forward, we plan to optimize the algorithm used for enforcement of access control further using techniques such as those defined in [19]. We would also like to implement modules for parsing messages similar to MongoDB Wire Messages for other NoSQL databases, thus extending ABAC to all NoSQL databases. The system design can be further improved for fine-grained access control allowing the policy to be subject to field values, in addition to field names, which is currently supported. Our long term goal is to work with the open-source community to have ABAC built into MongoDB as its integral part, thus going beyond the use of a proxy layer.

Acknowledgments

Research reported in this publication was supported by the National Science Foundation awards CNS-1564034, CNS-1624503 and CNS-1747728, the National Institutes of Health award R35GM134927, and a research gift received from Cisco University Research. The work of Shamik Sural is supported by CISCO University Research Program Fund, Silicon Valley Community Foundation award number 2020-220329 (3696). The content is solely the responsibility of the authors and does not necessarily represent the official views of the agencies funding the research.

Biographies



Eeshan Gupta graduated from the Department of Computer Science and Engineering at the Indian Institute of Technology, Kharagpur, India in 2021 with a dual degree, i.e., Bachelor of Technology and Master of Technology. He is currently working at Google. His research interests include database systems, computer security and distributed systems.



Shamik Sural is a full professor in the Department of Computer Science and Engineering, Indian Institute of Technology (IIT) Kharagpur. He received the Ph.D. degree from Jadavpur University, Kolkata, India in the year 2000. Before joining IIT Kharagpur in 2002, he spent more than a decade in the Information Technology industry working in India as well as in the USA. Shamik is a recipient of the Alexander von Humboldt Fellowship for Experienced Researchers and also Fulbright Nehru Academic and Professional Excellence Fellowship. He is a senior member of IEEE and has served as the Chairman of the IEEE Kharagpur Section in 2006. He is currently on the editorial boards of IEEE Transactions on Dependable & Secure Computing and IEEE Transactions on Services Computing. He has published more than two hundred research papers in reputed international journals and conferences. His research interests include computer security and data science.



Jaideep Vaidya is a Professor of Computer Information Systems with Rutgers University and is the Director of the Rutgers Institute for Data Science, Learning, and Applications. He has published over 190 papers in international conferences and journals. His research interests are in privacy, security, and data management. He is an IEEE Fellow, an ACM Distinguished Scientist, and is the Editor in Chief of IEEE TDSC.



Vijayalakshmi Atluri is a Professor of Computer Information Systems in the MSIS Department, and research director for the Center for Information Management, Integration and Connectivity (CIMIC) at Rutgers University. Her research interests include Information Security, Privacy, Databases, Workflow Management, Spatial Databases, Multimedia and Distributed Systems. She has published over 150 technical papers in peer reviewed journals and conference proceedings. She is a senior member of the IEEE Computer Society and member of the ACM.

References

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

- [2]. DB-Engine Ranking - popularity ranking of database management systems. Available: <https://db-engines.com/en/ranking>. [Accessed June 16, 2021]
- [3]. Okman L, Gal-Oz N, Gonen Y, Gudes E and Abramov J, "Security issues in NoSQL databases", Proc. IEEE 10th Int. Conf. Trust Security Privacy Comput. Commun., pp. 541–547, 2011.
- [4]. Colombo P, Ferrari E, "Access control technologies for Big Data management systems: literature review and future trends", Cyber-security 2, 3, 2019. Available: 10.1186/s42400-018-0020-9. [Accessed June 16, 2021]
- [5]. Sandhu RS, "Role-based access control", Advances in Computers. Vol. 46. Elsevier, pp. 237–286, 1998
- [6]. Hu VC, Kuhn DR, Ferraiolo DF and Voas J, "Attribute-Based Access Control," in Computer, vol. 48, no. 2, pp. 85–88, Feb. 2015.
- [7]. Hu VC, Ferraiolo D, Kuhn R, Friedman AR, Lang AJ, Cogdell MM, Schnitzer A, Sandlin K, Miller R and Scarfone K, "Guide to Attribute Based Access Control (ABAC) Definition and Considerations [includes updates as of 02-25-2019], NIST special publication, 800(162).
- [8]. MongoDB Wire Protocol. Available: <https://docs.mongodb.com/manual/reference/mongodb-wire-protocol/> [Accessed June 16, 2021]
- [9]. Das S, Sural S, Vaidya J and Atluri V, "Central Attribute Authority (CAA): A Vision for Seamless Sharing of Organizational Resources," 2019 First IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA), 2019, pp. 209–217.
- [10]. Bertino E, Bonatti PA, and Ferrari E, "TRBAC: A temporal role-based access control model", ACM Trans. Inf. Syst. Secur 4, 3, pp. 191–233, August 2001.
- [11]. Klimt B and Yang Y, "The Enron corpus: A new dataset for email classification research", Proc. Eur. Conf. Mach. Learn, pp. 217–226, 2004.
- [12]. Moniruzzaman A and Hossain S. "NoSQL Database: New Era of Databases for Big data Analytics - Classification, Characteristics and Comparison.", ArXiv abs/1307.0191, 2013
- [13]. MongoDB Official Documentation. Available: <https://docs.mongodb.com/>. [Accessed June 16, 2021]
- [14]. Cassandra Official Documentation Available: <https://cassandra.apache.org/doc/latest/>. [Accessed June 16, 2021]
- [15]. AWS Identity and Access Management. Available: <https://aws.amazon.com/iam/>. [Accessed June 16, 2021]
- [16]. Redis Official Documentation Available: <https://redis.io/documentation>. [Accessed June 16, 2021]
- [17]. The Neo4j Operations Manual. Available: <https://neo4j.com/docs/operations-manual/current/>. [Accessed June 16, 2021]
- [18]. BSON (Binary JSON) Serialization. Available: <https://bsonspec.org/>. [Accessed June 16, 2021]
- [19]. Nath R, Das S, Sural S, Vaidya J, and Atluri V, "PolTree: A Data Structure for Making Efficient Access Decisions in ABAC", Proceedings of the 24th ACM Symposium on Access Control Models and Technologies (SACMAT '19). Association for Computing Machinery, New York, NY, USA, pp. 25–35, 2019
- [20]. Colombo P, Ferrari E, "Fine-Grained Access Control Within NoSQL Document-Oriented Datastores", Data Sci. Eng 1, pp. 127–138, 2016.
- [21]. Ulusoy H, Kantarcioglu M, Pattuk E and Hamlen K, "Vigiles: Fine-Grained Access Control for MapReduce Systems," 2014 IEEE International Congress on Big Data, pp. 40–47, 2014.
- [22]. GitHub: EM-Proxy: EventMachine Proxy DSL for writing high-performance transparent / intercepting proxies in Ruby. Available: <https://github.com/igrigorik/em-proxy>. [Accessed June 16, 2021]
- [23]. Gupta M, Patwa F, and Sandhu R. "An attribute-based access control model for secure big data processing in Hadoop ecosystem." Proceedings of the Third ACM Workshop on Attribute-Based Access Control, pp. 13–24, 2018.

- [24]. Ferraiolo D, Chandramouli R, Hu V, and Kuhn R. "A comparison of attribute based access control (ABAC) standards for data service applications.", NIST Special Publication 800 (2016): 178, 2016.
- [25]. Jin X, Krishnan R, Sandhu R, "A unified attribute-based access control model covering DAC, MAC and RBAC.", IFIP Annual Conference on Data and Applications Security and Privacy, pp. 41–55. Springer, Berlin, Heidelberg, 2012.
- [26]. Colombo P and Ferrari E, "Towards a Unifying Attribute Based Access Control Approach for NoSQL Datastores," 2017 IEEE 33rd International Conference on Data Engineering (ICDE), 2017, pp. 709–720.
- [27]. Ong KW, Papakonstantinou Y, and Vernoux R. "The SQL++ unifying semi-structured query language, and an expressiveness benchmark of SQL-on-Hadoop, NoSQL and NewSQL databases." CoRR, abs/1405.3631, 2014.
- [28]. Longstaff J and Noble J, "Attribute Based Access Control for Big Data Applications by Query Modification," 2016 IEEE Second International Conference on Big Data Computing Service and Applications (BigDataService), 2016, pp. 58–65.
- [29]. Batra G, Atluri V, Vaidya J, and Sural S. "Deploying ABAC policies using RBAC systems.", Journal of computer security 27, no. 4, pp. 483–506, 2019. [PubMed: 31929684]
- [30]. Colombo P and Ferrari E, "Enhancing MongoDB with Purpose-Based Access Control," in IEEE Transactions on Dependable and Secure Computing, vol. 14, no. 6, pp. 591–604, 1 Nov.-Dec. 2017.
- [31]. Rizvi S, Mendelzon A, Sudarshan S, and Roy P, "Extending query rewriting techniques for fine-grained access control." Proceedings of the 2004 ACM SIGMOD international conference on Management of data, pp. 551–562., 2004.
- [32]. Colombo P and Ferrari E, "Towards Virtual Private NoSQL datastores," 2016 IEEE 32nd International Conference on Data Engineering (ICDE), 2016, pp. 193–204.
- [33]. Gupta E, Sural S, Vaidya J, and Atluri V, "Attribute-Based Access Control for NoSQL Databases.", Proceedings of the Eleventh ACM Conference on Data and Application Security and Privacy, pp. 317–319, 2021.
- [34]. GitHub: Mongo-ABAC: A gem for enforcing ABAC policy in MongoDB servers. Available: <https://github.com/eeshan9815/mongo-abac>. [Accessed October 10, 2021]
- [35]. GitHub: mongo-proxy: A gem for proxying MongoDB at the wire-protocol level. Available: <https://github.com/bakks/mongo-proxy>. [Accessed October 10, 2021]
- [36]. Sicari S, Rizzardi A, and Coen-Portisini A, "Security&privacy issues and challenges in NoSQL databases.", Computer Networks, Volume 206 (2022): 108828.
- [37]. Kulkarni D, "A fine-grained access control model for key-value systems.", Proceedings of the third ACM conference on Data and application security and privacy (CODASPY '13). Association for Computing Machinery, New York, NY, USA, 2013, pp. 161–164.
- [38]. Morgado C, Busichia Baioco G, Basso T and Moraes R, "A Security Model for Access Control in Graph-Oriented Databases," 2018 IEEE International Conference on Software Quality, Reliability and Security (QRS), 2018, pp. 135–142
- [39]. Mohamed A, Auer D, Hofer D, J. "Authorization Policy Extension for Graph Databases", Future Data and Security Engineering. FDSE 2020. Lecture Notes in Computer Science, vol 12466. Springer, Cham, 2020
- [40]. Colombo P, Ferrari E, "Evaluating the effects of access control policies within NoSQL systems", Future Generation Computer Systems, Volume 114, 2021, pp. 491–505.

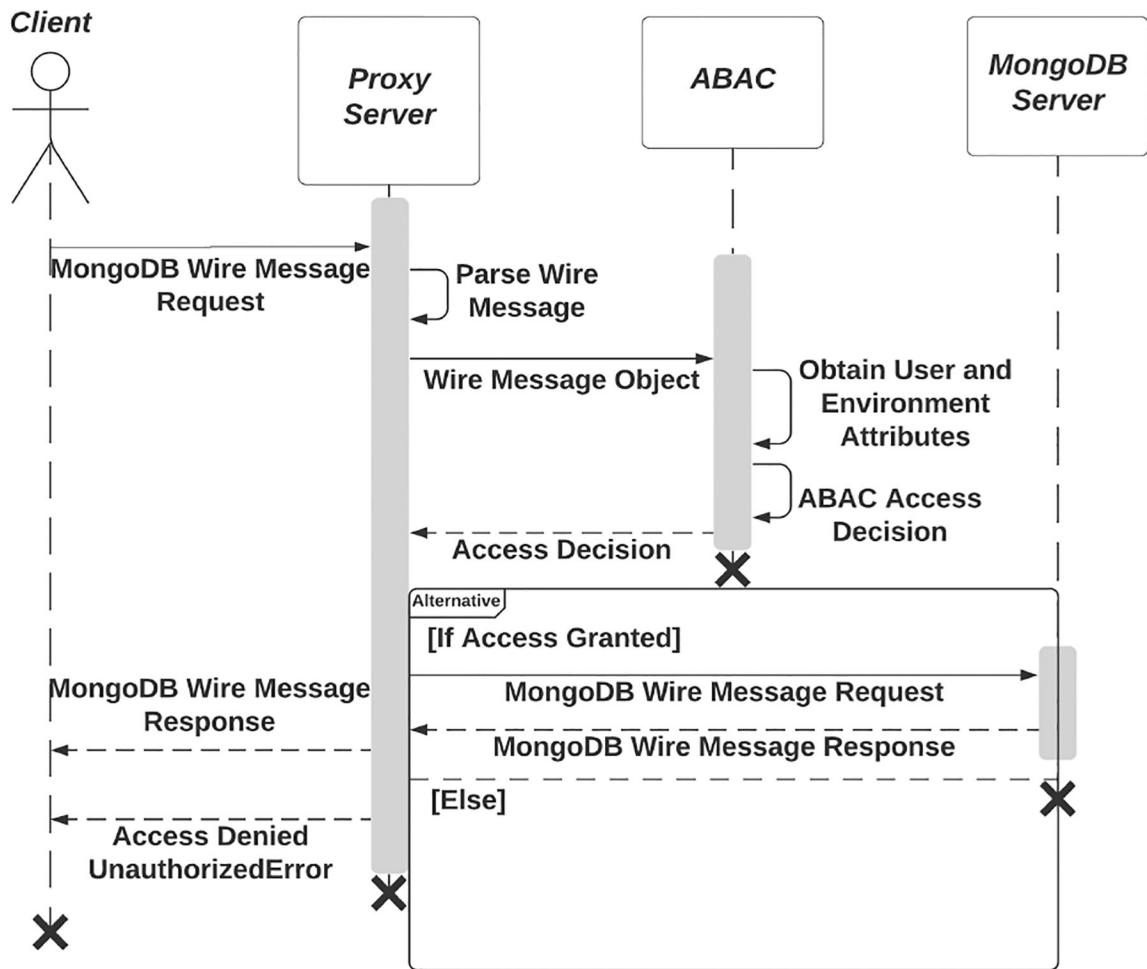


Fig. 1. Access control before forwarding request to the MongoDB server

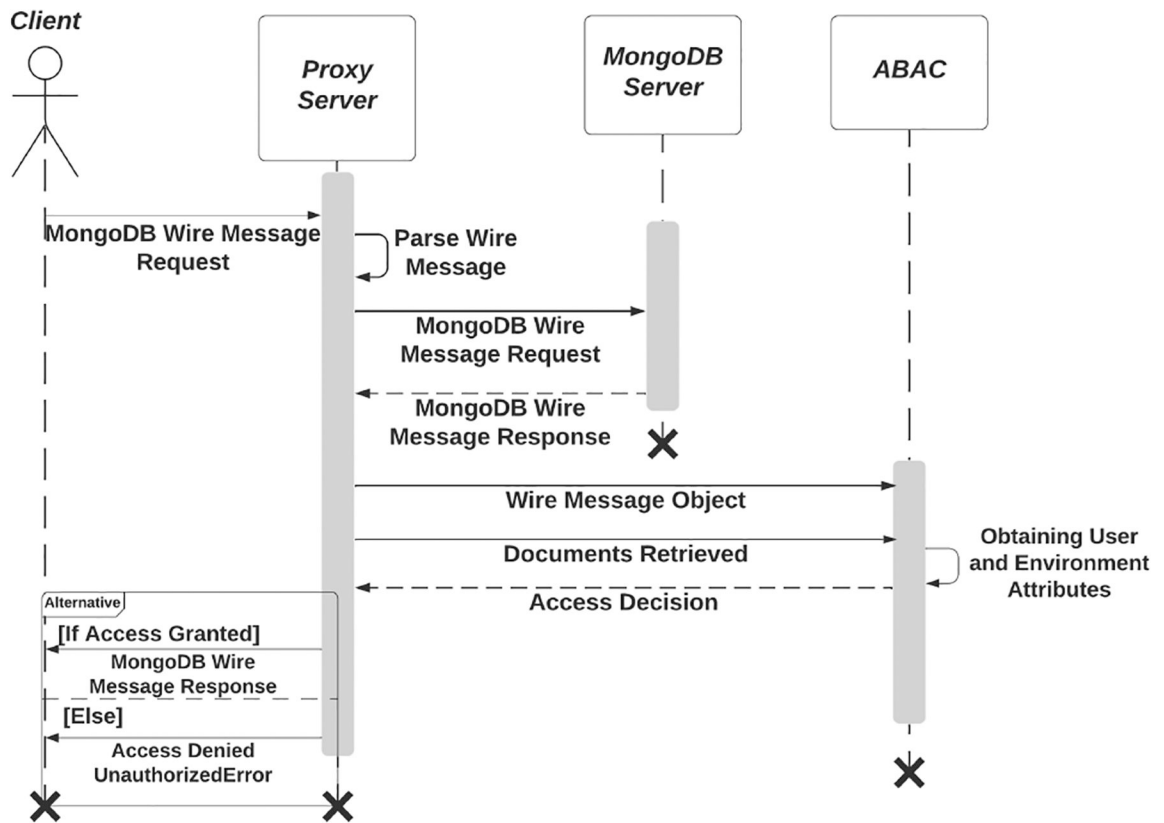
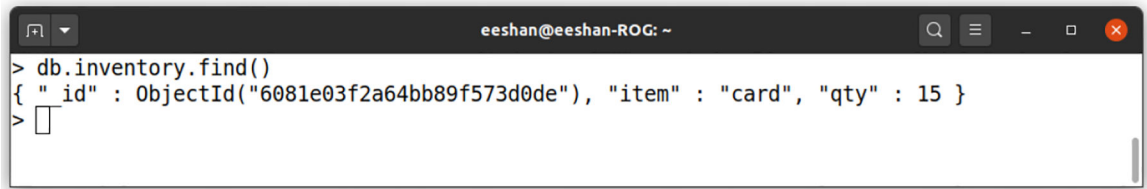


Fig. 2.
Access control after receiving reply from the MongoDB server

A terminal window with a dark background and light text. The title bar reads 'eeshan@eeshan-ROG: ~'. The prompt is '>'. The command entered is 'db.inventory.find()'. The output is a JSON object: '{ "id" : ObjectId("6081e03f2a64bb89f573d0de"), "item" : "card", "qty" : 15 }'. The prompt is '>' again.

```
eeshan@eeshan-ROG: ~  
> db.inventory.find()  
{ "id" : ObjectId("6081e03f2a64bb89f573d0de"), "item" : "card", "qty" : 15 }  
>
```

Fig. 3.
Query to the server made by Alice

```
eeshan@eeshan-ROG: ~/code/ABAC
wire message intercepted: {:flags=>0, :numberToSkip=>0, :numberToReturn=>0, :database=>"test", :collection=>"inventory", :query=>{}, :returnFieldSelector=>nil, :header=>{:messageLength=>48, :requestID=>4, :responseTo=>0, :opCode=>:query}}
user name: Alice
ip: 127.0.0.1
current time: 2021-04-24 22:41:00 +0530
current day: Saturday
collection: inventory
opcode: query
op: find
user attributes fetched: {"position"=>"Manager", "region"=>"India", "team"=>"Ads"}
object attributes fetched: {"region"=>"India", "readonly"=>"no"}

checking for rule: {"user_attributes"=>{"position"=>"Manager", "region"=>"India"}, "object_attributes"=>{"region"=>"India", "readonly"=>"no"}, "env"=>{"time"=>["weekends", "night"], "ip"=>["127\\.([0-9]+\\.([0-9]+\\.([0-9]+)\\.([0-9]+))]"}}, "permissions"=>{"inventory"=>["find"]}}
user and object attributes match with the rule
PeriodicEvent satisfied for ["weekends", "night"]
location matched with IP regex: 127\\.([0-9]+\\.([0-9]+\\.([0-9]+)\\.([0-9]+)

Permission granted to user Alice for find on inventory
```

Fig. 4.
Server logs for Alice's query



```
eeshan@eeshan-ROG: ~  
> db.inventory.find()  
Error: error: {  
  "$err" : "UnauthorizedError: not authorized to execute command query on collection  
  inventory of database test",  
  "code" : 13  
}
```

Fig. 5.
Query to the server made by user Bob


```

eeshan@eeshan-ROG: ~/code/ABAC
wire message intercepted: {:flags=>0, :numberToSkip=>0, :numberToReturn=>0, :database=>"test", :collection=>"inventory", :query=>{}, :returnFieldSelector=>nil, :header=>{:messageLength=>48, :requestID=>4, :responseTo=>0, :opCode=>:query}}
user name: Bob
ip: 127.0.0.1
current time: 2021-04-24 22:39:14 +0530
current day: Saturday
collection: inventory
opcode: query
op: find
user attributes fetched: {"position"=>"Developer", "region"=>"USA", "team"=>"Backend", "work"=>"yes"}
object attributes fetched: {"region"=>"India", "readonly"=>"no"}

checking for rule: {"user_attributes"=>{"position"=>"Manager", "region"=>"India"}, "object_attributes"=>{"region"=>"India", "readonly"=>"no"}, "env"=>{"time"=>["weekends", "night"], "ip"=>["127\\.\\.[0-9]+\\.\\.[0-9]+\\.\\.[0-9]+"]}, "permissions"=>{"inventory"=>["find"]}}
user and object attributes do not match with the rule
PeriodicEvent satisfied for ["weekends", "night"]
location matched with IP regex: 127\\.\\.[0-9]+\\.\\.[0-9]+\\.\\.[0-9]+

checking for rule: {"user_attributes"=>{"position"=>"Developer", "region"=>"USA"}, "object_attributes"=>{"region"=>"USA"}, "env"=>{"time"=>["office-hours"], "ip"=>["172\\.\\.[0-9]+\\.\\.[0-9]+\\.\\.[0-9]+", "192\\.\\.[0-9]+\\.\\.[0-9]+\\.\\.[0-9]+"]}, "permissions"=>{"inventory"=>["insert"], "profiles"=>["find", "insert"]}}
user and object attributes do not match with the rule
PeriodicEvent satisfied for ["office-hours"]
location not matched with IP regex: 172\\.\\.[0-9]+\\.\\.[0-9]+\\.\\.[0-9]+
location not matched with IP regex: 192\\.\\.[0-9]+\\.\\.[0-9]+\\.\\.[0-9]+

Permission denied to user Bob for find on inventory

```

Fig. 6.
Server logs for Bob's query



```
eeshan@eeshan-ROG: ~  
> db.inventory.find()  
Error: error: {  
  "$err" : "UnauthorizedError: not authorized to execute command query on collection  
  inventory of database test",  
  "code" : 13  
}
```

Fig. 7.
Query to the server made by Kate

```

eeshan@eeshan-ROG: ~/code/ABAC
wire message intercepted: {:flags=>0, :numberToSkip=>0, :numberToReturn=>0, :database=>"test", :collection=>"inventory", :query=>{}, :returnFieldSelector=>nil, :header=>{:messageLength=>48, :requestID=>12, :responseTo=>0, :opCode=>:query}}
user_name: Kate
ip: 127.0.0.1
current time: 2021-04-24 22:50:22 +0530
current day: Saturday
user attributes for Kate not found locally
digital certificate verified by /DC=org/DC=CAA/CN=Central Attribute Authority with serial 2
collection: inventory
opcode: query
op: find
user attributes fetched: {"position"=>"Manager", "region"=>"WestCoast", "abc"=>"xyz"}
object attributes fetched: {"region"=>"India", "readonly"=>"no"}

checking for rule: {"user_attributes"=>{"position"=>"Manager", "region"=>"India"}, "object_attributes"=>{"region"=>"India", "readonly"=>"no"}, "env"=>{"time"=>["weekends", "night"], "ip"=>["127\\.\\.[0-9]+\\.\\.[0-9]+\\.\\.[0-9]+"}], "permissions"=>{"inventory"=>["find"]}}
user and object attributes do not match with the rule
PeriodicEvent satisfied for ["weekends", "night"]
location matched with IP regex: 127\\.\\.[0-9]+\\.\\.[0-9]+\\.\\.[0-9]+

checking for rule: {"user_attributes"=>{"position"=>"Developer", "region"=>"USA"}, "object_attributes"=>{"region"=>"USA"}, "env"=>{"time"=>["office-hours"], "ip"=>["172\\.\\.[0-9]+\\.\\.[0-9]+\\.\\.[0-9]+", "192\\.\\.[0-9]+\\.\\.[0-9]+\\.\\.[0-9]+"}], "permissions"=>{"inventory"=>["insert"], "profiles"=>["find", "insert"]}}
user and object attributes do not match with the rule
PeriodicEvent satisfied for ["office-hours"]
location not matched with IP regex: 172\\.\\.[0-9]+\\.\\.[0-9]+\\.\\.[0-9]+
location not matched with IP regex: 192\\.\\.[0-9]+\\.\\.[0-9]+\\.\\.[0-9]+

Permission denied to user Kate for find on inventory

```

Fig. 8.
Server logs for Kate's query

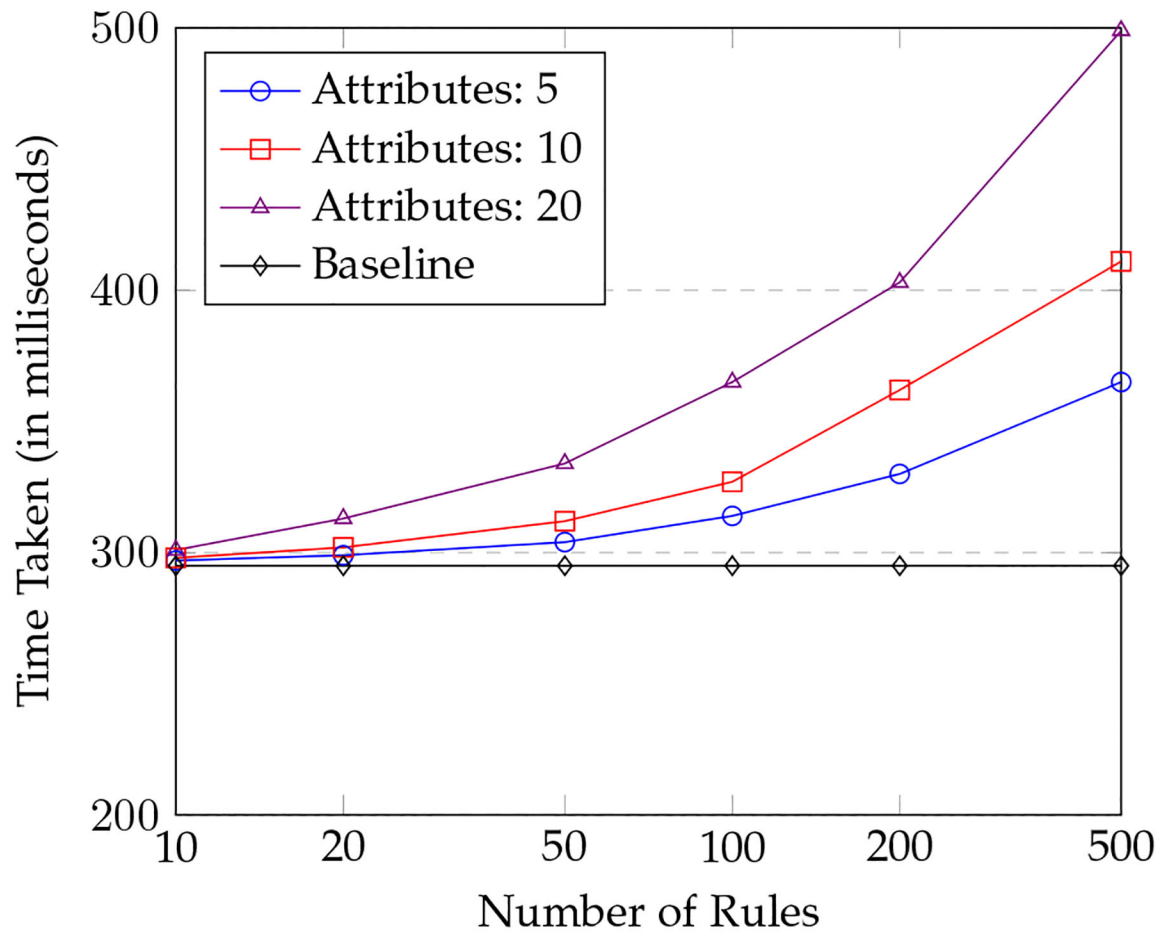


Fig. 9. Execution Time (in milliseconds) of *count* query for varying number of rules and user attributes when access is granted.

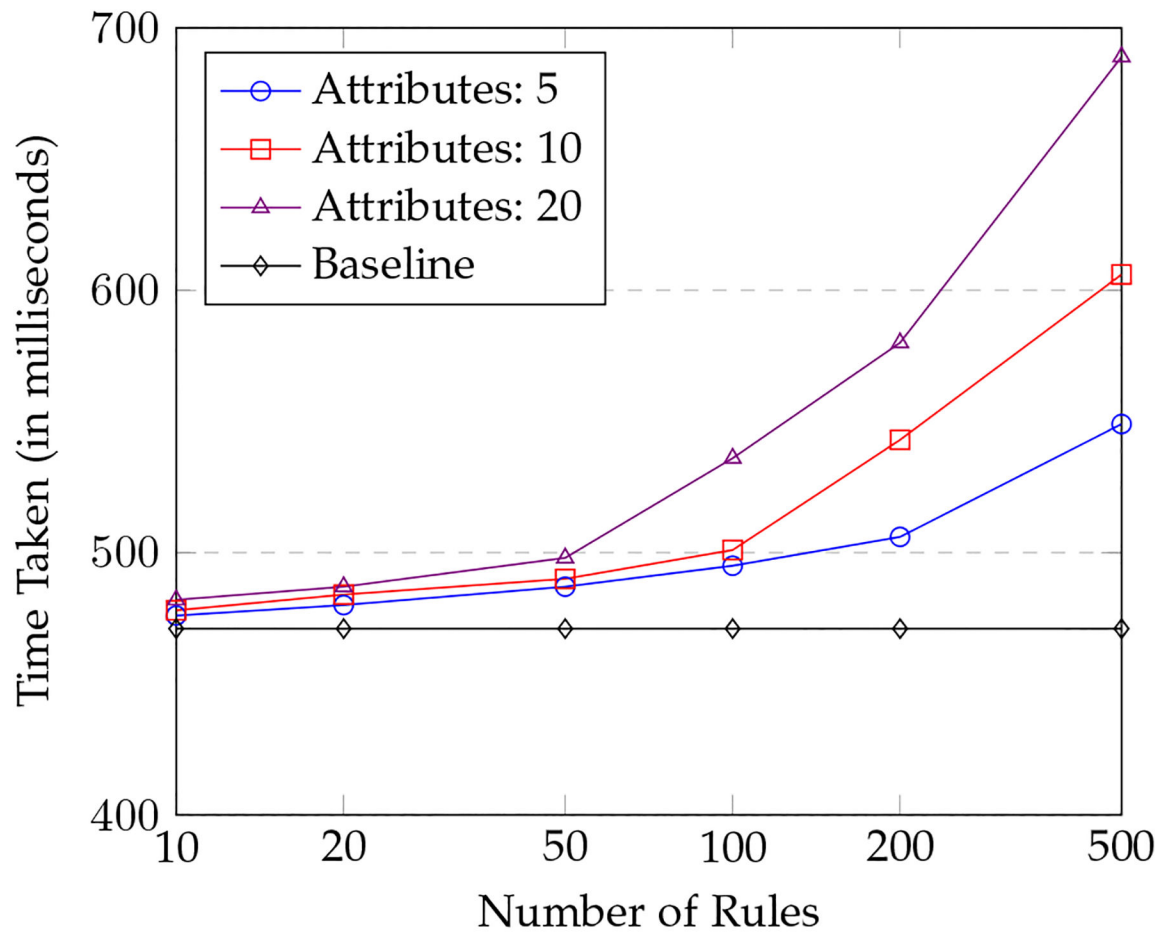


Fig. 10.

Execution Time (in milliseconds) of *find* query for varying number of rules and user attributes when access is granted.

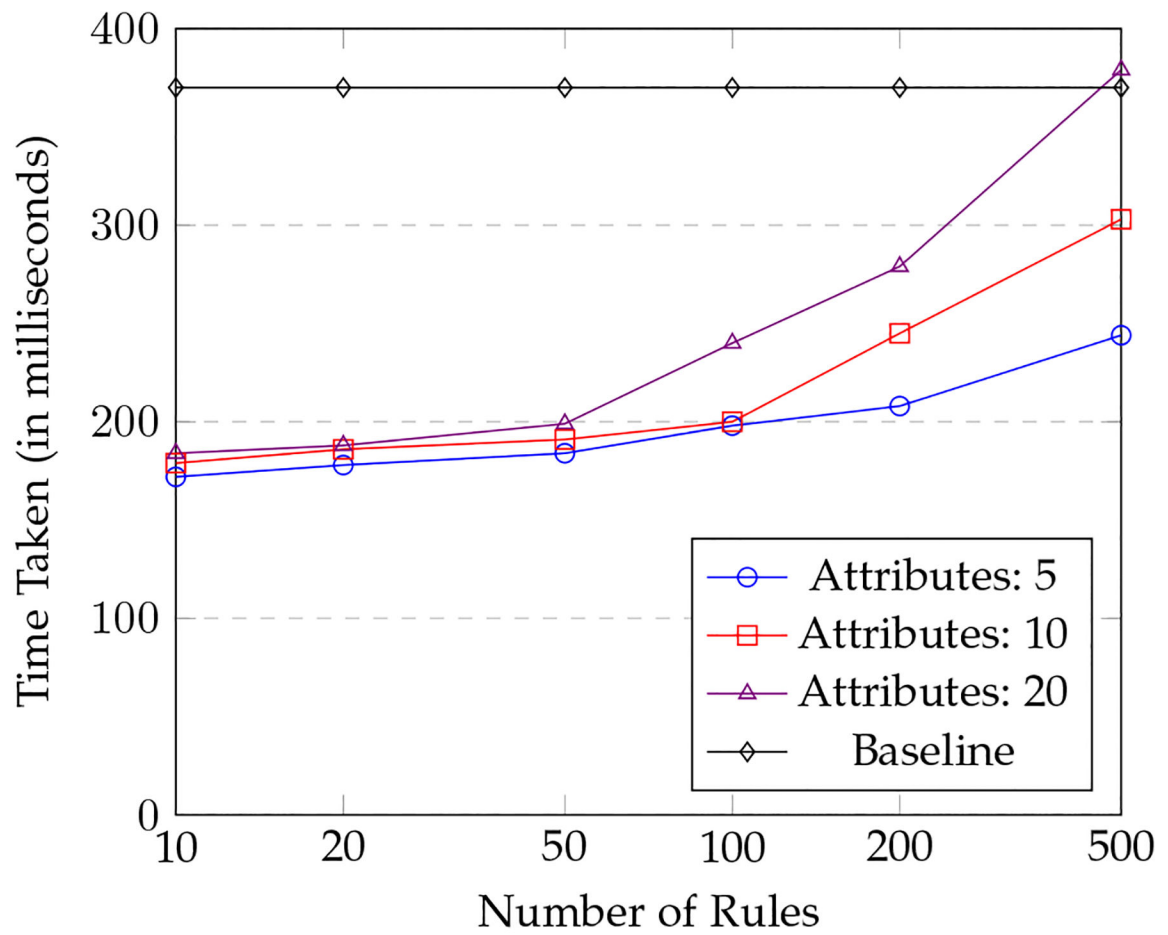


Fig. 11. Execution Time (in milliseconds) of *insert* query for varying number of rules and user attributes when access is denied.

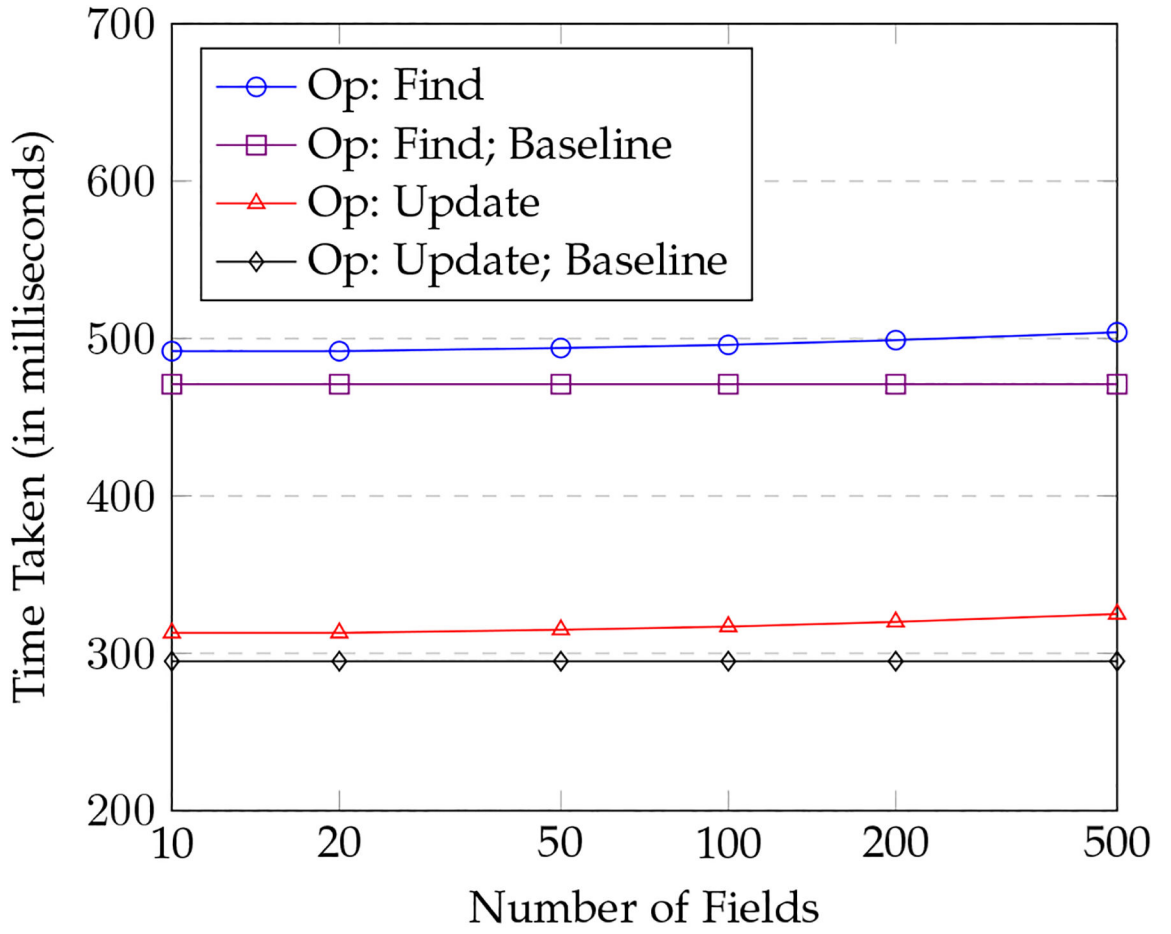


Fig. 12. Execution time (in milliseconds) of *find* and *update* queries for varying number of fields in the ABAC policy.

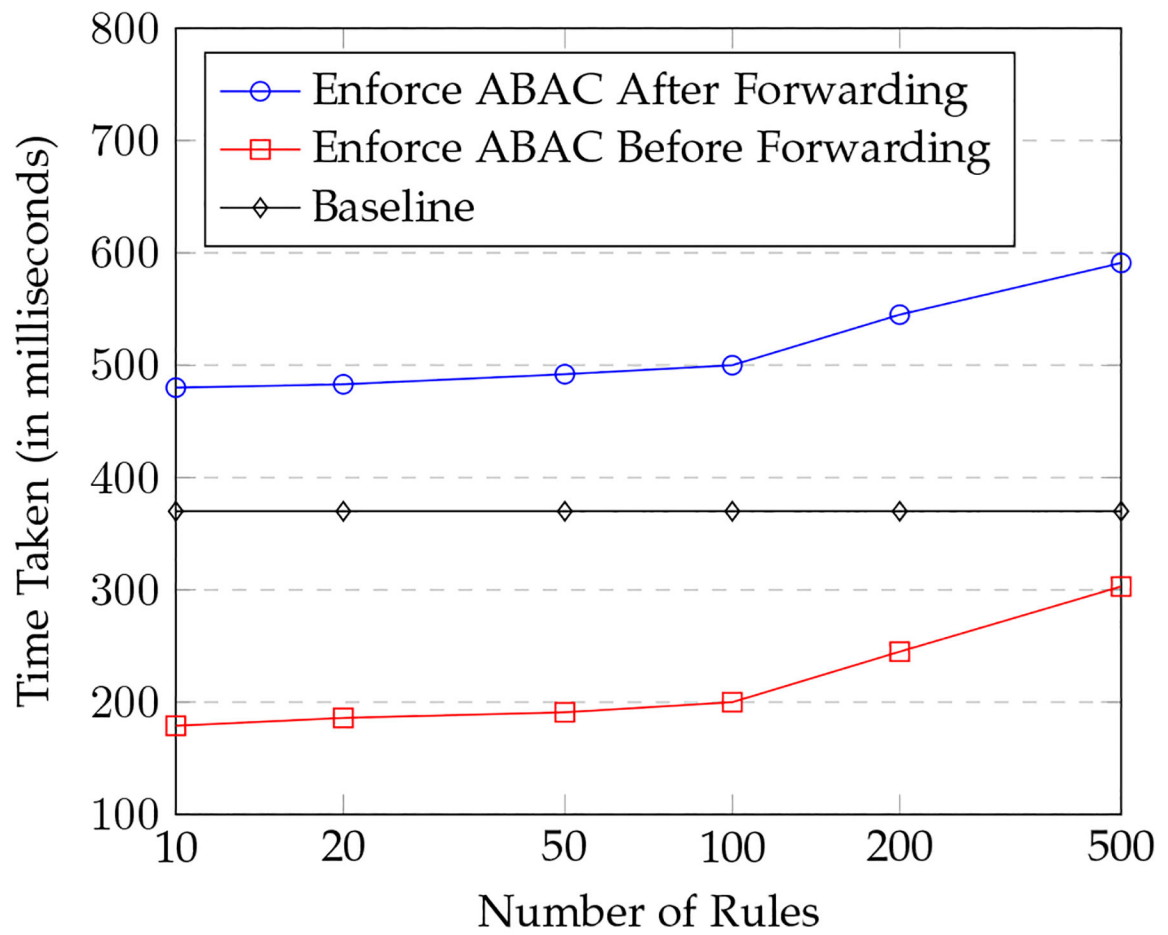


Fig. 13. Execution time (in milliseconds) of a query operation for two different architectures when access is denied.