Journal of Information Security and Applications 65 (2022) 103120

Contents lists available at ScienceDirect

Journal of Information Security and Applications

journal homepage: www.elsevier.com/locate/jisa

Security policies by design in NoSQL document databases

Carlos Blanco [a,b,*], Diego García-Saiz [a], David G. Rosado [b], Antonio Santos-Olmo [c], Jesús Peral [d], Alejandro Maté [d], Juan Trujillo [d], Eduardo Fernández-Medina [b]

[a] ISTR Research Group, Department of Computer Science and Electronics, University of Cantabria, Spain
[b] GSyA Research Group, Technologies and Information Systems Department, University of Castilla-La Mancha, Spain
[c] I+D+i Department, Marisma Shield S.L and Sicaman Nuevas Tecnologías S.L., Tomelloso, Spain
[d] LUCENTIA Research Group, Languages and Computing Systems Department, University of Alicante, Spain

ARTICLE INFO

Keywords:
Conceptual modelling
Big Data
NoSQL
Document databases
Security

ABSTRACT

The importance of data security is currently increasing owing to the number of data transactions that are continuously taking place. Large amounts of data are generated, stored, modified and transferred every second, signifying that databases require an appropriate capacity, control and protection that will enable them to maintain a secure environment for so much data. Big Data is becoming a prominent trend in our society, and increasing amounts of data, including sensitive and personal information, are being loaded into NoSQL and other Big Data technologies for analysis and processing. However, current security approaches do not take into account the special characteristics of these technologies, leaving sensitive and personal data unprotected and consequently risking considerable financial losses and brand damage. In this paper, we focus on NoSQL document databases and present a proposal for the design and implementation of security policies in this type of databases. We first follow the concept of security by design in order to propose a metamodel that allows the specification of both the structure and the security policies required for document databases. We also define an implementation model by analysing the implementation features provided by a specific NoSQL document database management system (MongoDB). Having obtained the design and implementation models, we follow the model-driven development philosophy and propose a set of transformation rules that allow the automatic generation of the final implementation of security policies. We additionally provide a technological solution in which the Eclipse Modelling Framework environment is employed in order to implement both the design metamodel (Emfatic) and the transformations (Epsilon, EGL). Finally, we apply the proposed framework to a case study carried out in the airport domain. This proposal, in addition to saving development time and costs, generates more robust solutions by considering security by design. This, therefore, abstracting the designer from both specific aspects of the target tool and having to choose the best strategies for the implementation of security policies.

1. Introduction

Large amounts of data are routinely generated and transferred by a multitude of mobile applications, social networks, scientific and professional tools and websites. Current tools and technologies must be able to handle the scale, speed, variety and complexity of these massive data sets [1,2]. Moreover, data is an important and essential asset for companies, public organisations and institutions. Suitable protection must, therefore, be guaranteed by making an effort to incorporate adequate security or privacy measures [3–9] so as to protect these data correctly.

One of the database technologies currently used is that of NoSQL datastores, which are employed in the data management back-end of several ICT applications such as Big Data owing to their high levels of scalability, performance and consistency [10,11]. However, despite their undeniable benefits, they have certain vulnerabilities and there are some obstacles to their widespread adoption, particularly with regard to security [12]. In fact, when NoSQL databases were initially designed, security was not considered to be an important feature [13], and the security is, therefore, provided on the middleware and operating system levels [14].

* Corresponding author at: ISTR Research Group, Department of Computer Science and Electronics, University of Cantabria, Spain.
E-mail addresses: Carlos.Blanco@unican.es (C. Blanco), Diego.Garcia@unican.es (D. García-Saiz), David.GRosado@uclm.es (D.G. Rosado), asolmo@sicaman-nt.com (A. Santos-Olmo), jperal@dlsi.ua.es (J. Peral), amate@dlsi.ua.es (A. Maté), jtrujillo@dlsi.ua.es (J. Trujillo), Eduardo.Fdezmedina@uclm.es (E. Fernández-Medina).

https://doi.org/10.1016/j.jisa.2022.103120

Available online 2 February 2022

Researchers and practitioners have, over the last decade, come to recognise that security features should be incorporated by means of a structured and systematic approach. This approach should be perfectly coupled with the design and other requirements of the system by combining principles from both software and security engineering [3,15], in what is known as “Security by design". One possible approach that has been widely used to incorporate security features throughout the life cycle of a system’s development is that of model-driven development [16–18]. The model-driven approach has been successfully applied during the development of secure systems, thus allowing security aspects to be integrated and automatically transformed at all levels of modelling. These ideas should also be used in secure NoSQL databases and in systems that make use of this NoSQL technology. One of these is Big Data, in which security efforts are currently focused on providing specific solutions at the implementation level [19–23], when an early identification of security requirements would generate more robust and higher quality solutions [24].

Furthermore, NoSQL databases are designed on the basis of different data models. These models can be broadly classified into four categories: key–value, column, graph-oriented and document [25]. According to the DB-Engines Ranking [26], MongoDB ranks as the most popular NoSQL datastore, in addition to which it uses a document-oriented data model. The objective of MongoDB is to increase the horizontal scalability of the data layer with a simple development complexity and the possibility of storing large amounts of data. NoSQL performs very impressively when it is necessary to scale through the use of several machines, and MongoDB is one of the NoSQL databases with the highest performance rates [27]. The aforementioned reasons had a great influence on our decision-making process when initially considering the possibility of focusing on document databases. Our intention was to use MongoDB as a technological tool for their implementation in such a way that we could also include the other data models in the future.

A recent version of MongoDB is MongoDB Enterprise Advanced, which was released in 2021 [28]. It has advanced security features that support the CIA triad (Confidentiality, Integrity and Availability), such as LDAP authentication and authorisation, Kerberos support, the encryption of data at-rest, FIPS-compliance, and the maintenance of audit logs. These capabilities extend MongoDB’s already comprehensive security framework, which includes Role-Based Access Control, PKI certificates, Read-Only views for field-level security and TLS data transport encryption. All these measures indicate the work and evolution that is taking place in NoSQL technologies to make them more secure. These measures cannot, however, be incorporated into all phases of the design and development of a database, since these solutions are employed at the implementation level, once the database has been built. This is one of the reasons why it is necessary to provide new proposals with which to incorporate security not only at the implementation level but also into the design, and to be able to develop more secure and robust NoSQL databases.

This paper provides a proposal for the design and implementation of security policies for NoSQL document databases. Its main contributions are the following:

- The definition of a metamodel for the design of document databases that will make it possible to establish security policies in an appropriate manner and relate them to the structural elements of this type of systems. Designers will be able to use this metamodel in order incorporate security policies from early stages (security by design), thus abstracting them from the technical aspects of specific tools.
- The analysis of the security policy implementation capabilities offered by a specific document database manager (MongoDB), the creation of its corresponding implementation model and the definition of the most appropriate implementation strategies for the casuistry found in the security policies.
- The inclusion of our proposal in a model-driven development approach, along with the definition of transformations between the design model and its corresponding implementation in MongoDB. This approach improves development times and costs, while facilitating its future extension to other tools and platforms.
- The creation of a technological solution for our proposal on the basis of the Eclipse Modelling Framework tools, along with the implementation of our design metamodel in Emfatic and the necessary transformations in Epsilon (EGL).
- The application of our proposal to a case study concerning an airport. This allows all the stages of the process to be observed, from the moment at which the designer models the system and the necessary security policies at the design level, to that at which the model is automatically transformed to its corresponding implementation in MongoDB. Verification that the solution obtained effectively satisfies the security policies defined in the design model.

The remainder of this paper is structured as follows: Section 2 presents an overview of our proposal. Section 3 shows the proposed metamodel with which to design NoSQL document databases, including the structural aspects and security policies. Section 4 presents the implementation model for a NoSQL document database management tool, MongoDB, while Section 5 shows the set of transformation rules (for both structural elements and security policies) that allows us to convert our design models into the implementation in MongoDB. Section 6 then presents a case study carried out with a dataset concerning passenger management at an airport, showing all the stages of our proposal (design model, automated transformation and verification that the security policies modelled are effectively satisfied in the final implementation). Section 7 discusses the proposals related to NoSQL database security and secure development that could contribute to solving the problem presented herein. Finally, the main contributions and our directions for future work are explained in Section 8.

2. Proposal overview

This section describes our proposal for the design and implementation of security policies for document databases (Fig. 1).

The components of our proposal can be summarised as follows. We first defined a specific metamodel for document databases at a level that allows the abstraction of the technical concepts of the database manager in which the system will eventually be implemented. This metamodel allows the modelling of the structural aspects of document databases (collections, simple fields, compound fields, indexes, etc.). A role-based access control system, together with a set of security policies that establish what actions can and cannot be performed on the collections and fields of the database, can also be modelled. Our proposal, therefore, includes security by design, thus providing independence from technology and implementation constraints. This means that these restrictions are taken into account in the decisions to be made in the subsequent development phases, thus favouring the higher quality and robustness of the final solutions. This strategy is an improvement in the traditional development process, in which security policies are usually incorporated once the database has been built. This later inclusion limits the security capabilities that exist in the database manager, and means that modifications cannot be made to an already planned design (or that those which are made generate extra effort and cost).

We then followed the model-driven development philosophy to design a set of transformation rules with which to automate the final process of implementing the database in a specific tool. The characteristics and limitations of the target tools made it necessary to first define an implementation model for the target tool and to carry out an analysis of the most appropriate implementation strategies (views, projections, etc.). This information was then used to define the transformation rules required in order to generate the corresponding implementation in the

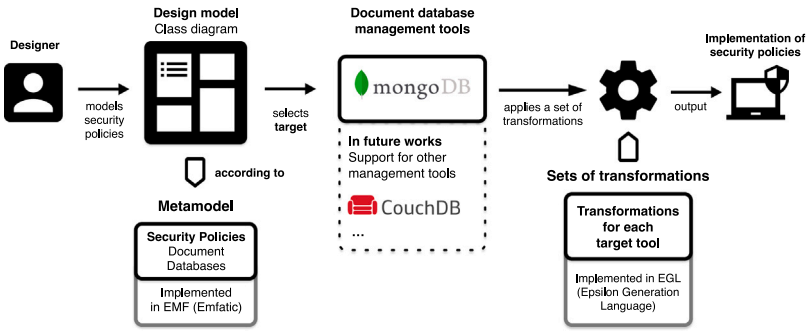

Fig. 1. Overview of our proposal.

target tool, which is based on the design model. This implies savings as regards costs and development time, while improving the final quality, thus ensuring that the established security policies are properly implemented. In this paper, we have considered MongoDB as the target database manager, as it is the most widely used in NoSQL databases. In this sense, we have designed a set of transformations to generate the corresponding MongoDB implementation from the design model. The incorporation of other implementation options in our proposal (other document database managers such as CouchDB) is identified as future work, being necessary to define a new set of transformations for each new manager to be incorporated.

Once the models and the transformations required to generate their corresponding implementation had been defined, it was necessary to create a technological solution with which to support the model-driven development process. This was done using the tools provided by Eclipse (the Eclipse Modelling Framework), implementing the metamodel in Ecore (using the Emfatic textual syntax) and the design model transformations in Epsilon (EGL).

From the designer's point of view, the steps required in order to use our proposal are the following. First, the designer creates a design model of the system, including both the structural aspects and the necessary security policies. This model is defined according to our metamodel. The designer then selects which target tool s/he wishes to use (in this case, MongoDB), and the corresponding set of transformations is executed. This automatically generates the implementation of both the structure and the security policies of the document database. This approach facilitates the designer's work, since we employ a simpler design model that frees him/her from the need to know technical aspects of specific tools and from having to make decisions regarding correct implementation strategies, since the transformations are in charge of managing these aspects.

3. Modelling secure policies in document databases

This section shows our proposal for a metamodel for document databases that allows the designer to model the document database at a level of abstraction that is independent of the technical implementation details (Fig. 2). This metamodel includes both the structural aspects specific to this type of systems and security elements that allow the establishment of security policies that are adjusted to its characteristics. We have also designed a set of restrictions with which the models must comply, which were created in accordance with this metamodel and will be defined at the end of this section (Table 1).

The elements considered in the metamodel have been identified through an analysis of the needs in the design of document databases, as well as an analysis of the characteristics offered by the management tools of this type of databases for their implementation. From a higher level of abstraction (conceptual level) we identified the concepts needed to represent the structure and security of a document databases.

In order to specify the structure of the document database we need to include elements such as collections, fields, identifiers or data types. Furthermore, we also need to include one aspect that this kind of systems considers and that is the possibility of defining simple and composed fields, so that in turn, composed fields can be formed from other simple or composed fields. This modelling problem can be addressed with a classical design pattern called composite [29]. This pattern proposes its modelling with a generalisation (Field) specialised in simple elements (SimpleField) and composed elements (ComposedField), and an association that indicates that each composed element can be formed in turn by several Fields (SimpleField or ComposedField).

On the other hand, in order to establish the security policy of the system, we need to use an access control method that allows us to classify the subjects and objects of the system and to associate security privileges to them. In this sense, we have opted for a role-based access control (RBAC) as it is widely accepted in the industry and it is used by practically all document database management tools. In this sense, users are grouped into roles that define security privileges on certain elements (collections or fields) and certain actions (the usual ones in databases: read, insert, update, delete). To facilitate the designer's work and avoid conflicts between security rules, we consider an open-world system where the designer has to specify only the security rules that deny certain privileges on certain elements to certain roles. Our proposal will then generate at the implementation level the positive and negative security rules needed to implement the system. From a lower level of abstraction, we analyse the features offered by document database management tools for the implementation (of both the structure and security of the system) and move up the abstraction level to obtain a common metamodel. Our metamodel contains the necessary information to generate the corresponding implementation in concrete tools, such as MongoDB. For instance, in this analysis we have identified structural aspects such as collection ids, data types, etc. and security aspects such as the security actions that can be applied (hide instances, hide fields, etc.).

Now, the proposed metamodel are described in more detail. The structural aspects (represented at the top of the figure) make it possible to model a document database (Database) through the use of a set of collections (Collection), which can, in turn, contain a set of fields (Field). For each field, it is possible to indicate the associated information regarding whether it is obligatory (required) and or it has intrinsic restrictions (constraint). Each field can, in turn, be simple (SimpleField) or composed (ComposedField). A simple field represents an attribute or a property with a value, which must be one (or various) of the types of data indicated as being permitted for that field (DataType). For example, a personal identification field can permit values of the type "string" and "int". It is also necessary to highlight that there are two special types of data: the arrays, which make it possible to represent sets of values, or the enumerations, which make it possible to establish a set of possible values. In our proposal, these values are indicated by means of a restriction of the field. It is also possible to define composed fields (ComposedField), which are structures formed of various fields, be they simple or composed. This concept is represented in the model by applying the Composite design pattern to the Field, SimpleField and

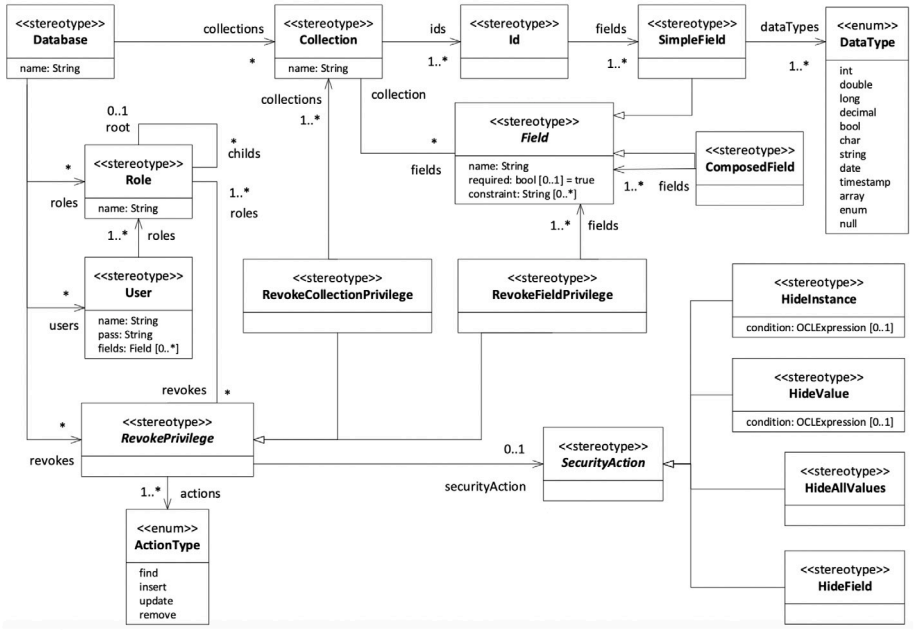

Fig. 2. Metamodel for secure policies in document databases.

ComposedField classes. Furthermore, each collection is identified by one or various identifiers (Id), which are, in turn, formed of one or various simple fields (SimpleField).

The other elements in our metamodel (bottom of the figure) are related to aspects of security, which allow us to establish security policies regarding the structural elements of the document database. We have employed a role-based access control (RBAC) policy for this task, since it allows us to establish security policies in a manner appropriate to this type of systems. Moreover, this strategy is chosen and supported by practically all document database management tools. Our metamodel makes it possible to establish a hierarchy of roles by making use of the Role metaclass and of its references to the parent (root) and children (children) elements. The User metaclass can then be employed to define specific users to whom a set of roles is assigned, such as a name, a password and, if necessary, additional fields of information.

We then establish a set of security policies that correctly manage the user privileges. This is done by considering an open-world approach in which all actions are permitted unless there is a security policy that explicitly withdraws permission for specific actions to be carried out. The open-world approach is appropriate for document databases, since these databases are oriented towards the massive use of data and it is, by default, necessary to allow access to them. Our metamodel has a metaclass whose purpose is to define the security privilege (RevokePrivilege), which allows the designer to indicate which policies for which objects are revoked for which subjects. The possible privileges are reflected in the ActionType enumeration and correspond with the reading (find), creation (insert), modification (update) and delete (delete) operations.

The objects that are affected by the security policy can be used as a basis on which to differentiate two types of rules: those that affect the complete collections and those that affect their fields. These can be considered as fine-grained rules that make it possible to represent more specific security constraints. In order to represent these two types of privileges and their singularities, the RevokePrivilege metaclass has been specialised into RevokeCollectionPrivilege and RevokeFieldPrivilege. A security policy defined at the collection level, therefore, allows us to remove policies for a set of roles concerning certain actions (read, create, modify or delete). For example, in the case of a very simple database related to purchases with Customer and Order collections, we could withdraw from a certain role the actions of creation, modification and deletion as regards the Order collection.

With regard to the fine-grained security policies, the main difference (apart from the type of objects that they affect, which are now fields rather than collections) lies in the fact that they also make it possible to establish a greater variety of security actions (securityAction), which will be carried out if the rule that has been defined is fulfilled. It is possible to: hide the values affected (the cells) by writing a null value in their place using the HideValue action; hide all the values in this field (by writing null in all the cells in that field) using HideAllValues; hide the field affected by all the instances (the complete column) using HideField, or hide the instances affected (the rows) using HideInstance. This last action (HideInstance) can also be used in policies at the collection level. Moreover, some of these security actions make it possible (by using an OCL expression) to establish a condition for the data, which will be evaluated in execution time.

To continue with the previous example, let us now focus on a hypothetical field that indicates whether a customer's request is confidential (by means of a Boolean value of true or false) and use this as a basis on which to describe the effects of applying one of these security actions:

- **hideValue** would modify all those fields referred to by null values in the rule. It would, for example, be possible to refer the rule to the fields "names" and "surnames" and to hide those values if the condition of the request being confidential was fulfilled.
- **hideAllValues** would write a null value in all those cells that referred to the field affected by the rule. We could, for example, focus on the confidential field and change all the values for null values, such that it would not be possible to infer which were confidential and which were not.
- **hideField** would act in a similar manner, but would completely hide the confidential field, such that we would see all the requests but would not know of the existence of the confidential field.
- **hideInstance** would hide the complete instance, such that if we were to establish the need to hide confidential requests as a condition, it would show only those instances and data that were not confidential.

The proposed metamodel is complemented with a series of restrictions that all the models specified must satisfy in accordance with the metamodel. These restrictions are shown in Table 1, together with their implementation in OCL, where CO1 is a structural restriction whose purpose is to seek uniqueness as regards the names of certain

Table 1
Restrictions that must be fulfilled according to the metamodel.

Constraints	
C01	Uniqueness of names. Two Role, User, Collection or Field elements with the same name cannot exist in the same Collection (same value in the name field).
	Context Database invself.roles->isUnique(name) *Context Database invself.users->isUnique(name)* *Context Database invself.collections->isUnique(name)* *Context Collection invself.fields->isUnique(name)*
C02	If a RevokePrivilege does not contain the action "find" in its actions field, it will not have an associated securityAction.
	Context RevokePrivilege *if self.actions->select(a|a="find")->size()=0* *then self.securityAction->size()=0* *enf if*
C03	If a RevokePrivilege of the type RevokeCollectionPrivilege has an associated securityAction, this must be of the HideInstance type. However, a RevokePrivilege of the type RevokeFieldPrivilege may have an associated securityAction of any type.
	ContextRevokeCollectionPrivilegeinv *self.securityAction->isEqual(HideInstance)*
C04	A RevokeFieldPrivilege cannot exist in a field of a collection X and a role R if there is a RevokeCollectionPrivilege for that collection X that revokes the action "find" for the role R.
	Context RevokeCollectionPrivilege *body* *RevokeCollectionPrivilege::allInstances()* *->forAll(rcp | rcp.actions->select(a|a="find")->size()=1* *implies* *RevokeFieldPrivilege.allInstances()->select(rfp|* *rcp.roles->select(r|rfp.roles->contains(r))->size=0 and* *rcp.collections->select(c|rfp.fields.collection->contains(c))* *->size()=0)*

Table 2
Implementation of metamodel for document databases.

```
class Database {attr String[1] name; val Collection[*] collections;
       val Role[*] roles; val User[*] users;
       val RevokePrivilege[*] revokes;}
class Collection {attr String[1] name; val Id[+] ids;
       val Field[*] #collection fields;}
abstract class Field {attr String[1] name;
       attr boolean[?] required = true; attr String[*] constraints;
       ref Collection[1] #fields collection;}
class SimpleField extends Field {attr DataType[+] dataTypes;}
class ComposedField extends Field {ref Field[+] fields;}
class Id {ref SimpleField[+] fields;}
class Role {attr String[1] name; val Role[*] #root child;
       ref Role[?] #child root;
      val RevokePrivilege[*] #roles revokes;}
class User {attr String[1] name; attr String[1] pass;
       ref Role[*] roles; ref Field[*] fields;}
abstract class RevokePrivilege{
       ref Role[+] #revokes roles; attr ActionTypes[+] actions;
       attr String[?] condition; attr ActionType[+] actions;
       attr SecurityAction[?] securityAction;}
class RevokeCollectionPrivilege extends RevokePrivilege {
       ref Collection[+] collections;}
class RevokeFieldPrivilege extends RevokePrivilege {
       ref Field[+] objects;}
enum DataType {int; double; long; decimal; bool; char; string;
       date; timestamp; enumerate; array; null;}
enum ActionType {find; insert; update; remove;}
abstract class SecurityAction {}
class HideInstance extends SecurityAction {
       attr OCLExpression[?] condition;}
class HideValue extends SecurityAction {
       attr OCLExpression[?] condition;}
class HideAllValues extends SecurityAction {}
class HideField extends SecurityAction {}
```

elements. The purpose of the other restrictions (CO2, CO3 and CO4) is, meanwhile, to seek the correct definition of the security policies.

Finally, our metamodel has been implemented in Ecore (Eclipse Modelling Framework) using the Emfatic textual syntax (Table 2).

4. Implementation model for MongoDB

This section shows the model of the document database system at a level of detail that allows its implementation in a specific document database manager: MongoDB. This model describes the structural and security elements available in the MongoDB document databases, and its objective is to assist the automatic transformation process of the database. We have consequently established the most appropriate implementation strategy for each of the elements that were defined in a more abstract manner in the previous model. Our proposal, therefore, frees the designer from the need to know the lowest level of details regarding implementation in specific database managers and from having to select the best strategy, which may lead to security errors if not performed correctly. The implementation model is described by separating its elements into two models: one containing the structural aspects (Fig. 3) and the other containing the security aspects (Fig. 4).

With regard to the structural part (Fig. 3), note that the principal element of the database (Database) is the collection (Collection) whose structure is described on the basis of the properties of which it is composed (Property) and whether or not they are required. Each property can, in turn, be defined as: a simple field (SimpleField) with a type of associated data; an enumeration (EnumField) with a set of permitted values; an array of values of a particular type (ArrayField), or a composed field (ComposedField) with various properties that can, in turn, be simple, enumerated, arrays or composed fields. Each of these properties can be associated with expressions that have conditions that must be fulfilled (Condition). There are also the indices (Index), which have been defined for a particular collection and are composed of a set of properties. The use of these elements makes it possible to implement the structural aspects defined in our metamodel: databases, collections, ids, simple and composed fields, constraints, data types, etc.

With regard to the security aspects (Fig. 4), on the one hand there is the concept of view (View) for a collection. If it is not desirable for the view to contain all the properties in the collection, then it is possible to carry out a projection indicating which properties belong to the view and which do not. It is also possible to establish conditions for each property. It would, for example, be possible to show only a customer's name, surname and age, and not their address. On the other hand, if it is desirable for the view to contain only certain instances, which makes it possible to establish expressions containing the condition to be evaluated. It would, for example, be possible to indicate that only customers over 18 years of age can be shown. The combination of these two strategies permits the implementation of several of the security actions defined in the metamodel: hiding fields, instances or fields and instances.

It is also possible to establish a hierarchy of roles (Role) and to indicate the privileges that each role has on the basis of which actions (find, insert, update, remove) are authorised for which resource (collections or views). Finally, it is possible to define a set of users (User) of the document database manager, indicating their usernames and passwords, and assigning each of them to one or more roles from which they will inherit their security policies.

The use of these elements enables the implementation of the RBAC system defined in the metamodel, along with the authorisation rules. The correct implementation of the security policies defined in the metamodel is not, however, a trivial task, since it must be done on the basis of the elements available in the final tool, MongoDB. It is, therefore, necessary to establish a strategy based on the creation of roles and views, and to represent the semantics of the security policies (permitted actions, security actions, etc.) in the most appropriate manner.

5. Generating a secure implementation from models

The previous two sections show a metamodel for document databases with which to represent both structural and security aspects at a high level of abstraction. They also show an implementation model that represents the strategies present in document database management systems (in this case, MongoDB) at a low level of abstraction for the implementation of structural and security aspects.

This section shows the definition of the transformations required in order to employ the high-level document database models (defined in accordance with our metamodel) so as to automate the secure implementation process in a specific database manager. In this case, the tool considered is MongoDB. The transformations carried out will be grouped into those that manage the structural or security aspects of the database.

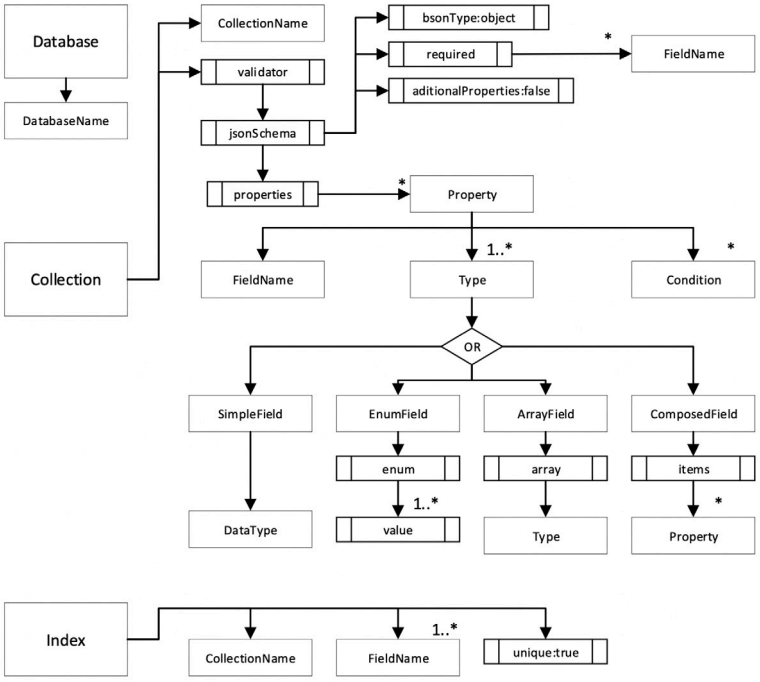

Fig. 3. Implementation model for MongoDB: structural concepts.

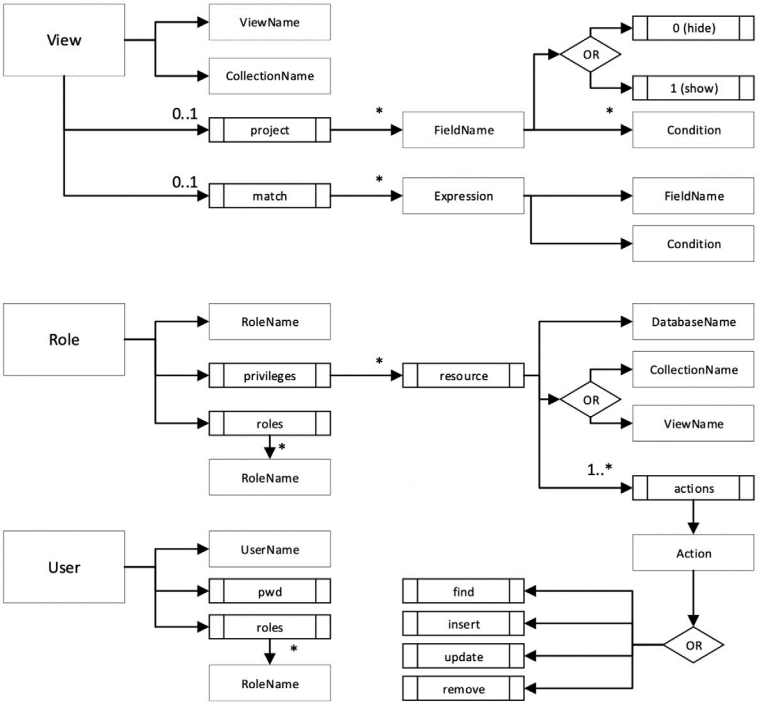

Fig. 4. Implementation model for MongoDB: security concepts.

5.1. Structural transformations

The first rules to be applied to the model are those in the set of transformation rules for the structural aspects (Fig. 5). These are in charge of generating the database, collections and fields in accordance with the information in the model. Once the database has been created (Database2Database transformation), the collections are processed, one by one (Collection2Collection transformation), in order to create both the corresponding collection and a JSON schema for validation purposes. The collection fields are then analysed in order to discover those that must be added to the validation schema and thus represent their restrictions (Field2Required transformation).

A detailed analysis of the fields in the collection is subsequently carried out (Field2Property transformation). A property containing all the information necessary as regards the name of the field, the restrictions to be applied and the types of data that are admitted (simple, enumeration, array or composed) is then created for each one. In the case of the admitted data type, the transformation is served by two auxiliary rules whose purpose is to process the simple and complex data types.

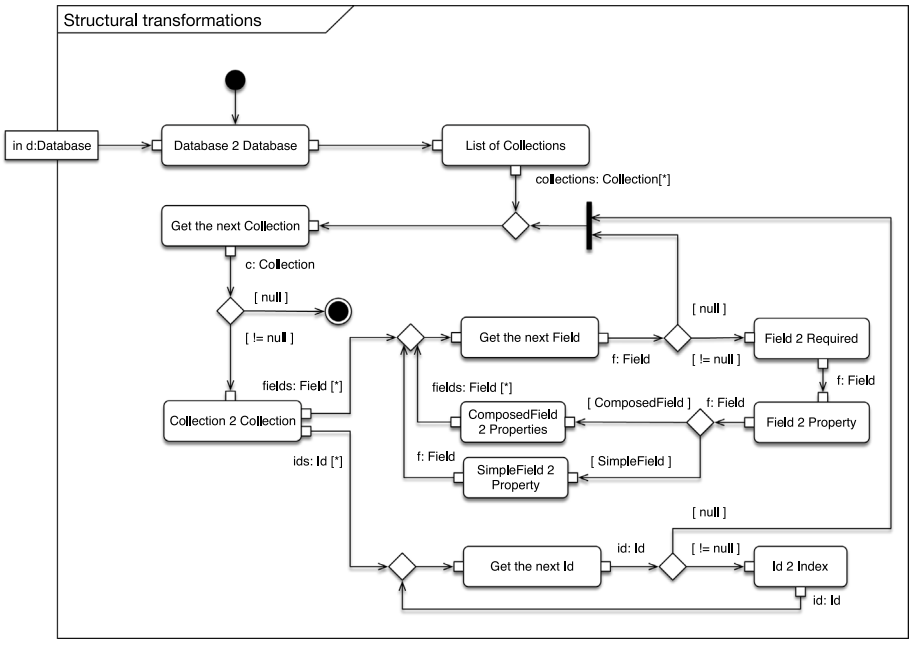

Fig. 5. Transformation rules (structural aspects).

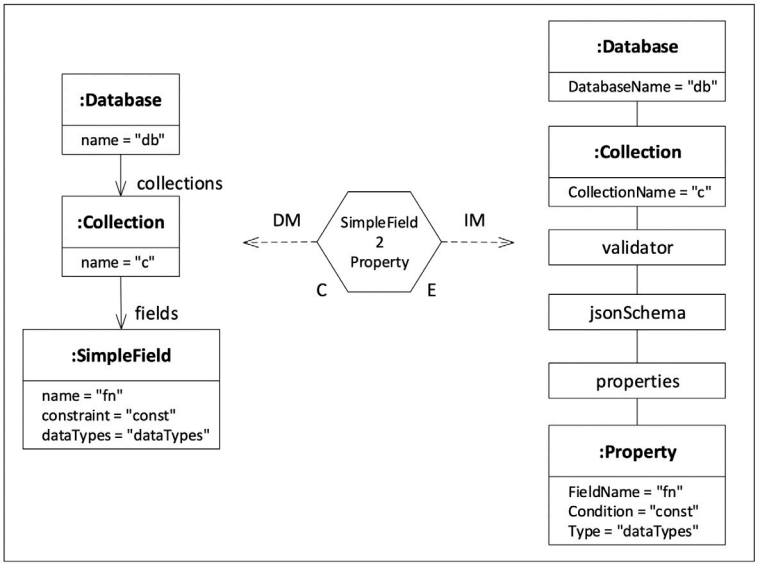

Fig. 6. Graphical notation for the SimpleField2Property relation.

The transformation in charge of processing the properties with simple data (SimpleField2Property transformation) is represented in Fig. 6 using the graphical syntax specifically proposed by the OMG Query View Transformation language. This makes it possible to see how the transformation seeks determined elements in the base model (on the left-hand side of the figure) and generates another set of elements in the output model (on the right-hand side of the figure). In this case, a simple transformation is shown. When provided with a Simple Field containing information related to its name, permitted data type and constraints that belongs to a determined collection and database, it generates a Property in the output model. This property contains said information and is also associated with said database and collection, although in this case it forms part of the validation schema of the collection. This transformation processes the fields containing a basic data type (int, char, String, etc.), an enumeration data type or arrays of basic data types.

The composed data types are processed in a similar manner (ComposedField2Properties transformation), although, since they are in turn composed of properties (simple or composed), each of them must be analysed anew (for the Field2Property transformation). Finally, the indices of each collection (Id2Index transformation) are created, and each collection may, in turn, have various indices composed of various simple fields.

5.2. Security policies transformation

Once the structural elements of the document database have been generated, the following set of transformations (Fig. 7) is in charge of defining the security elements required to ensure the correct fulfilment of the security policies defined in the model.

It is first necessary to analyse the hierarchy of the roles defined in the model in order to generate that configuration at the implementation level (Role2Role transformation). It is then necessary to discover

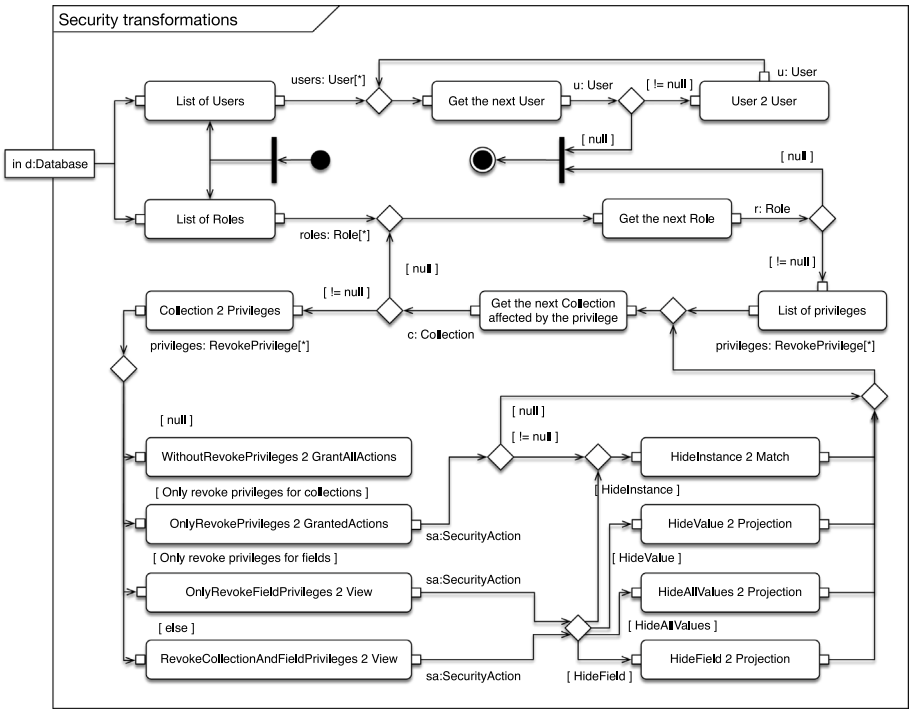

Fig. 7. Transformation rules (security policies).

whether associated security policies have been defined for each role in each collection, and what those policies are, in order to choose the best implementation strategy (Collection2Privileges transformation). This transformation is represented in greater detail in Fig. 8. Note that the transformation selects the set of policies (RevokePrivilege) associated with a specific role (rName) and a specific collection (cName) from the base model (left-hand side of the figure). This set of policies contains both policies at the collection level that refer to that collection and policies defined at the field level that refer to fields in that collection. In the output model (right-hand side of the figure), a resource to which more specific information will later be added (the creation of views, if necessary, the establishment of permitted actions, etc.) is generated for the appropriate role and collection. According to the types of policies that are associated, it is possible to find various situations in which different actions must be taken when establishing the implementation strategy required in order to respect these security policies. A specific transformation is available for each of these situations, such that once the situation in question is known, the generation of the code is delegated to that transformation (bottom of the figure).

The possible situations that could be encountered, along with the implementation strategies that must be applied, are the following:

- **Without security policies.** The role has no security policy associated with it for that collection (WithoutRevokePrivileges2GrantAllActions transformation). This is the simplest situation. In this case, it is necessary only to aggregate privileges regarding the resource that represents the collection with the information concerning the role, granting it all the actions (find, insert, update, remove).
- **Only security policies for collections.** The role has policies associated with the collection, but only with those policies (of the RevokeCollectionPrivilege type) that affect the collection in its entirety (OnlyRevokeCollectionPrivileges2GrantedActions transformation). In this case, similar action is taken, although permission is granted only for those actions that have not been revoked by the rule. If a security action of the HideInstance type is defined for the rule, then it will also be necessary to use an auxiliary rule (HideInstance2Match transformation) in order to implement a view with a filtering condition.

Table 3
Implementation of HideAllValues2Projection transformation.

```
[% for (db in document!Database) %]
[% for (c in db.collections) %]

// Collection
db.createCollection{
    "[%=c.CollectionName%]",{
        validator: {
            $jsonSchema: {
bsonType: "object",

// Required Fields
required: [
    [% for (sf in c.fields)]
    [% if(sf.required==true)%]
    "[%=sf.name%]",
]

//Properties
properties: {
[% for (sf in c.fields)%]
[% if(sf.isKindOf(SimpleField))%]

    //Field
    [%=sf.FieldName%]:{

    //Data types
        [%if(sf.dataTypes.contains(enum))%]
enum: [%=sf.condition%]

bsonType:
        [% for (dt in sf.dataTypes) %]
        [% if(dt.equals(enum)==false) %]
        "[%=sf.dataTypes%]",

    //Condition
        [%if(sf.constraint != null)%]
        {$cond: [%=sf.constraint%]}
    }
}}}}}
```

- **Only security policies for fields.** The role has policies associated with the collection, but they are only fine-grained policies (of the RevokeFieldPrivilege type) that affect fields in the collection (OnlyRevokeFieldPrivileges2View transformation). MongoDB permits several implementation mechanisms, and in this case, we select the mechanism employed for the creation of views. Views restrict the permitted action to solely “find” (and the other actions

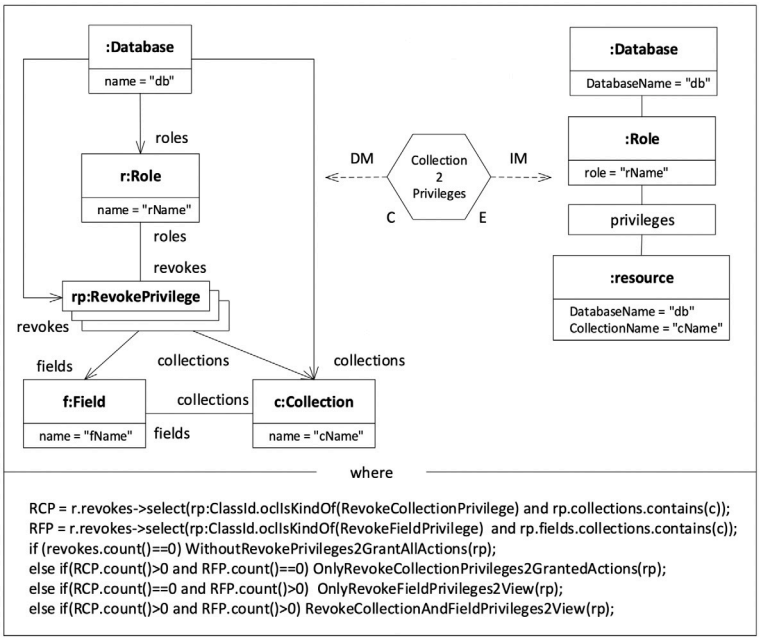

Fig. 8. Graphical notation for the Collection2Privileges relation.

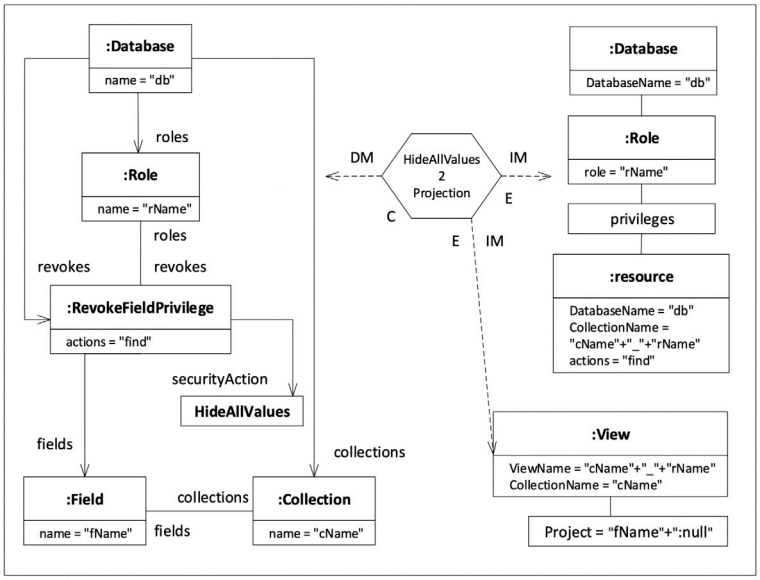

Fig. 9. Graphical notation for the HideAllValues2Projection relation.

will, therefore, be omitted), but allow us to establish the various security actions associated with the fine-grained rule by using "projections" (for the HideValue, HideAllValues or HideField security actions) or "match" (for the HideInstance security action). This is done using various auxiliary rules (HideInstance2Match, HideValue2Projection, HideAllValues2Projection and HideField2Projection transformations). Fig. 9 shows more details of one of these auxiliary rules (HideAllValues2Projection) which, given one or various fields, is in charge of carrying out a projection containing an indication that all the values are null.

– **Security policies for collections and fields.** The role has policies associated with the collection and they are of both types, at the collection and at the fine-grained levels (RevokeCollectionAndFieldPrivileges2View transformation). In this case, as we are limited by the implementation characteristics provided by MongoDB, it is necessary to follow a strategy similar to the previous one. We, therefore, establish a view in which we deal with the policies at the field level, along with their associated security actions. However, we omit the policies defined at the collection level, since the fine-grained policies that have been defined prevail over them.

In the last step, the specific users are aggregated with each of the roles defined (User2Role transformation). In this case, the name and password of each user and, optionally, a set of fields containing additional information related to the user, are indicated.

Having defined the transformations, we implemented them by using the Epsilon package, specifically EGL (Epsilon Generation Language). Table 3 shows an example of the implementation in EGL. In this example, it will be noted that various relations from the structural part of the document database in charge of generating the collections have been coded in the same EGL rule in order to indicate the fields required

and to process the simple fields. Other EGL rules would process the composed fields and generate the indices, just as another set of EGL rules would deal with the security aspects.

6. Case study

This section presents an example that allowed us to verify the applicability of our proposal. The example employed concerns a document database at an airport, and principally handles information regarding passengers and flights. Various security policies were, therefore, defined for it. This example will show how the three phases in our proposal were applied. It was first necessary (Phase 1) to model the database according to our metamodel (Section 6.1). It was then necessary (Phase 2) to apply the transformations defined in order to automatically generate a secure implementation (Section 6.2). Finally (Phase 3), we discovered that the solutions obtained respected the security policies defined (Section 6.3).

6.1. Design

The designer first modelled the document database using the metamodel defined in this paper. This made it possible to represent both the structural concepts (collections, simple and composed fields, identifiers, restrictions, etc.) and the security policies (configuration of roles, permissions for collections or fields, security actions, etc.). The model developed was then described, and was separated into that part of the model representing the structural aspects (Fig. 10) and that representing the security aspects (Figs. 11–13).

When focusing on the structural aspects (Fig. 10), it will be noted that various collections have been modelled. The passenger collection (Passenger) contains the passengers' personal information (name, address, age), along with information regarding whether or not they are suspicious (suspicious) and their level of risk (low, medium or high). It also contains information concerning the set of journeys in which each passenger is involved. Each journey is a collection (Trip) containing various associated data (the cost, seat number, whether they have checked in, whether they have boarded and the flight identification number). Each piece of baggage is, in turn, a collection containing identifying information: its weight, whether it has been inspected and whether it is suspicious.

The flights are represented in another collection (Flight), which contains information regarding the purpose of the flight (commercial or military), the passengers, the crew, the aircraft used, and the date and place of departure. These elements are, in turn, collections containing associated data, such as Place, which make it possible to know the gate, terminal, airport, city and country. Each flight collection additionally contains tracking information, which is represented as a set of tracking values obtained at specific instances. These tracking values are the date and the time, height, longitude, latitude and speed, and are grouped together in a composed field (TrackInfo).

The following figures correspond to the part of the model that specifies the security aspects (Figs. 11–13). We have defined a hierarchy of roles (Fig. 11), which is used to classify the users of the database and to define the security policies. There is a main role User that is specialised into passengers (Passenger) and the airport staff (Staff), which is, in turn, specialised into administrators (Admin) and security personnel (Security). The User and Staff roles have been defined as abstracts, such that it is possible only to create instances of the leaf nodes (Passenger, Admin and Security).

We define a series of security policies that we wish the document database system to satisfy (Figs. 12 and 13) .

The security policies are the following:

- **Security policy 1**. The only people who can carry out the insertion, modification and delete actions for the flight (Flight) and place (Place) collections are the administrators, while the read action can be carried out by any user. This has been modelled by defining a collection security policy (FlightAndPlaceInformation). This revokes permission from roles other than that of the administrator (Passenger and Security) as regards the insertion, modification and deletion of information concerning the collections in question (Flight and Place).
- **Security policy 2**. The passengers can consult information regarding flights only if the flights do not have a military purpose, while the airport personnel can consult information regarding all flights. We have defined a security policy (FlightPurpose) associated with the passenger role. This hides those flight instances (HideInstance security action) that fulfil the condition of being flights with a military purpose.
- **Security policy 3**. The information regarding the passengers (personal data, flights, baggage) can be consulted, inserted, modified or deleted only by members of the airport staff (administrators or security personnel) and never by the passengers themselves. We have established a security policy at the collection level (PassengerInformation) that revokes permission from the passengers as regards carrying out any type of action on the Passenger collection.

The fine-grained security policies specify security policy 3 in more detail. This means that it is defined at the collection level and indicates that the information concerning passengers can be processed only by the airport staff (administrators and security personnel). This specifically provides more limits as regards the information about passengers that the administrators can consult. The fine-grained security policies are the following:

- **Fine-grained security policy 3.1**. This policy specifies that, of the airport staff, the security personnel can consult information about all the passengers, but the administrators can consult information about only those that do not imply a security risk (who are not catalogued as being suspicious or high risk). They cannot see the names or addresses of the others (those that do imply a security risk). A security policy (PassengerSuspicious) has been established for the administration role. This acts on the name and address fields and hides the values in question (HideValue security action) if the condition of the passenger being suspicious or high risk is fulfilled.
- **Fine-grained security policy 3.2**. The information regarding the passengers' baggage can be consulted only by the security personnel and not by the administrators. A security policy (Baggage) has been established for the admin role that completely hides the baggage field (HideField security action). This security action allows administrators to consult the information regarding passengers and observe their journeys but not their associated baggage, or even know of its existence. That is to say, it is not the case that they will see various pieces of baggage with null values, but rather that they will not even know that there is an array of baggage associated with the journey.
- **Fine-grained security policy 3.3**. The administrators are not allowed to know the passengers' ages. A security policy (PassengerAge) has been established that acts on the administrators, modifying the values of the passenger age field with null values (HideAllValues security action).

6.2. Application of transformation rules for the automatic generation of the implementation

This subsection shows how the transformation rules were applied to the document database used in the airport case study (Figs. 10, 11, 12 and 13). Upon applying these transformations, we automatically obtain their corresponding implementation in MongoDB.

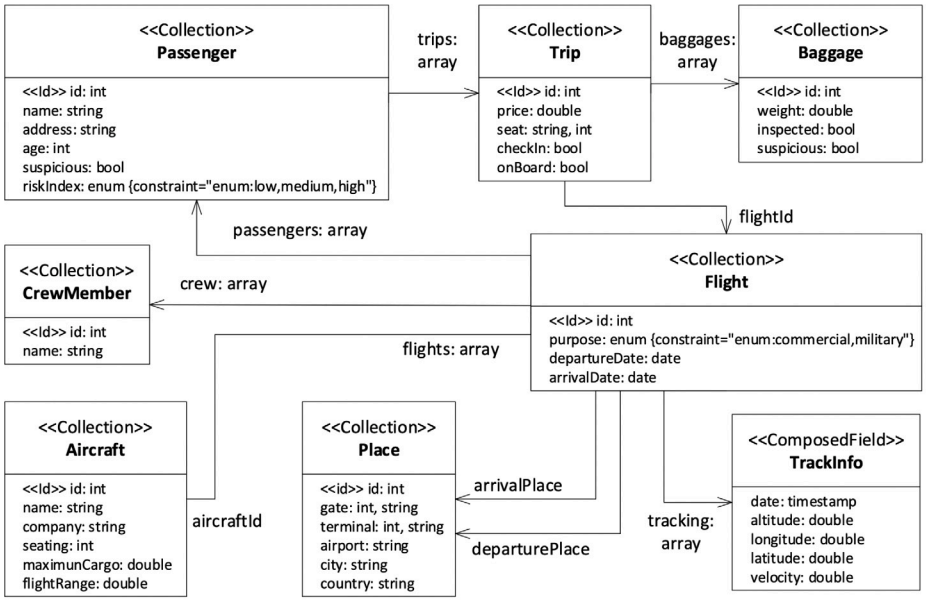

Fig. 10. Case study: structural elements.

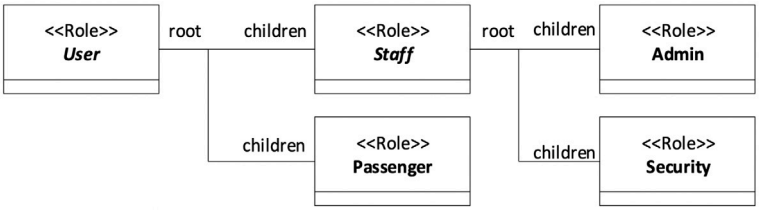

Fig. 11. Case study: hierarchy of security roles.

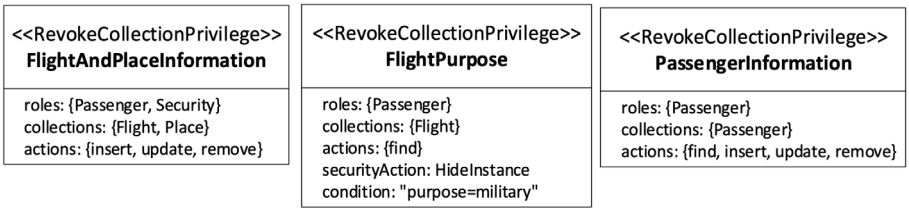

Fig. 12. Case study: security policies.

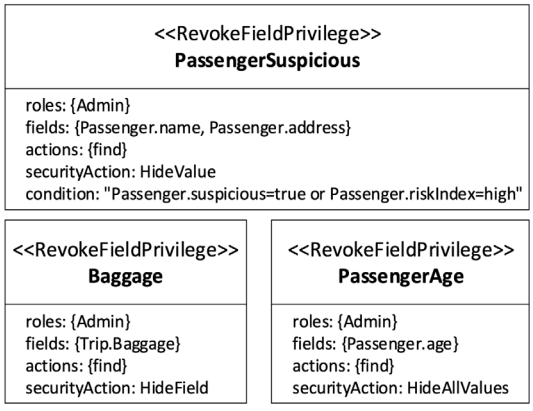

Fig. 13. Case study: fine-grained security policies.

Structural transformations

The first step consisted of generating the implementation of the structure of the document database (collections, fields, etc.) after having applied the set of transformation rules that handle the structural aspects (Fig. 5).

Once the database for the airport had been created, each of the collections in the model was processed in order to generate the code corresponding to that collection, together with its validation schema (Collection2Collection transformation). A list of the fields that were required was then added to this schema (Field2Required transformation). The information (name, data type and restrictions) concerning each simple or composed field related to the collection (SimpleField2Property and ComposedField2Property transformations) was also added.

Fig. 14 provides an example of how the SimpleField2Property transformation was applied to the enumerated field "riskIndex" in the "Passenger" collection. This made it possible to generate a property with the same name and type in the output model in the validation schema of the "Passenger" collection, in which the values that the enumerated field can take are reflected in the condition of the property.

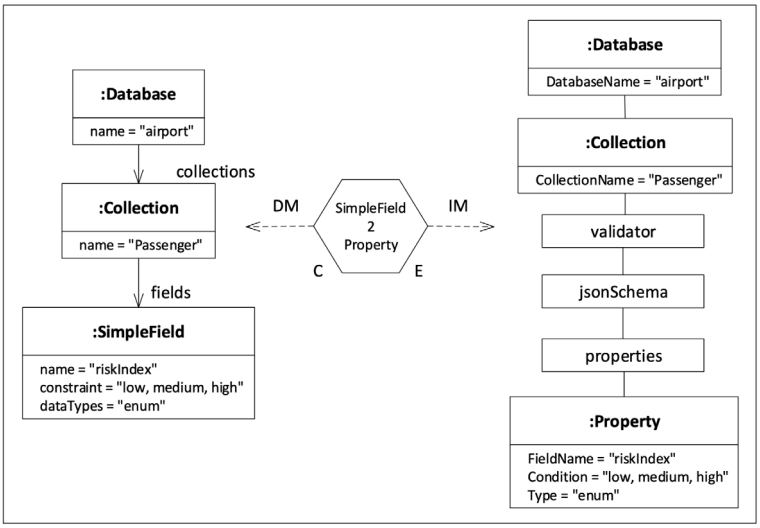

Fig. 14. Application of the SimpleField2Property transformation.

```
db.createCollection("Passenger", {
   validator: {
      $jsonSchema: {
         bsonType: "object",
         required: [ "_id", "name", "address", "age",
            "suspicious", "riskindex", "trips" ],
         properties: {
            _id: {
               bsonType: "int"
            },
            name: {
               bsonType: "string"
            },
            address: {
               bsonType: "string"
            },
            age: {
               bsonType: "int"
            },
            suspicious: {
               bsonType: "bool"
            },
            riskindex: {
               enum: [ "low", "medium", "high"]
            },
            trips: {
              bsonType: "array",
            }
         }}}})
```

Fig. 15. JSON validator for Passenger collection.

After processing the collections, it was necessary to create the indices required (Id2Index transformation). Fig. 15 shows the code generated for the Passenger collection, in which it is possible to see the validation schema. This indicates all the fields of which it is composed (property), together with their identifying names and the types of data allowed (bsonType). We should highlight that the riskIndex field is declared as a type of enumerated data in which the restriction has been specified as being the set of values that can take that enumeration. Moreover, the validation schema indicates that all the fields in the collection are required (indicated in required).

Security policy transformations

After generating the implementation of the structure of the document database, the set of transformations in charge of the security aspects acts (Fig. 7). The transformation is initiated by generating the hierarchy of roles defined in the model (Role2Role transformation). In this case, a role will be generated for User, Staff, Admin, Security and Passenger, although User and Staff are abstract and will not have users assigned to them. An analysis of each of these roles with their users (Security, Passenger and Admin) will be carried out as follows in order to show the set of authorisation rules associated with them. They will then be used as a basis on which to automatically apply an implementation strategy (Collection2Privileges transformation supported by transformations for each situation).

Let us begin with the Security role. It will be noted that it affects only a security policy associated with the Flight and Place collections (RevokeCollectionPrivilegeFlightAndPlaceInformation). Fig. 16 shows how the Collection2Privileges transformation is applied to the Security role and the Place collection. This situation corresponds to the case of a role being associated only with collection-level security policies, and it is granted permissions for actions that have not been revoked, signifying that it invokes the OnlyRevokeCollectionPrivileges2GrantedActions transformation.

Fig. 17 shows the code generated. Note that the Security role has been granted privileges to execute all the actions (find, insert, update, remove) for all the collections, with the exception of the Place and Flight collections, for which it can execute only the Find action. This is owing to the fact that the rule that affects it withdraws the Insert, Update and Remove actions for those collections.

We shall now analyse the Passenger role, which is associated with three of the security policies defined at the collection level. Let us return to the previous case, which is affected only by collection-level policies (OnlyRevokeCollectionPrivileges2GrantedActions transformation), although it implies several more complex transformation rules. The code generated is shown in Fig. 18.

There is a security policy (FlightAndPlaceInformation) for the Flight and Place collections that withdraws the Insert, Update and Remove actions from Passenger for these collections. This is indicated in the code representing the fact that the Find action is the only one permitted. Another similar policy (PassengerInformation) withdraws all the actions for the Passenger collection. This collection is represented in the code because it does not include that collection within its privileges. Finally, another policy (FlightPurpose) limits the consultations regarding Flight to only those flights whose purpose is not military. This implementation is carried out by generating a view (SecurityAction2Match transformation) called Flight Passenger in which a projection showing all the fields in a collection is carried out. There is also a match that filters those views whose Purpose field has a value of 'military'. This implementation is shown in 19.

Finally, we shall analyse the Admin role, which is affected by three fine-grained security policies (RevokeFieldPrivilege) that affect the fields in the Passenger and Trip collections. In this case, and as is shown in Fig. 20, all actions are authorised for all the collections, with

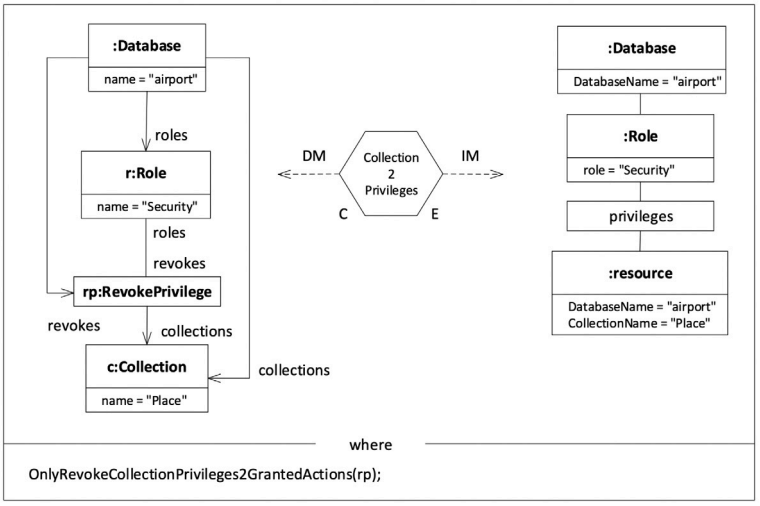

Fig. 16. Application of Collection2Privileges transformation.

```
db.createRole(
   {
     role: "Security",
     privileges: [
       { resource: { db: "airport", collection: "Place" },
           actions: [ "find" ] },
       { resource: { db: "airport", collection: "Flight" },
           actions: [ "find"] },
       { resource: { db: "airport", collection: "Aircarft" },
           actions: [ "find","insert","update","remove" ] },
       { resource: { db: "airport", collection: "Baggage" },
           actions: [ "find","insert","update","remove" ] },
       { resource: { db: "airport", collection: "CrewMember" },
           actions: [ "find","insert","update","remove" ] },
       { resource: { db: "airport", collection: "Passenger" },
           actions: [ "find","insert","update","remove" ] },
       { resource: { db: "airport", collection: "Trip" },
           actions: [ "find","insert","update","remove" ] },
     ],
     roles: []
   }
)
```

Fig. 17. Security role with only find security policies in Place and Flight collections.

the exception of Passenger and Trip, for which only the Find action is granted. Each security policy is then analysed at the level of the field that affects the role in order to generate the views required for their implementation (OnlyRevokeFieldPrivileges2View transformation).

One of the security policies (RevokeFieldPrivilegeBaggage) hides the Baggage field in the Flight collection (HideField security action). Fig. 21 shows how it is implemented using a view with a projection that shows all the other fields in the Flight collection except Baggage.

Another security policy (PassengerAge) affects the Passenger collection by indicating that the values regarding the passengers' ages must be hidden (HideAllValues). Fig. 22 shows how the HideAllValues2Projection rule is applied to that security policy, creating a view with a projection in which the Age field always shows a null value.

Another security policy that is related to the same collection (PassengerSuspicious) hides the values of the name and address fields (HideValue) if the passenger is suspicious or has a high risk index.

The implementation of both security policies is shown in Fig. 23, in which the view created for the Admin role and the Passenger collection carries out a projection in which the Age field is always null and in which the Name and Address fields show their original value or a null value, depending on whether they fulfil the condition established (null if suspicious is true or if riskIndex is high).

Finally, it is necessary to create the users of the database defined in the model associated with a specific role, and assign that role to them (User2Role transformation). This is shown in the example in Fig. 24.

6.3. Security policies fulfilment

In this Section, we explain how the implementation of the security policies generated from the transformation process works. Fig. 25 shows the information that a user with the Security role has retrieved from the Passenger collection using the "find" operation. In this case, given that the Security role has no security policies defined for this collection, all information about all passengers is shown.

However, given that the Admin role has restrictions as regards the Passenger collection, the information retrieved by a user who has this role will be different. This scenario is shown in Fig. 26, which illustrates that, in the case of the Admin role, the find operation must be performed for the Passenger admin view created in Section 3, given that this role is not permitted to apply a find operation directly to the Passenger collection. As will be observed, the "age" value is hidden with a null value for all passengers. The name and address fields are also hidden for the passenger with id 5201950 because his risk index value is high, thus fulfilling the PassengerSuspicious and PassengerAgesecurity security policies defined for this role in Section 3.

Fig. 27 shows the information obtained by the Security role when performing a find operation for the Flight collection. Given that this role has no security policies defined for it that affect the Flight collection, all data from all flights can be accessed without restrictions.

However, the Passenger role had a security policy for the Flight collection, which does not allow it to retrieve data regarding military flights. Fig. 28 illustrates the results obtained after performing a find operation for the Flight_passenger view, given that the Passenger role cannot directly access the Flight collection. As will be noted, the flight with _id 23162 is not shown because its purpose is military, as shown in Fig. 27.

Finally, the information retrieved from the Flight collection by the Security role is shown in Fig. 29. As occurred previously, this role has access to all data without restrictions. Fig. 30 shows the results of the same operation executed by the Admin role. The difference is that, for this last role, the "baggage" field is fully hidden, as required by the Baggage security policy defined in Section 3.

6.4. Lessons learned

The application of our proposal to the case study has allowed us to validate its applicability as well as to detect and improve several aspects.

On the one hand, we have been able to check how a designer can specify the structure and security policies of the database in a simple

```
db.createRole(
   {
     role: "Passenger",
     privileges: [
       { resource: { db: "airport", collection: "Flight_passenger" },
           actions: [ "find"] },
       { resource: { db: "airport", collection: "Place" },
           actions: [ "find" ] },
       { resource: { db: "airport", collection: "Aircarft" },
           actions: [ "find","insert","update","remove" ] },
       { resource: { db: "airport", collection: "Baggage" },
           actions: [ "find","insert","update","remove" ] },
       { resource: { db: "airport", collection: "CrewMember" },
           actions: [ "find","insert","update","remove" ] },
        { resource: { db: "airport", collection: "Trip" },
           actions: [ "find","insert","update","remove" ] }
     ],
     roles: []
   }
)
```

Fig. 18. Passenger role with only find security policies in Place and Flight collections and without any policy in Trip and Passenger collections.

```
db.createView(
    "Flight_passenger",
    "Flight",
    [
        {$project: {_id: 1, purpose: 1, departureDate: 1,
            arrivalDate: 1, tracking: 1, arrivalPlace: 1,
            departurePlace: 1, aircraftid: 1, crew: 1}},
        {$match: {"purpose": {$ne: "military"}}}

    ]
)
```

Fig. 19. View to enable Passengers to retrieve filtered Flights.

```
db.createRole(
   {
     role: "Admin",
     privileges: [
       { resource: { db: "airport", collection: "Trip_admin" },
           actions: [ "find"] },
       { resource: { db: "airport", collection: "Passenger_admin" },
           actions: [ "find"] },
       { resource: { db: "airport", collection: "Flight" },
           actions: [ "find","insert","update","remove" ] },
       { resource: { db: "airport", collection: "Aircarft" },
           actions: [ "find","insert","update","remove" ] },
       { resource: { db: "airport", collection: "CrewMember" },
           actions: [ "find","insert","update","remove" ] },
       { resource: { db: "airport", collection: "Baggage" },
           actions: [ "find","insert","update","remove" ] },
       { resource: { db: "airport", collection: "Place" },
           actions: [ "find","insert","update","remove" ] }
     ],
     roles: []
   }
)
```

Fig. 20. Admin role with find permissions limited for the views created from Passenger and Flight collections.

C.Blanco et al.

```
db.createView(
    "Trip_admin",
    "Trip",
    [
        {$project: {_id: 1, price: 1,
            seat: 1, checkIn: 1, onBoard: 1}}
    ]
)
```

Fig. 21. View employed to filter Trip data for Admin role.

way using our metamodel. It can be seen how the case study models (Figs. 10, 11, 12 and 13) are much more concise and visual than the system implementation. For example, each security policy is modelled as a class with several attributes and at the implementation level requires the definition of privileges, views, projections, etc. Moreover, being models independent of the target tools, the designer is freed from knowing details about the syntax and strategy to be followed for its correct implementation in a given document database manager.

On the other hand, the case study has allowed us to analyse alternatives for the modelling and implementation of each structural and security aspect and to select those most beneficial for the use of our proposal by the designer.

Initially, we started by adopting a mixed strategy for the representation of security policies that allowed us to define in the same model both rules that grant privileges and rules that revoke them. This aspect gives flexibility in modelling but also makes the designer worry about conflicts between positive and negative rules and their consistency. In favour of facilitating modelling, we finally decided to adopt an open-world approach in which the designer only indicates the revoked privileges and from them the necessary positive and negative permissions are generated at the implementation level. For example, the rule "FlightAndPlaceInformation" (Fig. 12) indicates that insert, update and remove privileges are denied for the Flight and Place collections. At the implementation level, this is represented by several policies (one per collection): one to grant read privilege to Flight, one to grant read to Place, and several to grant read, insert, update and remove to the other collections.

Another aspect we analysed was the inclusion of the possible security actions that can be associated with a revoke privilege (hide instance, hide value, hide all values or hide field). Depending on whether the revoke privilege is associated with collections or fields, you can apply different security actions. We evaluated several alternatives, from the use of an enumeration to a specialisation of the security actions in two subtypes (collection and field). Moreover, some of these actions can only be applied to the read privilege. For these reasons, we finally chose to use an intermediate solution in which we modelled the security actions with inheritance and added to the metamodel the necessary constraints to control the exceptional cases.

Furthermore, when developing the transformation rules from the models to their corresponding implementation, we added minor changes to the metamodel that would facilitate the flow of application of these rules. For example, we made the association between Role and RevokePrivilege navigable in both directions, which makes it easy to obtain all the rules associated with a given role.

7. Related work

Addressing security early in a software development project has the potential to avoid, fix and correct security flaws in the later stages of development. Security issues should ideally be addressed as early as

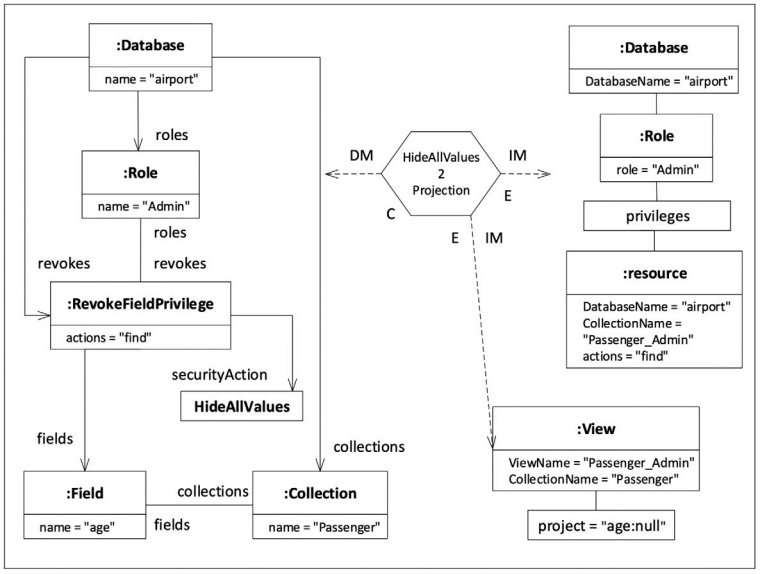

Fig. 22. Application of the HideAllValues2Projection transformation.

```
db.createView(
    "Passenger_admin",
    "Passenger",
    [
        {$project: {_id: 1,
                    name: {$cond: {if:  { $or: [{$eq: ["$suspicious",true]},
                        {$eq: ["$riskIndex","high"]}]  },
                        then: null, else: "$name" }},
                    address: {$cond: {if:  { $or: [{$eq: ["$suspicious",true]},
                        {$eq: ["$riskIndex","high"]}]  },
                        then: null, else: "$address" }},
                    suspicious: 1, riskindex: 1, age: null
                }
        }
    ]
)
```

Fig. 23. View to filter Passenger data for Admin role.

```
db.createUser(
    {
      user: "Security",
      pwd: "passSec",
      roles: [ { role: "Security", db: "airport" } ]
    }
)
```

Fig. 24. Security user with the security role assigned.

```
> db.Passenger.find({});
{ _id : 678009, name : "John S. Doe",
    address : "First Avenue 45, London, UK",
    age : 25, suspicious : false, riskindex : "low",
    trips : [ 556778, 2244565, 323121 ] }

{ _id : 176779, name : "Jane H. Doe",
    address : "First Avenue 45, London, UK",
    age : 27, suspicious : false, riskindex : "low",
    trips : [ 556778, 2244565, 323121 ] }

{ _id : 5201950, name : "Charles Widmore",
    address : "The Island, Somewhere in the Pacific",
    age : 69, suspicious : true, riskindex : "high",
    trips : [ 815 ] }
```

Fig. 25. Find operation in Passenger collection by using Security role.

```
> db.Passenger_admin.find({});
{ _id : 678009, name : "John S. Doe",
    address : "First Avenue 45, London, UK",
    age : null, suspicious : false, riskindex : "low",
    trips : [ 556778, 2244565, 323121 ] }

{ _id : 176779, name : "Jane H. Doe",
    address : "First Avenue 45, London, UK",
    age : null, suspicious : false, riskindex : "low",
    trips : [ 556778, 2244565, 323121 ] }

{ _id : 5201950, name : null,
    address : null,
    age : null, suspicious : true, riskindex : "high",
    trips : [ 815 ] }
```

Fig. 26. Find operation in Passenger_admin view by using Admin role.

the software analysis and design phase. This is the core principle of "Security by design", which is advocated by industry and academia [30,31], although new "security-by-design" proposals are appearing in different environments such as Big Data [32], Cloud Computing [33,34], IoT [35] and Cyber–physical Systems [36,37]. There are also proposals as in [38] where the authors present a novel security-by-design methodology based on Security Service Level Agreements (SLAs), which can be integrated within modern development processes and that is able to support the risk management life-cycle in an almost-completely automated way.

```
> db.Flight.find({});
{ _id : 35891, purpose : "commercial",
    departureDate : ISODate("2019-01-05T16:00:00Z"),
    arrivalDate : ISODate("2019-01-05T23:04:00Z"),
    tracking : [ "tri987", "tri988" ],
    arrivalPlace : 11, departurePlace : 23,
    aircraftid : 8907, crew : [ "crew1", "crew2" ] }

{ _id : 45122, purpose : "commercial",
    departureDate : ISODate("2018-12-30T19:05:00Z"),
    arrivalDate : ISODate("2018-12-31T02:04:00Z"),
    tracking : [ "tri111", "tri112" ],
    arrivalPlace : 56, departurePlace : 123,
    aircraftid : 1256, crew : [ "crew3", "crew4" ] }

{ _id : 23162, purpose : "military",
    departureDate : ISODate("2020-01-11T08:11:00Z"),
    arrivalDate : ISODate("2020-01-11T19:26:00Z"),
    tracking : [ "tri0777", "tri778" ],
    arrivalPlace : 45, departurePlace : 28,
    aircraftid : 1200, crew : [ "crew12", "crew3", "crew25" ] }
```

Fig. 27. Find operation in Flight collection by using Security role.

```
> db.Flight_passenger.find({});
{ _id : 35891, purpose : "commercial",
    departureDate : ISODate("2019-01-05T16:00:00Z"),
    arrivalDate : ISODate("2019-01-05T23:04:00Z"),
    tracking : [ "tri987", "tri988" ],
    arrivalPlace : 11, departurePlace : 23,
    aircraftid : 8907, crew : [ "crew1", "crew2" ] }

{ _id : 45122, purpose : "commercial",
    departureDate : ISODate("2018-12-30T19:05:00Z"),
    arrivalDate : ISODate("2018-12-31T02:04:00Z"),
    tracking : [ "tri111", "tri112" ],
    arrivalPlace : 56, departurePlace : 123,
    aircraftid : 1256, crew : [ "crew3", "crew4" ] }
```

Fig. 28. Find operation in Flight_passenger view by using Passenger role.

```
> db.Trip.find({})
{ _id : 45678, price : 340.09, seat : "1A",
    checkIn : true, onBoard : false,
    baggages : [ 1232898, 3232879 ],
    flightid : 219898 }

{ _id : 12458, price : 890.11, seat : "20D",
    checkIn : false, onBoard : false,
    baggages : [ 2212122 ],
    flightid : 818898 }

{ _id : 11223, price : 90.01, seat : "45A",
    checkIn : true, onBoard : false,
    baggages : [ 9816273 ],
    flightid : 719897 }
```

Fig. 29. Find operation in Trip collection by using Security role.

```
> db.Trip_admin.find({})
{ _id : 45678, price : 340.09, seat : "1A",
    checkIn : true, onBoard : false,
    flightid : 219898 }

{ _id : 12458, price : 890.11, seat : "20D",
    checkIn : false, onBoard : false,
    flightid : 818898 }

{ _id : 11223, price : 90.01, seat : "45A",
    checkIn : true, onBoard : false,
    flightid : 719897 }
```

Fig. 30. Find operation in Trip_admin view by using Admin role.

Model-Driven Security [17] is an approach that can support the process of modelling security requirements [39] at a high level of abstraction in the early stage of software development before the transaction process and the generation of codes, thus increasing the level of system quality [40]. In this context, some of the best known proposals are UMLSec [41] and SecureUML [42], among others [43].

Several initiatives have been proposed in order to incorporate security into the software development life cycle (SDLC) by using different software models. Some initiatives propose that modifications be made to the traditional lifecycle by incorporating security activities or tasks [44] with the aim of creating enhanced methodologies with security aspects [45]. Some examples of this are CLASP [46], Microsoft SDL [47] or Seven touchpoints [48]. Other proposals are focused on the definition of security architectures [49–51] or security patterns [52,53], which help to integrate security into the SDLC and allow to incorporate all security aspects during the design.

If the incorporation of security aspects into the software lifecycle from the early stages of development is fundamental as regards obtaining a secure and robust system, it is also important for the design and development of databases. In fact, Bugiotti et al. [54] propose a database design methodology for NoSQL databases that relies on an abstract data model whose objective is to represent NoSQL systems in a system-independent manner. However, it is also necessary to model proposals for the design and management of NoSQL databases using well-known security techniques, recommendations, and best practices, thus allowing the design of efficient, robust and secure databases. Most of the existing works are focused on the security aspects of MongoDB or Cassandra [55]. Many of them are limited to finding solutions to concrete problems in restricted scenarios, and are oriented towards document [56] or columnar data [57], and always at the implementation level [58]. Other works focus on cryptographic techniques or mechanisms for cloud databases [19,59,60] and modelling at a higher level of abstraction, using Big Data concepts that are not specific to any NoSQL technology [61]. There are also some service-oriented proposals but they do not target NoSQL databases, such as the work of Ghazi et al. in [62] which have developed a comprehensive database security-as-a-service (DB-SECaaS) system, a three-dimensional approach to securing data in cloud is presented in [63]. Yamany et al. [64] presented an intelligent service-oriented architecture security framework, and Song et al. [65] proposed a generic framework of data protection-as-service (DPaaS), which, as mentioned above, are not focused on NoSQL.

Among the model-oriented approaches, we can find the work of Akoka et al. [66], which defines an approach for MDE-based NoSQL databases in which forward and reverse engineering processes are combined, or [61], in which a query-driven data modelling process for Apache Cassandra that defines mapping rules and patterns but does not support the integration of security policies is described. The work in [67] proposes NoSQL database design method using conceptual data model based on Peter Chen's framework, again without any security aspect. The authors of [68] define a common conceptual level model for various types of NoSQL databases. Those of [69] propose an automatic MDA-based approach that provides a set of transformations, formalised with the QVT language, with which to translate UML conceptual models into NoSQL models. In [70], the same authors propose an automatic approach for the implementation of UML conceptual models in NoSQL systems, including the mapping of the associated OCL constraints to the code required to verify them. In [71], the authors present a process with which to extract a model from a collection of JSON documents stored in MongoDB, while those of [72] propose a process with which to extract models from a NoSQL document database that can include several collections. We should also mention [54], which describes a mapping from a NoSQL document database to a relational model. ToNoSQLModel process is presented in [73], and is an MDA-based approach employed to extract a physical model by starting from a NoSQL document database and includes links between different collections. A transformation process that ensures the transition from the NoSQL logic model to a NoSQL physical mode by applying a chain of transformations that translate it into a succession of models in order to arrive at a target NoSQL model that complies with existing solutions is proposed in [74]. In [75] the authors propose a methodology for

the logical design of NoSQL document databases converting conceptual modelling for suitable and efficient logical representation considering the expected workload of the application. Li et al. proposed an approach with which to transform UML class diagrams into an HBase data model [76]. All these proposals propose intermodel transformation rules for NoSQL without considering the security aspects or policies of the databases at any point in the transformations. The work in [77] proposes a security model, based on the use of metadata, to provide a mechanism of access control for NoSQL graph-oriented database management system. One proposal that does consider access control policies for NoSQL databases, but from a modernisation or reverse-engineering perspective, is the work of [78]. This presents the first modernisation approach for the introduction of security into NoSQL document databases through the improvement of access control by using a domain ontology to detect sensitive information and creating a conceptual model of the database.

In conclusion, the current proposals for the modelling and development of NoSQL databases do not adequately consider security policies in all stages. Solutions focused on the implementation stage provide isolated security aspects or mechanisms. Solutions focused on earlier stages of development, meanwhile, provide modelling and integration solutions, but consider only structural aspects, and security aspects are, in most cases, ignored or not appropriate.

8. Conclusions

NoSQL databases are beginning to acquire considerable importance and are in great demand. This is owing principally to the increase in Big Data technologies, which has obliged organisations to change from relational databases to other not necessarily relational models, such as document, graph-oriented models, etc. NoSQL databases are scalable, distributed and highly reliable. There is, however, currently an important obstacle to their generalised adoption: data security. This is owing to the fact that security is usually added at the end, once the system is already implemented, which leads to solutions that are prone to security failures and that are poorly suited to security requirements. Our proposal is focused on the design and implementation of security policies in NoSQL document databases by including security from early stages of the development process (security by design). Security is, therefore, taken into account in the system design decisions and its subsequent implementation, thus obtaining more robust systems. To this end, we first define a metamodel that allows us to define both the structure and the security policies associated with the document database at the design level. This signifies that, in order to include security policies in the system, the designer directly uses design models (defined according to this metamodel). This facilitates his/her work, since it allows him/her to be abstracted from the technical aspects of the tools and the need to know which the best implementation strategies are.

Our proposal then analyses the characteristics offered by a specific document database management tool, MongoDB. We subsequently define a model for the proper implementation of both structural aspects and security policies in this tool.

Once both models have been defined, our proposal facilitates the design and implementation of the system, along with the security policies, using a model-driven development approach. This is done by defining the relationships that are required between the design model and the implementation model in MongoDB. These relationships select the best implementation strategy, thus abstracting the designer from this task (use of views, projections, etc.). This is completed with the provision of a technological solution based on the Eclipse Modelling Framework in which we implement the source metamodel (in Emfatic) and the necessary transformations (in Epsilon, EGL). The use of the model-driven approach not only saves time and development costs, but has also made it possible to obtain an architecture that can be easily extended to other tools and technologies.

We have applied the proposal to a case study of an airport in order to show the whole process involved in the design and implementation of the system, together with its associated security policies. In this case study, a design model was first created in order to make it simple to see how the system is defined without the inclusion of technical aspects of specific tools. The transformations defined were then generated automatically in order to carry out their corresponding implementation in MongoDB, thus freeing the designer from having to choose specific implementation strategies. All of the above made it possible to discover that the solution obtained really did satisfy the security policies initially modelled.

As future work, we plan to improve our model-driven architecture in several respects. On the one hand, we intend to include new target document database management tools (such as CouchDB or CouchBase). A new tool would be included through the reuse of our metamodel, which would make it necessary to analyse the security features offered by that tool. This would be used as the basis on which to define the strategies required in order to implement the security policies. Finally, we would define a set of transformations that would automate the code generation, starting from our design model.

On the other hand, the architecture could be improved by embracing other Big Data and NoSQL technologies (columnar, graph-oriented, key–value, etc.).

In this sense, to incorporate a new technology to our proposal (such as graph-oriented databases) it will be necessary to carry out a process similar to the one shown in this paper. It would be necessary to analyse, on the one hand, the specific structural and security characteristics of this technology and, on the other hand, the characteristics offered by the management tools for its final implementation. Based on this, we would specify our own metamodel for designing this kind of systems, as well as a set of transformations for each management tool in which we wish to generate the corresponding implementation.

Once we have several metamodels centred on different technologies (document, graph-oriented, columnar, etc.), we could go one step further, which would consist of defining a single metamodel at a higher level of abstraction. This metamodel would include the elements necessary to define the structural and security requirements of the system, independently of the technology that is subsequently decided to be used in its design. To build this metamodel, we would need to identify elements common to BigData and NoSQL systems that will be specify as different elements depending on the technology used.

For example, a structural element that could be called “container” and represents a group of fields could correspond, depending on the technology chosen, with a collection, node or column family. Or a security requirement could correspond to different security policies specifying different security actions, according to what is possible in that technology.

In the definition of this metamodel, we should ensure that it includes all the necessary information for the next design stage, in which a metamodel specific to a particular technology will be used. Integrating this requirements model in our proposal and applying the model-driven development approach, we could define transformation rules that would derive the corresponding design model in a chosen technology (document, graph-oriented, columnar, etc.) and from there, the designer would add or modify the necessary aspects in the model and finally would generate the implementation in a specific tool.

CRediT authorship contribution statement

Carlos Blanco: Conceptualisation, Methodology, Software, Investigation, Writing – original draft, Writing – review & editing. **Diego García-Saiz:** Conceptualisation, Software, Validation, Investigation, Writing – original draft. **David G. Rosado:** Conceptualisation, Investigation, Writing – original draft, Writing – review & editing. **Antonio Santos-Olmo:** Conceptualisation, Investigation, Writing – review &

editing. **Jesús Peral:** Conceptualisation, Resources, Data curation. **Alejandro Maté:** Conceptualisation, Resources, Data curation. **Juan Trujillo:** Supervision, Investigation, Project administration, Funding acquisition. **Eduardo Fernández-Medina:** Conceptualisation, Investigation, Supervision, Project administration, Funding acquisition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

This work has been developed within the AETHER-UCLM (PID2020-112540RB-C42), AETHER-UA (PID2020-112540RB-C43) and PRECON-I4 (TIN2017-86520-C3-3-R) projects funded by "Ministerio de Ciencia e Innovación, Spain", and the AURORA (SBPLY/21/180501/000079) and GENESIS (SBPLY/17/180501/000202) projects funded by "Consejería de Educación, Cultura *y* Deportes, Junta de Comunidades de Castilla-La Mancha, Fondo Europeo de Desarrollo Regional FEDER".

References

[1] Al-Sai ZA, Abdullah R, heikal Husin M. Big data impacts and challenges: A review. In: 2019 IEEE Jordan international joint conference on electrical engineering and information technology. IEEE; 2019, p. 150–5. http://dx.doi.org/10.1109/JEEIT.2019.8717484, https://ieeexplore.ieee.org/document/8717484/.

[2] Singh N, Lai KH, Vejvar M, Cheng TC. Big data technology: Challenges, prospects, and realities. IEEE Eng Manag Rev 2019;47(1):58–66. http://dx.doi.org/10.1109/EMR.2019.2900208.

[3] Kantarcioglu M, Ferrari E. Research challenges at the intersection of big data, security and privacy. Front Big Data 2019;2:1. http://dx.doi.org/10.3389/fdata.2019.00001.

[4] Mendelson A. Security and privacy in the age of big data and machine learning. Computer 2019;52(12):65–70. http://dx.doi.org/10.1109/MC.2019.2943137.

[5] Moreno J, Serrano MA, Fernandez-Medina E, Fernandez EB. Towards a security reference architecture for big data. In: CEUR workshop proceedings. vol. 2062, 2018.

[6] Rawat DB, Doku R, Garuba M. Cybersecurity in big data era: From securing big data to data-driven security. IEEE Trans Serv Comput 2019;1. http://dx.doi.org/10.1109/tsc.2019.2907247.

[7] Shamsi JA, Khojaye MA. Understanding privacy violations in big data systems. IT Prof 2018;20(3):73–81. http://dx.doi.org/10.1109/MITP.2018.032501750, https://ieeexplore.ieee.org/document/8378964/.

[8] Tang M, Alazab M, Luo Y. Big data for cybersecurity: Vulnerability disclosure trends and dependencies. IEEE Trans Big Data 2019;5(3):317–29. http://dx.doi.org/10.1109/TBDATA.2017.2723570.

[9] Venkatraman S, Venkatraman R. Big data security challenges and strategies. AIMS Math 2019;4(3):860–79. http://dx.doi.org/10.3934/math.2019.3.860, http://www.aimspress.com/article/10.3934/math.2019.3.860.

[10] Diogo M, Cabral B, Bernardino J. Consistency models of NoSQL databases. Future Internet 2019;11(2):43. http://dx.doi.org/10.3390/fi11020043.

[11] Roy-Hubara N, Sturm A. Design methods for the new database era: a systematic literature review. Softw Syst Model 2020;19(2):297–312. http://dx.doi.org/10.1007/s10270-019-00739-8.

[12] Sahafizadeh E, Nematbakhsh MA. A survey on security issues in big data and NoSQL. Adv Comput Sci Int J 2015;4(4):68–72. http://dx.doi.org/10.1109/ICITST.2015.7412089, http://www.acsij.org/acsij/article/view/80.

[13] Ramzan S, Bajwa IS, Kazmi R, Amna. Challenges in NoSQL-based distributed data storage: A systematic literature review. Electronics (Switzerland) 2019;8(5):488. http://dx.doi.org/10.3390/electronics8050488.

[14] Kurpanik J, Pańkowska M. NOSQL problem literature review. Studia Ekon. 2015;234:80–100.

[15] Moreno J, Fernandez EB, Serrano MA, Fernandez-Medina E. Secure development of big data ecosystems. IEEE Access 2019;7:96604–19. http://dx.doi.org/10.1109/ACCESS.2019.2929330.

[16] Basin D, Clavel M, Egea M, De Dios MA, Dania C. A model-driven methodology for developing secure data-management applications. IEEE Trans Softw Eng 2014;40(4):324–37. http://dx.doi.org/10.1109/TSE.2013.2297116.

[17] Lúcio L, Zhang Q, Nguyen PH, Amrani M, Klein J, Vangheluwe H, et al. Advances in model-driven security. In: Advances in computers. vol. 93, Elsevier; 2014, p. 103–52. http://dx.doi.org/10.1016/B978-0-12-800162-2.00003-8.

[18] Blanco C, Fernández-Medina E, Trujillo J. Modernizing secure OLAP applications with a model-driven approach. Comput J 2014;58(10):2351–67. http://dx.doi.org/10.1093/comjnl/bxu070.

[19] Alenezi M, Usama M, Almustafa K, Iqbal W, Raza MA, Khan T. An efficient, secure, and queryable encryption for NoSQL-based databases hosted on untrusted cloud environments. Int J Inf Secur Priv 2019;13(2):14–31. http://dx.doi.org/10.4018/IJISP.2019040102.

[20] Mohammed NM, Niazi M, Alshayeb M, Mahmood S. Exploring software security approaches in software development lifecycle: A systematic mapping study. Comput Stand Interfaces 2017;50:107–15. http://dx.doi.org/10.1016/j.csi.2016.10.001.

[21] Nguyen PH, Kramer M, Klein J, Traon YL. An extensive systematic review on the model-driven development of secure systems. Inf Softw Technol 2015;68:62–81. http://dx.doi.org/10.1016/j.infsof.2015.08.006.

[22] Souag A, Mazo R, Salinesi C, Comyn-Wattiau I. Reusable knowledge in security requirements engineering: a systematic mapping study. Requir Eng 2016;21(2):251–83. http://dx.doi.org/10.1007/s00766-015-0220-8.

[23] Yadav D, Maheshwari DH, Chandra DU. Big data hadoop: Security and privacy. SSRN Electron J 2019. http://dx.doi.org/10.2139/ssrn.3350308.

[24] Gupta A, Verma A, Kalra P, Kumar L. Big data: A security compliance model. In: Proceedings of the 2014 conference on IT in business, industry and government: an international conference by CSI on big Data. 2014, http://dx.doi.org/10.1109/CSIBIG.2014.7056963.

[25] Gupta S, Singh NK, Tomar DS. Analysis of NoSQL database vulnerabilities. 2018, https://papers.ssrn.com/sol3/papers.cfm?abstract{_}id=3172769.

[26] DB-Engines. DB-engines ranking of graph DBMS. 2016, http://db-engines.com/en/ranking/graph+dbms.

[27] García-Ruiz CM, Oliver A, Peral J, Trujillo J, Blanco C, Fernández-Medina E. Management of sensitive data on NoSQL databases. In: CEUR workshop proceedings. vol. 1979, 2017, p. 156–69, http://ceur-ws.org/Vol-1979/paper-150.pdf.

[28] MongoDB Inc. MongoDB security architecture. Tech. rep., 2021, November. https://www.mongodb.com/white-papers.

[29] Gamma E, Helm R, Johnson R, Vlissides JM. Design patterns: Elements of reusable object-oriented software. 1st ed.. Addison-Wesley Professional; 1994, http://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented.

[30] van den Berghe A, Scandariato R, Yskout K, Joosen W. Design notations for secure software: a systematic literature review. Softw Syst Model 2017;16(3):809–31. http://dx.doi.org/10.1007/s10270-015-0486-9.

[31] Moral-García S, Moral-Rubio S, Fernández EB, Fernández-Medina E. Enterprise security pattern: A model-driven architecture instance. Comput Stand Interfaces 2014;36(4):748–58. http://dx.doi.org/10.1016/j.csi.2013.12.009.

[32] Awaysheh FM, Aladwan MN, Alazab M, Alawadi S, Cabaleiro JC, Pena TF. Security by design for big data frameworks over cloud computing. IEEE Trans Eng Manag 2021;1–18. http://dx.doi.org/10.1109/TEM.2020.3045661.

[33] Verginadis Y, Michalas A, Gouvas P, Schiefer G, Hübsch G, Paraskakis I. PaaSword: A holistic data privacy and security by design framework for cloud services. J Grid Comput 2017;15(2):219–34. http://dx.doi.org/10.1007/s10723-017-9394-2, http://link.springer.com/10.1007/s10723-017-9394-2.

[34] Veloudis S, Paraskakis I, Petsos C, Verginadis Y, Patiniotakis I, Gouvas P, et al. Achieving security-by-design through ontology-driven attribute-based access control in cloud environments. Future Gener Comput Syst 2019;93:373–91. http://dx.doi.org/10.1016/j.future.2018.08.042, https://linkinghub.elsevier.com/retrieve/pii/S0167739X17320782.

[35] Vallois V, Guenane F, Mehaoua A. Reference architectures for security-by-design IoT: Comparative study. In: 2019 Fifth conference on mobile and secure services. IEEE; 2019, p. 1–6. http://dx.doi.org/10.1109/MOBISECSERV.2019.8686650, https://ieeexplore.ieee.org/document/8686650/.

[36] Geismann J, Gerking C, Bodden E. Towards ensuring security by design in cyber-physical systems engineering processes. In: Proceedings of the 2018 international conference on software and system process. New York, NY, USA: ACM; 2018, p. 123–7. http://dx.doi.org/10.1145/3202710.3203159, https://dl.acm.org/doi/10.1145/3202710.3203159.

[37] Peisert S, Margulies J, Nicol DM, Khurana H, Sawall C. Designed-in security for cyber-physical systems. IEEE Secur Priv 2014;12(5):9–12. http://dx.doi.org/10.1109/MSP.2014.90, http://ieeexplore.ieee.org/document/6924670/.

[38] Casola V, De Benedictis A, Rak M, Villano U. A novel security-by-design methodology: Modeling and assessing security by SLAs with a quantitative approach. J Syst Softw 2020;163:110537. http://dx.doi.org/10.1016/j.jss.2020.110537.

[39] Bulusu ST, Laborde R, Wazan AS, Barrère F, Benzekri A. A requirements engineering-based approach for evaluating security requirements engineering methodologies. In: Advances in intelligent systems and computing. vol. 738, Springer Verlag; 2018, p. 517–25. http://dx.doi.org/10.1007/978-3-319-77028-4_67.

[40] Masmali O, Badreddin O. Model driven security : A systematic mapping study. Softw Eng 2019;7(2):30–8.

[41] Jürjens J. Secure Systems Development With UML. Springer-Verlag; 2005, p. 1–309. http://dx.doi.org/10.1007/b137706, http://www.books24x7.com/marc.asp?bookid=16288.

[42] Lodderstedt T, Basin D, Doser J. SecureUML: A UML-based modeling language for model-driven security. In: Lecture notes in computer science (including subseries lecture notes in artificial intelligence and lecture notes in bioinformatics), vol. 2460 LNCS, Dresden: Springer; 2002, p. 426–41. http://dx.doi.org/10.1007/3-540-45800-x_33.

[43] Mashkoor A, Egyed A, Wille R. Model-driven engineering of safety and security systems: A systematic mapping study. 2020, arXiv preprint arXiv:2004.08471. http://arxiv.org/abs/2004.08471.

[44] M.Surakhi O, Hudaib A, AlShraideh M, Khanafseh M. A survey on design methods for secure software development. Int J Comput Technol 2017;16(7):7047–64. http://dx.doi.org/10.24297/ijct.v16i7.6467.

[45] Kanniah SL, Mahrin MN. A review on factors influencing implementation of secure software development practices. Int J Comput Syst Eng 2016;10(8):3032–9, https://zenodo.org/record/1127256.

[46] Viega J. Security in the software development lifecycle: An introduction to {CLASP}, the comprehensive lightweight application security process. Secure Software, Inc., McLean, Virginia, USA, White Paper.

[47] Howard M, Lipner S. The security development lifecycle: sdl: A process for developing demonstrably more secure software. Microsoft Press; 2006, p. 352, http://www.amazon.com/Security-Development-Lifecycle-Michael-Howard/dp/0735622140.

[48] McGraw G. Software Security: Building Security In. Addison-Wesley Professional; 2006, p. 6. http://dx.doi.org/10.1109/ISSRE.2006.43.

[49] Chondamrongkul N, Sun J, Warren I. Formal security analysis for software architecture design: An expressive framework to emerging architectural styles. Sci Comput Program 2021;206:102631. http://dx.doi.org/10.1016/j.scico.2021.102631, https://linkinghub.elsevier.com/retrieve/pii/S0167642321000241.

[50] Pedraza-Garcia G, Astudillo H, Correal D. A methodological approach to apply security tactics in software architecture design. In: 2014 IEEE Colombian conference on communications and computing, COLCOM 2014 - conference proceedings. 2014, http://dx.doi.org/10.1109/ColComCon.2014.6860432.

[51] Moreno J, Rosado DG, Sánchez LE, Serrano MA, Fernández-Medina E. Security reference architecture for cyber-physical systems (CPS). J.UCS 2021;27(6):609–34. http://dx.doi.org/10.3897/jucs.68539, https://lib.jucs.org/article/68539/.

[52] Alvi AK, Zulkernine M. A security pattern detection framework for building more secure software. J Syst Softw 2021;171:110838. http://dx.doi.org/10.1016/j.jss.2020.110838, https://linkinghub.elsevier.com/retrieve/pii/S0164121220302296.

[53] Yoshioka N, Washizaki H, Maruyama K. A survey on security patterns. Prog Inform 2008;(5):35. http://dx.doi.org/10.2201/NiiPi.2008.5.5, http://www.hillside.net/patterns/definition.html http://www.nii.ac.jp/pi/n5/5{_}35.html.

[54] Bugiotti F, Cabibbo L, Atzeni P, Torlone R. Database design for NoSQL systems. In: Lecture notes in computer science (including subseries lecture notes in artificial intelligence and lecture notes in bioinformatics), vol. 8824, Springer; 2014, p. 223–31. http://dx.doi.org/10.1007/978-3-319-12206-9_18.

[55] Zugaj W. Analysis of standard security features for selected NoSQL systems. Am J Inf Sci Technol 2019;3(2):41. http://dx.doi.org/10.11648/j.ajist.20190302.12.

[56] Pasqualin D, Souza G, Buratti EL, de Almeida EC, Del Fabro MD, Weingaertner D. A case study of the aggregation query model in read-mostly NoSQL document stores. In: Proceedings of the 20th international database engineering & applications symposium on. New York, New York, USA: ACM Press; 2016, p. 224–9. http://dx.doi.org/10.1145/2938503.2938546.

[57] Weintraub G, Gudes E. Data integrity verification in column-oriented NoSQL databases. In: Lecture notes in computer science (including subseries lecture notes in artificial intelligence and lecture notes in bioinformatics), vol. 10980 LNCS, Springer Verlag; 2018, p. 165–81. http://dx.doi.org/10.1007/978-3-319-95729-6_11.

[58] Solsol IL, Vargas HF, Díaz GM. Security mechanisms in NoSQL dbms's: A technical review. In: Communications in computer and information science. vol. 1154 CCIS, 2020, p. 215–28. http://dx.doi.org/10.1007/978-3-030-46785-2_18.

[59] Ahmadian M, Plochan F, Roessler Z, Marinescu DC. SecureNoSQL: An approach for secure search of encrypted NoSQL databases in the public cloud. Int J Inf Manag 2017;37(2):63–74. http://dx.doi.org/10.1016/j.ijinfomgt.2016.11.005.

[60] Ferretti L, Colajanni M, Marchetti M. Supporting security and consistency for cloud database. In: Lecture notes in computer science (including subseries lecture notes in artificial intelligence and lecture notes in bioinformatics), vol. 7672 LNCS, 2012, p. 179–93. http://dx.doi.org/10.1007/978-3-642-35362-8_15.

[61] Liu L. Security and privacy requirements engineering revisited in the big data era. In: Proceedings - 2016 IEEE 24th international requirements engineering conference workshops. Institute of Electrical and Electronics Engineers Inc.; 2017, p. 55. http://dx.doi.org/10.1109/REW.2016.7.

[62] Ghazi Y, Masood R, Rauf A, Shibli MA, Hassan O. DB-SECaaS: a cloud-based protection system for document-oriented NoSQL databases. EURASIP J Inf Secur 2016;2016(1):16. http://dx.doi.org/10.1186/s13635-016-0040-5, https://jis-eurasipjournals.springeropen.com/articles/10.1186/s13635-016-0040-5.

[63] Prasad P, Ojha B, Shahi RR, Lal R, Vaish A, Goel U. 3 dimensional security in cloud computing. In: 2011 3rd international conference on computer research and development. vol. 3, 2011, p. 198–201. http://dx.doi.org/10.1109/ICCRD.2011.5764279.

[64] EL Yamany HF, Capretz MA, Allison DS. Intelligent security and access control framework for service-oriented architecture. Inf Softw Technol 2010;52(2):220–36. http://dx.doi.org/10.1016/j.infsof.2009.10.005, https://linkinghub.elsevier.com/retrieve/pii/S0950584909001761.

[65] Song D, Shi E, Fischer I, Shankar U. Cloud data protection for the masses. Computer 2012;45(1):39–45. http://dx.doi.org/10.1109/MC.2012.1, http://ieeexplore.ieee.org/document/6127995/.

[66] Akoka J, Comyn-Wattiau I. Roundtrip engineering of NoSQL databases. Enterp Model Inf Syst Archit (EMISAJ) 2018;13:281–92.

[67] Shin K, Hwang C, Jung H. NoSQL database design using UML conceptual data model based on peter chen's framework. Int J Appl Eng Res 2017;12(5):632–6.

[68] Banerjee S, Sarkar A. Modeling NoSQL databases: from conceptual to logical level design. In: 3rd International conference applications and innovations in mobile computing. (February 2016):2016, p. 10–2.

[69] Abdelhedi F, Ait Brahim A, Atigui F, Zurfluh G. MDA-based approach for NoSQL databases modelling. In: Lecture notes in computer science (including subseries lecture notes in artificial intelligence and lecture notes in bioinformatics), vol. 10440 LNCS, 2017, p. 88–102. http://dx.doi.org/10.1007/978-3-319-64283-3_7, http://link.springer.com/10.1007/978-3-319-64283-3{_}7.

[70] Abdelhedi F, Ait Brahim A, Zurfluh G. Applying a model-driven approach for UML/OCL constraints: Application to NoSQL databases. In: Panetto H, Debruyne C, Hepp M, Lewis D, Ardagna CA, Meersman R, editors. In: Lecture notes in computer science (including subseries lecture notes in artificial intelligence and lecture notes in bioinformatics), vol. 11877 LNCS, Cham: Springer International Publishing; 2019, p. 646–60. http://dx.doi.org/10.1007/978-3-030-33246-4_40.

[71] Klettke M, Störl U, Scherzinger S. Schema extraction and structural outlier detection for JSON-based NoSQL data stores. In: Lecture notes in informatics (LNI), proceedings - series of the gesellschaft fur informatik (GI). vol. 241, 2015, p. 425–44.

[72] Sevilla Ruiz D, Morales SF, García Molina J. Inferring versioned schemas from NoSQL databases and its applications. In: Lecture notes in computer science (including subseries lecture notes in artificial intelligence and lecture notes in bioinformatics), vol. 9381, Cham: Springer; 2015, p. 467–80. http://dx.doi.org/10.1007/978-3-319-25264-3_35, http://link.springer.com/10.1007/978-3-319-25264-3{_}35.

[73] Brahim A, Ferhat R, Zurfluh G. Model driven extraction of NoSQL databases schema: Case of MongoDB. In: Proceedings of the 11th international joint conference on knowledge discovery, knowledge engineering and knowledge management. vol. 1, SCITEPRESS - Science and Technology Publications; 2019, p. 145–54. http://dx.doi.org/10.5220/0008176201450154.

[74] Fatima Kalna, Abdessamad Belangour, Mouad Banane AE. MDA transformation process of a PIM logical decision-making from NoSQL database to big data NoSQL PSM. Int J Eng Adv Technol 2019;9(1):4208–15. http://dx.doi.org/10.35940/ijeat.a1619.109119.

[75] de Lima C, dos Santos Mello R. A workload-driven logical design approach for NoSQL document databases. In: Proceedings of the 17th international conference on information integration and web-based applications & services. New York, NY, USA: ACM; 2015, p. 1–10. http://dx.doi.org/10.1145/2837185.2837218.

[76] Li Y, Gu P, Zhang C. Transforming UML class diagrams into HBase based on meta-model. In: Proceedings - 2014 international conference on information science, electronics and electrical engineering. vol. 2, 2014, p. 720–4. http://dx.doi.org/10.1109/InfoSEEE.2014.6947760.

[77] Morgado C, Busichia Baioco G, Basso T, Moraes R. A security model for access control in graph-oriented databases. In: 2018 IEEE international conference on software quality, reliability and security (QRS). IEEE; 2018, p. 135–42. http://dx.doi.org/10.1109/QRS.2018.00027, https://ieeexplore.ieee.org/document/8424965/.

[78] Maté A, Peral J, Trujillo J, Blanco C, García-Saiz D, Fernández-Medina E. Improving security in NoSQL document databases through model-driven modernization. Knowl Inf Syst 2021. http://dx.doi.org/10.1007/s10115-021-01589-x.