SeqBAC: A Sequence-Based Access Control Model

Diogo Domingues Regateiro[1], Óscar Mortágua Pereira[2], Rui L. Aguiar[3]
Instituto de Telecomunicações
DETI, University of Aveiro
Aveiro, Portugal
{diogoregateiro[1], omp[2], ruilaa[3]}@ua.pt

***Abstract*—Access control, when used in the context of database applications, is aimed to supervise the requests made by legitimate users to access sensitive data. These requests represent actions that a user can perform on a database and they typically read or write data. While this supervision can be formalized at a higher level, e.g. using an access control model such as RBAC, in the end, the data access is done through each authorized action. Therefore, the current access control models enforce their policies on an action by action basis, being unable to support relations of order between them. In many database applications, access to data is not done randomly, but by following very specific sequences of actions which are not supervised. This paper argues that a better security policy can be achieved by supervising these sequences. Thus, previous research is leveraged to propose a formalized model, capable of enforcing access control over the sequences of actions that can complement existing access control models.**

Keywords-information security, access control, sequence enforcement, database security, SeqBAC.

I. INTRODUCTION

Access control is a mechanism that limits the activity of legitimate users in a system. There are many strategies to enforce access control in a system, of which we emphasize the main four: discretionary access control (DAC), mandatory access control (MAC), attribute-based access control (ABAC) and role-based access control (RBAC). However, these access control models are not one-size fits all solutions, and so many other access control models exist [1][2][3][4][5]. Nevertheless, RBAC has risen as the dominating access control model used, especially for relational database applications. In this model, a user can only perform some action if he has been given permission to enact the role that governs said action. When it comes to data, actions are usually single read or write operations.

However, actions are not always independent of one another, some are used to collect values that are passed on to subsequent actions or to achieve some higher-level use case. A basic example of this would be a doctor prescribing some drug to a patient while allergies must be accounted for. First, the information about the patient would be selected, then any information about potential allergies for that user queried. This information is then used to filter the drugs that can be prescribed and added to the patient's profile. The referenced access control models do not support this kind of relation between actions to be encoded in the policies. A possible solution is for this dependency logic to be put into the application layer. Unfortunately, if the application is incorrectly implemented or exploitable, it may be possible to bypass this logic. If this is the case, it would be possible to perform authorized actions in unforeseen ways, such as prescribing a drug without allergies being checked. Another solution is to use stored procedures defined on the DBMS. However, they are not always supported and are not easily manageable since it is difficult to know which actions are being used and in which order.

A solution to prevent this dependency logic between actions from being bypassed is to design and validate the sequences in which actions are being executed. This approach benefits from the fact that designing sequences of actions can be done early in a project lifecycle, helping with implementation later. Moreover, a model based on these sequences could validate the sequences automatically and in real-time. In contrast, manually written code (e.g. application logic, stored procedures, etc.) is prone to implementation mistakes. These mistakes can compromise the correctness of the system, such as forgetting a crucial validation check, which can go unnoticed for extended periods of time. Additionally, by imposing an order of execution, it is possible to provide values for parameters directly from previous actions. This would lead to a more secure solution because it becomes possible to know the origin of the values passed as parameters instead of being only provided by the user.

Thus, we present a sequence-based access control model (SeqBAC) that aims to be able to encode the relations between actions that a user is authorized to perform and guarantee that they are executed in the sequence that they were meant to be. Additionally, the work presented in [6] could be used to implement a tool that could generate the code necessary to follow the defined sequences automatically and access the data. A previous iteration of this model [7] has seen a proof of concept, showing that such a concept is possible to execute and implement at a very basic level. This paper formalizes the results obtained from that proof of concept, expanding the concepts to include sequence branching and subsequence calls.

This paper is divided as follows: section II provides some of the state of the art, section III describes the model in terms of the desired policy, section IV introduces the necessary definitions to formalize the model, section V provides the model formalization and section VI provides our conclusions about the work.

II. RELATED WORK

There are many access control models in the literature. From the previously mentioned DAC, MAC, RBAC, and ABAC, to the Bell-LaPadula [8] model for government and military

This work is funded by National Funds through FCT - Fundação para a Ciência e a Tecnologia under the project UID/EEA/50008/2013 and SFRH/BD/109911/2015.
DOI reference number: 10.18293/SEKE2018-099

applications, and many others [1][2][3][4]. Other access control models based on finite state machines also exist[9][10], but they usually operate at a higher level, controlling authentication and other access control related tasks. The model proposed in this paper aims to control the lower level database operations.

The Extensible Access Control Markup Language (XACML) [11][12] has been proposed as a standard which includes a language for defining access control policies based on ABAC, an architecture for enforcing them and a processing model which describes how requests are evaluated. However, defining policies to control the sequence of actions a user may perform on a system is not within the original scope of ABAC. XACML could be extended to support the set of rules required to implement SeqBAC policies, but it would just be one implementation of the model herein described. Other access control languages exist, such as the Enterprise Privacy Authorization Language (EPAL) [12][13] which has been proposed to protect the customer's data privacy within a company, and it is another possible language where our model may be implemented on.

Barker [5] states that existing access control models all use the same basic concepts, which are then applied in a restrictive manner. Thus, Barker proposes a meta-model for access control based on these basic concepts and shows some examples of how other access control models are supported by this model. This approach was studied but ultimately seemed inadequate for the access control model being presented, as actions have several order relations between them, i.e. actions can exist in several sequences, and this was not supported by Barker's meta-model. Staab and Muller [14] also introduced the MITRA framework, which is another meta-model for information flow where trust and reputation architectures are in place.

Non-deterministic access control models could also be considered for implementation of sequences of actions, especially when branches are considered and users can follow any of them. Several of these models exist, of which we emphasize: probabilistic models to determine risk [15][16][17], cognitive-based systems [18] and fuzzy theory-based models [19][20]. Ultimately, the SeqBAC model presented here is meant to be deterministic, which would allow security experts to conduct auditing and to keep a level of assurance that no unexpected access decision is made.

The model in this paper builds upon a CRUD expression driven access control model [7][21] and generalizes it so that it no longer depends on the RBAC model to authorize to execute the actions. In [21], an architecture was proposed to enforce access control based on RBAC, and driven by the CRUD expressions that are naturally part of the domain of the applications using the architecture. In [7], an extension to the RBAC model was proposed to support basic sequences of CRUD expressions to complement the role-based approach. We allow sequences of CRUD expressions to be ruled by roles, and users can follow these sequences if they are allowed to play the role. However, this extension was limited to simple chains of CRUD expressions and it lacked a proper formalized model.

While to the best of our knowledge no attempts have been made to create a model that can enforce sequences of actions to access the data, the process of altering an existing model to re-purpose it for other scenarios is widely used in the literature. Many examples of this practice exist [2] where the RBAC model is extended with geographic information for the purposes of using a user's location to allow or deny access to data. The formalized model in this paper addresses this gap and generalizes the work in [7] to sequences of actions.

III. BASE POLICY

In this section, the SeqBAC model being proposed will be described in terms of the simplest base scenario that it aims to support.

A simple policy could contain just an ordered set of actions and parameter tuples, where an authorized user could execute the first action, then the second, etc. However, scenarios such as the drug prescription, described in section I, could require a doctor to go back in the order of execution to add a different drug to a prescription after a previous one had been found to cause allergic reactions on a patient. Thus, it is necessary to define which actions a user can take at some point in a use case execution.

Since the policy is meant to restrict the order in which actions can be performed on a database by a legitimate user in the terms described above, such a policy should contain the following:

- A set of actions A and their input parameters P.
- A set E of directed transition relations between actions.
- The set of users U allowed to execute the policy.

The set of users can be defined explicitly or implicitly through some condition the users must satisfy, such as be playing some role, possess a set of attributes, etc.

This type of policy is more descriptive than other policies, such as RBAC policies, due to the set of transition relations between the actions. There is an initial action, which each user executes first, and then the users can execute other actions by following the transition relations between them. Given the fact that actions are defined with transitions between them, a policy in SeqBAC defines a sequence of actions, hereby known as an action flowchart.

Hence, the SeqBAC model supervises a set of actions that can be executed over the set of available data and will limit the executions of these actions to certain sequences. This list of actions is not required to be complete *apriori* for the model to be used: new actions may be added at any given time by the system's administrator or other authorized users. Furthermore, the sequences of actions may branch, allowing the users multiple actions to choose from to allow flexibility. Sequences of actions may also be reused in other sequences when their purpose is needed in several situations, and this will be pursued in more detail in section V.C.

The importance of these policies is that, in many cases, actions over data are not executed randomly. Instead, actions are executed following some notion of order that is related some use case. However, these sequences are not generally encoded in the access control mechanisms, which enforces access control on an action by action manner. This can lead to unintended results when malicious users can execute actions by impersonating a

legitimate user.

To exemplify one such policy, consider the following set of four actions {A, B, C, D} in the context of an online shop. Action A queries the database for client information to authenticate it, action B queries for the checkout cart of the client, action C allows the client to review its payment options and action D allows it to place an order. In this scenario, we will require action A to be done always first, following it by action B to show the current checkout cart to the client once the authentication succeeds. Then, once the client decides to place the order, the client may want to review and update their payment options before doing so, making action C optional and finalizing with action D.

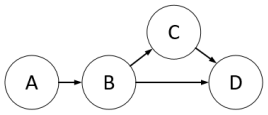


Figure 1. Scenario representation of the relations between actions.

This scenario is represented in Figure 1, where each action is represented by its letter and an arrow connects one action to the next one that naturally follows as per description. If a malicious user breaches the application and tries to obtain payment details, it is unable to do so. The action to review the payment options is in the middle of a sequence of actions, and to reach it the malicious user would have to execute actions A and B. Since action A is used for authentication, unless the malicious user possesses the authentication credentials or breaches the DBMS itself, it will be impossible to access the payment options of the users.

IV. Model Concepts

Formally, the SeqBAC model is a set of action flowcharts associated with each user. It follows closely the existing concepts that govern flowcharts, and as such we will use flowchart notation to formalize the model.

First, consider the set of defined actions *A* defined in formula 1 and a parameter *P* defined in formula 2, where actions a_1 to a_N are the actions defined in the system, *name* is a string of characters and *datatype* represents the datatype of the associated parameter in the database, if relevant.

$$A = \{a_1, a_2, \ldots a_N\} \tag{1}$$

$$P = (name, datatype) \tag{2}$$

With these concepts, we can define the set of action nodes *V* in our model's flowcharts, where each node is a pair of an action and a set of parameters as shown in formula 3. An action allows an authorized user to access or modify some subset of the data.

$$V = \{(a, \{P\})\}, a \in A \tag{3}$$

Having defined the action nodes of the flowchart we now need to define the transitions between them. The set of valid transitions *E* are defined as a set of ordered 2-element from *V*, which forms the unidirectional transitions between elements of *V*. Formally, the set of transitions *E* is defined in formula 4.

$$E = \{(u, v) : u, v \in V\} \tag{4}$$

Formula 5 defines the flowchart *G*, used to model a high-level use case. Each flowchart is an ordered pair of a set of action nodes *V* and a set of transitions *E* that connects two action nodes.

$$G = (V, E) \tag{5}$$

Additionally, we define the functions *ActionSet(G)* to return the set of action nodes of the flowchart *G* and *TransitionSet(G)* to return the set of transitions.

Finally, formula 6 defines the set of flowcharts *SOF* that makes up the model. Each user *U* is then given a subset of SOF_U which they are authorized to execute, as shown in formula 7.

$$SOF = \{G_1, G_2, \ldots G_M\} \tag{6}$$

$$SOF_U \subseteq SOF \tag{7}$$

V. Model Definition

In order to enforce SeqBAC, a way to define how a user can be tracked along a flowchart $G \in SOF$, is needed. Furthermore, the possible move operations that a user can perform at a given node, as well as what type of information can flow from one node to the next is required.

From the information provided in the previous section, a definition of the SeqBAC model can now be created.

Definition 1: SeqBAC. The SeqBAC model has the following components:

- *A*, *P*, and *U* denote actions, parameters, and users, respectively;
- *V* denotes an action node, which is a tuple of an action and a set of parameters necessary to execute the action;
- $E \subseteq V \times V$ is a relation between action nodes, describing the authorized transitions;
- $G = (V,E)$ and SOF denote an action flowchart, with the set of actions and the authorized transitions between them and the set of all defined action flowcharts, respectively;
- auth $: U \longrightarrow 2^G$ is a function that determines if a user is allowed to access a certain action flowchart: $\text{auth}(u_i) \subseteq SOF, u_i \in U$.

Given that enforcing access control over sequences of actions is the primary concern, it is required to locate the position of a user within a sequence at any time. To achieve this, the User Access Pointer is defined.

Definition 2: User Access Pointer. Given a SOF, the user access pointer (UAP) is a pair of elements *(G, v)* that uniquely identifies a flowchart $G \in SOF$ and the current node $v \in V$ the user is allowed to use.

This UAP allows a system to keep track of which flowchart the user is using and on which node within it. We will now define how the UAP can be updated in order to move the user within a flowchart, which uses an operation called *Stepping* and it involves moving along a transition of the flowchart.

Definition 3: Stepping. Consider the flowchart *G* and its UAP

on step *n* of the flowchart traversal, denoted UAP_n. Stepping is the process in which UAP_{n+1} is generated by referencing a new node such that:

$$\forall_x \forall_y \left(\begin{array}{c} \left(UAP_n = (G, x) \wedge UAP_{n+1} = (G, y) \right) \Rightarrow \\ (x, y) \in TransitionSet(G) \end{array} \right) \quad (8)$$

Definition 3 constraints moving from node to node, and therefore updating the UAP, along the transitions between them as previously informally described. When a UAP first references the initial node of the flowchart, we consider it to be in step 1 (UAP_1) and the step counter increments with each stepping. A UAP on step 0 (UAP_0) is not referencing any node and is used when no flowchart is currently being traversed by the user.

There are several different situations in which *Stepping* may be used, and they differ on the in-degree and the out-degree of the nodes involved, i.e. the number of transitions in and out of a node respectively, as well as the direction of the transitions between them. The transition between nodes will be detailed first without considering the information that can flow between them.

A. Stepping

We will now describe several scenarios in which *Stepping* operations may occur and how they are handled within the context of our model. The most trivial *Stepping* operation occurs when we have two nodes *A* and *B*, where node *A* has an out-degree of 1 and node *B* an in-degree of 1, as shown in Figure 2. In this case, if the UAP is at node *A*, then by Definition 3 it follows that it can only move to node *B*.

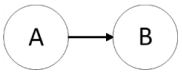
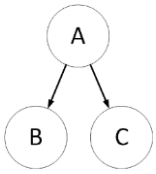


Figure 2. Stepping's trivial case.

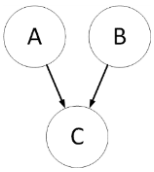
We will now analyze the different type of *Stepping* operations that can occur when the in-degrees and out-degrees differ. When the out-degree of node *A* is bigger than 1, then we have more than one node that can satisfy Definition 3 and we say that it causes *Splitting* in the sequence. In this case, the user can decide which node to go to. This case is analogous to the piece of pseudo-code on Figure 3, where the action *A* is always executed, and then either action *B* or *C* are executed depending on some condition criteria.




```
do A
if(some condition) then
   do B
else
   do C
end if
```

Figure 3. Splitting example and associated code.

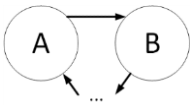
We can now consider the opposite situation, where the in-degree of a node is bigger than 1. This is shown in Figure 4 and it represents a situation where two or more branches of a sequence join at the node. For the user, it is exactly the same situation as in a trivial *Stepping* operation, but he would have been able to reach that node through some other sequence of nodes in the flowchart. In the example, node *C* is simultaneously reachable from both nodes *A* and *B*. This case is analogous to the piece of pseudo-code on Figure 4, where either action *A* or *B* are executed depending on some condition criteria. Then, action *C* is executed independently of the condition criteria.




```
if(some condition) then
   do A
else
   do B
end if
do C
```

Figure 4. Merging example and associated code.

So far, we've only discussed situations were *Stepping* moves the user forward in a flowchart. However, if a transition exists in both directions between two nodes, then it is possible for the user to step between those two nodes as many times as it wants given that the only restriction to *Stepping* is the Definition 3. We call this situation a *Cycle* in the sequence, as demonstrated in Figure 5, and it could potentially be created with many in-between nodes. While this can make sense in some situations, it should be possible to restrict the number of times the user can go back to the same node. Additional restrictions will be discussed in section V.B.2). This case is analogous to the piece of pseudo-code on Figure 5, where the actions *A*, *B*, etc. can be executed in a cycle. The cycle ends when the execution transitions out of the cycle, normally on the last node.




```
while(some condition)
    do A
   do B
   …
end while
```

Figure 5. Cycle example and associated code.

One particular case of a *Cycle* is when one is created using only one node that can transition to itself. We call this case a *Loop* and it allows a user to reuse the same node several times, possibly with information obtained from past accesses. Bulk operations can benefit from this type of transition.

B. Information Flow

One important fact to consider is that most data access is not static, i.e. parameters can be used to select or modify data. In fact, some of the data used as input parameters might need to have its source validated to ensure that the user does not access data that he is not authorized to access. To achieve this type of data validation it is proposed that nodes can use data from previous nodes as parameters. We refer to this concept as the information flow and we will define it in this section.

1) User Context

To handle the passing of information between nodes, we need to create a user context that contains all the data a user accessed and that a node can use to parameterize the data access. To define this context, we will first define the *Accessed* predicate that indicates whether a node *v* was accessed in the past or not.

Definition 4: Accessed(G, v). A node $v \in ActionSet(G)$ for a given flowchart *G* is said to have been accessed when a user's UAP possessed a reference to node *v* on at least one step leading up to the current step *N*:

$$\forall_{v \in ActionSet(G)} \exists_{n \leq N} \left(Accessed(G, v) \Rightarrow UAP_n = (G, v) \right) \quad (9)$$

Definition 5: User Context. Given a user U and each graph G from its set of flowcharts SOF_U, the User Context of user U (UC_U) is a pair of elements containing the current UAP on step N (UAP_N) and a set of previously accessed nodes by user U, which can be referenced by the predicate $AccessedSet(UC_U)$.

$$UC_U = (UAP_N, \{v : v \in ActionSet(G) \wedge Accessed(G, v)\}) \quad (10)$$

This UC_U is updated each time a *Stepping* operation occurs and it can be used by a node to obtain values to parametrize the data access. This means that the user does not provide the parameters himself, ensuring that the data is valid and that the user can request it. However, the UC must be made secure to prevent a user from breaching it in any way. Additionally, when a user stop traversing a flowchart, the UC_U should be emptied so the previously accessed data cannot be used out of context. Formally, the reset operation of the UC_U empties the set of accessed nodes and puts the current UAP back to step 0:

$$\text{Reset}(UC_U) : UC_U = (UAP_0, \emptyset) \quad (11)$$

There are several ways to implement this method of passing a result of a previous action to another as a parameter. However, one way to do this could involve the usage of a protected server that caches the request results sent to each client, and then it would automatically pass the necessary information to each action when the client requests it to be executed.

2) Information Flow Restriction

One important aspect to ensure that data is not accessed out of context is the ability to prevent an action from using some specific data obtained through a previous action as parameter inputs. Thus, a method to remove unneeded data from the UC and free resources on the system is desirable.

To address this, we introduce the ability to revoke access to the data obtained from a previously accessed node. The revocation is done automatically using a list of nodes targeted for revocation, which we call the revocation list. This list can exist in the transitions between nodes or on each node. We will analyze both options.

Definition 6: Revocation List. Given a flowchart $G \in SOF$, a revocation list R is a set of previously accessed nodes in $ActionSet(G)$ that must prevent further access to their data.

$$R = \{v' \in ActionSet(G)\} \quad (12)$$

Since the enumerated nodes in the list prevent any further access to their data, it cannot be used as parameter values for consequent access attempts. We must now update the user context definition to contemplate the revocation list, i.e. the user context for some user U and a flowchart G must now contain, at a given step N, the list of previously accessed nodes that have not appeared on any revocation list, as shown in formula 13.

$$UC_U = (UAP_N, \{v : v \in ActionSet(G) \wedge Accessed(G, v) \setminus R\}) \quad (13)$$

When the revocation list is encoded in the transitions between nodes, the set of transitions will then be defined by an ordered triplet of two nodes and a revocation list:

$$E = \{(u, v, R) : u, v \in V\} \quad (14)$$

When a *Stepping* operation is carried out, the revocation list associated with it is processed and the nodes in the list can be removed from the list of nodes in the user context.

In the other solution, i.e. placing the revocation list at each node, the revocation list is associated with each node in the flowchart G instead:

$$V = \{(a, \{P\}, R)\}, a \in A \quad (15)$$

Whenever a *Stepping* operation is carried out and some node v is referenced by the UAP, following the restriction imposed in Definition 3, the user context must be updated with the revocation list by subtracting the list from the user context's list of accessed nodes. Both approaches are valid and which one is used depends solely on the ease of implementation.

C. Subsequence Calls

We will now describe the process by which sequences of actions may be reused on other sequences, also known as a sub-sequence call. Sub-sequence calls are operations that move the UAP to the root of another flowchart G to perform some action that is required by multiple other flowcharts, making it a common action. Figure 6 shows this idea, where the node A' on the flowchart to the right is a call to the entire sub-sequence to the left. Two main types of sub-sequence calls are considered in this work: dependent and independent.

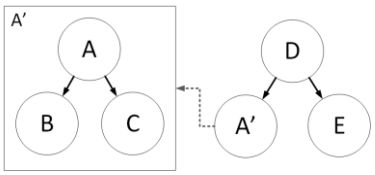


Figure 6. Sub-sequence calls example.

Dependent calls are distinguished from independent calls by their need to use the current UC. As the name implies, dependent calls require the UC to be passed to the sub-sequence, allowing access to the data obtained from the initial flowchart.

Independent calls, however, do not require the data from the initial flowchart to perform the data access. This ensures that data is not misused and passed only on a need-to-know basis. Just like the *Stepping* operation, the sub-sequence call allows the user to move to another node except the node exists on a different flowchart. When the UAP reaches the end of the sub-sequence G', the UC from it is copied to the initial flowchart G, allowing the node the user moves into in the original flowchart G to access the data obtained, if any. This doesn't violate the definition of the User Context (Definition 5) since the set of accessed nodes do not have to belong to a single flowchart.

More formally, when a call to a sub-sequence finishes and returns a UC', the initial UC_n is updated to UC_{n+1} as follows:

$$UC_{n+1} = \left(UAP_n, AccessedSet(UC_n) \vee AccessedSet(UC') \right) \quad (16)$$

Note that the original UAP remains the same as it was before the sub-sequence call was made, and it is only updated when the user execution returns to the original flowchart. It is also important to note that the dependent calls and the UC' returned by the sub-sequence call can have revocation lists associated with them as described in section V.B.2). However, since the new UC' is copied from the original UC, the revocation list of a

dependent call only affects *UC'*, leaving *UC* untouched. The *UC* can be restricted using a revocation list after the sub-sequence call terminates and the user moves to the next node via *Stepping*.

Additionally, when the UC is copied for the execution of a sub-sequence, the node of the original sequence is not passed along with it. This means that when a sub-sequence finishes execution, the system still needs to know where to point the user to continue the sequence execution. Therefore, every time a subsequence call is performed, the current UC should be saved.

A natural solution to save the UCs in is a stack, in which the last item to be stored is the first to the removed. This way, when a sub-sequence is called, the UC used is put into the stack and copied to the sub-sequence. When a sub-sequence terminates, the last UC is removed from the stack, merged with the current one, and the previous UAP recovered as shown in formula 16.

D. Implementation

While this paper is concerned primarily with defining the SeqBAC model, it can also be beneficial to consider how such a model can be implemented. The unique way that actions interact with each other through transition relations makes it clear that a graph is the type of structure seems to be the most appropriate for storing SeqBAC policies. Furthermore, graph databases, such as Neo4j, allow to define properties on each node and edge. This feature could be used to store things such as subsequence calls by referencing another graph, the revocation list for accessed data pruning purposes and other data.

However, storing the actions and the relations between them is not enough. If the access control system only checks if a user is authorized to execute some action, then either the user knows the sequences or can experience many authorization errors. This can be expected to happen since there can be many ways to fulfill a use case and a defined sequence may only account for a specific way to do it. A solution to this implementation issue is the development of a tool that can parse the defined sequences and then generate the necessary code that follows them automatically. A previous work, presented in [6], shows how this could be used to integrate a similar idea into existing DBMS solutions. This way, when an application using SeqBAC is being developed, the developers do not need to master the policies defined as action flowcharts.

VI. Conclusion

In this paper, the SeqBAC model was introduced and formalized. The model was designed to enforce access control policies over sequences of actions, allowing users to execute them in controlled sequences, and extends a previous work.

This paper also considers how this model could be implemented, addressing the issue of developers having to master the defined sequences of actions with the proposal of using a tool to parse the defined sequences and generating the code to use them automatically. While this model requires some work to define the sequences of actions when compared to other models that allow unrestricted access to data, it helps to ensure that the use cases are implemented correctly faster.

Regarding future work, an actual implementation of the example discussed in section V.D is thought to be next natural step. Additionally, tools to define policies, validate source-code and generate mechanisms based on the defined policies are also being considered. These would allow developers to know easily what operations are available at any point during the execution of a sequence, preventing the need for them to master the policies, and to ensure the overall correctness of their code.

References

[1] E. Bertino, P. A. Bonatti, and E. Ferrari, “Trbac,” *Proc. fifth ACM Work. Role-based access Control - RBAC '00*, no. May 2016, pp. 21–30, 2000.

[2] M. L. Damiani, E. Bertino, B. Catania, and P. Perlasca, “Geo-Rbac,” *ACM Trans. Inf. Syst. Secur.*, vol. 10, no. 1, p. 2–es, 2007.

[3] E. Tarameshloo and P. W. L. Fong, “Access control models for geo-social computing systems,” in *Proceedings of the 19th ACM symposium on Access control models and technologies - SACMAT '14*, 2014, pp. 115–126.

[4] I. Ray and M. Toahchoodee, “A Spatio-temporal Role-Based Access Control Model,” *Data Appl. Secur. XXI*, vol. 4602, pp. 211–226, 2007.

[5] S. Barker, “The next 700 access control models or a unifying meta-model?,” *Proc. 14th ACM Symp. Access Control Model. Technol.*, pp. 187–196, 2009.

[6] Ó. M. Pereira, D. D. Regateiro, and R. L. Aguiar, “Secure, Dynamic and Distributed Access Control Stack for Database Applications,” *Int. J. Softw. Eng. Knowl. Eng.*, vol. 25, no. 09n10, pp. 1703–1708, Nov. 2015.

[7] Ó. M. Pereira, D. D. Regateiro, and R. L. Aguiar, “Extending RBAC Model to Control Sequences of CRUD Expressions,” *SEKE'14 - Intl. Conf. Softw. Eng. Knowl. Eng.*, 2014.

[8] D. Bell and L. LaPadula, “Secure Computer Systems: A Mathematical Model. Volume II.,” vol. II, no. May 1973, 1973.

[9] Bin Duan and Bing Liu, “Design of security state machine of access control for control object based on IEC 61850,” in *2006 IEEE Power Engineering Society General Meeting*, 2006, p. 3 pp.

[10] M. Thirumaran, P. Dhavachelvan, D. Aishwarya, and R. Shanmugapriya, “Finite State Machine based Access Control Mechanism for Web Service Work Flow Management,” *IERI Procedia*, vol. 4, pp. 391–397, 2013.

[11] B. Parducci and H. Lockhart, “eXtensible Access Control Markup Language (XACML) Version 3.0,” *OASIS Standard*, 2013.

[12] G. Yee, *Privacy Protection for E-Services*. Idea Group Inc (IGI), 2006.

[13] J.-W. Byun, *Toward Privacy-preserving Database Management Systems - Access Control and Data Anonymization*. ProQuest, 2007.

[14] E. Staab and G. Muller, “MITRA: A Meta-Model for Information Flow in Trust and Reputation Architectures,” *arXiv Prepr. arXiv1207.0405*, p. 19, Jul. 2012.

[15] R. McGraw, “Risk-Adaptable Access Control (RAdAC),” *inPrivilege Manag. Work. NIST–National Inst. Stand. Technol. Technol. Lab.*, 2009.

[16] D. R. dos Santos, R. Marinho, G. R. Schmitt, C. M. Westphall, and C. B. Westphall, “A framework and risk assessment approaches for risk-based access control in the cloud,” *J. Netw. Comput. Appl.*, vol. 74, pp. 86–97, Oct. 2016.

[17] S. Kandala, R. Sandhu, and V. Bhamidipati, “An Attribute Based Framework for Risk-Adaptive Access Control Models,” in *2011 Sixth International Conference on Availability, Reliability and Security*, 2011, pp. 236–241.

[18] IBM, “Cognitive Security White Paper,” 2016. [Online]. Available: http://cognitivesecuritywhitepaper.mybluemix.net/. [Accessed: 11-Jan-2017].

[19] C. Martínez-García, G. Navarro-Arribas, and J. Borrell, “Fuzzy Role-Based Access Control,” *Inf. Process. Lett.*, vol. 111, no. 10, pp. 483–487, 2011.

[20] J. Kacprzyk, S. Zadrożny, and G. De Tré, “Fuzziness in database management systems: Half a century of developments and future prospects,” *Fuzzy Sets Syst.*, vol. 281, pp. 300–307, Dec. 2015.

[21] Óscar Mortágua Pereira, D. D. Regateiro, and R. L. Aguiar, “Role-Based Access Control Mechanisms,” *... (ISCC), 2014 IEEE ...*, vol. 2, no. 1, pp. 1–7, Jun. 2014.