Monotonic and Non-monotonic Context Delegation

Mouiad AL-Wahah and Csilla Farkas
College of Engineering and Computing, University of South Carolina, 301 Main St, Columbia, SC, U.S.A.

Keywords: Description Logic, Monotonic, Non-monotonic, OWL Ontology, Context Delegation, Access Control, Authorization.

Abstract: Delegating access privileges is a common practice of access control mechanisms. Delegation is usually used for distributing responsibilities of task management among entities. Delegation comes in two forms, GRANT and TRANSFER. In GRANT delegation, a successful delegation operation allows delegated privileges to be available to both the delegator and delegatee. In TRANSFER delegation, delegated privileges are no longer available to the delegator. Although several delegation approaches have been proposed, current models do not consider the issue of context delegation in context-based access control policies. We present two ontology-based context delegation approaches. Monotonic context delegation, which adopts GRANT version of delegation, and non-monotonic for TRANSFER version of delegation. The approach presented here provides a dynamic and adaptive privilege delegation for access control policies. We employ Description logic (DL) and Logic Programming (LP) technologies for modeling contexts, delegation and CBAC privileges. We have designed three lightweight Web Ontology Language (OWL) ontologies, CTX, CBAC, and DEL, for context, Context-Based Access Control (CBAC), and delegation, respectively. We show that semantic-based techniques can be used to support adaptive and dynamic context delegation for CBAC policies. We provide the formal framework of the approaches and show that they are sound, consistent and preserve least-privilege principle.

1 INTRODUCTION

In dynamic environments like pervasive systems, ubiquitous computing, ambient intelligence, and more recently, Internet of Things (IoT), there is an association between a system and its environment and the system always adapts to its ever-changing environment. This situation makes the Context-Based Access Control (CBAC) the method of choice for such environments.

The fundamental idea behind delegation is that some active entity in a system, the delegator, delegates privileges to another active entity, the delegatee, to carry out some functions on behalf of the former. Delegation comes in two forms (Crampton and Khambhammettu, 2008), GRANT and TRANSFER. In GRANT delegation, a successful delegation operation allows delegated permissions to be available to both the delegator and delegatee. In TRANSFER delegation, the delegated permissions are no longer available to the delegator.

There is a significant previous work on Context-Based Access Control (CBAC) (Bhatti et al., 2005; Toninelli et al., 2006; Kulkarni and Tripathi, 2008; Shen and Cheng, 2011; Corradi et al., 2004; Trnka and Cerny, 2016; Kagal et al., 2006; Gusmeroli et al., 2013). However, support to delegate CBAC privileges is limited. For example, approaches described in (Bhatti et al., 2005; Toninelli et al., 2006; Kulkarni and Tripathi, 2008; Shen and Cheng, 2011; Bellavista and Montanari, 2017) do not provide any delegation services. Most of the existing delegation methods are based on traditional access control models, such as Role-Based Access Control (RBAC) models (Trnka and Cerny, 2016; Zhang et al., 2001). Methods such as attribute-based delegation (Servos and Osborn, 2016; Servos and Osborn, 2017) and capability-based delegation (Kagal et al., 2006; Gusmeroli et al., 2013) requires that the underlying access control policy is changed. Moreover, none of the methods address the issue of context delegation when the access authorization is a context-dependent. Furthermore, only a few approaches have extensively studied TRANSFER delegation due to the complexity of enforcing TRANSFER delegation mechanisms. This paper extends the work presented by the same authors in (AL-Wahah and Farkas, 2018).

The main contributions of our approach are: 1) Our method provides dynamic and adaptive context

Al-Wahah, M. and Farkas, C.
Monotonic and Non-monotonic Context Delegation.
DOI: 10.5220/0007400904490460
In *Proceedings of the 5th International Conference on Information Systems Security and Privacy (ICISSP 2019)*, pages 449-460
ISBN: 978-989-758-359-9

delegation that does not modify the original access control policy in case of monotonic delegation. For non-monotonic delegation, the approach incurs only selective changes to the underlying access control policy. Dynamic adaptability is provided through policy adjustment operations that keep the policy rules on update with the latest context changes. Adaptation is needed, in our approach, in situations: (i) When context of the requester/resource/environment changes, that change should be reflected on the access control policy. (ii) In TRANSFER delegation, the delegated permissions have to be moved (temporarily or permanently) from the delegator to the delegatee and this needs the policy to be updated to cope with this situation. 2) Our approach can be adopted by existing CBAC systems which do not provide delegation services. 3) Our semantic-based delegation model supports capabilities such as checking the access control and delegation policies for conflict and consistency, explaining inferences and helping to instantiate and validate the variables in dynamic environments. 4) We provide the formal proof that context delegation and its delegation-supported authorization enforcement are sound, consistent and preserve the least privilege principle.

The rest of this paper is organized as follows: In section 2, we present the context-based access control system modeling. Section 3 is dedicated to semantic-based context delegation. In section 4, we present the properties of our approach and the proofs of these properties, and finally in section 5, we conclude with suggestions for future work.

2 CBAC SYSTEM MODELING

In this section, we give a brief overview of the Context-Based access control. "Context" has been defined by Dey et al. (Dey et al., 2001) as "any information that is useful for characterizing the state or the activity of an entity or the world in which this entity operates." In CBAC, the system administrator (or resource owner) specifies a set of contexts and defines for each context c the set of applicable permissions. When an entity (a user) operates under a certain context, (s)he acquires the set of permissions (if any) that are associated with the active context. When (s)he changes the active context, the previous permissions are automatically revoked, and the new permissions acquired (Corradi et al., 2004). Hence, Context plays the role of a bridge between the requester and the access permissions. If a requester has a context c at the time in which the request is made and that c is associated with permission p, then (s)he can access a resource using permission p.

2.1 Context-based Access Control Model

Access requests are evaluated based on the contexts associated with the subject and the requested. The request is matched with context metadata that specify and activate the policy rule that to be enforced. We use rule-based Logic Programming (LP) to encode context and policy rules.

Definition 1. (Access Control Policy (ACP) Rules.) Access control policy rule is given as a 6-tuple $\langle s, sc, r, rc, p, ac \rangle$, where $s \in Subject$, $r \in Resource$, sc, $rc \in Context$, where sc is the subject's context and rc is the resource context, $p \in Permission$=$\{$"*Deny*","*Permit*"$\}$, and $ac \in Action$=$\{read, write, delegate, revoke\}$. Each rule is instantiated by an access request, using the model ontologies and rules, and is evaluated at runtime to reach a decision.

Definition 2. (Access Request (AR).) An Access request is given as a triple $\langle s, r, ac \rangle$, where $s \in Subject$, $r \in Resource$, $ac \in Action$.

For example, an access request denoted as $ar = \langle s, r, "read" \rangle$, represents the case when subject s is requesting a "*read*" access to a resource r. The policy engine requests the contexts of s and r, and evaluates the permission p for the request ar. Assume the contexts of s and r are sc and rc, respectively. If using the contexts sc and rc, the policy engine can derive a permission, i.e., p is "*Permit*", and there is no conflict, it grants the access permission for the request. Otherwise, it denies the request. This kind of access authorization does not involve the case when delegation is present within access control policy setting. Algorithm 1 represents this kind of access authorization.

Definition 3. (Monotonic Reasoning.) Let O_1 and O_2 two DL ontologies, and c is a DL axiom. O_1 entails c (equivalently, c is a logical consequence of O_1), written as $O_1 \models c$ if we have $O_2 \subseteq O_1$ and $O_2 \models c$. In secure authorization terms, monotonic authorization reasoning means that positive (and negative) authorizations will not be altered when new facts are added into the knowledge base. Hence, what is previously inferred as a permitted action still holds even after new facts are asserted (or inferred) into the knowledge base.

Definition 4. (Closed World Assumption (CWA) (Pratt, 1994).) Closed World Assumption is the assumption that what is not known to be true is false. In CWA, absence of information is interpreted as negative information. CWA assumes complete informa-

tion about a given state of affairs, which is useful for constraining information and validating data in an application such as a relational database.

Algorithm 1: Simple Access Authorization.

Input: *CBAC*, CBAC ontology and *RQ* is an Access Request. *RS* is Jena rule-set
Output: Access decision, either "Deny" or "Permit".

```
RT ← parse(RQ)                                   ▷ RT = ⟨s, r, ac⟩
if RT = access then
    sc ← getContext(s);
    rc ← getContext(r);
    p ← evaluate(s, sc, r, rc, CBAC, RS);
    if p = "Permit" AND noConflict(p, s, sc, r, rc, CBAC, RS) then
        return("Permit");
        exit();
    end if
else
    return("Deny");
    exit();
end if
```

Definition 5. (Non-monotonic Reasoning.) Let KB_1 and KB_2 be two knowledge bases, and c is a consequence. We may have that $KB_2 \subseteq KB_1$ and $KB_2 \models_d c$, but $KB_1 \not\models_d c$. In secure authorization terms, non-monotonic authorization reasoning means that positive (and negative) authorizations may be altered when new facts are added into the knowledge base. Hence, what is previously inferred as a permitted action may not hold when new facts are asserted (or inferred) into the knowledge base.

Definition 6. (Open World Assumption (OWA) (Hitzler et al., 2010).) The Open World Assumption is the assumption that what is not known to be true, is unknown. In OWA, absence of information is interpreted as unknown information that may be added later. OWA assumes incomplete information about a given state of affairs, which is useful for extending information in an application such as ontology-based applications and Semantic Web.

For the sake of this paper, CWA is better suited, because we need to capture who has access permission to a certain resource and not the reverse. CWA assumes non-deductibility as a failure to prevent granting access rights. For example, the access authorization algorithm needs to know who has the permission to access a certain resource but not all those who have not the access permission to that resource. For this reason, we have used Jena forward chaining rules to assert access authorization decisions and also for blocking delegation-based access permissions. Jena rules are used to enforce the CWA after all semantic inferences are made.

Definition 7. (Least Privilege Principle (Schneider, 2003).) The least privilege principle states that a subject should be given only those privileges that it needs in order to complete its task

2.2 Ontology-based Context Model

To model the context, we adopt a Description Logic (DL)-based method that partially resembles the method adopted by Bellavista and Montanari (Bellavista and Montanari, 2017). However, our context representation differs than that adopted by (Bellavista and Montanari, 2017). They have tightly coupled the subject's context (they call it the requester context), the resource's context, the environmental context and the time context in one context (protection context). In our model, the subject's context and resource's context are separated. To support context delegation, we modify the subject's context only. We represent our model using the OWL-DL ontologies, the reader is referred to (Hitzler et al., 2010) and (https://www.w3.org/OWL/, Data accessed on April 13 2018) for additional description on the current OWL standard.

Our context model is built around the concept of contextual attribute, information which models contextual attributes of the physical/logical environment such as location and temperature. Specific context subclasses can be represented under Generic Concept *Context*. Each subcontext class consists of attribute values and constants. In our model, the generic context of the subject is given by the following DL axiom:

$$\begin{aligned} &SContext \equiv Context \sqcap (User \sqcap \exists hasID.IDentity \\ &\sqcap \exists hasRole.Role \sqcap \exists hasGroup.Group) \\ &\sqcap (Environment \sqcap \exists hasLocation.Location) \sqcap \\ &(TElement \sqcap \exists hasTime.Time\_Interval) \sqcap \exists \\ &hasID.Identifier \end{aligned} \tag{1}$$

For example, A reference context of *OnDutyDoctor* is represented as follows:

$$\begin{aligned} &OnDutyDoctor \equiv Context \sqcap (User \sqcap \exists hasID.IDentity \\ &\sqcap \exists hasRole.Role\{Doctor\} \sqcap \exists hasGroup.Group \\ &\{InShiftDoctors\}) \sqcap (Environment\{WorkingEnvrnt\} \\ &\sqcap \exists hasLocation.Location\{Hospital\}) \sqcap (TElement \\ &\{WorkingTime\} \sqcap \exists hasTime\{xsd : dateTime \\ &[\geq 2018-04-06T09:00:00, \leq 2018-04-06 \\ &T17:00:00]\}) \sqcap \exists hasID.\{0\} \end{aligned} \tag{2}$$

Note that the concept *OnDutyDoctor* includes all the characteristics specifications of the generic concept

SContext. We call this context a *reference context*. It holds the high-level context of an entity which will be used later as a reference when we need to instantiate the active context of that entity. The *active context* holds the entity context at a specific instant of time. For example, when an entity requests an access to a resource. Active contexts are similar to their *reference contexts* counterparts. However, they differ in that they do not have range values in their definitions. Active context reflects a real snapshot of an entity's context at a specific time instant. For example, the following DL axiom describes a certain user context at 2018-04-06T14:23:00, which represents 2:23 pm on April 6, 2018:

$$OnDutyDoctor\{Bob\} \equiv Context \sqcap (User\{Bob\} \sqcap \exists hasID.IDentity\{Doctor563\} \sqcap \exists hasRole.Role\{Doctor\} \sqcap \exists hasGroup.Group\{InShiftDoctors\}) \sqcap (Environment\{WorkingEnvrnt\} \sqcap \exists hasLocation.Location\{HospitalA\}) \sqcap (TElement\{WorkingTime\} \sqcap existshasTime.Time\_Instance\{xsd : dateTime[2018-04-06T14:23:00]\}) \sqcap \exists hasID.\{0\} \quad (3)$$

This concept states that *Bob* is *OnDutyDoctor* at time 2:23 pm on April 6, 2018, if he is a user, has a role of *Doctor*, belongs to a group that is called *InShiftDoctor*, within a *WorkingEnvrnt*, at location *Hospital* and during the *WorkingTime*.

The context ontology can be extended or shrinked by adding or removing subcontexts or by adding or removing contextual attributes to the subcontexts.

3 SEMANTIC-BASED CONTEXT DELEGATION

The purpose of delegation is to grant/transfer access privileges from one entity, the delegator, to another entity, the delegatee. We require that the delegator must have the access privilege that is associated with context to be delegated. Delegating a subset of contextual attributes may result in a number of problems. These problems:

- Colluding (Servos and Osborn, 2016; Servos and Osborn, 2017), i.e., two entities may satisfy a policy that they could not if they acted individually. Our approach solves this problem by not allowing the delegated context's attributes to be merged with the delegatee context's attributes that are of the same type. For example, consider Figure. 1 below, Joe and Alice can potentially collude to satisfy a policy that they could not if they acted individually. The policy {role(Lab-Analyst),location(Hospital)} can not be satisfied neither by Alice nor by Joe, if acting alone, but they can deceive the system's policy through attribute-delegation.

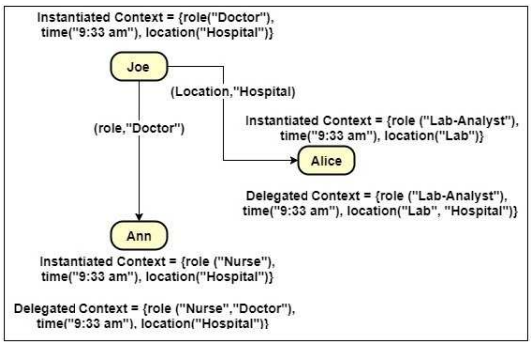

Figure 1: Attribute delegation problems example.

  When this context is delegated by our approach, the delegatee's context, c_2, is checked if it already has time and role attributes, and if these satisfy the delegation constraints, the corresponding contextual attributes of the delegator will not be merged with them. Instead, a new context, c_3, is created for the delegatee, and the chosen contextual attributes are assigned to it. This way there will be no colluding among the contextual attributes because c_2 is not the same as c_3.

- Inconsistent policy, i.e., the delegated privileges are conflicting the user's original privileges. For example, Joe's delegation of the *role* "Doctor" to Ann will result in Ann's delegated context that contains two values for the *role* attribute, "Doctor" and "Nurse". If there was a policy rule such that "*role* <> "*Nurse*"", then two different results would be possible depending on the value of *role* attribute. Our approach avoids inconsistent policies by evaluating delegator's context together with the delegatee's context.

At the time of delegation, the delegator must have the context c that is to be delegated to the delegatee. After the delegation is successfully completed, delegatee can use the delegated context and the privilege(s) associated with it to access to a resource r Our approach imposes constrains on context delegation. The constraints may be specified by the delegator or the system security officer. These constraints further restrict the delegation. Intuitively, if the delegatee's context satisfies the constraints, then the delegation is permitted. Otherwise, the delegation will be aborted. Our model architecture is shown in Figure 2.

Definition 8. (Delegation Request (DR).) Delegation request is given as a 6-tuple $\langle s_1, s_2, r, ac, DCs, Par \rangle$,

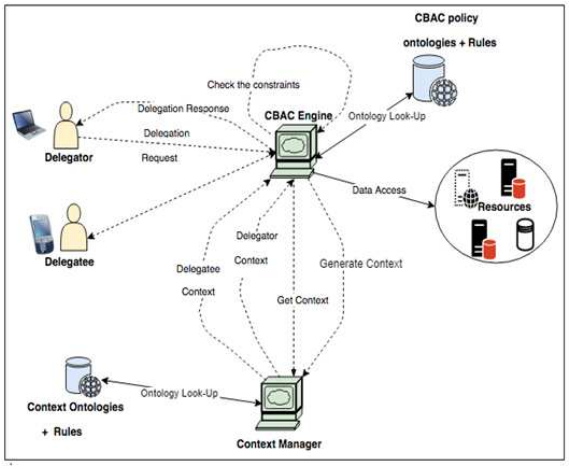

Figure 2: The proposed system architecture.

where s_1, $s_2 \in Subject$ and they represent the delegator and delegatee, respectively. $r \in Resource$, the resource to make the delegation over, $ac \in Action$, the action and must be equal to "$delegate$", $DCs \subseteq Constraint$ represents the set of constraints imposed by the delegator on delegatee's context, and Par is a finite set of delegation parameters, other than the delegation constraints, which are specified by the delegator. Delegation parameters, Par, are given by:

$$Par = (n_1, v_1), ..., (n_m, v_m) \quad (4)$$

Where n_i represents the parameter $name_i$ and v_i is the value of this parameter. Two important parameters in Par are very crucial for our approach. $DelType$ and $DelStatus$, $DelType$ is used to specify whether the delegation is of type $GRANT$ or $TRANSFER$. If $DelType = TRANSFER$, then the delegator must specify which kind of $TRANSFER$ to work with. One of the two values for the parameter $DelStatus$ is used to do so, which is either $Permanent$ or $Temporary$. The meaning of these values is clear, $Permanent$ is used for permanent delegation and $Temporary$ is for temporary delegation.

The DCs are represented as a set of pairs:

$$DCs = (CA_1, Cons_1), ...(CA_n, Cons_n) \quad (5)$$

Where CA_i represents an attribute i and $Cons_i$ is the delegation constraints set i (if any) that is imposed over CA by the delegator and must be satisfied by the delegatee's contextual attributes.

3.1 Delegation Policies

Every delegation operation is subject to predefined delegation policies. Delegation policies are rules that restrict the delegation. We represent our delegation policies in a predicate form as follows:

$can\_delegate(s_1, c_1, s_2, c_2, \backslash GRANT", DCs)$: subject s_1 can delegate context c_1 to subject s_2 if s_2's context satisfies delegation constraints DCs.

$can\_delegate(s_1, c_1, s_2, c_2, \backslash TRANSFER", DCs)$: subject s_1 can delegate context c_1 to subject s_2 if s_2's context (the reference context) satisfies delegation constraints DCs.

$can\_revoke(s_1, s_2, c_1, casCaded)$: delegator s_1 can revoke the delegated context c_1 from s_2 if s_1 is authorized to do so, i.e., it was the delegator of c_1 and the delegation is of type $\backslash GRANT"$. In case of $\backslash TRANSFER"$ delegation, the system administrator is responsible for revoking delegated context c_1. Note that, the issue of cascading revoke has been studied extensively and we do not address this issue in this paper.

3.2 Delegation Ontology

Delegation ontology, DEL, shown in Figure 3, states that the $Delegator$ is either a Source of Authority (SoA), and in this case, she does not lose her access rights when she transfers the Context associated with her to the $Delegatee$. Or she is not the SoA, and in this case, she is going to lose her access rights to a resource when she transfers her access rights to a $Delegatee$. Only the SoA have the right to set the value $isSoA$. This will keep delegation chain under the control of SoA and guarantee that the delegation authorization will not go infinite. $Delegation$ has two types $DelType$ to deal with the two cases of delegation. The first type is when the $Delegator$ transfers ("Transfer") her access rights to the $Delegatee$ and the second type is when the $Delegator$ grants ("Grant") her access rights to the $Delegatee$. A $Delegation$ can be delegated by a $Delegator$ who grants/transfers a $Context$ in case the $Delegator$ is also the SoA) to a $Delegatee$ if that $Delegatee$'s context satisfies delegation constraints. The delegated $Context$ qualifies the $Delegatee$ to accomplish a specific type of $Permission$ that is associated with the $Context$ on a certain $Resource$. If the $Delegation$ $isDelegatable$ and the $MaxDepth$ of the $Delegation$ chain does not reach its limit yet, then the $Delegatee$ can be a $Delegator$. When $Delegation$ is endowed to a $Delegatee$, the constraints $Constraint$ are applied to that granted/transferred $Delegation$ according to what the $Delegator$ specifies. Delegation ontology also specifies that a $Delegation$ can be authenticated using some $Credential$. This meant to be used by the CBAC engine to authenticate delegators before they can reach/change their delegation settings.

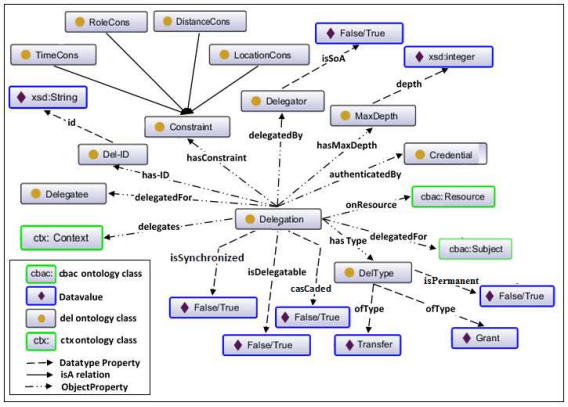

Figure 3: Delegation ontology.

3.3 Delegation Operations

We assume that each delegation operation delegates only one context at a time. If the delegator has multiple contexts (one is the instantiated context and the others may be gained by previous delegations) and (s)he wishes to delegate more than one context to the same delegatee, (s)he can do that in multiple delegation operations. The delegation operation takes the form $delegate(s_1, c_1, s_2, c_2, \backslash GRANT", Par)$.

Delegator s_1 delegates context c_1 to delegatee s_2. After checking delegation constraints satisfaction as we have illustrated in the previous subsection, the delegation algorithm (see Algorithm 3) creates a delegation instance with an identifier del_{id}. The delegation instance gets part of its values from the delegation request, namely from *Par* and *DCs*. We define the following parameters, *MaxDepth* is the depth of the delegation. It specifies the number of times the context can be delegated. This value is set by the first delegator ($isSoA = true$, see Figure 3). The *isDelegatable* is a Boolean value that determines whether the context is delegatable. If $isDelegatable = false$, then the algorithm automatically sets *MaxDepth* to 0.

3.4 Delegation Constraints

We represent delegation constraints, denoted as *Cons*, using Semantic Web Rule Language safe rules (SWRL-safe). SWRL combines OWL ontologies with Horn Logic rules, extending the set of OWL axioms to include Horn-like rules. SWRL rules have the syntax $Antecedent -> Consequent$, where each *Antecedent* and *Consequent* consists of atoms. These atoms can be of the form $C(x)$, $P(x,y)$, $sameAs(x,y)$ or $differentFrom(x,y)$, where C is an OWL class, P is an OWL property, and x, y are either variables, OWL individuals or OWL data values. The *Consequent* atom will be true if all atoms in the *Antecedent* are true.

For example, suppose that *Bob* has *OnDutyDoctor* as a reference context as has been shown in equation 2. Now suppose that *Bob* wants to set delegation constraint on the *time* contextual attribute before delegating his context (his reference context) to another user, *Ann*. *Ann* is a consultant doctor and she has the following reference context:

$$OnDutyConsult \equiv Context \sqcap (User \sqcap \exists hasID.IDentity \sqcap \exists hasRole.Role\{Consult\} \sqcap \exists hasGroup.Group \{InShiftConsult\}) \sqcap (Environment\{WorkingEnvrnt\} \sqcap \exists hasLocation.Location\{HospitalB\}) \sqcap (TElement \{WorkingTime\} \sqcap \exists hasTime\{xsd : dateTime [\geq 2018-04-06T09:00:00, \leq 2018-04-06 T17:00:00]\}) \sqcap \exists hasID.\{0\} \quad (6)$$

The delegation constraint is $(01{:}00pm \geq time \geq 10{:}00am)$, that is, it can only be delegated between 10:00 am and 01:00 pm. At the time of delegation, *Ann* has an active context as shown below:

$$OnDutyConsult\{Ann\} \equiv Context \sqcap (User\{Ann\} \sqcap \exists hasID.IDentity\{Doctor643\} \sqcap \exists hasRole.Role \{Doctor\} \sqcap \exists hasGroup.Group\{InShiftConsult\}) \sqcap (Environment\{WorkingEnvrnt\} \sqcap \exists hasLocation .Location\{Hospital\}) \sqcap (TElement\{WorkingTime\} \sqcap existshasTime.Time\_Instance\{xsd : dateTime [2018-04-06T12:30:00]\}) \sqcap \exists hasID.\{0\} \quad (7)$$

The policy engine checks, then, if the delegation constraints are satisfied or not. The policy engine uses the following SWRL rule to check the time constraint:

$$TimeCons(?t_3) \wedge notBefore(?t_3, ?cons1) \wedge swrlb : greaterThanOrEqual(?cons1, 10:00) \wedge notAfter (?t_3, cons2) \wedge swrlb : lessThanOrEqual(?cons2, 01:00) -> satisfied(?t_3) \quad (8)$$

where t_3=*Time_Instance* and is extracted from *Ann*'s active context and is equal to 12:30:11 *pm* (on April 6, 2018), and the constraints *cons*1=10:00 *am* and *cons*2=01:00 *pm* from the delegation constraints set by *Bob*.

3.5 Enforcement of Delegated Authorization

Non-monotonic context delegation (and delegation in general) needs to revise or contract the underlying policy rules to make them adaptable to delegation operation. To this end, non-monotonic delegation

should be able to block, permanently or temporarily, some access rules and prevent them from being activated. Doing so, however, will enforce the policy engine to use negation to infer failure of some rules within the policy rule set. DL's ontologies and SWRL rules do not permit this kind of reasoning because they based on the principle of Open World Assumption (OWA). OWA principle does not allow us to conclude that a SWRL atom is false because OWA assumes it may become true in the future, and hence, it assumes that atom's value is "Unknown". On the contrary, in Logic Programming (LP), which is based on Closed World Assumption (CWA), every atom or proposition that is not proved to be true is considered as a false atom.

The naive approach for doing so can simply be achieved by removing the context c from the CBAC knowledge base. However, this method ignores the fact that one context may have access privileges to multiple resources. Figure 4 illustrates this situation, when subject s with context c can access resources $r_1 ... r_5$. When trying to prevent subject s with context c from accessing resources r_3 and r_4 by deleting context c, then subject s will also be prevented to access resources r_1, r_2, r_5.

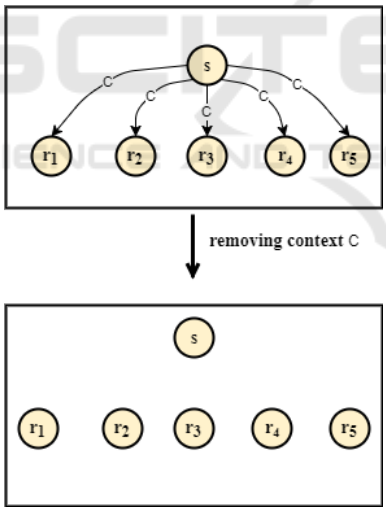

Figure 4: Naive context deletion.

To remedy this situation, we propose to selectively block subject with specific context from accessing certain resources via labeling the access path from that context to the blocked resources using OWL object properties. The predicate $update - cbac()$ is used to block a subject s with context c and deprive it from accessing resource r. This is done by inserting fresh OWL Object property instances, *isTBlocked* or *isPBlocked*, between context c and resource r. *isTBlocked* is used for temporarily blockade, while *isPBlocked* is used for permanent blockade Figure 5. Both *isTBlocked* or *isPBlocked* are defined in CBAC ontology.

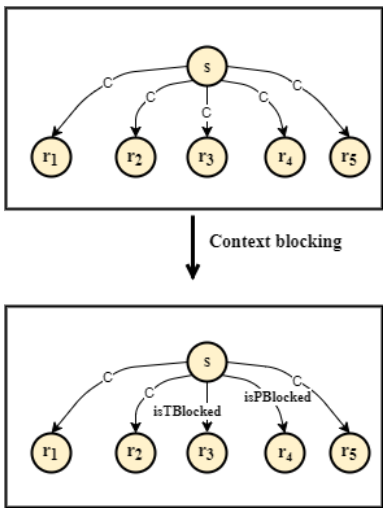

Figure 5: Selective context blockade.

The enforcement of access control authorization with delegation is achieved via selective blocking of the context (and hence their associated subjects) from accessing certain resources. Algorithm 2 is used for this purpose.

The context delegation (both monotonic and non-monotonic) is implemented via answering the delegation request using algorithm 3.

Algorithm 3 takes *CBAC*, *Del*, *Ctx*, *RS* and *RQ*, for Context-Based Access Control, delegation and context ontologies, respectively. *RS* is the access control rule-set and *RQ* is an Access Request. The algorithm returns the updated ontologies *CBAC*, *Del*, *Ctx* and the updated *RS*. The function $parse(RQ)$ is used for parsing the request and to check its syntax correctness. Function $dismantle(RT)$ is used to further partition the request into its elements, namely the delegator s_1, the delegatee s_2, the resource r, delegation constraints *DCs* and delegation parameters *Par*. The function $isAuthorized(s_1, sc_1, r, rc, CBAC, RS)$ is used to check whether s_1 has the privilege to access resource r. It uses Algorithm 2 to do that. Function $extractCAs(X)$ is used to extract the contextual attributes *CAs* from a given context axiom X. The function $type = extract\_type(Par)$ extracts, from a set of parameters found in *Par*, the type of delegation. The $checkSatisfiability(DCs, CAs)$ function checks the satisfiability of contextual attributes of s_2 against delegation constraints *DCs* set by s_1. The approach proceeds as follows:

1) The delegator prepares a delegation request and sends it to the policy engine.

2) The policy engine parses the request and starts the delegation process.

Algorithm 2: Delegation-Supported Access Authorization.

Input: *CBAC*, CBAC ontology, *Del* is a delegation ontology, *RQ* is an Access Request, *RS* is access control rule-set.
Output: Access decision, either "Deny" or "Permit".

1: $RT \leftarrow parse(RQ)$ ▷ $RT = \langle s, r, ac \rangle$
2: **if** $RT = access$ **then**
3: $sc \leftarrow getContext(s)$;
4: **if** $sc.Id \neq Null$ **then**
5: $d\_inst \leftarrow consult(Del, sc.Id)$
6: **if** Del has an RDF triple:($del : d\_ins\ del : OnResource\ cbac : r$) **then**
7: $rc \leftarrow getContext(r)$;
8: $p \leftarrow evaluate(s, sc, r, rc, CBAC, RS)$;
9: **end if**
10: **if** $p =$ "*Permit*" AND $noConflict(p, s, sc, r, rc, CBAC, RS)$ **then**
11: $Permit - Access$;
12: $exit()$;
13: **else**
14: $Deny - Access$;
15: $exit()$;
16: **end if**
17: **end if**
18: **if** $sc.Id = Null$ **then**
19: **if** $Not(blocked(sc, r, CBAC))$ **then**
20: $rc \leftarrow getContext(r)$;
21: $p \leftarrow evaluate(s, sc, r, rc, CBAC, RS)$;
22: **end if**
23: **if** $p =$ "*Permit*" AND $noConflict(p, s, sc, r, rc, CBAC, RS)$ **then**
24: $Permit - Access$;
25: $exit()$;
26: **else**
27: $Deny - Access$;
28: $exit()$;
29: **end if**
30: **end if**
31: **else**
32: $return$("*Not access request*");
33: **end if**

3) The policy engine extracts the delegation constraints, asks the context manager for the delegator's context, and checks if the delegator has the delegation right.
4) If the delegator is authorized, the policy engine asks the context manager for the delegatee's (s_2) context and checks for satisfiability of the delegation.
5) If the delegation is satisfiable, the policy engine creates a delegation instance using the delegation ontology and the parameters specified in the delegation request.

Algorithm 3: Context Delegation Request Answering.

Input: *CBAC*, *Del*, *Ctx* are CBAC, delegation, and context Ontologies. *RS* is the access control rule-set. *RQ* is an Access Request.
Output: $UCBAC, UCtx, UDel$ /*Updated CBAC, context and delegation ontologies.*/

1: $RT \leftarrow parse(RQ)$
2: **if** $RT = RQ$ **then**
3: $evaluateRQ(RT)$;
4: $exit()$;
5: **end if**
6: $\langle s_1, s_2, r, ac, DCs, Par \rangle \leftarrow dismantle(RT)$;
7: $sc_1 \leftarrow getContext(s_1)$;
8: $rc \leftarrow getContext(r)$;
9: **if** $isAuthorized(s_1, sc_1, r, rc, CBAC, RS) = false$ **then**
10: $output$("s_1 *is not authorized to access* r");
11: $exit()$;
12: **end if**
13: $sc_2 \leftarrow getContext(s_2)$;
14: $CAs \leftarrow extractCAs(sc_2)$;
15: $T \leftarrow checkSatisfiability(DCs, CAs)$;
16: **if** $T = false$ **then**
17: $output$("*The context is not delegatable*");
18: $exit()$;
19: **end if**
20: $type \leftarrow extract\_type(Par)$
21: **if** $type =$ "*GRANT*" **then**
22: $UDel \leftarrow createDelegationinstance(Del, \langle s_1, s_2, r, ac, DCs, Par \rangle, del_{id})$
23: $UCtx \leftarrow createContext(Cx_2, Ctx, del_{id})$
24: $return(UDel, UCtx)$
25: $exit()$
26: **else**
27: $UDel \leftarrow createDelegationinstance(Del, \langle s_1, s_2, r, ac, DCs, Par \rangle, del_{id})$
28: $UCtx \leftarrow createContext(Cx_2, Ctx, del_{id})$
29: $UCBAC \leftarrow update\_cbac(CBAC, UDel, del_{id}, sc_1, r)$
30: $return(UCBAC, UDel, UCtx)$
31: $exit()$
32: **end if**

6) The policy engine sends a request to the context manager, accompanied with a delegation identifier, del_{id}, to construct a generated context for s_2. This context is a copy of the delegator reference context but it is associated with the delegatee.
7) The context manager creates the generated context for s_2 and associates it with the identifier del_{id} provided by the policy engine with the request.
8) Now the delegatee has two contexts, the instantiated context and the generated context and (s)he can choose which one to work with.

4 PROPERTIES AND PROOFS

In this section, we provide the proof that context delegation approach, as well as enforcing the context delegation in access authorization are sound, consistent and preserve the Least-Privilege principle.

The function $evaluate()$ is given by:

function $evaluate(s,sc,r,rc,CBAC,RS)$
$1: RulesNo \leftarrow getrulesnumber(RS);$
$2: infmodel \leftarrow reason(CBAC);$
$3:$ **for** $i = 1$ *to* $RulesNo$ **do**
$4:$ **if** $RS[i]$ *has the form* :
$(?rq \quad rdf:type \quad cbac:Request)$
$(?rq \quad ctx:hasAction \quad cbac:ac)$
$(?rq \quad cbac:hasSubject \quad ?s)$
$(?rq \quad cbac:hasResource \quad ?r)$
$(?s \quad rdf:type \quad ctx:sc)$
$(?r \quad rdf:type \quad ctx:rc)$
$(?rq \quad cbac:hasDecision \quad ?d)$
$->(?d \quad cbac:hasEffect \quad cbac:p)]$ **then**
$5:$ **if** $p =$ "*Permit*" **then**
$6:$ $return(p);$
$7:$ $exit();$
$8:$ **else**
$9:$ $p =$ "*Deny*"
$10:$ $return(p);$
$11:$ $exit();$
$12:$ **end** $\{evaluate\}$

The function $evaluate(s,sc,r,rc,CBAC,RS)$ takes the access control ontology CBAC, Figure 6 as input and achieves the DL-based reasoning to get the $infmodel$. The access control policy rules, represented as Jena forward inference rules, will be applied on $infmodel$ to derive an access decision. The function $getrulesnumber(RS)$ returns the number of rules in the access control policy rule-set RS. The Boolean function $noConflict(p,s,sc,r,rc,CBAC,RS)$ works just like $evaluate(s,sc,r,rc,CBAC,RS)$ function but it returns a Boolean value if two conflicting rules ("*Permit*" and "*Deny*") are fired at the same time.

Lemma 1. *Given RS and CBAC correctly represented, the function evaluate(s,sc,r,rc,CBAC,RS) will always return a correct authorization decision.*

Proof. Assume the function $evaluate(s,sc,r,rc,CBAC,RS)$, when it is called, returns false authorization decision. This decision could be derived in three cases: (i) The first case

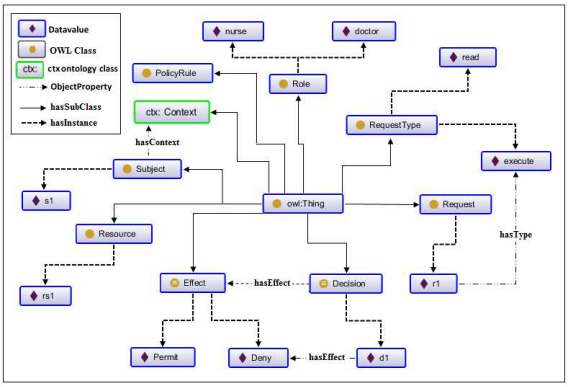

Figure 6: CBAC ontology.

is an example of traditional security violations that could take place in any system. A resource owner (or security officer) misconfigures her security settings, such as incorrectly encoding Jena rules for access control policy. As a result, the system will adhere to this misconfigured setting.(ii) There is no misconfigured security settings. In this case, one of the following three conditions must be held: (a) The DL-based reasoning process (step 2) returns inconsistent model. This assumption is invalid because we use DL-based reasoner Pellet which is sound and complete with respect to OWL-DL. Hence, The system knowledge base is always have correct and consistent information. (b) $RS \nvDash p$, access control rule-set does not model the decision p (p is neither "*Permit*" or "*Deny*"). If $p \in RS$, then $RS \models p$. If $p \notin RS$, then p must be generated by a Jena rule of the form:

$$a_1 \wedge ... \wedge a_n \rightarrow p \tag{9}$$

where $a_1...a_n$ are RDF (Resource Description Framework) triples and p is an RDF triple (or set of triples), and for each $a_i \in a_1...a_n$, $RS \models a_i$, either by inference or by assertion. If $RS \models a_i$ for all $a_1...a_n$, then $a_1...a_n \models p$, for p is either "*Permit*" or "*Deny*" (more precisely, "*Deny*" is the default value returned by the function $evaluate(s,sc,r,rc,CBAC,RS)$ when there is no rule of the form $a_1...a_n \models p$). Hence, $RS \models p$ which contradicts our assumption that $RS \nvDash p$. (c) RS and/or $CBAC$ are incorrectly represented. This contradicts (a) above and our assumption that $CBAC$ and RS are correctly represented. Hence, the function $evaluate(s,sc,r,rc,CBAC,RS)$ will eventually derive a correct access decision p. □

Corollary 1. *Given a CBAC and RS correctly represented, the function noConflict(p,s,sc,r,rc,CBAC,RS) will derive the correct Boolean value and terminates.*

Proof. Function $noConflict(p,s,sc,r,rc,CBAC,RS)$ works just like the function

evaluate(*s*,*sc*,*r*,*rc*,*CBAC*,*RS*) does but after finding a decision *p*="*Permit*", it iteratively searches for *p*="*Deny*". If it happens to find two conflicting decisions, it will return a *true* value. Otherwise, $false$ value is returned. In Lemma 1, we have proved that the function *evaluate*(*s*,*sc*,*r*,*rc*,*CBAC*,*RS*) is sound (returns only correct answers). Hence, *noConflict*(*p*,*s*,*sc*,*r*,*rc*,*CBAC*,*RS*) follows from Lemma 1. □

Theorem 1. *(Algorithm 2 Soundness.) Given a domain knowledge base that is correctly represented using ontologies CBAC, CTX, DEL, and a set of Jena rules RS.*

Proof. The proof is naturally follows from Lemma 1 and Corollary 1. □

Lemma 2. *Function checkSatisfiability*(*DCs*;*CAs*) *is decidable, sound and consistent.*

Proof. We use only SWRL-Safe rules in our delegation and CBAC ontologies. A SWRL rule is DL-safe if every variable in the rule head occurs in a Datalog atom in the body. SWRL-Safe rules are known to be decidable, sound, consistent and complete (refer to (Motik et al., 2005) for the complete proof). □

function $update\_cbac(CBAC, UDel, del_{id}, sc_1, r)$
1 : $D \leftarrow$ *get_Instance*($UDel, del_{id}$);
2 : **if** *D.isPermanent* = "*False*" **then**
3 : $UCBAC \leftarrow CBAC \cup opAssertion(sc_1, r,$ "*isTBlock*")
4 : **else**
5 : $UCBAC \leftarrow CBAC \cup opAssertion(sc_1, r,$ "*isPBlock*");
10 : *return*(*UCBAC*);
11 : *exit*();
12 : **end** {*update_cbac*}

Lemma 3. *The function update_cbac(CBAC,UDel,*del_{id}*,*sc_1*,r) is decidable, sound and consistent.*

Proof. All ontology-based operations in this function, namely *get_Instance*() and *opAssertion*() (for object Property assertion) are decidable, sound and complete. □

Theorem 2. *(Algorithm 2 Soundness.) Given a domain knowledge base that is correctly represented using ontologies CBAC, CTX, DEL, and a set of Jena rules RS.*

Proof. The proof is naturally follows from Lemma 1 and Corollary 1. □

Theorem 3. *(Algorithm 3 Soundness.) Given a domain knowledge base that is correctly represented using ontologies CBAC, CTX, DEL, and a set of Jena rules RS.*

Proof. The proof is naturally follows from proof of Theorem 2, Lemma 2 and Lemma 3. □

We have shown that Algorithms 2 and 3 are sound. Remains to prove that the model in which context delegation is used is consistent, no contradicting decisions are derived for one access request.

Theorem 4. *(Model Consistency.) Given a model represented by a domain knowledge base that is correctly represented using ontologies CBAC, CTX, DEL, and a set of Jena rules RS. Delegation operation always results in consistent access authorization decisions ("Permit" and "Deny" are never both derived as answers for the same access request).*

Proof. Let $\mathcal{M}$ be the model of our approach and is given by $\mathcal{M}$=*CBAC*$\cup$ *CTX* $\cup$ *DEL*. Our model employs two types of reasoning techniques, monotonic reasoning which is achieved by DL-based Pellet reasoner on the model ontologies before any access authorization or delegation operation is made and non-monotonic reasoning, which is applied using Jena forward chaining. Monotonic reasoning has been proven to be decidable, sound, complete and consistent (Baader et al., 2003; Hitzler et al., 2010). That means for every sound and consistent model $\mathcal{M}$, there is an inference model $\mathcal{MF} = Inf(\mathcal{M})$, where Inf is the DL-based reasoning operation, $\mathcal{MF}$ is also sound and consistent model. Hence, while the model is correctly represented (this is given), so all DL-based inferences produce sound and consistent output models. However, this fact does not solve the problem of possible inconsistencies that may be resulted when deriving an answer for a request (it is possible to reach to a conclusion that the decision is both "Permit" and "Deny" while both of them belong to the same class in the CBAC ontology and not for disjoint classes). Our model solves this problem by using Jena rules (function *noConflict*(*p*,*s*,*sc*,*r*,*rc*,*CBAC*,*RS*)) which returns a true value when conflicting decision is reached. Hence, the model $\mathcal{M}$ is consistent and any model $\mathcal{MF}$ that is resulted by either monotonic reasoning or non-monotonic reasoning is also consistent (in respective to conflicting access decisions). □

Theorem 5. *(Least-privilege Principle.) Given a domain knowledge base that is correctly represented using ontologies CBAC, CTX, DEL, and a set of Jena rules RS. Algorithms 2 and 3 (Delegation-Supported Context Delegation) always preserve the*

Least-Privilege Principle when executes a delegation request.

Proof. In this proof, we combine two proofs of Algorithms 2 and 3 preserve Least-Privilege Principle. Assume that Algorithms 2 and 3 do not preserve Least-Privilege Principle. This would happen in two situations:

(i) Delegatee acquires more privileges that (s)he supposed to have according to the policy in use. For example, delegatee may acquire, as a result of context delegation operation, access to more than the specified resource(s) or/and can use more than one access type (write/read/execute) on a specific resource. Our approach restricts the delegatee access to a specific resource (and the type of this access) according to the delegation ontology. In (step 4) of Algorithm 2, we check whether the context is a delegated or an instantiated context ($sc.Id = X$ or $sc.Id = Null$, X is any non-zero integer value). If it is a delegated context, then the algorithm checks if the delegation instance *d_ins* is applicable on a resource *r* by checking existence of RDF triple (*del*[1] : *d_ins del* : *OnResource cbac* : *r*) in delegation ontology (step 6). Only in this case the delegatee with context *sc* is permitted to access resource *r* in ontology CBAC. Otherwise, access is denied. But this violates our assumption that Algorithms 2 and 3 do not preserve Least-Privilege Principle. Hence, Algorithms 2 and 3 preserve Least-Privilege Principle for the delegatee accessing a resource in the knowledge based.

(ii) Delegator has transferred her context *sc* to the delegatee but she is still able to use it for accessing resource *r*. Algorithm 3 (step 29) updates the CBAC ontology by inserting an RDF triple (*ctx* : *c cbac* : *isPBlock cbac* : *r*), for permanent delegation, and (*ctx* : *c cbac* : *isTBlock cbac* : *r*), for temporary delegation). In this way, when delegator tries to access resource *r* using a context *sc* that she has already permanently or temporarily transferred it to the delegatee, then algorithm will discover that this context is blocked from access resource *r* (step 19 of Algorithm 2). Again, this violates our assumption that Algorithms 2 and 3 do not preserve Least-Privilege Principle. Hence, Algorithms 2 and 3 preserve Least-Privilege Principle for the delegator accessing a resource in the knowledge based.

□

[1] del, cbac and ctx are prefixes for delegation, CBAC and context ontologies

5 CONCLUSION AND FUTURE WORK

In this paper we have proposed two ontology-based context delegation approaches. Monotonic context delegation, which adopts GRANT version of delegation, and non-monotonic for TRANSFER version of delegation. The approaches provide dynamic and adaptive mechanism for monotonic and non-monotonic privilege delegation. The proposed monotonic delegation does not cause any change to the underlying access control policy. For non-monotonic delegation, the approach incurs only selective changes to the underlying access control policy. The approaches presented in this paper are modeled using semantic-based technologies and can be used by existing CBAC systems which do not provide delegation capability. Furthermore, we have provided the formal proofs that context delegation and its delegation-supported authorization enforcement are sound, consistent and preserve the least privilege principle. We have implemented the model using real networks. The ontologies and some related preliminary coding can be found on https://github.com/Mouiad1975/Context-Delegation

For future direction, we are working on extending our model by using RESTful web services with Java (Jersey/JAX-RS). Another extension we are working on is the semantic-based obligation for situations that need to make sure some obligations are respected by subjects already got the access authorization to access certain resources.

REFERENCES

AL-Wahah, M. and Farkas, C. (2018). Context delegation for context-based access control. In *2nd International Workshop on A.I. in Security*, pages 70–79. ECML.

Baader, F., Calvanese, D., McGuinness, D., Patel-Schneider, P., and Nardi, D. (2003). *The description logic handbook: Theory, implementation and applications*. Cambridge university press.

Bellavista, P. and Montanari, R. (2017). Context awareness for adaptive access control management in iot environments. *Security and Privacy in Cyber-Physical Systems: Foundations, Principles and Applications*, pages 157–178.

Bhatti, R., Bertino, E., and Ghafoor, A. (2005). A trust-based context-aware access control model for web-services. *Distributed and Parallel Databases*, 18(1):83–105.

Corradi, A., Montanari, R., and Tibaldi, D. (2004). Context-based access control management in ubiquitous environments. In *Proceedings of the 3rd IEEE International Symposium on Network Computing and Applications*, pages 253–260. IEEE.

Crampton, J. and Khambhammettu, H. (2008). Delegation in role-based access control. *International Journal of Information Security*, 7(2):123–136.

Dey, A., Abowd, D., and Salber, D. (2001). A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Hum.-Comput. Interact.*, 16(2):97–166.

Gusmeroli, S., Piccione, S., and Rotondi, D. (2013). A capability-based security approach to manage access control in the internet of things. *Mathematical and Computer Modelling*, 58(5-6):1189–1205.

Hitzler, P., Krötzsch, M., and Rudolph, S. (2010). *Foundations of Semantic Web Technologies*. Chapman and Hall/CRC Press.

Kagal, L., Berners-Lee, T., Connolly, D., and Weitzner, D. (2006). Self-describing delegation networks for the web. In *Policies for Distributed Systems and Networks, 2006. Policy 2006. Seventh IEEE International Workshop on*, pages 10–pp. IEEE.

Kulkarni, D. and Tripathi, A. (2008). Context-aware role-based access control in pervasive computing systems. In *Proceedings of the 13th ACM symposium on Access control models and technologies*, pages 113–122. ACM.

Motik, B., Sattler, U., and Studer, R. (2005). Query answering for owl-dl with rules. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(1):41–60.

Pratt, I. (1994). *Closed World Assumptions*, pages 65–84. Springer.

Schneider, F. (2003). Least privilege and more [computer security]. *IEEE Security & Privacy*, 99(5):55–59.

Servos, D. and Osborn, S. (2016). Strategies for incorporating delegation into attribute-based access control (abac). In *International Symposium on Foundations and Practice of Security*, pages 320–328. Springer.

Servos, D. and Osborn, S. (2017). Current research and open problems in attribute-based access control. *ACM Computing Surveys (CSUR)*, 49(4):1–65.

Shen, H. and Cheng, Y. (2011). A semantic context-based model for mobile web services access control. *International Journal of Computer Network and Information Security*, 3(1):18.

Toninelli, A., Montanari, R., Kagal, L., and Lassila, O. (2006). A semantic context-aware access control framework for secure collaborations in pervasive computing environments. In *International semantic web conference*, pages 473–486. Springer.

Trnka, M. and Cerny, T. (2016). On security level usage in context-aware role-based access control. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, pages 1192–1195. ACM.

Zhang, L., Ahn, G., and Chu, B. (2001). A rule-based framework for role based delegation. In *Proceedings of the sixth ACM symposium on Access control models and technologies*, pages 153–162. ACM.