DOI 10.12694/scpe.v17i3.1180
Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 17, Number 3, pp. 189–205. http://www.scpe.org

ANALYSIS AND VERIFICATION OF XACML POLICIES IN A MEDICAL CLOUD ENVIRONMENT*

MERYEME AYACHE † MOHAMMED ERRADI † AHMED KHOUMSI ‡ AND BERND FREISLEBEN §

Abstract. The connectivity of devices, machines and people via Cloud infrastructure can support collaborations among doctors and specialists from different medical organisations. Such collaborations may lead to data sharing and joint tasks and activities. Hence, the collaborating organisations are responsible for managing and protecting data they share. Therefore, they should define a set of access control policies regulating the exchange of data they own. However, existing Cloud services do not offer tools to analyse these policies. In this paper, we propose a Cloud Policy Verification Service (*CPVS*) for the analysis and the verification of access control policies specified using XACML. The analysis process detects anomalies at two policy levels: a) intra-policy: detects discrepancies between rules within a single security policy (*conflicting rules* and *redundancies*), and b) inter-policies: detects anomalies between several security policies such as *inconsistency* and *similarity*. The verification process consists in verifying the *completeness* property which guarantees that each access request is either accepted or denied by the access control policy. In order to demonstrate the efficiency of our method, we also provide the time and space complexities. Finally, we present the implementation of our method and demonstrate how efficiently our approach can detect policy anomalies.

Key words: Formal Verification, Cloud Computing, XACML Policies, Automata, Completeness, Security Anomaly Detection.

AMS subject classifications. 68Q60, 68Q85, 68M14

1. Introduction.

The use of connected devices (mobiles, sensors, scanners etc.) permits the creation of Electronic Personal Records (EPR) to monitor patients' health states remotely. The EPR of a patient consists of medical histories, diagnoses, medications, immunization dates, etc [11]. A first advantage of an EPR is that it provides accurate, up-to-date, and complete information about patients. Another advantage is that it supports collaborations among different doctors in diverse medical organisations.

Cloud computing offers a suitable platform for such collaborations [1]. For instance, the storage service offered by the Cloud can be considered as a shared pool where medical organisations can store and share their data. A patient's EPR usually contains confidential data and hence each medical organisation needs to define a set of policies to regulate the access to the outsourced data. However, current Cloud solutions do not offer users the ability to define their own policies. To address this issue, we have developed, in previous work, a middleware (denoted *curlX* [6]) that permits the enforcement of users' security policies in *Openstack* [27] (an open source Cloud solution). *curlX* uses XACML [25] (eXtensible Access Control Markup Language) to specify access control policies [7]. Yet, XACML has many limitations in terms of policy anomaly detection [16]. For instance, XACML lacks a mechanism to detect conflicts and redundancies.

In this paper, we present a formal approach based on automata to detect anomalies in XACML policies such as: conflicting rules, redundancies, inconsistencies, and policy similarities; and to verify the completeness property that guarantees that each access request is either accepted or denied by the access control policy. The approach is implemented as a Cloud service denoted *CPVS* (Cloud Policy Verification Service) and integrated into *curlX*. The advantage of the proposed approach is that it detects several discrepancies in the XACML policies using the same formal model, in contrast to other existing approaches which make use of different models to detect distinct anomalies.

The rest of the paper is organised as follows: Section 2 presents related work. In Section 3, we present an overview of XACML policies and automata. Section 4 describes the architecture of *curlX* taking into account the new verification service. In Section 5, we present the procedure to transform XACML policies into automata. Sections 6 and 7 define the approaches to detect anomalies and verify completeness in XACML policies. We calculate the time and space complexities in Section 8. Section 9 discusses the implementation and evaluation of our proposed approach. Finally, Section 10 concludes the paper and outlines areas for future research.

* This work is supported by the BMBF (PMARS Program) and the DAAD (German-Arab Transformation Partnership)
†ENSIAS, Mohammed V University in Rabat, Morocco (meryemeayache@gmail.com; mohamed.erradi@gmail.com)
‡Department of Electrical & Computer Engineering, University of Sherbrooke, Canada (ahmed.khoumsi@usherbrooke.ca)
§Department of Mathematics & Computer Science, Philipps-Universität Marburg, Germany (freisleb@informatik.uni-marburg.de)

2. Related Work. In a collaborative healthcare process, doctors and specialists from different medical organisations share patient's data in order to make a better diagnosis. Due to the current Big data exponential data growth, solutions that store, process [10] and manage medical data are of a great interest. In this direction, cloud computing represents a cost-effective solution for such needs [1, 2]. For instance, Marzini et al. [23] make use of the cloud elasticity to manage basic activities in healthcare scenarios. On the other hand, the usage of cloud computing for medical environments raises several issues such as reliability and security.

Regarding the reliability issue, Gawanmeh et al. [14] present a state of the art review on the verification of reliability in healthcare systems using either simulation-based verification, formal methods such as automata and prism [28, 29], or semi-formal methods. In the work presented in this paper, we focus on the formal verification of the security aspect, especially access control.

Access control protects the system's resources against unauthorized access via a set of policies. Jansen [18] proposed XACML as a policy specification language for cloud applications. Yet, XACML policies may contain conflicting and redundant rules, since XACML policies are sometimes managed by more than one administrator [16]. Moreover, in collaborative applications, the XACML policies are aggregated from collaborative parties which may raise conflicts between rules in different policies.

Several works make use of verification techniques such as model checking in order to detect XACML policy anomalies. For instance, to detect conflicts between rules in a given policy, Martin et al. [22] encode the rules in Coq [8], a tool built specifically for formal theorem proving. A rule is a Coq record with two fields: the first field has the effect type, and the second field contains the srac type that combines the four elements of XACML: subject-resource-action-condition. In order to compare the elements of type srac independently, the authors split them into a defined normal form *DNF*. If two rules have overlap (srac types are identical) with different effects, the rules are then in conflict. Otherwise, if the effects are similar, then the rules are redundant. However, using Coq does not allow the automatic anomaly detection after the insertion of new rules, since their proposed approach does not interact directly with the policies' original format. In contrast, Mourad et al. [24] use the Unified Modelling Language (UML) to detect conflicting and redundant rules prior to their enforcement in the system. However, this technique does not allow completeness verification.

Regarding inter-policy conflict detection, Ramli [30] uses Answer Set Programming (ASP) in order to detect incompleteness, conflicting and unreachable XACML policies. As a limitation of this approach, it is difficult to model XACML expressions dealing with types of attributes that do not belong to AnsProlog [31], such as strings. Huonder [17] proposes another approach to detect and resolve conflicts in XACML policies based on mapping each target to n-dimensional space and overlapping the policies with different effects. The intersection of all dimensions defines an inter-policy conflict. Yet, this technique cannot verify the policy's completeness property.

Besides verification-based techniques, many research efforts consider representing XACML policies as decision trees to detect and resolve conflicts. In this direction, Hu et al. [16] make use of Binary Decision Diagram (BDD). In this work, each XACML attribute is encoded into an atomic boolean expression. The rules are then functions of these expressions. Fisler et al. [12] suggested an extended version of BDD called Multi-Terminal Binary Decision Diagram (MTBDD). Also, Gouglidis et al. [15] transform the XACML policies into Computation Tree Logic (CTL). These tree-based approaches have a main drawback, the state explosion in the decision trees.

A comparison of the proposed approach in this paper with the seven existing methods is presented in Table 2.1. We have adopted the metrics proposed by Li et al. [20]:

1. **Completeness**: it guarantees that any access request has a response by the access control policy: permit or deny.
2. **Policy anomalies** can be divided into two categories, namely intra-policy anomalies (conflicting rules and redundancy) and inter-policy anomalies (inconsistency and similarity):
   - *Conflicting rules*: two rules are in conflict in the same security policy.
   - *Redundancy*: the existence of two rules that have the same effect (permit or deny) such that one of the two rules can be removed without changing the result of the policy.
   - *Inconsistency*: the existence of two or more rules in different policies that are in conflict.
   - *Policy similarity*: two policies can be similar and represented differently.

TABLE 2.1
The Capabilities of the Proposed Access Control Verification Approaches

Approach	Technique	Completeness	Policy anomalies				Flexibility
			Conflicting rules	Redundancy	Inconsistency	Similarity	
[22]	Coq	No	Yes	No	No	No	No
[24]	UML	No	Yes	Yes	No	No	No
[30]	ASP	Yes	No	No	Yes	No	No
[17]	n-dimensions	No	No	No	Yes	No	No
[16]	BDD	No	Yes	Yes	Yes	No	No
[12]	MTBDD	No	No	No	No	Yes	No
[15]	CTL	No	Yes	No	No	No	Yes
Our Proposed method	Automata	Yes	Yes	Yes	Yes	Yes	Yes

3. **Flexibility**: It indicates whether a method can detect anomalies at run-time (i.e. detects if the new inserted rule raises anomalies with the existing rules prior to its enforcement).

Table 2.1 underlines our contributions compared to existing works. The proposed approach uses automata to represent the XACML policies. This formalism, allow us to detect several XACML anomalies (intra and inter policies conflicts) and to verify the completeness property using the same formal model. In addition, the approach has the ability to detect conflicting rules at run-time. In fact, the proposed approach models each security policy with an automaton. To verify if a new rule raises conflicts with the existing ones, the proposed approach consists in applying the synchronous product to the rule's automaton and the policy's automaton. The conflict detection process is then applied to the resulted automaton. Hence, there is no need to integrate the new rule into the policy to do the verification.

3. Preliminaries. The proposed approach consists in verifying XACML (eXtensible Access Control Markup Language) security policies using automata. Therefore, in this section, we define the two concepts: automata and XACML.

3.1. Automata. *Finite state automata* (or briefly automata) are used, for example, for pattern matching in text editors [3], for lexical analysis in compilers, for communication protocol specifications, for language recognition [9], and for firewall design analysis [19]. An automaton can be formally defined by $\mathcal{A} = (\Sigma, Q, q^0, Q^f, \delta)$ where Σ is a finite set of events (also called alphabet), Q is a finite set of states, q^0 is the initial state and $Q^f \subseteq Q$ is a finite set of final states. $\delta : Q \times \Sigma \to Q$ is the transition function, where $\delta(q, \sigma) = r$ means that the execution of the event σ (or the reading of the term σ) from state q leads to state r. $\delta(q, \sigma) = r$ can also be written as $q\sigma r$.

An automaton $\mathcal{A}$ consists of states linked by labelled transitions, and represented by a graph whose nodes and arcs are the states and the transitions of $\mathcal{A}$, respectively. There is one initial state (with a small incoming arrow) and one or more final states (double circled).

In this paper, we use the notation $S = \{\sigma_1, \sigma_2, \ldots, \sigma_p\}$ (it can be also denoted as $S = \{\sigma_1\} \cup \{\sigma_2\} \cup \ldots \{\sigma_p\}$) for a set of events. The notation qSr means that if q is the current state, then every event σ_k from the set S leads to the state r. The arc labelled S, linking q and r, is equivalent to many arcs labelled $\sigma_1, \ldots, \sigma_p$ linking q and r.

A finite event sequence (more briefly, sequence) is accepted by $\mathcal{A}$ if it starts in the initial state q^0 and terminates in one of the final states of $\mathcal{A}$. The language of $\mathcal{A}$, denoted $\mathcal{L}_\mathcal{A}$, is the set of sequences accepted by $\mathcal{A}$.

The rich theory of automata allows us to compose models of systems, behaviours, mechanisms due to the operations that can be performed over automata. For instance, the synchronous products of two automata $\mathcal{A}_1$ and $\mathcal{A}_2$ over the alphabet Σ is an automaton over the alphabet Σ whose language is $\mathcal{L}_{A_1} \cap \mathcal{L}_{A_2}$. This intersection permits us to track the behaviour of the global system (consisting of the two subsystems modelled by A_1 and A_2) in order to detect anomalies and conflicts.

3.2. XACML. XACML (eXtensible Access Control Markup Language) has been widely used as a policy specification language in both academia and industry. Its first version was released by Anderson et al. [4] in 2003 and used in the context of distributed systems [21]. Two years later, OASIS extended the old version and

proposed XACML 3.0 [25]. XACML assumes an architecture containing a PDP (Policy Decision Point), PEP (Policy Enforcement Point) and PAP (Policy Administration Point). The XACML request to access a specific resource is redirected to PEP. The PEP then extracts the attributes from the request and sends them to the PDP which searches in the policy repository for the appropriate policy that matches the request. The PDP then sends a response to the requester, which can be: *Permit*, *Deny*, *Not Applicable* or *Indeterminate*. The first two responses are obvious. *Not Applicable* is applied if no rule or policy matches the request. *Indeterminate* is applied if the system cannot interpret the request due to the lack of attributes or problems of connection. The PAP is responsible to associate the new added rules to the appropriate policy.

The main component of XACML *Policy* is composed of a *Target* that identifies the capabilities that should be exposed by the requester (the targeted resources for example), and some *Rules*. Each *Rule* contains facts (*Subjects*, *Resources*, *Actions* and *Environment*) for access control decisions and an *Effect* that can be either *Permit* or *Deny*.

Policies can be combined using *PolicySet* that specifies the combining algorithms in case if two security policies provoke permit/deny conflicts. XACML offers four combining algorithms:

- permit-override: If at least one policy is evaluated as "*permit*", the integrated output will also be "*permit*".
- deny-override: If at least one policy is evaluated as "*deny*", the integrated output will also be "*deny*".
- first applicable: The result of the combining algorithm is the result of the first policy that evaluates to Permit or Deny.
- only-one-applicable: The result is the one of the only applicable policy. If we have more than one policy, then the result is Not Applicable.

A XACML policy contains hundreds and thousands of rules, which make it difficult to detect policy conflicts directly from the XML file. Yet, identifying conflicts in XACML policies is a primordial task for their designers. In fact, the choice of the combining algorithms relies essentially on the information from conflict diagnosis. The XACML policies may contain two kinds of conflicts: intra-policy (conflict between rules of the same policy) and inter-policy (conflict between rules of several policies defined under the *PolicySet*).

4. CPVS: Cloud Policy Verification Service. Cloud computing offers several services, such as computing, authentication, and storage. These services could support collaborations among different organisations. Yet, such collaborations need to be regulated by a set of access control policies to protect the shared resources. However, the current Cloud architectures do not provide to the users the capability to define their own access control policies (high level control policies). For instance, *Openstack* is a widely used Cloud open source software that offers a storage service via *Swift* [5]. Although the *Swift* component supports fine-grained access control to objects (resources), it remains specific and at a low level of control. To address this issue, we have developed a middleware denoted *curlX* [6] that permits the collaborators to express their own security policies using XACML and enforce them using the cloud primitives such us curl (client url request) library. The user sends a curl request to the middleware asking for accessing a resource stored in *Swift* (see Fig. 4.1). The curl request is of the form:

$\mathbf{curl} - X <PUT|POST> -i - HX - Auth - Token :<TOKEN> -HX - Container - Read :<ACL> <STORAGE - URL>/<container>.$

The authentication token and the storage url are provided by Keystone. Keystone is an authentication service in *Openstack*. It provides each authenticated user by tokens that expire after a specific time delay. The curl request is redirected to the translator component to be transformed into XACML request and redirected to the PEP. After retrieving the main attributes (subjects, resources, and action), PEP sends them to PDP in order to search for an adequate policy, and then decide if the request is permitted or not.

In the first version of *curlX* [7], we did not take into account the analysis of XACML policies. For this purpose, in this paper, we integrate a new component into *curlX* denoted *CPVS* (Cloud Policy Verification Service). Figure 4.1 presents the global architecture of *curlX* after the integration of *CPVS*. It is a policy verification framework that consists of the following four verification modules (see Fig. 4.2):

- Intra-policy anomaly detection: responsible for detecting conflicting and redundant rules in a single security policy.

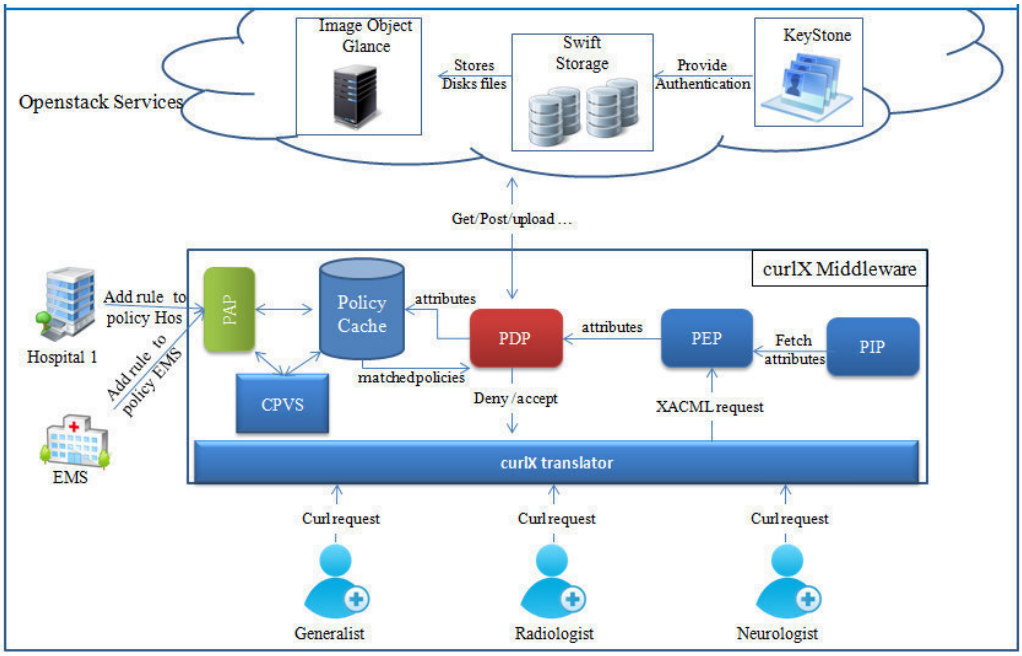

FIG. 4.1. *The global architecture of curlX*

- Inter-policy anomaly detection: if the policy cache contains combined security policies, this module is used to detect inconsistency and similarity between the combined policies.
- flexibility: medical organisations may add a new rule to their security policies. However, this new rule may create a conflict with the existing ones. Hence, this module is responsible for verifying if the new rule generates any conflicts or redundancies prior to its enforcement.
- Policy properties verification: verifies the completeness for each policy stored in the cache.

Each verification module communicates with the policy cache via the *Xparser* sub-module that parses the XACML policies and extracts its components in a hierarchical way.

5. Modelling XACML Policies by Automata.

5.1. Use Case: Stroke Accident. Healthcare organisations provide several services to their patients: emergency services, day procedures, diagnostic services, therapy services, etc. For each service, an organisation may have to produce documents (e.g. personal records, X-ray, brain scan, electroencephalography (EEG), etc). Those documents are typically stored in the organisation's data center. An organisation's data should be accessible by stakeholders from other organisations, so that they can collaborate in an elaborated diagnosis. However, information sharing must be regulated in order to guarantee the integrity and confidentiality of the shared information. This leads to the necessity of having policies regulating the medical data sharing.

Hereafter, we consider a reference scenario of a stroke accident presented by the Moroccan emergency medical service of Rabat [26]. In this scenario, three kinds of medical organisations are involved: two hospitals (H_1 and H_2), one emergency medical service (EMS) and two university hospitals (UH_1 and UH_2). These organisations are involved in a collaborative session (presented by a sequence of accesses) in order to perform an effective diagnosis to the transferred patient. In fact, doctors located in the host hospital (the hospital where the patient has been transferred) can make use of the experiences of specialists located in other medical organisations.

Our proposed scenario is that of a patient that has an accident and is transferred to the nearest hospital, which we call host hospital and denote by H_1. The Moroccan medical system relies on distributed storage: the patient can have his medical file in another hospital H_2. Once the patient is in H_1, the generalist calls EMS and a regulator receives the call and inserts the patient's information in the system. The regulator looks for

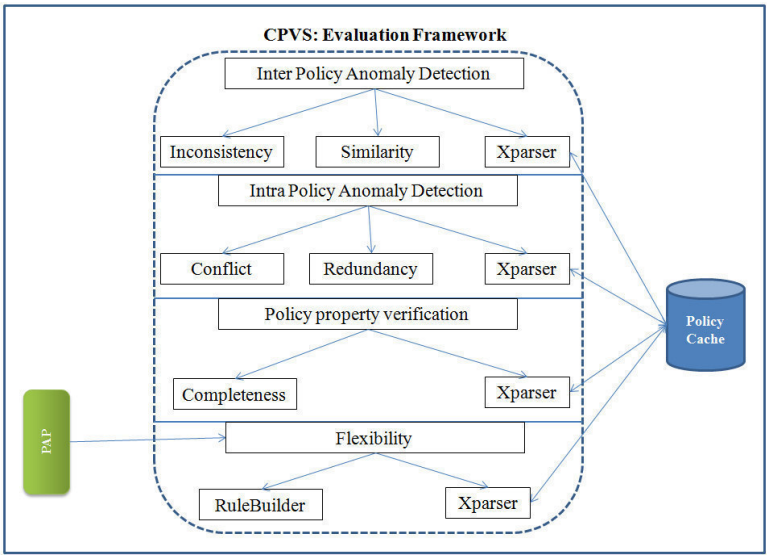

FIG. 4.2. *The four modules of CPVS (Cloud Policy Verification Service)*

the available specialist doctors: radiologists, cardiologists, neurologists etc., in other hospitals or in university hospitals. Once he finds a list of available doctors, the regulator creates a collaborative session. During the collaborative session (whose aim is to elaborate a diagnosis), the specialist doctors may ask the host hospital to provide them with some scans and a part of the medical file of the patient.

Each organisation regulates the access to its own resources by enforcing a set of access control rules. For example, we consider the rule R_1 in a hospital H_1 that permits the radiologists of H_1 to write into the personal record (PR) and the scans of all the patients belonging to this hospital. In the rest of the paper, we adopt three essential notions that are used in the security policies: subjects, objects (resources), and actions. Therefore, in the next subsection we formally describe each one of them. This description is essential for the construction of automata.

5.2. Formal Description of the Collaborating Organisations. Let Org denote the set of organisations involved in the collaborative session. Each organisation has:

- Subjects: they are human resources. Formally, we have:
  Subjects = Doctors ∪ Nurses where
  Doctors = $\bigcup_{x \in Org} doctors_x$ where
  $doctors_x = generalists_x \cup radiologists_x \cup regulators_x \cup cardiologists_x \cup neurologists_x$
- Objects: they are physical and computer resources (hardware, software). For the sake of simplicity, we consider here only the following categories: personal records (PR), scans, audio, lists of available doctors ($listDoctors_x$), and the histories of the collaborative session's discussion ($collSessDisc_x$). We obtain formulas like:
  Objects = $\bigcup_{x \in Org} objects_x$ where
  $objects_x = PR_x \cup scans_x \cup audios_x \cup listDoctors_x \cup collSessDisc_x$

For the rest of the paper, we consider four types of scans: MRI (Magnetic Resonance Imaging), CAT (Computed Axial Tomography), EEG (Electroencephalography) and MRA (Magnetic Resonance Angiogram). For the purpose of illustrating our study, we will consider only the following three categories of medical organisations that are present in most scenarios: hospitals, university hospitals, and emergency medical services. For the sake of simplicity, we will restrict their sets of subjects and objects as follows (Figure 5.1):

1. **Hospitals** H_1, H_2, H_3, ...
   The subjects of a hospital are: generalists, radiologists, and nurses. Formally:

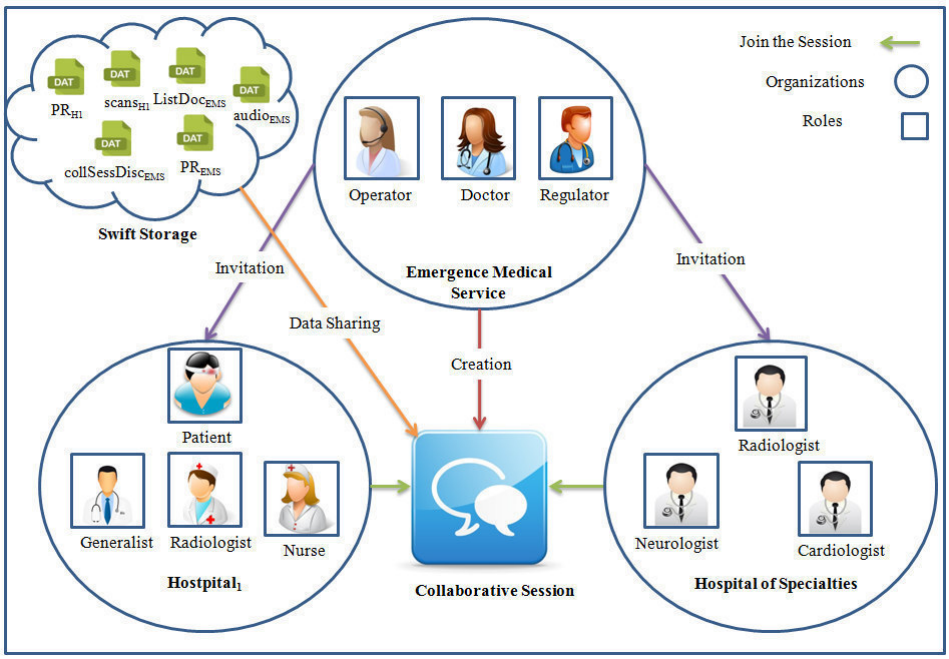

FIG. 5.1. *Organisation involved in the collaborative diagnosis.*

$subjects_{H_i} = generalists_{H_i} \cup radiologists_{H_i} \cup nurses_{H_i}$
The objects of a hospital are: personal records of patients regularly followed by the hospital, or of patients transferred to the hospital in emergency cases; and scans consisting of MRI, MRA, CAT, and EEG. Formally:
$objects_{H_i} = PR_{H_i} \cup MRI_{H_i} \cup MRA_{H_i} \cup CAT_{H_i} \cup EEG_{H_i}$.

2. **University hospitals** UH_1, UH_2, UH_3, ...
The subjects of a university hospital are specialist doctors (for our case, neurologists, cardiologists and radiologists). Formally:
$subjects_{UH_i} = neurologists_{UH_i} \cup radiologists_{UH_i} \cup cardiologists_{UH_i}$.

3. **Emergency medical services** EMS_1, EMS_2,...
The subjects of an emergency medical service are regulators and generalists. Formally:
$subjects_{EMS_i} = generalists_{EMS_i} \cup regulators_{EMS_i}$.
The objects of an emergency medical service are personal records of the hosted people and who passed through the emergency case, audio records, list of the doctors of hospitals and university hospitals, and history of the collaborative session's discussions. Formally:
$objects_{EMS_i} = PR_{EMS_i} \cup audios_{EMS_i} \cup collSessDisc_{EMS_i} \cup listDoctors_{EMS_i}$.

5.3. From XACML Policies to Automata. XACML policies have three levels, namely *PolicySet*, *Policy* and *Rule*. *Rule* is the single entity that describes the particular access control policy. Therefore, in this paper, we focus mainly on the formalisation of *Rule*. *Policy* is the sequence (combination) of several rules. *PolicySet* is the sequence (combination) of two or more policies.

A rule is formally defined by triplet (S, O, a_U), where S is a set of subjects, O is a set of objects, and a_U represents a permission. More precisely, in a_U we have a = Deny or Permit, and U is a set of actions like read and write. The meaning of (S, O, a_U) is:

- if a=Permit, then any action in U applied by a subject of S to an object of O is permitted;
- if a=Deny, then any action in U applied by a subject of S to an object of O is forbidden.

The rule described at the end of Sect. 5.1 can be written as follows: $(radiologist_{H_1}, scans_{H_1} \cup PR_{H_1}, p_{write})$. This gives the right to the radiologists of hospital H_1 to perform the write action on the two categories of objects: the scans and personal records of all patients of hospital H_1.

In XACML, a rule is described by: an *Effect* and a *Target*. The *Effect* can have two values: "*Permit*" and "*Deny*". The *Target* is a combination of *Match* elements. Each *Match* element describes an attribute that a *Request* should match in order to activate a policy. There are four attribute categories in XACML 3.0, namely: (a) subject attribute is the entity requesting the access, e.g., generalist, radiologist, etc; (b) resource attribute is the object or the required data, e.g., EEG, MRA, etc; and (c) action attribute defines the type of access requested, e.g, read, write, delete, etc. The evaluation of the *Match* attributes extracted from the rule permits the evaluation of the request. Even if the request matches one of the rules, the algorithm continues until the last rule in the *PolicySet*.

Our proposed automata-based approach is realized as follows: From the XACML representation of a policy F, we construct an automaton $\mathcal{A}$ that models F, and then our analyses of F are done on $\mathcal{A}$. The automaton $\mathcal{A}$ generated from a policy F has the following characteristics: from the initial state of $\mathcal{A}$, we have several possible paths where each path consists of *a pair* of transitions that leads to a final state associated to a permission a_U (see Sect. 3.1). Each path represents a rule (S, O, a_U) of F as follows: the first and second transitions are labelled S and O respectively, and the reached state is associated to a_U. The set of paths of $\mathcal{A}$ represents therefore a set of rules that constitute F. Table 5.1 indicates how the constituents of a XACML policy are represented in the corresponding automaton.

Table 5.1
How the constituents of a XACML policy are represented in the corresponding automaton

XACML Policies	Finite State Automaton
Rule	Word
Set of subjects and objects	Alphabet
ActionMatch attributes	Actions associated to the final states
SubjectMatch attributes	Labels of first transitions
ResourceMatch attributes	Labels of second transitions

Consider a medical organisation x and its security policy consisting of rules $x_1, x_2, \ldots, x_n$, where n is the number of rules. The construction of the automaton from the policy is done in four steps [19]:

- **Step 1: Attribute extraction from a XACML policy.** Algorithm 1 parses the XACML policy using the function *getDetailPolicy* that extracts rules from the XML file and expresses each one of them formally by a triplet (S, O, a_U). Each rule has: *Effect* (a), *SubjectMatch* (S), *ResourceMatch* (O), *Action* (U).
- **Step 2: Automaton for each rule.** each rule x_i = (S, O, a_U) obtained in Step 1 is described by an automaton with four states x_i^0, x_i^1, and x_i^2 and $x_i^{\#}$, where x_i^0 represents the initial state, x_i^2 and $x_i^{\#}$ are final states. The pair of states x_i^0 and x_i^1 are linked by a transition labelled S, and the pair of transition x_i^1 and x_i^2 are linked by a transition labelled O. The permission a_U is associated to the final state x_i^2. Transitions labelled $\neq$S and $\neq$O link x_i^0 and x_i^1 to $x_i^{\#}$ respectively. $\neq$S denotes the set of subjects of all the collaborative medical organisations, except the subjects of S. $\neq$O denotes the set of objects of the medical organisation owning the policy, except the objects of O. The final state x_i^2 is called *match state*, because it is reached for any request matching the attribute values of the rule x_i, i.e. for any subject s $\in$ S and object o $\in$ O. The final state $x_i^{\#}$ is called *no-match state*, because it is reached if the request does not match the attribute values of x_i. As an example, Fig. 5.2 represents the automaton obtained from the rule presented in the end of Section 5.1.
- **Step 3: Standardize the intervals of the automata.** The automata obtained in Step 2 do not have the same alphabet, the objective here is therefore to rewrite the transitions of the automata so that they have the same alphabet (this rewriting is useful for Step 4). This is realized by partitioning each of the domains of subjects and objects into a set of disjoint sets. Such partitioning permits to express a transition of an automaton as a union of sets of the partition.

Algorithm 1 Algorithm of Step 1

Input: XACML Policy
Output: S, O, a, U

procedure GETDETAILPOLICY(Policy.xml)
  $document \leftarrow parse(Policy.xml)$
  $root \leftarrow document.getDocumentElement()$ ▷ parses the tags of the xml file
  while $root \neq EndofDocument$ **do**
    for $i \leftarrow 0, nbRootNodes$ **do**
      if $node.getName =$ "$Rule$" and $node.getAttributes \neq null$ **then**
        $a \leftarrow node.getAttributes.getNamedItem($"$Effect$"$)$
        if node.getNodeName = "SubjectMatch" **then**
          $S \leftarrow Attribute.getTextContent$
        else if node.getNodeName = "ResourceMatch" **then**
          $O \leftarrow Attribute.getTextContent$
        else if node.getNodeName() = "ActionMatch" **then**
          $U \leftarrow Attribute.getTextContent$
        end if
      end if
    end for
  end while
  return S, O, a, U
end procedure

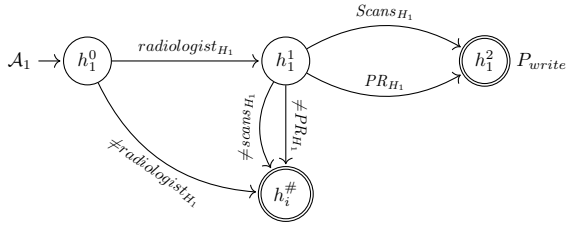

FIG. 5.2. *Automaton $\mathcal{A}_1$ obtained in Step 1 for the rule R_1.*

For example, the automaton of Fig. 5.2 is transformed into the automaton of Fig. 5.3. The set $scans_{H_1}$ has been partitioned into 4 sets $\mathrm{MRI}_{H_1}, \mathrm{MRA}_{H_1}, \mathrm{CAT}_{H_1}$ and EEG_{H_1}, which implies that the transition labelled $scans_{H_1}$ is replaced by four transitions labelled $\mathrm{MRI}_{H_1}, \mathrm{MRA}_{H_1}, \mathrm{CAT}_{H_1}$ and EEG_{H_1}, respectively.

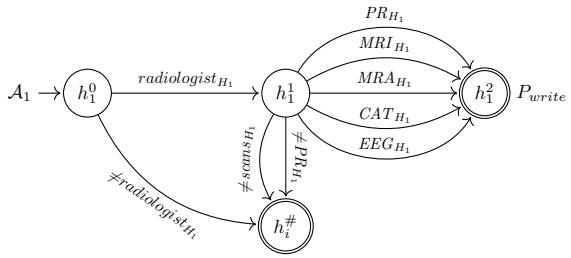

FIG. 5.3. *Automaton $\mathcal{A}_1^*$ obtained from $\mathcal{A}_1$ of Fig. 5.2*

- **Step 4: Product of automata**. In order to model the security policy defined in a XACML policy file, we combine the automata resulting from Step 3 by an operator called *synchronous product*. Hereafter, we consider the policy presented in Table 5.2 as an example. It contains seven rules regulating access to different resources (objects).
  The resulting automaton representing the policy of an organisation x is denoted $\mathcal{A}_x$. Each of its states is a combination of states $(u_1, u_2, \ldots, u_n)$ of the various combined automata, hence each $u_i = x_i^j$ or $x_i^\#$, for $j = 1$ or 2. A final state $x_{\ldots}^2$ may be associated to one or more permissions. For the sake of clarity, a state of $\mathcal{A}_x$ is noted $x_{i_1,i_2,\ldots}^j$, where we indicate only the indices i_k such that $u_{i_k} = x_{i_k}^j$ (i.e. $u_{i_k} \neq x_{i_k}^\#$), for $j = 1$ or 2. For example, the initial state is noted $x_{1,2,\ldots,n}^0$. A state is noted $x^\#$ if all its

TABLE 5.2
Example of a XACML policy

RuleID	Effect	SubjectMatch	ResourceMatch	ActionMatch
R_1	Permit	generalist	PR	read
R_2	Permit	neurologist	EEG	read
R_3	Permit	radiologist	Scans	write
R_4	Deny	radiologist	Scans	write
R_5	Deny	generalist	PR	read
R_6	Permit	neurologist	EEG	read
R_7	Permit	generalist	PR	read

components are $x_i^{\#}$. For example, if we apply the four steps to the policy of Table 5.2, we obtain the automaton of Fig. 5.4.

Let us explain the notation used for match-states, for example the match-state $h^2_{1,5,7}$ associated to permissions (P_{read}, D_{read}, P_{read}). This state is reached by the pair of transitions (generalist, PR), i.e. when a generalist wants to have access to a personal record of a patient. The three indices 1, 5, and 7 mean that this access is matched by the rules 1, 5 and 7. The three permissions are respectively associated to the three indices, i.e.: R_1 and R_7 permit the read access, and R_5 forbids the read access.

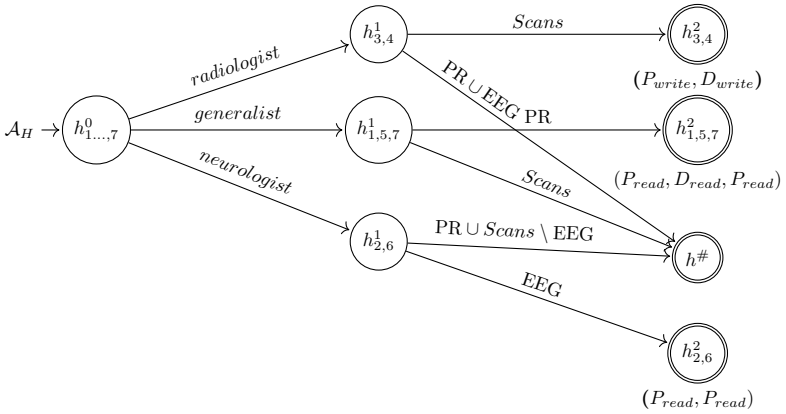

FIG. 5.4. *Automata $\mathcal{A}_H$ Modelling the Security Policy of Table 5.2.*

6. Intra- and Inter-security Policy Anomaly Detection. In a XACML policy, we can specify the policies of several entities (or organisations, like hospitals): the policy of each organisation is represented by *Policy*. *PolicySet* specifies how the different *Policy* components are combined. We study therefore two types of anomalies: intra-policy anomalies that correspond to anomalies between rules of a same policy, and inter-policies anomalies that correspond to anomalies between rules of different policies. In this paper, we consider two types of intra-policy anomalies: redundancy and conflicting rules, and two types of inter-policy anomalies: similarity and inconsistency.

6.1. Intra-policy anomaly detection. Let us first consider intra-policy anomalies of a policy F and show how they are detected by the automaton of F.

6.1.1. Detecting redundant rules. In a policy F, a rule R_j is redundant to a rule R_i if the result of F is not changed by removing R_j and keeping R_i.

PROPOSITION 1. *Consider a policy F and its automaton $\mathcal{A}_F$. A rule R_j is redundant to a rule R_i if for every match-state $x^2_{j_1,j_2...}$ of $\mathcal{A}_F$:*

1. *the match-state has the index j only if it has also the index i, and*
2. *i and j are associated to the same permission.*

Consider, for example, the policy of Table 5.2 and its automaton of Fig. 5.4. Rules 2 and 6 are mutually redundant to each other, because: 1) the indices 2 and 6 are in the state $h^2_{2,6}$ and there is no other state where the indices are not together, and 2) the same permission P_{read} is associated to both indices. Therefore, we can remove either R_2 or R_6 without changing the result of the policy.

6.1.2. Detecting conflicting rules. In a policy F, two rules R_i and R_j are conflicting if they can match the same subjects and objects (s,o) and have different permissions.

PROPOSITION 2. *Consider a policy F and its automaton $\mathcal{A}_F$. Rules R_i and R_j are conflicting if there exists a match-state $x^2_{j_1,j_2\cdots}$ of $\mathcal{A}_F$ such that:*

1. *the match-state has the indices i and j, and*
2. *i and j are associated to different permissions.*

Consider, for example, the policy of Table 5.2 and its automaton of Fig. 5.4. The match-state $h^2_{3,4}$ implies that R_3 and R_4 are conflicting. Intuitively, R_3 permits radiologists to read scans while R_4 forbids it. Also, the match-state $h^2_{1,5,7}$ implies that R_5 is conflicting with R_1 and R_7. Intuitively, R_1 and R_7 permits generalists to read PR while R_5 forbids it.

Algorithm 2 in Appendix A regroups the steps of detecting intra-policy conflicts. The algorithm consists in extracting the final states using the function *getFinalNodes*. The algorithm compares then the permissions associated to each final state, if they are different, the rules are considered as conflicting rules, and otherwise they are redundant.

6.2. Inter-policy anomaly detection. Let us now consider inter-policy anomalies of two policies F_1 and F_2. To detect this type of anomalies, we need here to construct an automaton that combines the automata $\mathcal{A}_{F1}$ and $\mathcal{A}_{F2}$. This is equivalent to consider the global policy F obtained by putting together the rules of F_1 and F_2 and then constructing the automaton $\mathcal{A}_F$ of F.

6.2.1. Detecting similar policies. Two policies F_1 and F_2 are similar if in every situation, the decision of F_1 is similar to the decision of F_2.

PROPOSITION 3. *Consider two policies F_1 and F_2 and the automaton $\mathcal{A}_F$ of the policy F that regroups F_1 and F_2. F_1 and F_2 are similar if in F: every rule of F_1 is redundant to a rule of F_2, and every rule of F_1 is redundant to a rule of F_2.*

Proposition 3 implies that similarity can be verified by detecting redundancy between the rules of the policies using Proposition 1.

6.2.2. Detecting inconsistent policies. Two policies F_1 and F_2 are inconsistent if there exists a situation where they have contradictory (i.e., different) decisions.

PROPOSITION 4. *Consider two policies F_1 and F_2 and the automaton $\mathcal{A}_F$ of the policy F that regroups F_1 and F_2. F_1 and F_2 are inconsistent if in F: there exist a rule of F_1 and a rule of F_2 which are conflicting.*

Proposition 4 implies that inconsistency can be verified by detecting conflicting rules using Proposition 2.

Therefore, the same logic of the two procedures of Algorithm 2 can be applied to detect inconsistency and policy similarity. The only difference is the input of the algorithm: instead of an automaton corresponding to one policy, the input is replaced by a synthesized automaton of two policies (Algorithm 3 in Appendix A).

7. Verification of the Completeness Property. Besides the intra- and inter-security policy anomalies, it is important to assure the evaluation of security properties to guarantee the correctness of access control policies. Most of the existing Cloud verification methods focus mainly on the system behaviour verification and do not take into consideration the security policies. Therefore, designing a dedicated tool that targets the verification of security properties in the Cloud is an important issue to be addressed [13]. In this section, we describe a formal method based on the automata generated in Section 5 to detect and verify the completeness property. Completeness guarantees that each access request is either accepted or denied by the access control policy.

PROPOSITION 5. *A security policy $\mathcal{P}$ is complete if and only if the corresponding synthesized automaton $\mathcal{A}_P$ has no* no-match state.

For instance, the security policy presented in Table 5.2 is incomplete because its corresponding automaton in Fig. 5.4 has a *no-match state* denoted $h^{\#}$. The 3 paths leading to $h^{\#}$ correspond to the following 3 situations:

- A radiologist requests access to a PR or EEG.
- A generalist requests access to a scan.
- A neurologist requests access to a PR or a scan that is not EEG.

Intuitively, the security policy of Table 5.2 does not take any decision in these 3 situations. Algorithm 4 in Appendix A presents the procedure *isComplete* that verifies if the input automaton has a no-match state.

8. Evaluation of Space and Time Complexities. Let n be the number of rules of a policy, and d_1 and d_2 be the numbers of bits to code the subjects and objects, respectively. Hence, the maximum possible number of subjects and objects are 2^{d_1} and 2^{d_2}, respectively. Let $D = d_1 + d_2$. We consider two notions called *great fields* and *small fields* defined by Khoumsi et al. [19]. A *great field* is a field whose domain contains more than n values, and a *small field* is a field whose domain contains at most n values. We then consider two variables: μ, the number of *great fields*; and δ, the sum of the number of bits to code the *small fields*.

By adapting the results of Khoumsi et al. [19] to our context, we obtain Proposition 6 (the proof of this proposition is in Appendix B).

PROPOSITION 6. *The space and time complexities of automata construction and completeness detection are in* $O(n^{\mu+1} \times 2^{\delta})$*, which is bounded by both* $O(n^3)$ *and* $O(n \times 2^D)$.

The bounds of the complexities for the procedures of policy analysis are obtained by multiplying the above values by n. Hence, we obtain the following proposition:

PROPOSITION 7. *The space and time complexities of redundancy and conflict detections are in* $O(n^{\mu+2} \times 2^{\delta})$*, which is bounded by* $O(n^4)$ *and* $O(n^2 \times 2^D)$.

The latter result holds also for detecting similarity and inconsistency between two policies, but by replacing n by $n_1 + n_2$, where n_1 and n_2 are the numbers of rules of the two policies.

As an example, we can consider a policy with $n = 500$ rules where the maximum number of subjects is 256 (so the subjects are coded in 8 bits: $2^8 = 256$) and where the maximum number of objects is 131072 (hence the objects are coded in 17 bits: $2^{17} = 131072$). Hence:

- D = 25 = 8+17 = total number of bits to code the great and small fields.
- $\mu = 1$ = number of great fields: there is one great field which is "objects", because $2^{17} > 500$.
- $2 - \mu = 1$ = number of small fields: there is one small field which is "subjects", because $2^8 \leq 500$.
- $\delta = 8$ = number of bits to code the small field subjects

If we use the expression $O(n^{\mu+1} \times 2^{\delta})$ which depends on the great and small fields, we obtain: $O(n^{\mu+1} \times 2^{\delta}) = O((500^2) \times (2^8)) = O(64$ millions). However, if we use the two expressions $O(n^2)$ and $O(n \times 2^D)$ which do not depend on the great and small fields, we obtain:

- $O(n^2) = O(500^3) = O(125$ millions)
- $O(n \times 2^D) = O(500 \times 2^{25}) = O(16.7$ billions)

From the example, we can conclude that by considering the great and small fields, we obtain a more precise estimation of the complexity. Note that:

- when all the fields are great, we obtain $O(n^{\mu+1} \times 2^{\delta}) = O(n^2)$,
- when all the fields are small, we obtain $O(n^{\mu+1} \times 2^{\delta}) = O(n \times 2^D)$.

From Proposition 6, the time and space complexities of automata construction and completeness detection are upper-bounded by $O(n^3)$ and $O(n \times 2^D)$. Let *Ns* and *No* be the maximum numbers of subjects and objects, respectively. We have $2^D = 2^{d_1} \times 2^{d_2}$, $Ns = 2^{d_1}$ and $No = 2^{d_2}$. Therefore, $O(n \times 2^D) = O(n \times Ns \times No)$. We deduce that our complexities of automata construction and completeness detection exceeds neither the order of the polynomial n^3 nor the order of $n \times Ns \times No$.

With the same reasoning on Proposition 7 we obtain that our complexities of redundancy and conflict detections exceed neither the order of the polynomial n^4 nor the order of $n^2 \times Ns \times No$.

In conclusion, our complexities are at most polynomial in n and linear in *Ns* and *No*.

9. Implementation and Evaluation. We have implemented our policy analysis service *CPVS* (Cloud Policy Verification Service) in Java. This service is integrated into *curlX*, a middleware integrated into *Openstack*. Based on our policy anomaly analysis mechanism, *CPVS* consists of four core components: Inter-policy anomaly detection module, intra-policy anomaly detection module, policy property verification module, and flexibility module. The modules are described in detail in Section 4. *CPVS* makes use of the DOM API provided by the Sun XACML implementation in order to parse the XACML policies and extract the attributes.

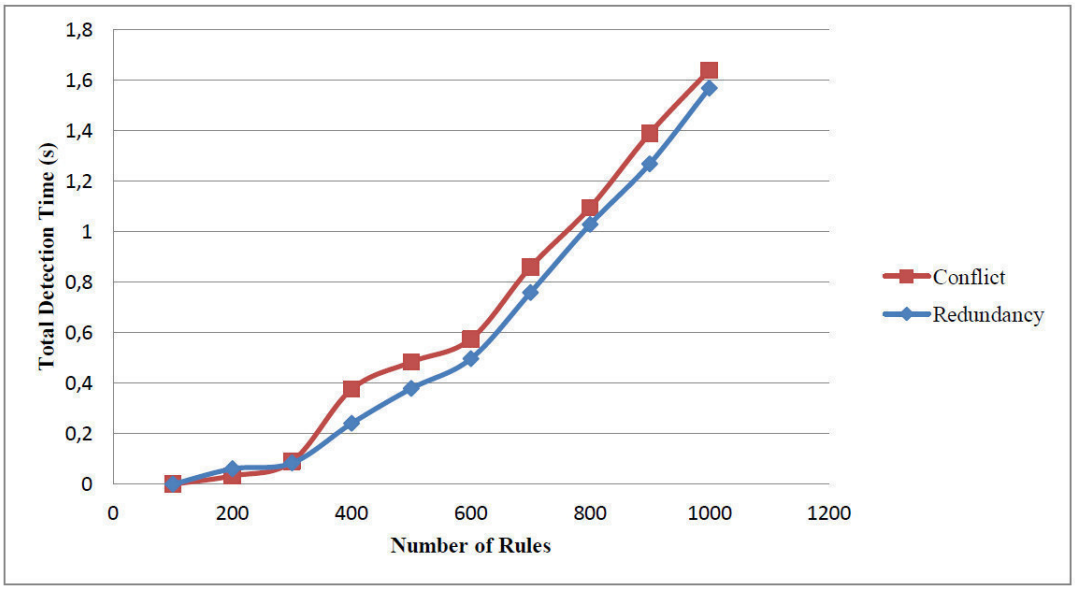

FIG. 9.1. *Performance Improvement for Intra-policy Anomaly Detection*

We have implemented a Domain Specific Language (DSL) to support the construction of automata. In order to evaluate the efficiency and effectiveness of the proposed solution, first we need large policy data sets. Unfortunately, no one has been published due to confidentiality constraints. Hence, we have developed a random policy generator in order to generate a large number of XACML policies.

For scalability, it is also important to note that creating subjects and objects with no semantic relationship (categorisation) is an inefficient approach. It is better to regroup the subjects and objects in subsets or categories to reduce their sizes. For instance, we can have 10 objects: 4 files consisting of prescriptions and 3 scans (*EEG*, *MRI*, *BrainScan*), and 3 documents containing information about the patient. For this example, we have two sub-categories: Scans and PR (prescriptions and documents). In this way, instead of having 10 atomic objects, we have only 2 subsets. This reduces the number of states in the final policy automaton, which then reduces the time of anomaly detection.

We evaluated the efficiency and effectiveness of *CPVS* for synthetic XACML policies using 10 synthetic generated policies. Our experiments were performed on an Intel Core 2 Duo CPU 2.00 GHz with 3.00 GB RAM running on Windows 7. We adopted three types of performance measurement related to intra-policy anomaly detection, inter-policy anomaly detection and incompleteness detection.

The time required by *CPVS* to detect anomalies, such as redundancy and conflicts, depends on the time of parsing and comparing the final states of the automaton. From Fig. 9.1, we can notice that the times of conflict and redundancy detections are quasi equal, which reflects the results of time complexity of Section 8.

Furthermore, we generated synthetic policies consisting of 100 rules, and we combined them using the synchronous product. Figure 9.2 presents the performance of *CPVS* to detect inconsistency and similarity between different set of policies (2, 4 . . . 10).

The verification of completeness, which consists in finding the *no-match* state in the policy's automata, depends on the location of such a state. The performance of such verification is quasi constant (Fig. 9.3). It remains 2 ms for policies that contain 100, 200, 300 and 400 rules, and then it goes to 3 ms for the four other policies.

10. Conclusion. We have proposed a formal approach based on automata to detect XACML policy anomalies and verify the policy completeness. Our proposed approach consists of four steps: (1) extracting attributes from XACML policies; (2) modelling each rule by an automaton; (3) standardising the sets of transitions in automata; and finally (4) forming products of automata to model the security policy. The

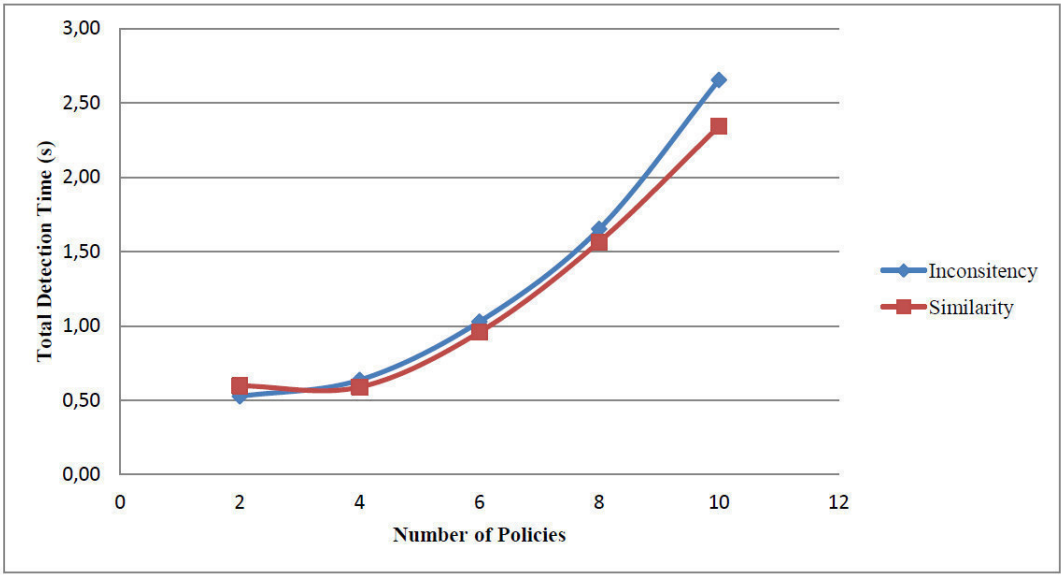

FIG. 9.2. *Performance Improvement for Inter-policy Anomaly Detection*

resulted automaton is used to detect anomalies based on analysing its final states. We evaluated the time and space complexities for anomaly detection. The approach has been implemented in a Cloud service called *CPVS* (Cloud Policy Verification Service) integrated into a middleware denoted *curlX*. The advantage of our approach is that it detects several anomalies in XACML policies at two different levels using the same formal model. In fact, it can detect intra-policy anomalies such as conflicts and redundancies, and inter-policy anomalies such as inconsistencies and similarities.

In future work, we intend to propose a formal approach to resolve the detected anomalies based on a dynamic aspect. The resolution takes into consideration the XACML combining algorithms. Moreover, the concept of delegation is often used in the domain of e-health, so we intend to verify the impact of delegation of roles on the security policies.

Acknowledgements. The authors want to thank Najib Fouhami for his help during the implementation and Yahya Benkaouz for his helpful discussions. This work is supported by the BMBF (PMARS Program) and the DAAD (German-Arab Transformation Partnership).

REFERENCES

[1] B. CALABRESE AND M. CANNATARO, *Cloud computing in healthcare and biomedicine*, Scalable Computing: Practice and Experience, vol. 16, no. 1, pp. 1–18, 2015.
[2] S. P. AHUJA, S. MANI, AND J. ZAMBRANO, *A survey of the state of cloud computing in healthcare*, Network and Communication Technologies, vol. 1, no. 2, pp. 12-19, 2012.
[3] V. ALFRED, *Algorithms for finding patterns in strings*, Algorithms and Complexity, vol. 1, pp. 255-300, 2014.
[4] A. ANDERSON, A. NADALIN, B. PARDUCCI, D. ENGOVATOV, H. LOCKHART, M. KUDO, P. HUMENN, S. GODIK, S. ANDERSON, S. CROCKER, T. MOSES, *Extensible access control markup language (xacml) version 1.0*, OASIS, 2003.
[5] J. ARNOLD, *OpenStack Swift: Using, Administering, and Developing for Swift Object Storage.* " O'Reilly Media, Inc.", 2014.
[6] M. AYACHE, M. ERRADI, AND B. FREISLEBEN, *curlx: A middleware to enforce access control policies within a cloud environment*, in 2015 IEEE Conference on Communications and Network Security (CNS), pp. 771–772, IEEE, 2015.
[7] M. AYACHE, M. ERRADI, AND B. FREISLEBEN, *Access control policies enforcement in a cloud environment: Openstack*, in 2015 11th International Conference on Information Assurance and Security (IAS), pp. 25–30, IEEE, 2015.
[8] F. BLANQUI, *Introduction to the coq proof assistant*, Lecture notes available on https://who. rocq. inria. fr/Frederic. Blanqui, 2013.
[9] H. BOUAMOR, H. SAJJAD, N. DURRANI, AND K. OFLAZER, *Qcmuq@ qalb-2015 shared task: Combining character level mt and error-tolerant finite-state recognition for arabic spelling correction*, in ANLP Workshop 2015, p. 144, 2015.

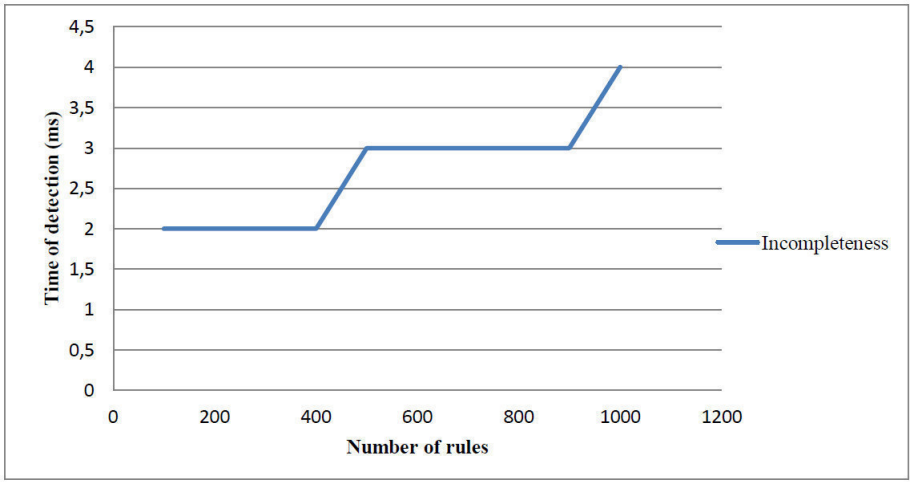

FIG. 9.3. *Performance of the Completeness Property Verification*

[10] O. CHOUDHURY, N. L. HAZEKAMP, D. THAIN, AND S. EMRICH, *Accelerating comparative genomics workflows in a distributed environment with optimized data partitioning*, in 2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), pp. 711–719, IEEE, 2014.

[11] J. A. EVANS, *Electronic medical records system*, July 13 1999. US Patent 5,924,074.

[12] K. FISLER, S. KRISHNAMURTHI, L. A. MEYEROVICH, AND M. C. TSCHANTZ, *Verification and change-impact analysis of access-control policies*, in 27th International conference on Software engineering, pp. 196–205, ACM, 2005.

[13] A. GAWANMEH AND A. ALOMARI, *Challenges in formal methods for testing and verification of cloud computing systems*, Scalable Computing: Practice and Experience, vol. 16, no. 3, pp. 321–332, 2015.

[14] A. GAWANMEH, H. AL-HAMADI, M. AL-QUTAYRI, S.-K. CHIN, AND K. SALEEM, *Reliability analysis of healthcare information systems: State of the art and future directions*, in 2015 17th International Conference on E-health Networking, Application & Services (HealthCom), pp. 68–74, IEEE, 2015.

[15] A. GOUGLIDIS, I. MAVRIDIS, AND V. C. HU, *Security policy verification for multi-domains in cloud systems*, International Journal of Information Security, vol. 13, no. 2, pp. 97–111, 2014.

[16] H. HU, G.-J. AHN, AND K. KULKARNI, *Discovery and resolution of anomalies in web access control policies*, IEEE Transactions on Dependable and Secure Computing, vol. 10, no. 6, pp. 341–354, 2013.

[17] F. HUONDER, *Conflict detection and resolution of xacml policies*, Master's thesis, University of Applied Sciences Rapperswil, 2010.

[18] W. JANSEN, *Cloud hooks: Security and privacy issues in cloud computing*, in 2011 44th Hawaii International Conference on System Sciences (HICSS), pp. 1–10, IEEE, 2011.

[19] A. KHOUMSI, W. KROMBI, AND M. ERRADI, *A formal approach to verify completeness and detect anomalies in firewall security policies*, in Foundations and Practice of Security, pp. 221–236, Springer, 2014.

[20] A. LI, Q. LI, V. C. HU, AND J. DI, *Evaluating the capability and performance of access control policy verification tools*, in Military Communications Conference, MILCOM 2015-2015 IEEE, pp. 366–371, IEEE, 2015.

[21] M. LORCH, S. PROCTOR, R. LEPRO, D. KAFURA, AND S. SHAH, *First experiences using xacml for access control in distributed systems*, in 2003 ACM Workshop on XML security, pp. 25–37, ACM, 2003.

[22] M. ST-MARTIN AND A. P. FELTY, *A verified algorithm for detecting conflicts in XACML access control rules*, in 5th ACM SIGPLAN Conference on Certified Programs and Proofs, Saint Petersburg, FL, USA, January 20-22, 2016, 2016.

[23] E. MARZINI, P. MORI, S. DI BONA, D. GUERRI, M. LETTERE, AND L. RICCI, *A tool for managing the x1. v1 platform on the cloud*, Scalable Computing: Practice and Experience, vol. 16, no. 1, pp. 103–120, 2015.

[24] A. MOURAD, H. TOUT, C. TALHI, H. OTROK, AND H. YAHYAOUI, *From model-driven specification to design-level set-based analysis of xacml policies*, Computers & Electrical Engineering 52, pp. 65-79, 2016.

[25] OASIS, Extensible access control markup language (xacml) version 2.0, 2005.

[26] M. OUZZIF, M. HAMDANI, H. MOUNTASSIR, AND M. ERRADI, *A formal modeling approach for emergency crisis response in health during catastrophic situation*, in International Conference on Information Systems for Crisis Response and Management in Mediterranean Countries, pp. 112–119, Springer, 2014.

[27] K. PEPPLE, *Deploying openstack.* " O'Reilly Media, Inc.", 2011.

[28] U. PERVEZ, O. HASAN, K. LATIF, S. TAHAR, A. GAWANMEH, AND M. S. HAMDI, *Formal reliability analysis of a typical fhir standard based e-health system using prism*, in 2014 IEEE 16th International Conference on e-Health Networking, Applications and Services (Healthcom), pp. 43–48, IEEE, 2014.

[29] U. PERVEZ, A. MAHMOOD, O. HASAN, K. LATIF, AND A. GAWANMEH, *Formal reliability analysis of device interoperability middleware (dim) based e-health system using prism*, in 2015 17th International Conference on E-health Networking, Application & Services (HealthCom), pp. 108–113, IEEE, 2015.

[30] C. D. P. K. Ramli, *Detecting incompleteness, conflicting and unreachability xacml policies using answer set programming*, arXiv preprint arXiv:1503.02732, 2015.

[31] A. Sureshkumar, M. De Vos, M. Brain, and J. Fitch, *Ape: An ansprolog* environment*, in First International Workshop on Software Engineering for Answer Set Programming (SEA), vol. 7, pp. 101–115, 2007.

Appendix A. Algorithms of Anomaly Detecing and Completness Verification.

Algorithm 2 Intra-policy Anomaly Detection

Input: Policy Automaton
Output: Redundancy Set $\mathcal{RS}$ and Conflicts Set $\mathcal{CS}$

procedure IsRedundant(Automaton) ▷ Verify if there are any redundant rules
  $nodes \leftarrow getFinalNodes(Automaton)$ ▷ Extract the final states from the automaton
  while $node.size \neq 0$ and $nodes \neq null$ **do**
    for $i \leftarrow 0, node.size$ **do**
      for $j \leftarrow i+1, node.size$ **do**
        if $node.get(i) = node.get(j)$ **then** ▷ The permissions of the final states are equal
          RS.add(i,j) ▷ Add both rules i and j to the redundant Set
        end if
      end for
    end for
  end while
  return $\mathcal{RS}$
end procedure
procedure hasConflict(Automaton) ▷ Verifies if there are any conflicting rules
  $nodes \leftarrow getFinalNodes(Automaton)$
  while $node.size \neq 0$ and $nodes \neq null$ **do**
    for $i \leftarrow 0, node.size$ **do**
      for $j \leftarrow i+1, node.size$ **do**
        if $node.get(i) \neq node.get(j)$ **then** ▷ The permissions of the final states are not equal
          CS.add(i,j) ▷ Add both rules i and j to the conflict set
        end if
      end for
    end for
  end while
  return $\mathcal{CS}$
end procedure

Algorithm 3 Inter-policy Anomaly Detection

Input: $Policy_1\mathcal{P}_1$, $Policy_2\mathrm{P}_2$
Output: Inconsistency Set $\mathcal{IS}$ and Similarity Set $\mathcal{SS}$

procedure InterPolicyAnalyzer(Automaton) ▷ Detects the inconsistency and the similarity
  $ProductAutomaton \leftarrow generateProductAutomaton(\mathcal{P}_1, \mathcal{P}_2)$ ▷ The automaton of the global policy
  $\mathcal{IS} \leftarrow IsRedundant(ProductAutomaton)$
  $\mathcal{SS} \leftarrow hasConflict(ProductAutomaton)$
  Return $\mathcal{IS}$ and $\mathcal{SS}$
end procedure

Algorithm 4 Verification of the Completeness Property

Input: Policy Automaton
Output: Verification of Completeness (C)

procedure isComplete(Automaton)
  $nodes \leftarrow getFinalNodes(Automaton)$
  while $node.size \neq 0$ and $nodes \neq null$ **do**
    for $i \leftarrow 0, node.size$ **do**
      if $node.get(i) = E_i$ **then** ▷ E_i represents a *non-match state*
        return The policy is not complete
      end if
    end for
  end while
end procedure

Appendix B. Proof of Proposition 6. We use the notation $\Psi_i = \min(2^{d_i};\, n)$. We omit the complexity of Step 1 because it needs a fixed, and finite time O(1)).

B.1. Complexity of Step 2. The space and time complexities to construct one state or one transition of $\mathcal{A}_i$ are in O(1). Each $\mathcal{A}_i$ contains 4 states and a limited number of transitions from each state. Hence, the space and time complexities to construct each $\mathcal{A}_i$ are in O(1). Since we have to construct n automata, the space and time complexities of Step 2 are in O(n).

B.2. Complexity of Step 3. This step consists in replacing each set of objects and subjects by the corresponding transitions. The number of transitions from r_i^j of $\mathcal{A}_i^*$ is bounded by both O(2^{d_j}) and O(n) which means O(Ψ_i). The bound O(2^{d_j}) is because 2^{d_j} is the number of possible values of either subjects or objects, which is necessarily $\geq$ than the number of transitions from r_i^j. Hence, the space and time complexities to construct all the transitions of $\mathcal{A}_i^*$ are in O($\Psi_0 + \Psi_1$). Therefore, the space and time complexities to construct all the $\mathcal{A}_i^*$ are in O($n \times (\Psi_0 + \Psi_1)$).

B.3. Complexity of Step 4. Let us consider the construction of $\mathcal{A}_F$ in Step 4 level by level, where the states of level i are those reached after i transitions from the initial state. At each level i, the transitions links level i-1 to level i. The space and time complexities to construct a state r = $<r_1;\ldots; r_n>$ of $\mathcal{A}_F$ are in O(n), because we need to construct and store the n components of the state. The space and time complexities to construct a transition between two constructed states of levels i and i+1 are in O(1), because we need to store the label of the transition.

Level 0: The unique state is the initial state $r^0 = <q_1^0;\ldots; q_n^0>$. The space and time complexities of its construction are in O(n).

Level 1: Using the same reasoning as in the proof of Step 3, the number of transitions from r^0 is in O(Ψ_0). Hence, the number of states at level 1 is in O(Ψ_0). Therefore, the space and time complexities to construct all the transitions from level 0 to level 1 are in O(Ψ_0), and the space and time complexities to construct all the states at level 1 are in O($n \times \Psi_0$).

Level 2: The number of transitions from each state of level 1 is in O(Ψ_1). Since the number of states of level 1 is in O(Ψ_0), we obtain that the number of states at level 2 and the number of transitions from level 1 to level 2 are in O($\Psi_0 \times \Psi_1$). Therefore, the space and time complexities to construct all the states at level 2 are in O($n \times \Psi_0 \times \Psi_1$), and the space and time complexities to construct all the transitions from level 1 to level 2 are O($\Psi_0 \times \Psi_1$). At each level j, the number of states is also bounded by 2^n, because each state is defined by n 2-value components r_i ($r_i = q_i^j$ or $r_i = E_i$, for $i = 1 \ldots n$). But this bound has no influence due to the assumptions n>D and $2^n > n^2$.

All levels: By adding the complexities of all levels, we obtain that the space and time complexities of constructing $\mathcal{A}_F$ are in O($n \times \Psi_0$)+ O($n \times \Psi_0 \times \Psi_1$).
From $d_i \geq 1$ and $n >$ D, we obtain $\Psi_i \geq 2$, from which we deduce that Ψ_0+($\Psi_0 \times \Psi_1$) $\leq 2\times(\Psi_0 \times \Psi_1)$. Hence, the space and time complexities of constructing $\mathcal{A}_F$ are in O($n \times \Psi_0 \times \Psi_1$).

Associating permissions: It remains to compute complexities of associating permissions to the match states of $\mathcal{A}_F$. The space complexity of associating a permission to a match state of $\mathcal{A}_F$ is in O(1), because we only need to store the permission associated to the state. The time complexity of associating permissions to all match states of $\mathcal{A}_F$ is in O(n), because we may need to consult the n components of the state.

B.4. Total complexity. Since Steps 1 to 3 are less complex than Step 4, we obtain that the space and time complexities for constructing $\mathcal{A}_F$ are in O($n \times \Psi_0 \times \Psi_1$). By definition of μ and δ, we obtain $n \times \Psi_0 \times \Psi_1 = n^{\mu+1} \times 2^{\delta}$, which can be easily shown to be smaller than both n^3 and $n \times 2^D$.

Edited by: Amjad Gawanmeh
Received: Feb 1, 2016
Accepted: Jul 3, 2016