Context-aware Automatic Access Policy Specification for IoT Environments

[1]Ashraf Alkhresheh, [2]Khalid Elgazzar, [1]Hossam S. Hassanein
[1]School of computing, Queen's University, Kingston, ON K7L 3N6, Canada
[2]Department of Electrical, Computer and Software Engineering
University of Ontario Institute of Technology, Oshawa, ON L1H 7K4 Canada
Email: khashraf@cs.queensu.ca, Khalid.Elgazzar@uoit.ca, hossam@cs.queensu.ca

***Abstract*—Data privacy becomes a primary impediment to the realization of the IoT vision. One approach to the IoT security and privacy problem is to restrict access to sensitive data via access control and authorization models. Yet access context in IoT changes frequently raising the need for flexible and dynamic access control policies. Towards developing dynamic access control policies, context-based access control techniques are being investigated due to their robustness in assigning dynamic access permissions according to changes in context. In this paper, we propose to automate the generation of access control policies to overcome the inflexibility in traditional access policy specification techniques, and improve its adaptability to dynamic IoT environments. In our framework, we use context, attributes, and predication to describe the core access control elements. In response to access requests, our algorithm automatically produces conflict-free access control policies and makes the final access decisions at runtime. Our framework prevents non-authorized data accesses, and satisfies privacy constraints for authorized access requests in highly dynamic IoT environments. Our preliminary evaluation shows that the proposed approach offers greater flexibility and improved scalability than the current state-of-the-art methods.**

I. Introduction

A recent study conducted by IoT analytics [1] to rank IoT applications based on their popularity reported that in 2016, on average, there were 60,000 Google searches issued every month for the term "smart home," 41,000 for "smart city," and 33,000 for "wearables." However, the proliferation of smart devices in our surroundings enables pervasive collection, processing and dissemination of personal information, raising considerable privacy concerns among IoT users. As such, data privacy inhibits the widespread adoption of IoT. In order to unleash the full potential of IoT, smart devices should make their data accessible to interested parties (e.g., smartphones, web services) in a controlled manner; otherwise, potential IoT privacy breaches will outweigh its benefits.

One approach to the security and privacy problem in IoT is to restrict access to sensitive data via access control and authorization models. Traditional access control approaches such as Role Based Access Control (RBAC) [2], Attribute Based Access Control (ABAC) [3] and Task-Based Access Control (TBAC) [4] control access to shared resources based on static considerations such as user identity, role, and attributes. However, these approaches fall short in high dynamic computing environments as is the IoT, wherein most interaction scenarios, the identity, role or attributes of whom we will be sharing our resources with might not be known in advance. Thus, instead of relying on user identity, role or attributes only, an access control policy should consider other information that is related to the access in hand, such as the condition of the surrounding environment in which the access request takes place.

Operational factors (e.g., processors, memories and operating systems), situational factors (e.g., location, time, and network security configuration) and environmental factors (e.g., temperature, humidity) are examples of information that can greatly affect access control decisions and subsequently the performance of an access control system. Hereinafter, we refer to this information collectively as context information. Context refers then to any information that is considered relevant to interacting IoT entities as well as environmental conditions under which these interactions take place [5].

To point out some unique access control challenges in dynamic IoT environments, we begin by considering a dynamic access scenario where a former student is meeting their supervisor at the office. According to supervisor's calendar, the meeting is scheduled for two hours. When the student comes on campus, based on their current context (i.e., student's profile, location and time), they will be able granted access to some resources campus-wide such as smart parking, Wi-Fi network. As the student's context changes (e.g., the student enters a building), they will dynamically be assigned additional access privileges such as entrance to their lab. While waiting for the supervisor, the student may not have privileges to access the lab facilities. However, when the supervisor arrives, the student's context changes again so that during the meeting time, the coexistence of both the supervisor and student in the lab enables the student, to access lab-specific privileges (e.g., the room temperature of the lab, printer and other lab utilities).

In such scenarios, it would be impossible to define all the necessary policies for all possible situations in advance. For example, let us consider the case of a meeting that continues beyond its allotted time. It is important to ensure that the student can continue to access campus and lab resources as long as the meeting is taking place. If an unforeseen change happens, it would be necessary to modify previous policies to reflect the new context of the meeting very quickly. Without policy adaptation support, the supervisor would have to define a new policy to grant the student access to resources after

978-1-5386-2070-0/18/$31.00 ©2018 IEEE

the scheduled time of when the meeting was to end had past. However, this solution complicates policy management as policy administrator (i.e., the supervisor or the university) might not be the policy owner or the system might not be able to specify new policies promptly. This simple scenario demonstrates how dynamic context changes affect user privileges and complicates policy management in IoT. Therefore, dynamic access control policies that handle such context changing situations are inevitably a core requirement for IoT.

In this work, we focus our research on automatic policy specification as a core feature that any dynamic access control system should support. It improves the overall flexibility in managing access control policies and adaptability in dynamic IoT environments.

II. Background and Related Work

Access control (AC) is the process of enforcing the system security requirements on protected services and resources [6]. AC determines whether an entity has sufficient privileges to access system resources and what is or is not allowed for the entity to perform. The level of authorization that an entity can be assigned is determined by evaluating its associated properties against the rules. For example, entity associated properties would include group membership roles, credit payment for content, proximity, access history and privacy preferences.

Three primary abstractions under AC systems need to be defined: policies, models, and mechanisms [6]. While AC policies determine the high-level security rules according to which AC must be regulated, AC models provide a formal description of the AC security policy and the procedures. AC mechanisms provide the low-level functions that implement and enforce the security rules defined in the AC policies and formalized by the AC model. AC core elements represent the input upon which functions of an AC mechanism enforces the access rules defined in the AC policy. These elements include *Subject* which is the person, device, or a process that actively causes information to flow between system objects or that changes the system state, *Object* which is a passive entity that contains or receives information (e.g., thermostat, door lock, printer), *Operation* which refers to a certain action (e.g., read and write) invoked by a subject and applied to an object.

In RBAC, access to a resource is determined based on the relationship between users (subjects) and the organization or the administrator (policy creator) who controls access to resources (objects). RBAC simplifies policy management by grouping users with similar access needs into *roles*, which typically reflects the corporate structure of an organization. RBAC is easy to manage as users can be assigned to one or several roles according to their duties within the organization, which allows users to have multiple levels of access to the same resource. In addition, RBAC enables for many-to-one relationships between users and roles. Therefore, RBAC simplifies policy management by allowing for a set of access permissions to be set once for one role that is (the role) assigned to multiple users who have same access needs.However, IoT has different types of users with different access needs; this introduces many administrative and policy enforcement challenges in RBAC. For example, grouping users into roles would make it difficult to define granular access control for individual users. This results in creating additional roles to exclude specific users who fall into a particular group but do not necessarily need to have the same permissions granted as to other members of the group. Thus, RBAC is an inefficient approach in dynamic IoT environments.

ABAC controls access based on three considerations: a set of arbitrary attributes associated with users and resources, the environmental conditions that are relevant to the ongoing access request [3][7], and a set of policies that are specified based on these attributes and conditions. ABAC focuses on attributes instead of identity or roles and controls access by evaluating rules against a wide range of attributes. Controlling access based on attributes simplifies policy management as there is no need to associate access permissions directly with users, roles or groups. This flexibility enables the creation of access rules without specifying individual relationships between each subject and object. In addition, ABAC access policy is scalable in the sense that subjects need not be known in advance to the system or to the object to which access is requested. Therefore, subjects can join and leave the system without the need to modify the underlying rules that associate to them. Furthermore, ABAC can make access decisions at runtime, where changes in attributes will immediately change access decisions between subsequent access requests, making ABAC a preferred model for dynamic environments.

A common issue associated with traditional access control policies is that they describe core access control elements (i.e., subject, operation, object) and their matching conditions holistically and statically. Holistic in the sense that all attributes and conditions that govern the association of the three elements are tightly coupled in the policy specification process and static in that access policies are specified at the setup time and do not change. When an access request is received, the access decision is made by verifying the conditions of the access request against a set of predefined static access control policies. If all attributes of the access request satisfy the conditions of a certain access control rule(s), access is permitted or denied based on a definition of the applicable rule(s). Figure 1 shows the decision making process in static access control with the three core elements tightly coupled.

The tight coupling describes parallel relationships between the core access control elements (i.e., subject-operation-object relationships). It limits the flexibility in policy expression and adaptation to dynamic access scenarios. In addition, modification in access policy would result in significant policy maintenance overhead (e.g., policy conflict resolution).

In an attempt to overcome the rigidity in the policy specification process, Han et al. [8] propose to adopt a hierarchical description and matching method in their policy specification process. They use attributes to describe core elements by predicates as basic facts, and layered matching rules to describe relationships between each two of the core elements (i.e., subject-operation, operation-object, and object-subject rela-

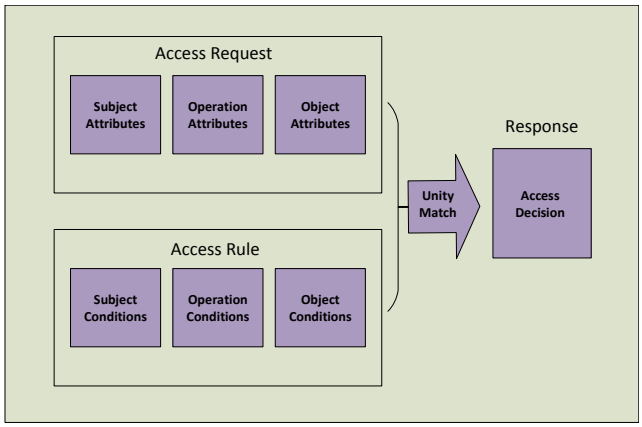

Fig. 1: Access control decision making based on static policy specification.

tionships) as hierarchical facts. Then, their proposed algorithm dynamically calculates the final relationships (i.e., subject-operation-object) and stores them into a policy repository. Finally, an access decision is made by analyzing and checking the conditions of the access request against the set of rules in the policy repository. If there exists a rule(s) that matches the ongoing request, access is permitted. Otherwise, access is denied by default. However, their approach incurs significant runtime overhead as well as offline maintenance when rules or access conditions change. This solution is not scalable and will not be feasible in a dynamic and time-sensitive IoT deployment.

III. Automatic Access Policy Framework Description

In this section, we describe the proposed automatic policy specification framework. Our framework is built around the concept of context which defines the set of environmental conditions under which protected IoT resource can be accessed, while access control policies define a set of allowed operations on that resource for each context. Once these contexts are defined by the resource administrator, they act as a *guard context* that regulates access to the protected resource. We also define *operational context* as the set of conditions associated with the requesting entity, protected resource, and the surrounding environment at runtime. Thus, if the operational context matches the required guard context, the requesting entity can only perform operations that are allowed under such context on the protected resource.

In fact, a guard context is a set of predefined application-dependent values or thresholds represented as key-value pairs for example (location: 'lab room'), (time,'<9:00 am'), (temperature:'$>$ 37 $^\circ$ && $<$ 40 $^\circ$'), (humidity:'$>$50 %'). These key-value pairs describe the environmental conditions defined by the resource administrator under which a requesting entity of certain attributes (e.g., identity, role and access credits) can perform operation of certain attributes such as (type: 'read'), (granularity: 'per minute') on a protected entity of certain attributes such as (CPU utilization: '$<$70 %'), (energy level: '$>$80 %').

Similar to guard context, operational context is represented by key-value pairs. , However, values in operational context are real-time measurements that reflect the real-world conditions of the interacting IoT entities at the time of access. If these measurements satisfy the guard context conditions that are defined separately for each element, then access is permitted. Otherwise, access is by default denied.

In the following, we describe our framework components.

A. *Primitive Facts (PF): Attributes and Guard Context*

In this section, we describe the core access control elements (subject, operation, and object) using abstractions, in a key-value pair representation, which contain both the attributes that characterize elements and the guard context that determines the qualification context (or constraints) relevant to each element that controls access to resources. We build these basic abstractions and represent them in predicates as follows:

- Element Abstraction:
  element(X) is a descriptor represented by a tuple of the form {`type:value, key1: value1, key2: value2, ...`}. The tuple contains a key *type* whose value is {`subject|operation|object`} to indicate this tuple is pertaining to the which of the access core elements. A descriptor is likely to contain a *time* and *location* keys that identify a certain time frame and location that control access to resources. For example, a subject must be in location x and time t to perform an operation of a certain object. The element descriptor is a unified fact defined by the object administrator on attributes of the access elements and their associated context. For example, the following basic abstraction represents a factual tuple for a subject x:
  element x= {`type:"subject",name:"Any", study-level: "undergraduate",advisee: "True",location:"IoTResearchLab",time: "6:00-16:00",coexistence: "True"`}. This descriptor contains a type key, three subject attributes that basically identify the subject, and three access constraints that determine the guard context required to gain access to a protected resource. In this example, the key `coexistence` refers to collocation of both the requesting subject and object administrator.

- Request Abstraction:
  The ideal request abstraction consists of a request template that defines the request elements and operational context c. The request descriptor contains patterns of the form p= {`type:value, key1: value1, key2: value2, ...`}. For example, the following tuple represents a print service request (pertaining only to operation): {`type:"operation",name:"print",loca-`

`tion:"6th floor",time:"8:00-14:00"}`. A request descriptor matches a resource if there is an operation in the fact base (i.e., set of facts defined by administrator) that satisfies every pattern in the request template.

It is worth noting that for guard context, individual constraints can be set on simple context information (e.g., time, location, number of users, temperature readings, etc.) or it can be set on complex context information inferred by semantic rules. These are application-dependent contexts, such as personal relationship (e.g., friend of, family member, primary physician), spatial relationship (e.g., coexistence of two or more subjects, objects or both) or situational context (e.g., threat level and emergency cases).

B. *Automatic Access Policy Specification*

Figure 2 depicts the process of our proposed dynamic access policy specification and access decision making. Our access policy specification algorithm takes the primitive facts and the access request as inputs and automatically produces the access control decision as an output

The algorithm shown in Listing 1 extracts the attributes and operational context information associated with the access request at runtime and uses the primitive facts defined by the policy administrator, to dynamically compute access policies in response to the access request at hand.

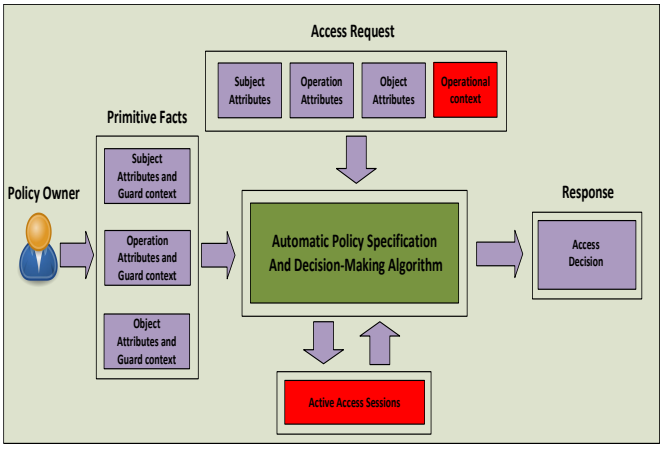

Fig. 2: Access control decision-making based on dynamic policy specification.

The matching process in Listing 1 performs two steps:

1) **Access request analysis.** In our work, an access request *(AR)* can be interpreted as a query of three parameters: subject *x*, operation *y*, object *z* that asks the access engine whether subject *x* is permitted to perform operation *y* on object *z* under the operational context associated with the *AR*. Object *z* is not always necessary in the request. For example, a subject *x* may request a print service (i.e., operation *y*), but does specify which object of the system may provide this service. In such a case, the system must be able to find an object that accepts the requested operation and is either accessible to subject *x* or allows open access.
When a new *AR* is received, our algorithm, `lines 2-9`, extracts all attributes of the requesting subject, requested operation, and target object along with the operational context associated with each element of the access request. Then, it generates four patterns as follow:
$$\textbf{\textit{p1}}\{x\},\ \textbf{\textit{p2}}\{y\},\ \textbf{\textit{p3}}\{z\},\ \textit{and}\ \textbf{\textit{p4}}\{c\}.$$
2) **Access decision making.** To evaluate the access request, our algorithm, `lines 11-17`, searches the *PF* for all combinations of the access control elements that exactly match the request patterns. Otherwise, access is denied by default.

Listing 1: Automatic access control policy specification and Decision-Making

```
   input : Primitive Facts PF, Access Request AR
   output: Access Decision: Permit/Deny
1  begin
2      for AR do
3          Extract all attributes and operational context
             values of the subject, x = {type : "subject",
4          key1 : value1, key2 : value2 ..... }
5          Extract all attributes and operational context
             values of the operation, y =
             {type : "operation",
6          key1 : value1, key2 : value2 ..... }
7          Extract all attributes and operational context
             values of the object (if exist), z =
             {type : "object",
8          key1 : value1, key2 : value2 ..... }
9      end
10     Generate PF-query{x},PF-query{y}, PF-query{z}.
11     Result := Deny
12     for all element ∈ PF do
13         if ∃ element(x) and element(y) and element(z) in
             PF: then
14             Result:= Permit
15             Break
16         end
17     end
18 end
```

IV. Use Case: Access to University Resources

Bob is a graduate student who has a meeting with his supervisor Alice at the office. According to Alice, all students under her supervision may access resources in the office and lab rooms only from within these rooms and during specific time frames. Examples of the resources that Alice is sharing: lab door lock (DL), air conditioner (AC) located in her office and printers (P) located in both rooms. As a resource administrator, Alice defines the attributes and guard

TABLE I: Student attributes and guard context.

Attribute/Guard-Context	Range
Type	$subject$
student-name	Any
study-level	$\{graduate, undergraduate\}$
advisee	$\{True, False\}$
Location	$\{Office, Lab\}$
time	24 hours
coexistence	$\{True, False\}$

TABLE II: Operation attributes and guard context.

Attribute/Guard-Context	Range
Type	$operation$
operation-name	$\{print, controlDL, controlAC\}$
study-level	$\{graduate, undergraduate\}$
location	$\{Office\ , Lab\}$
time	24 hours

context for the three types of resources she administers based on the following regulations and assumptions:

- A subject is described by four attributes and three contextual conditions as in Table I.
- An operation is described by three attributes and two contextual conditions as in Table II.
- An object is described by four attributes and two contextual conditions as in Table III.

Assume the following are the facts Alice sets to control access to her devices:

R1: Graduate students may have local access to the lab room and printers at any time.

R2: Undergraduate students may only access the lab and its printers from 8:00 to 16:00.

R3: Graduate advisees can control office AC and printer only when Alice coexists with them in the office.

A. *Primitive fact generation*

According to the resource profiles in Tables I to III and Alice's access regulations R1-R3, the system can generate a set of primitive facts as shown in Tables IV. These facts can be generated through a user-friendly interface from the fact description and presented to Alice for final approval.

B. *How it works*

While waiting for his supervisor, Bob decides to enter the lab and print the meeting agenda at approximately 12:00 PM. In order to get access to the lab resources, Bob needs to issue two access requests: the first request is to enter the lab room and second is to access the printer. When Bob submits the request, our algorithm extracts all attributes and operational context associated with this access request and generates the following queries to the access policy (i.e., PF):

TABLE III: Object attributes and guard context.

Attribute/Guard-Context	Range
type	$object$
object-name	$\{DL, P, AC\}$
study-level	$\{graduate, undergraduate\}$
operation-name	$\{print, controlDL, controlAC\}$
location	$\{\ Office\ , Lab\}$
time	24 hours

TABLE IV: Primitive facts.

PF_1	{ type: "subject", study-level: "graduate", location: "lab"}
PF_2	{type:"subject", study-level:"undergraduate", location: "lab", time: "08:14:00}.
PF_3	{type: "operation", operation-name: "controlDL", study-level: "graduate", location: "lab"}
PF_4	{type: "operation", operation-name: "print", study-level: "graduate", location: "lab"}
PF_5	{type: "operation", operation-name: "controlDL", study-level: "undergraduate", location: "lab", time:"08:00-16:00"}
PF_6	{type: "operation", operation-name: "print", study-level: "undergraduate", location: "lab", time: "08:00-16:00"}
PF_7	{type: "object", object-name: "DL", study-level: "graduate", operation-name: "controlDL", location: "lab"}
PF_8	{type: "object", object-name: "P", study-level: "graduate", operation-name: "print", location: "lab"}
PF_9	{type: "object",object-name:"DL",study-level: "undergraduate",operation-name:"controlDL", location: "lab",time:"08:00-16:00"}
PF_{10}	{type: "object", object-name: "P", study-level: "undergraduate",operation-name: "print", location: "lab",time:"08:00-16:00"}
PF_{11}	{type: "subject", study-level: "graduate", advisee: "true", location: "office", coexistence: "true"}.
PF_{12}	{type:"operation",operation-name:"controlAC", study-level: "graduate",location: "office"}.
PF_{13}	{type:"operation",operation-name: "print", study-level:"graduate", location: "office"}.
PF_{14}	{type: "object",object-name: "DL", study-level: "graduate",operation-name: "controlDL", location: "office"}.
PF_{16}	{type: "object", object-name:"P",study-level: "graduate",operation-name:"print", location: "office"}.
PF_{17}	{type: "object",object-name: "AC", study-level: "graduate", operation-name: "controlAC", location: "office"}.

p1={type: "subject", student-name: "Bob", study-level: "graduate", advisee: "True"}.

p2={type: "operation", "operation-name:

"controlDL", study-level: "graduate"}.

p3={type: "object", object-name: "DL", study-level: "graduate", operation-name: "controlDL"}.

p4={type: "context", location: "lab", time: "12:00"}.

Based on key-value pair matching, the request patterns collectively match the primitive facts PF_1,PF_3 and PF_7. Therefore, access to the lab room is granted. Similarly, in addition to the previous $p1$ and $p4$ the following patterns are extracted from a print access request:

p5={type: "operation","operation-name: "print ", study-level:"graduate"}.

p6={type: "object", object-name:"P", study-level: "graduate", operation-name: "print"}.

These patterns $p1, p5, p6$ and $p4$ collectively match PF_1,PF_4 and PF_8. Therefore, access to the printer in the lab room is granted.

Now, suppose that the context changes when Alice arrives and asks Bob to meet with her in her office, our context manager will capture changes in Bob's context (e.g., location: "office" and coexistence: "true") and inform the access engine to recheck the validly of ongoing requests. Bob's access privileges might be affected or even current access rights could be revoked. In our scenario, Bob now may access the printer in Alice's office. For any upcoming print requests issued by Bob, the system will notify Bob with the list of available services ranked by proximity and let Bob decide which service to choose. Should Bob issue another print request, the algorithm extracts the new context and generates the following context pattern, in addition to the previous ones, and send them all to the access engine for evaluation:

p7={type: "context",location: "office", time: "12:00", coexistence: "True"}.

This request ($p1, p5, p6$ and $p7$) matches to PF_{11}, PF_{13} and PF_{16} and a print access to Bob on Alice's office printer is granted.

It is worth mentioning that our framework allows the policy administrator to define different primitive facts for the same object such that the same subject may have multiple access levels on this object. In addition, for those users with multiple access levels, our algorithm always grants the least access privileges a subject may require to accomplish certain tasks. Thus preventing unnecessary or unintended accesses that may impair the privacy of the object administrator. For example, assume in our use case scenario that the printer in Alice's office has two printing modes {BW, color} where graduate students may only print color documents in the time frame 14:00 - 16:00. In this case, Alice needs to define a new operation for color printing or update the current print operation by adding supported modes as follows:

{type: "object", object-name: "P", operation-name: "print", mode: { "BW, color"}, study-level: "graduate", location: "office", time: "14:00-16:00"}
{type: "operation", operation-name: "print", default: "BW", mode: "color", study-level: "graduate", location: "office", time: "14:00-16:00"}

If Bob issues an access request during the above mentioned time frame to access the printer in Alice's office without specifying the printing mode, the matching algorithm will use the default key; otherwise, the requested mode will be evaluated. If the request matches multiple operations due to relaxed constraints, the algorithm will always grant the least privileges. For example, if printing mode is not specified, the print request will always grant the BW printing mode.

V. Evaluation of Access Policy Management

To evaluate the proposed approach, we compare our automatic policy specification method with traditional approach such as RBAC. We perform the comparison from the policy management perspective including adding a new user, adding new access rule and policy conflict resolution.

A. Adding new user

Let's assume Jane is a new graduate student who has just joined Alice's research team. To add Jane as a new user in RBAC, the system administrator needs perform the following steps:

1) Analyze Jane's access needs based on her attributes (e.g., graduate student, under the supervision of Alice).
2) If there exists an equivalent role, assign Jane to the appropriate organizational role (e.g., graduate student) that satisfies her access needs. Otherwise, create a new role.

When Jane issues an access request, the request is matched to a set of manually preconfigured and static access rules that are stored in an access policy database, where user-role and role-permission assignments are tightly coupled in the specification of each rule. Then, the access request is either granted or denied based on the implementation of applicable access rule(s).

Our approach makes it easier and scalable to add Jane as new user. The policy administrator does not need to add/define Jane's attributes to the basic facts maintained by the system, unless she has unique attributes that has not been defined already (e.g., Jane is a postdoc or research assistant). Further, our algorithm does not maintain permanent access policies, but rather it generates the applicable policies on the fly at runtime and makes access decisions based on the primitive facts. Thus, Jane's request will be evaluated using submitted credentials against primitive facts.

B. Adding new rule

In traditional access control systems, access rules may be specified by different authorities at different times for several purposes. Therefore, these rules may conflict with each other, allowing unintentional access to sensitive information. A straightforward definition of rules conflict is provided by Jajodia et al. [9], which states that a conflict occurs when contradicting access rights are granted to an individual entity. Detecting rule conflict is challenging. It involves identification of the conflicting rules and detecting the type of conflict among them at runtime. In addition, it requires dynamic verification to ensure conformance with security requirements specified by access policies.

Typically, access control systems adopt various conflict detection methods [10, 11] and conflict resolution strategies (e.g., first applicable rule, if one applicable rule, permit dominate, deny dominate) [12] to resolve conflicts in access policies or rules. In particular, Rule-Based RBAC with negative authorization [13] was proposed to overcome some limitations in RBAC including conflict resolution policies. Our approach does not store access policies in their final representation permanently, but rather the context-rich primitive facts directly related to access control elements. Access decisions are made on the fly solely on primitive facts. Thus, the system guarantees a conflict-free access policy. Every time a context changes, the system triggers a re-evaluation of ongoing granted access; context changes may mandate limitations on or even revocation of granted access. For example, to prevent graduate students from accessing printers in the lab room, we only need to update/delete primitive fact(s) that have the following three pairs: {`study-level:"graduate ", object-name:"P", location:"lab"`}. Therefore, the matching algorithm will not result in positive access decisions that require the existence of any of the deleted primitive facts. Hence, all subsequent access requests from graduate student to printers in the lab will be denied. To make it easier to enable back these permissions, we can add a key-pair attribute to each primitive fact that shows its status such that: `status:{"active", "disabled"}`, where `active` means it is in effect and `disable` means it's currently suspended. The system will then only match against active facts and ignore those that are suspended.

VI. Conclusion

In IoT environments, users and resources are expected to join and leave the system quite often. This may result in frequent changes in the context under which these entities may interact. This raises the essential need for dynamic access policy management to protect resources from unauthorized access without too much upfront overhead as well as to maintain high system flexibility and scalability. In this work, we propose an automatic access policy specification framework based on primitive facts that describe the core access control elements in predicates. Primitive facts are represented by an extendable tuple of key-value pairs that include both attributes that characterize the entity and context under which access is granted. Our algorithm generates access control policies and makes access decisions on the fly at the access request time. It improves the adaptability of the access policy in highly dynamic computing environments such as IoT.

Acknowledgment

This research is supported by a grant from the Natural Sciences and Engineering Research Council of Canada (NSERC) under grant number: STPGP 479248.

References

[1] Bartje Janina. The top 10 IoT application areas – based on real IoT projects. 2016.

[2] Ravi S Sandhu, Edward J Coyne, Hal L Feinstein, and Charles E Youman. Role-based access control models. *Computer*, 29(2):38–47, 1996.

[3] V. C. Hu, D. R. Kuhn, and D. F. Ferraiolo. Attribute-based access control. *Computer*, 48(2):85–88, 2015.

[4] Ji-Bo Deng and Fan Hong. Task-based access control model [j]. *Journal of Software*, 1:011, 2003.

[5] Anind K Dey. Understanding and using context. *Personal and ubiquitous computing*, 5(1):4–7, 2001.

[6] Vincent C Hu, David Ferraiolo, and D Richard Kuhn. *Assessment of access control systems*. US Department of Commerce, National Institute of Standards and Technology, 2006.

[7] Vincent C Hu, David Ferraiolo, Rick Kuhn, Arthur R Friedman, Alan J Lang, Margaret M Cogdell, Adam Schnitzer, Kenneth Sandlin, Robert Miller, Karen Scarfone, et al. Guide to attribute based access control (abac) definition and considerations (draft). *NIST special publication*, 800(162), 2013.

[8] Dao-jun Han, Ling Gong, and Fen Qin. A dynamic access control policy based on hierarchical description. In *Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC), 2016 International Conference on*, pages 76–80, 2016.

[9] Sushil Jajodia, Pierangela Samarati, Maria Luisa Sapino, and VS Subrahmanian. Flexible support for multiple access control policies. *ACM Transactions on Database Systems (TODS)*, 26(2):214–260, 2001.

[10] John C Strassner and Gregory W Cox. Performing policy conflict detection and resolution using semantic analysis, December 4 2012. US Patent 8,327,414.

[11] Gregory W Cox, David L Raymer, and John C Strassner. Efficient policy conflict detection, July 9 2013. US Patent 8,484,693.

[12] Anne Anderson. Extensible access control markup language (xacml). *Technology Report*, 2003.

[13] Mohammad A Al-Kahtani and Ravi Sandhu. Rule-based rbac with negative authorization. In *Computer Security Applications Conference, 2004. 20th Annual*, pages 405–415. IEEE, 2004.