An Attribute-Based Protection Model for JSON Documents

Prosunjit Biswas, Ravi Sandhu, and Ram Krishnan

Institute for Cyber Security
University of Texas at San Antonio
prosun.csedu@gmail.com, {ravi.sandhu, ram.krishnan}@utsa.edu

Abstract. There has been considerable research in specifying authorization policies for XML documents. Most of these approaches consider only *hierarchical structure* of underlying data. They define authorization policies by directly identifying XML nodes in the policies. These approaches work well for hierarchical structure but are not suitable for other required characteristics we identify in this paper as *semantical association* and *scatteredness*.
This paper presents an attribute based protection model for JSON documents. We assign *security-label* attribute values to JSON elements and specify authorization policies using these values. By using security-label attribute, we leverage semantical association and scatteredness properties. Our protection mechanism defines two types of policies called authorization and labeling policies. We present an operational model to specify authorization policies and different models for defining labeling policies. Finally, we demonstrate a proof-of-concept for the proposed models in the Swift service of OpenStack IaaS cloud.

1 Introduction

JavaScript Object Notation (JSON) is a human and machine readable representation for text data. It is widely used because of its simple and concise structure. For example, Twitter uses JSON as the only supported format for exchange of data starting from API v1.1 [5] and YouTube recommends uses of JSON for speed from its latest API [6]. JSON is being adapted increasingly in large and scalable document databases such as MongoDB [4], Apache Casandra [2] and CouchDB [3]. Besides these, JSON is also widely used in lightweight data storages for example in configuration files, online catalogs or applications with embedded-storage.

In spite of high adoption from industries, JSON has received little attention from academic researchers. To the best of our knowledge, there is no formal work published on the protection of JSON documents.

On the other hand, considerable work has been done for protection of XML documents. Although syntactically JSON and XML formats are

different, semantically both of them form a rooted tree hierarchical structure. In fact, JSON data can equivalently be represented in XML form and vice versa. This brings an obvious question - whether we can utilize authorization models used for XML documents for protection of JSON data.

Before we answer the preceding question, we look into some of the salient characteristics of data represented in JSON (or XML) format, given below.

- **Hierarchical relationship.** Data often exhibits hierarchical relationship. For example, a residential address consists of pieces like house number, street name, district/town and state name organized into an strictly hierarchical structure.
- **Semantical association.** Different pieces of data are often related semantically and may need same level of protection. For example, phone number, email address, Skype name may all represent contact information and require same level of protection.
- **Scatteredness.** Related information can be scattered around a document. For example, different pieces of contact information might be located in different places in a document. Some pieces of data can even be repeated in more than one place in the same document or across documents.

Interestingly, most of XML authorization models [8–10, 17] consider *structural hierarchy* only. These models have an implicit assumption that information has been organized in the intended hierarchical form. These models attach authorization policies directly on nodes in the XML tree and propagate them using the hierarchical structure. For example, Damiani et al. [15] specify authorization policy as a tuple $\langle subject, object, action, sign, type \rangle$ where subject is specified as user, user group, IP address or semantic name; object is specified with XPath expression; example of actions are read or write; signs are positive and negative; and example of types are local, global and DTD which determines the level of propagation. In this model, if similar data items requiring same level of protection are placed in structurally unrelated nodes, it is required to attach same authorization policy to all these nodes. This results in duplication of authorization policies which is caused by lack of recognition of semantical association and scatteredness properties.

Duplication incurs significant overhead in maintenance of authorization policies. For instance, if requirements for storing or publishing contact information (e.g. email, phone, fax) change, it is required to update

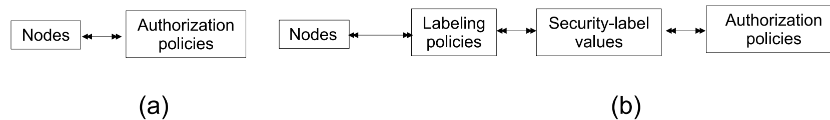

Fig. 1. (a) Existing XML models (b) the proposed model

policies for all different pieces of data that represent contact information. Organizations often collect different types of data including personal identifiable information of employees and customers. So, they are compliant to different internal and external parties including government and standard bodies. This increases the likelihood that authorization requirements change frequently over time.

While most XML authorization models directly identify nodes in their authorization policies, our proposed model adds a level of abstraction by using *security-label* attribute values. The proposed model specifies two types of policies called *authorization policies* and *labeling policies*. Authorization policies are specified using *security-label* attribute values. These values are assigned to JSON data using labeling policies. A conceptual overview of existing XML authorization models and our proposed model is shown schematically in Figure 1(a) and 1(b) respectively. By using security-label attribute values to connect nodes and policies, we can assign semantically related or scattered data same attribute values. This eliminates the need to specify duplicated policies.

The proposed model additionally offers flexibility in specification and maintenance of authorization and labeling policies. These two types of policies can now be managed separately and independently. For instance, given *security-label* attribute values, higher level, organization-wide policy makers can specify authorization policies using these values without knowing details of JSON structure. On the other hand, local administrators knowledgeable about details of specific JSON documents can specify labeling policies.

We believe, the presented model can easily be generalized for data represented in trees and be instantiated for other representations, for example, YAML [1]. For simplicity, we only focus on JSON here.

The contributions of this paper are as follows. We have identified underlying characteristics of data represented in XML/JSON form. While, existing XML authorization models address only *structural hierarchy*, we additionally focus on *semantical association* and *scatterredness* properties. We have designed an attribute-based protection mechanism for JSON documents including an operational and two different labeling models.

We have demonstrated a proof-of-concept for the proposed models in the Swift service of OpenStack IaaS cloud platform.

The rest of the paper is organized as follows. In Section 2, we discuss underlying concepts of JSON documents and existing works relating to the protection of these documents. Section 3 presents the operational model. The labeling models are described in Section 4. Section 5 discusses the proof-of-concept implementation of our proposed models. Finally, we conclude the paper in Section 6.

2 Background and related work

In this section, we briefly review JSON and discuss related work.

2.1 JSON (JavaScript Object Notation)

JSON or JavaScript Object Notation is a format for representing textual data in a structured way. In JSON, data is represented in one of two forms — as an *object* or an *array* of values. A JSON object is defined as a collection of *key,value* pairs where a *key* is simply a string representing a name and a *value* is one of the following primitive types—string, number, boolean, null or another object or an array. The definition of a JSON object is recursive in that an object may contain other objects. An array is defined as a set of an ordered collection of values. JSON data manifests following characteristics.

- JSON data forms a rooted tree hierarchical structure.
- In the tree, leaf nodes represent values and a non-leaf nodes represent keys.
- A node in the tree, can be uniquely identified by a unique path.

Figure 2(a) shows the content of a JSON document where strings representing values have been replaced by *"..."* for ease of presentation. Figure 2(b) shows the corresponding tree representation. Any node in the tree can be uniquely represented by JSONPath [18] which is a standard representation of paths for JSON documents.

2.2 Related work

There is limited academic research published on security of JSON data. To the best of our knowledge, we are the first to propose a protection model for it.

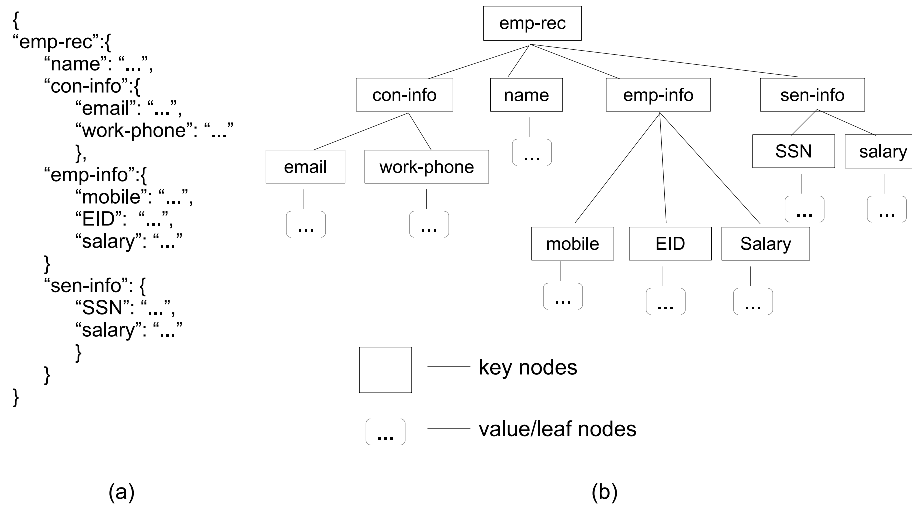

Fig. 2. Example of (a) JSON data (b) corresponding JSON tree

On the other hand, XML security has long been investigated by many researchers. A fundamental line of work in this area is about specifying authorization policies for the protection of XML documents [8–10, 17]. All of these models attach authorization policies directly on nodes in the XML tree. Most of these models use XPath [14] to specify a node in the tree. For example, Damiani et al. [15] specify authorization policies as a tuple of ⟨*subject, object, action, sign, type* ⟩ where an *object* is identified by an URI (Uniform Resource Identifier) along with a XPath expression.

Another direction of work is about effective enforcement of authorization mechanisms for secure and efficient query evaluation. For example, in [16] the authors derive *security views* comprising exactly the set of accessible nodes for different user groups. Based on the security view, they provide a unique DTD view for each user group. Similar works in this direction include [19, 20] which use query preprocessing approaches. These models uses preprocessed finite automatas for authorization policies, document and Schema/DTD, and determine if a query is safe before running it. Unsafe queries can be rewritten.

The idea of associating labels with protected objects has been proposed before. For example, in *purpose based access control (PBAC)* [13], the authors associate *intended purposes* with data items and *access purpose* with users. If *access purpose* of a user is included in the *intended purposes* of the requested objects, the request is granted. Our approach is similar. While PBAC manages intended purposes using RBAC [22],

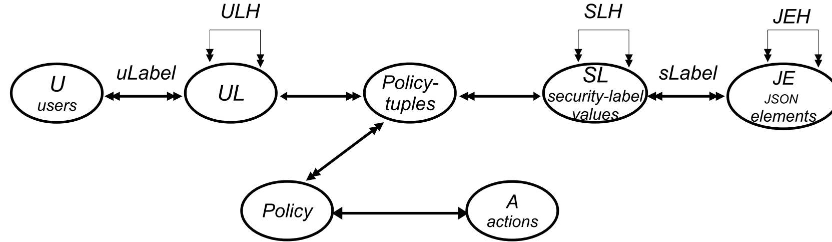

Fig. 3. The Attribute-based Operational Model (*AtOM*)

we use attributes with attribute-based access control (ABAC). Most significantly, PBAC does not specify how to annotate objects with *access purposes*, which we emphasize in this paper via labeling policies. Adam et al. [7], have applied *concepts* and *slots* on digital objects which work at a finer grained content level. They have also specified an access control model based on expressions using concepts and slots. This model also does not specify how to assign concepts and slots to objects.

The concept of attaching organized labels to users and objects and controlling access based on these labels is the underlying idea of Lattice Based Access Control (LBAC [21]), sometime also referred as Mandatory Access Control (MAC). The operational model, *AtOM*, presented in Section 3 resembles LBAC but it is fundamentally different from LBAC. *AtOM* is based on enumerated authorization policy ABAC model named EAP-ABAC ([12, 11]). EAP-ABAC is a general purpose ABAC model which supports larger set of attributes contrary to single label in LBAC and based on enumerated authorization policies. Correlation between EAP-ABAC and LBAC is presented in [12].

3 The operational model

This section presents the Attribute-based Operational Model (*AtOM*) for protection of JSON documents. *AtOM* adapts enumerated authorization policies from [12, 11].

Figure 3 presents components of *AtOM*. In the figure, the set of users is represented by U. Each user is assigned to one or more values of an attribute named *user-label* or *uLabel* in short. These values are selected from the set of all possible user-label values UL which are partially ordered. The partial order is represented by ULH. An example showing user-label values and hierarchy is presented in Figure 4(a). On the other hand, the set of JSON elements are specified as *JE*. JSON elements may subsume other JSON elements, and form a tree structured hierarchy. The

Table 1. Definition of *AtOM*

I. Sets and relations
- U, JE and A (set of users, JSON elements and actions resp.)
- JEH (hierarchy of JSON elements, represented by $\succeq_j$)
- UL and ULH (finite set of uLabel values and their partial order denoted as $\succeq_{ul}$ resp)
- SL and SLH (finite set of security-label values and their partial denoted as $\succeq_{sl}$ resp)
- *uLabel* and *sLabel* (attribute functions on users and JSON objects resp.) Formally, $uLabel : U \rightarrow 2^{UL}$; $sLabel : JO \rightarrow 2^{SL}$
II. Policy components
- $Policy\text{-}tuples = UL \times SL$
- $Policy_a \subseteq Policy\text{-}tuples$ for $a \in A$
- $Policy = \{Policy_a \mid a \in A\}$
III. Authorization function
- $can_access(u : U, a : A, o : JE) = (\exists(ul, sl) \in Policy_a)[ul \in uLabel(u) \wedge sl \in sLabel(o)]$
- $is_authorized(u : U, a : A, je_i : JE) = (can_access(u, a, je_j))[je_i \succeq_{sl} je_j]$

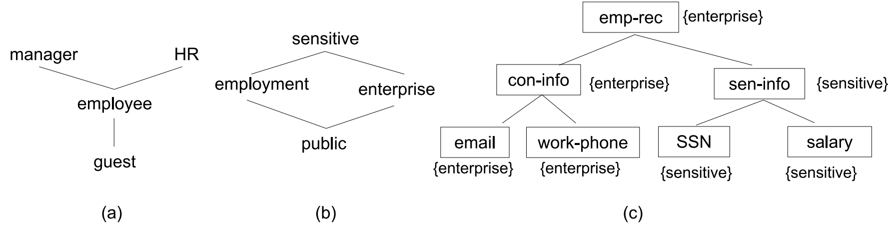

Fig. 4. (a) User-label values, (b) Security-label values and (c) Annotated JSON tree

hierarchy is represented by *JEH*. Each JSON element is assigned values of an attribute named *security-label* or *sLabel* in short. These values are selected from the set of security-label values SL which are also partially ordered. The partial order is represented by *SLH*. An example showing security-label values and hierarchy is presented in Figure 4(b). A JSON tree annotated with security-label values is given in Figure 4(c). These components and relationship among them are formally specified in Segment I of Table 1.

In Figure 3, the set of authorization policies is represented by *Policy*. There exists one authorization policy per action which is shown by the one-to-one relation between *Policy* and *A*. In Table 1, $Policy_{read}$ presents the authorization policy for action read. An authorization policy may contain one or more micro-policies and one micro-policy can be associated with more than one authorization policies. This is represented by the many-to-many relation between *Policy* and *Policy-tuples*. $Policy_{read}$, as mentioned above, contains four policy-tuples including *(manager, sensitive)*. The tuple *(manager, sensitive)* while contained in policy $Policy_{read}$ specifies that users who are manager can read objects that have been

Table 2. Example of an authorization policy and authorization requests

I. Enumerated authorization policies
$Policy_{read} \equiv$ { *(manager,sensitive), (HR,employment), (employee, enterprise), (guest, public)*}
II. Authorization requests
$is_authorized(Alice, read, emp\text{-}rec) = true$, assuming $uLabel(Alice) = \{manager\}$
$is_authorized(Bob, read, emp\text{-}rec) = false$, assuming $uLabel(Bob) = \{employee\}$
$is_authorized(Bob, read, con\text{-}info) = true$, assuming $uLabel(Bob) = \{employee\}$
$is_authorized(Charlie, read, sen\text{-}info) = false$, assuming $uLabel(Charlie) = \{HR\}$

assigned values sensitive. Formally, we represent a policy-tuple a pair of atomic values (ul, sl) where $ul \in UL$ and $sl \in SL$. The formal definition of policies and policy-tuples is given in Segment II of Table 1. We use the terms policy-tuples and micro-policies equivalently to represent sub-policies.

The authorization function $is_authorized()$ is specified in Section III of Table 1. We define the helper function $can_access(u, a, o)$ which specifies that the user u can access the object o for action a if there exists a policy-tuple in $Policy_a$ for that allows it. A user is authorized to perform an action on the requested JSON element if he can access the requested element and all its sub-elements. For example, let us assume, Alice as a manager wants to read *emp-rec* which has been assigned value *enterprise* as shown in Figure 4(c). The tuple *(manager, sensitive)* in $Policy_{read}$ specifies that Alice can read object labeled with sensitive or junior values. Thus, the request *is_authorized(Alice, read, emp_rec)* is evaluated true. On the other hand, assuming Bob as an employee, the request *is_authorized(Bob, read, emp-rec)* is evaluated false as an employee cannot read *sen-info* which is sub-element of *emp-rec*. Additional examples of authorization request is given in Segment II of Table 2.

4 Labeling policies

In this section, we discuss specification of labeling policies for the operational model given in Section 3. We broadly categorize the policies used in the operational model into specification of authorization policies and assignment of security-label values or labeling policies. Policy scope of the operational model is schematically shown in Figure 5. Here, we focus on the later type of policies.

We specify two different approaches to assign security-label values to elements in a JSON document, viz. content-based and path-based. These approaches are fundamentally different in how a JSON element is specified. While a path is described starting from the root node of the

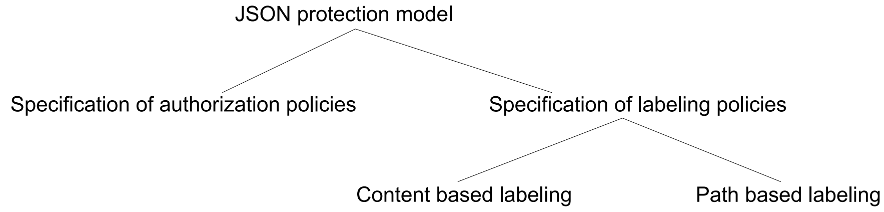

Fig. 5. Policy scope

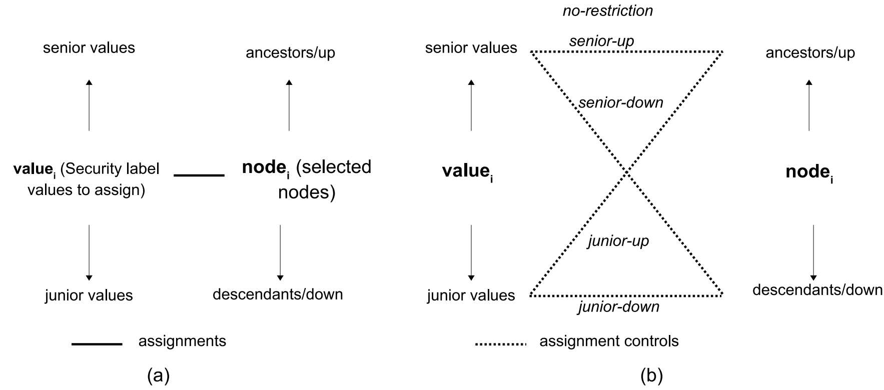

Fig. 6. (a) Assignment of security label values (b) assignment controls

tree, content is specified starting from the leaf nodes of the tree. These two contrasting approaches offer flexibility in assignments and propagation of security-label values.

4.1 Control on labeling policies

For specification of labeling policies, we define two types of restriction that control assignments and propagations of *security-label* values. In the first type, we restrict how security-label values are selected and assigned on tree nodes. We call this *assignment-control*. In the second type, we specify how assigned values are propagated along nodes in the tree. We call this *propagation-control*.

The motivation of *assignment-control* is to restrict arbitrary assignments of security-label values. This enables administrators to restrict future assignments after some assignments have been carried out. These controls are specified during the assignments. If any attempting assignment does not comply with *assignment-control*s of existing assignments, it will be discarded. We define five possible options for *assignment-control* as *no-restiction*, *senior-up*, *senior-down*, *junior-up* and *junior-down*. The

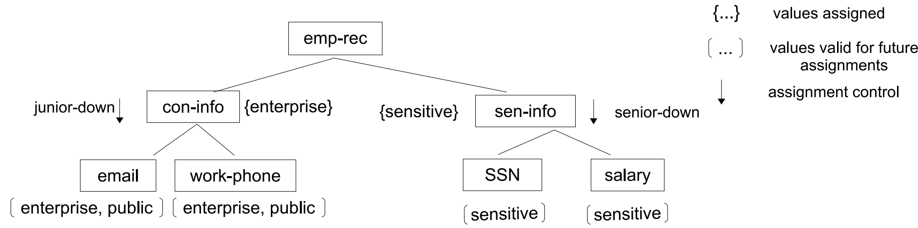

Fig. 7. Assignments with assignment controls

type *no-restriction* does not specify any restriction. If we assign a value $value_i$ in $node_i$, with *senior-up* restriction, all up/ancestors of $node_i$ must be assigned values senior to $value_i$ possibly including $value_i$. In type *senior-down* restriction, all down/descendants of $node_i$ must be assigned values senior to $value_i$ possibly including $value_i$. Similarly, the types *junior-up* and *junior-down*, specify that ancestors and descendants of $node_i$ must be assigned values junior to $value_i$, possibly including $value_i$. Figure 6 schematically illustrates *assignment-control*. In Figure 7, the node *con-info* is assigned a value *enterprise* with option *junior-down* which regulates that its descendant nodes namely {email, work-phone} must be assigned values *enterprise* or its juniors, in this case from the set {enterprise, public} (using security-label values given in Figure 4(b)). In the same figure, the node *sen-info* is assigned value *sensitive* with option *senior-down* which mandates that its descendant nodes namely {SSN, salary} must be assigned values from *sensitive* or its seniors in this case from the set {sensitive}.

Once we assign security-label values on an element in a JSON tree, the value can be propagated to other elements in the tree. We define following types for *propagation-control* as *no-prop*, *one-level up*, *one-level down*, *cascading up* and *cascading down.* Assigned values are not propagated in type *no-prop.* From a node, assigned values are propagated to parent and all its siblings in the type *one-level up.* Assigned values are propagated to all ancestor nodes in type *cascading up.* Similarly, from a selected item, assigned values are propagated to direct children in type *one-level down* and to all descendants in type *cascading down.*

4.2 Content-based labeling

This section shows how to assign security-label values by matching content and propagating the labels.

We adapt the concept of *query object* available in MongoDB [4] which matches content in a JSON document. Query objects discover content

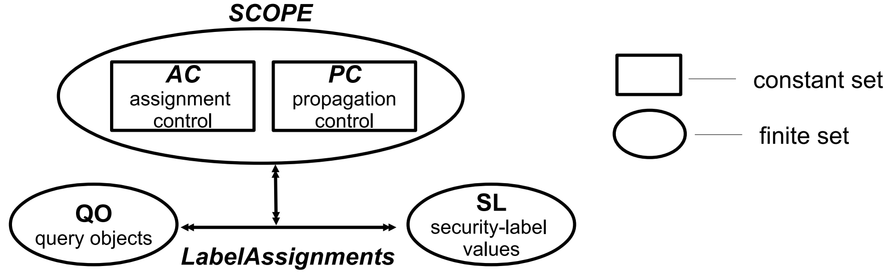

Fig. 8. Content-based labeling model

Table 3. Definition of content-based labeling

I. Basic sets and relations
- QO (set of query objects).
- AC (assignment control) $AC = \{$*no-restriction, senior-up, junior-up*$\}$.
- PC (propagation control) $PC = \{$*no-prop*, *one-level-up*, *cascade-up*$\}$.
- $SCOPE \subseteq AC \times PC$
- SL (set of security-label values).
II. Assignments of security-label values
- $LabelAssignments \subseteq QO \times SCOPE \times 2^{SL}$

starting from the *value nodes* of the JSON tree. It accepts regular expression to find *value nodes* or *key nodes* conveniently. MongoDB has built-in functions to express regular expressions and compare values matched by the regular expressions.

A model to assign security-label values based on query objects is given in Figure 8. In the figure, QO represents the set of all query objects and SL is the set of security-label values. The set AC represents *assignment-control* and PC represents *propagation-control* discussed earlier. AC and PC together define labeling scopes. A labeling scope determines how values are assigned and propagated in the tree. As content is matched from the value/leaf nodes of the tree, we consider assignment and propagation control only for the ancestors of the matching nodes.

The formal definition of the model is given in Table 3. Segment I of the table specify basic sets and relations. In Segment II, the relation *LabelAssignments* defines rules for assigning security-label values. An assignment rule is a triple of a query object to match content, a scope and a set of values to be assigned. Section I of Table 4 gives some examples of query objects and their interpretation in plain English. Segment II of Table 4, presents examples of assignment policies based on query objects.

Table 4. Examples of query objects and content-based labeling policies

I. Query objects
- ob1 = {"email": { $regex:"/.*@example.com/"} } (matches email addresses from domain example.com)
- ob2 = { $elemMatch: { $regex: "RE_EMAIL" } } (matches any key having value corresponding to the given regular expression)
- ob3 = {$elemMatch:{ $regex: "RE_SSN"}, $elemMatch: {"RE_CREDIT_CARD"}} (matches all objects containing both social security and credit card number)
II. LabelAssignments
- LabelAssignments= { (ob1, (no-prop, unrestricted), {enterprise}), (ob2, (no-prop, unrestricted), {enterprise}), (ob3, (no-prop, restricted), { sensitive} }

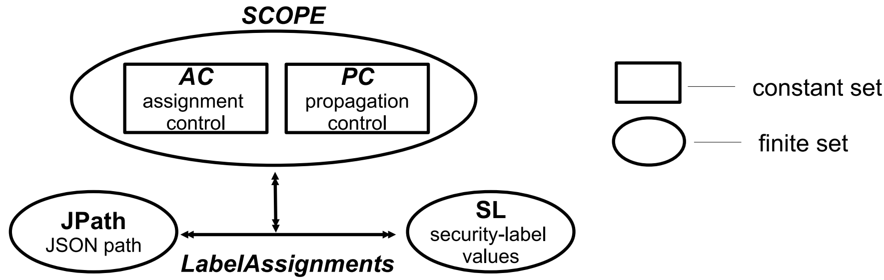

Fig. 9. Path-based labeling model

4.3 Path-based labeling

In this section, we show how we assign security-label values by matching paths in the JSON tree and propagate them along the tree.

We adapt *JSONPath* [18] to specify path-based labeling policies. This model is very similar to the content-based labeling model except we use JSONPath instead of query objects. While, query objects are matched starting from the leaf nodes, JSONPath specifies elements starting from the root node (or any node in case of relative path) and traverses towards leaf of the tree. As a result, this model apply assignment control and propagation control towards descendants of matching nodes. The components of the model and its formal definition are given in Figure 9 and Table 5 respectively. Examples of JSON paths and path based labeling policies are presented in Segment I and II of Table 6.

5 Implementation in OpenStack Swift

We have implemented our proposed operational model and path-based labeling scheme in OpenStack IaaS cloud platform using OpenStack Keystone as the authorization service provider and OpenStack Swift as the

Table 5. Definition of Path-based labeling

I. Basic sets and relations
- $JPath$ (set of JSONPaths).
- AC (assignment control) $AC = \{$*no-restriction, senior-down, junior-down*$\}$.
- PC (propagation control) $PC = \{$*no-prop*, *one-level-up*, *cascade-up*$\}$.
- $SCOPE \subseteq AC \times PC$, relation to assign and propagate values.
- SL (set of security-label values).
II. Assignments of security-label values
- $LabelAssignments \subseteq JPath \times SCOPE \times 2^{SL}$ (assign security-label values on JSON elements matched and propagate values based on defined scope)

Table 6. Examples of JSONPath and path-based labeling policies

I. JSONPaths
- path-to-email=$.emp-rec.con-info.email
- path-to-salary=$.emp-rec.sen-info.salary
II. LabelAssignments
- LabelAssignments= { (path-to-email, (no-prop, unrestricted), {enterprise}), (path-to-salary, (no-prop, unrestricted), {sensitive}) }

storage service provider. Our choice of OpenStack is motivated by its support for independent and inter-operable services and a well defined RESTful API set.

We have modified OpenStack Keystone and Swift services to accommodate required changes. A reference architecture of our testbed is given in Figure 10. Details of the implementation is shown in Figure 11. Required changes are presented as highlighted rectangles in Figure 11.

5.1 Changes in OpenStack Keystone

OpenStack Keystone uses roles and role-based policies to provide authorization decisions. In our implementation, we uses roles to hold user-label attribute values. A set of valid security-label values are also stored as part of the Keystone service.

Among two different types of policies - authorization and labeling policies, the former is managed in the Keystone service. We assume, a higher level administrators (possibly at the level of organization) adds, removes or updates these authorization policies. We add a policy table in Keystone database to store these enumerated authorization policies.

5.2 Changes in OpenStack Swift

In Swift side, we store *security-label* values assigned to JSON objects and path-based labeling policies applied to them. Security-label values

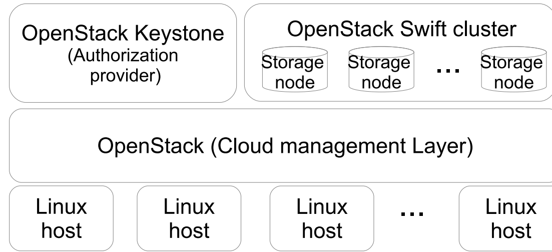

Fig. 10. Reference architecture of the implementation testbed

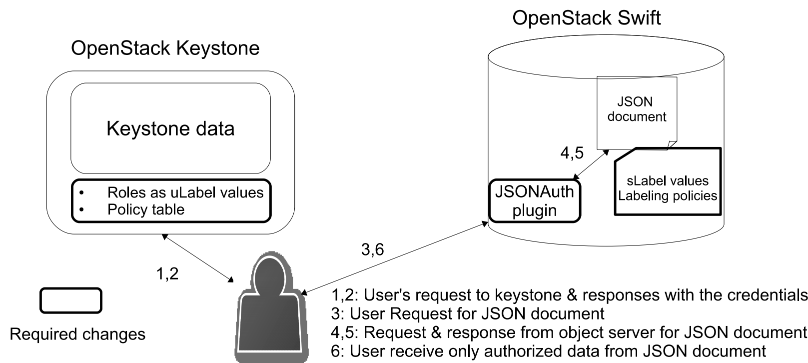

Fig. 11. Implementation in OpenStack IaaS cloud platform

and labeling policies are stored as metadata of the stored objects, JSON documents in this case. For simplicity, we assume object owner (Swift account holder in this case) can update security-label values or labeling policies for stored JSON document.

During evaluation, we intercept every requests to Swift (from the Swift-proxy server) and reroute a request to be passed through *JSONAuth plugin* if it is a request for a JSON document. In this case, the request additionally carries a requested path and authorization policies applicable to the user. JSONAuth plug-in retrieves the requested JSON document, apply path-based labeling policies to annotate the document and uses authorization policies to determine if the user is authorized for the requested content of the file.

5.3 Evaluation

An evaluation of our implementation is shown in Figure 12. The evaluation has been made against concurrent download requests to the Swift proxy server. The X-axis shows size of the JSON document requested for

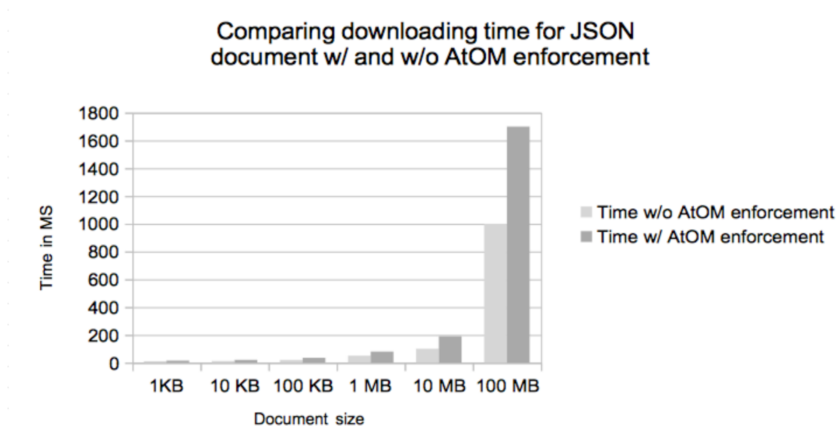

Fig. 12. Performance evaluation

download while the Y-axis shows the average download time for 10 concurrent request. Our evaluation shows a performance hit of nearly 60% over no authorization protection.

6 Conclusion

This paper presents an attribute based protection model for JSON documents. In the proposed model, JSON elements are annotated with *security-label* attribute values with *labeling policies*. We specify *authorization policies* using these attribute values. The advantage of the separation of labeling and authorization policies is that they can be specified and administered independently possibly by different level of administrators. In this regard, we have presented an operational model to specify authorization policies that evaluates access request. Further, we have specified two different models for assigning security-label attribute values on JSON elements based on content and paths. We have presented a proof-of-concept of the proposed models in OpenStack IaaS cloud platform.

Acknowledgement

This research is partially supported by NSF Grants CNS-1111925 and CNS-1423481.

References

1. The official YAML website. www.yaml.org. accessed 07/2016.

2. Apache Cassandra. http://cassandra.apache.org/. accessed 09/2015.
3. Apache CouchDB^{TM}. http://couchdb.apache.org/. accessed 09/2015.
4. MongoDB. http://www.mongodb.org/. accessed 09/2015.
5. Twitter API. https://dev.twitter.com/docs/api/1.1/overview. accessed 09/2015.
6. Youtube API. https://developers.google.com/youtube/v3/. accessed 09/2015.
7. Nabil R Adam, Vijayalakshmi Atluri, Elisa Bertino, and Elena Ferrari. A content-based authorization model for digital libraries. *IEEE KDE*, 14(2):296–315, 2002.
8. Elisa Bertino, Silvana Castano, Elena Ferrari, and Marco Mesiti. Controlled access and dissemination of XML documents. In *2nd ACM WIDM*, pages 22–27, 1999.
9. Elisa Bertino, Silvana Castano, Elena Ferrari, and Marco Mesiti. Specifying and enforcing access control policies for XML document sources. *World Wide Web*, 3(3):139–151, Springer, 2000.
10. Elisa Bertino and Elena Ferrari. Secure and selective dissemination of XML documents. *ACM TISSEC*, 5(3):290–331, 2002.
11. Prosunjit Biswas, Ravi Sandhu, and Ram Krishnan. A comparison of logical-formula and enumerated authorization policy ABAC models. In *DBSEC*. Springer, 2016.
12. Prosunjit Biswas, Ravi Sandhu, and Ram Krishnan. Label-based access control: An ABAC model with enumerated authorization policy. In *Proc. of the 2016 ACM Int. Workshop on Attribute Based Access Control*, pages 1–12, 2016.
13. Ji-Won Byun, Elisa Bertino, and Ninghui Li. Purpose based access control of complex data for privacy protection. In *10th ACM SACMAT*, 2005.
14. James Clark and Steve DeRose. XML path language (XPath) version 1.0, 1999.
15. Ernesto Damiani, Sabrina De Capitani di Vimercati, Stefano Paraboschi, and Pierangela Samarati. A fine-grained access control system for XML documents. *ACM TISSEC*, 5(2):169–202, 2002.
16. Wenfei Fan, Chee-Yong Chan, and Minos Garofalakis. Secure XML querying with security views. In *ACM SIGMOD/PODS*, pages 587–598, 2004.
17. Irini Fundulaki and Maarten Marx. Specifying access control policies for XML documents with XPath. In *9th ACM SACMAT*, pages 61–69, 2004.
18. Stefan Goessner. JSONPath Syntax. http://goessner.net/articles/JsonPath/. accessed 09/2015.
19. Bo Luo, Dongwon Lee, Wang-Chien Lee, and Peng Liu. Qfilter: fine-grained run-time XML access control via NFA-based query rewriting. In *ACM CIKM*, 2004.
20. Makoto Murata, Akihiko Tozawa, Michiharu Kudo, and Satoshi Hada. XML access control using static analysis. *ACM TISSEC*, 9(3):292–324, 2006.
21. Ravi S. Sandhu. Lattice-based access control models. *IEEE Computer*, 26(11):9–19, 1993.
22. Ravi S Sandhu, Edward J Coyne, Hal L Feinstein, and Charles E Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.