Attribute-based Fine Grained Access Control for Triple Stores

Ankur Padia, Tim Finin, and Anupam Joshi

University of Maryland, Baltimore County, Baltimore, MD 21250, USA
padiaankur@gmail.com, {finin, joshi}@umbc.edu

Abstract. The maturation of semantic web standards and associated web-based data representations like schema.org have made RDF a popular model for representing graph data and semi-structured knowledge. However, most existing SPARQL endpoint supports simple access control mechanism preventing its use for many applications. To protect the data stored in RDF stores, we describe a framework to support attribute-based fine grained access control and explore its feasibility. We implemented a prototype of the system and used it to carry out an initial analysis on the relation between access control policies, query execution time, and size of the RDF dataset.

Keywords: RDF triple store, attribute-based access control, information assurance, security

1 Introduction

The maturation of Semantic Web standards and associated web-based data representations like schema.org have made RDF a popular model for representing graph data and semi-structured knowledge. Triple stores are used to hold and query an RDF dataset and often expose a SPARQL endpoint service on the Web for public access. Most existing SPARQL endpoints support very simple access control mechanisms, if any at all, preventing their use for many applications where fine-grained privacy or data security is important.

Considerable research has been done to provide security to semantic data [1–6] - data represented using semantic web standard like RDF, RDFS and OWL. The work can be classified into four categories: *Rule* based, *Named Graph* or *View* based, *Description Logic (DL)* based and *triple pattern* based. There are also some standard frameworks such as Web Access Control [7] that can provide access control for semantic data. However, each of the approaches has shortcomings, especially when granularity of access control and the open environment of the web are considered. The enforcement of fine-grained policies on complex queries can require significant processing. An open environment allows users to submit ad hoc queries, preventing a system from pre-processing queries and storing the results for later use.

We present an approach to provide fine-grained attribute-based access control for triple stores to facilitate privacy and security. Our approach has the advantage of being a "pluggable" filter for generic SPARQL endpoints and thereby protecting a triple store that allows users to submit SPARQL queries.

The paper is organized as follows. After reviewing related literature in Section 2, we explain our framework to support attribute-based fine grained access control in Section 3 with an example to show representation of access policies in triple-based pattern and a mechanism to enforce access policies on user's query. In Section 4, we demonstrate the feasibility and present an initial analysis on the relation between access control policies, query execution time, and size of the RDF dataset. In Section 5 we summarize our work and briefly describe our plans for future work.

2 Literature review

The problem of enhancing the security and privacy of RDF data with access control policies has been studied for many years. Research works on access control for RDF data can be categorized based on the techniques employed [8] as (1) Named Graph or view, (2) Rule, (3) Description Logic and (4) Triple pattern. Each approach aims to provide security at a different level of granularity, either fine-grained or course-grained. In case of fine-grained access controls, policies are defined on a triple's subject, object or predicate, an RDF statement, or on a set of RDF statements based on some regular expression. On the other hand, policies for course-grained access controls are defined on a materialized named graph, or on a triple store.

Rule based approaches make use of rule languages like *SWRL*[1] or *n3*[2] to represent access control policies. Rule based policies can provide fine grained access control [2], as well as course grained access control [9, 10]. However, depending on granularity level, there can be high redundancy among the policies as the set of conditions used among the policies are same [3].

Approaches based on Named Graphs [11] combine multiple RDF graphs in a single document or repository and name each with an URI. Based on the access control policy, each user of the system is associated with corresponding set of named graphs [3]. The use of named graphs can simplify the process of providing access control, but the creation and materialization of the additional named graphs needed for access control requires considerable effort. Materializing the named graphs also comes with an expense of storage space and multiple simultaneous updates [12]. Moreover, named graphs can introduce redundancy for triple stores containing multiple cycles and multiple paths from a fixed source class to a fixed destination class.

In description logic (DL) [13] based policy representations, validity of the access control is determined using off-the-shelf reasoners. There are several advantages of DL based approach [4] over other approaches. First, policies can be

[1] `http://www.w3.org/Submission/SWRL/`
[2] `http://www.w3.org/TeamSubmission/n3/`

more expressive since they can rely on DL reasoning. Second, policies can be easily analyzed for consistency, comparison, verification and querying using off-the-shelf reasoners. Third, other security standards, like XACML[3], can be modeled in OWL-DL [14, 15]. However, since the policies are defined on T–Box entities and access control is determined using subsumption reasoning, the approach is typically limited to be all or none with respect to the instances present in the A–Box. To best of our knowledge, only [16] tried to provide fined grained access control by labeling axioms present in knowledge base. However, such an approach require a lattice structure, with respect to subsumption operator, among axioms present in the T–Box. Most of the time, such a requirement is difficult to meet, as their might not be any specific schema representing underlying RDF data. Moreover, data stored in a triple store might follow RDF semantic standards but might not adhere any DL profile.

Triple pattern based approaches like [17] provide fine grained access control by annotating each triple either as accessible or inaccessible for an user, and hence converting a triple store to a quad store. However, as the triple is marked for an user, such an approach makes it difficult to determine the accessibility of the same triple for multiple users. Approaches like [5] uses SWRL and the *Lightweight Access Control Schema (LACS)* vocabulary to authenticate and provides fine-grained access control by predefining a set of SPARQL queries and associating them with the users of the system. Such an approach provides the flexibility to associate the same triple with multiple users but cannot answer user queries that require a combination of two or more predefined SPARQL queries.

Approaches like [18] provide fine-grained access control by masking, to hide details, the entities present in a triple and thereby distributing the materialized copy of a triple store. As distribution is materialized, [18] is more effective for triple stores with a small number of triple count. Our approach differs from [19] by giving importance to the user attributes. Also, the use of the FILTER operator to restrict access can produce empty query result. For example consider an taxonomy with *Animal*, *SocialAnimal*, and *WildAnimal* as classes and *Animal* as the superclass of the remaining classes. A user's query to find all instances of *SocialAnimal* will be rewritten as all instances of *SocialAnimal* and FILTER all instances of *Animal* [20], which ends up being empty [21]. An attribute-based fine-grained approach [22] make use of the SeRQL query language but lacks some of the operators that are present in SPARQL [23, 24]. [25, 26] provide access control but does not support attribute-based access.

With respect to above approaches, we provide an initial attribute-based framework to (1) authenticate users, (2) associate the same triple to multiple users, (3) avoid redundancy and (4) demonstrate feasibility and present an initial analysis of the approach.

[3] `https://www.oasis-open.org/committees/xacml/`

3 Framework

In this section, we briefly describe our framework for attribute-based fine-grained access control. The framework is built on the notion of access control represented in form of triple patterns, such that it can be added directly or can be used to modify user's SPARQL query. As shown in Figure 1, the framework is divided into six different modules (1) user data, (2) authentication, (3) background data, (4) triple-based access policies, (5) triple policy enforcement and (6) triple store. We briefly describe each of the modules below.

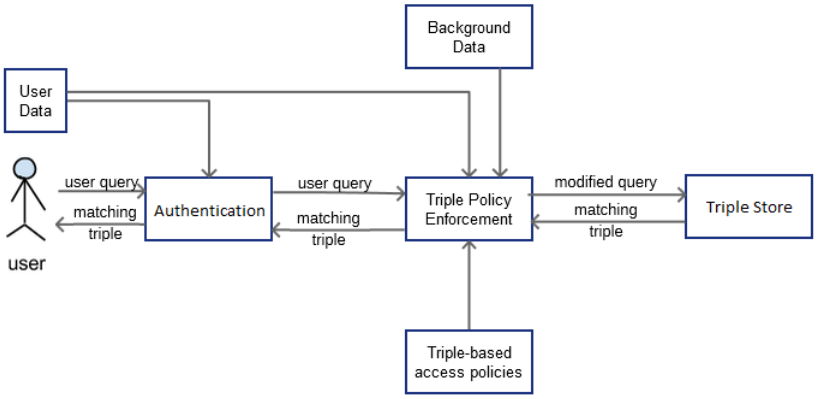

Fig. 1. Framework to provide attribute-based fine grained access control.

I. User data: User data is a set of attributes of the user, and are submitted with user's SPARQL query.

II. Authentication: The authentication module is used to verify the identity of the user.

III. Background data: Background data is used to provide additional information beside the details supplied by the user. Background data can either be context information or some other knowledge sources which can facilitate in enforcing policies.

IV. Triple-based access policies: Triple-based access polices is a set of rules used to restrict the scope of submitted user queries, and are represented in form of triple patterns.

V. Triple policy enforcement: Triple policy enforcement applies access policies on submitted user query, such that only authenticated set of triples are accessed by the user.

VI. Triple store: Triple store is used to store information in form of triples.

As shown in Figure 1, a user submits a query along with corresponding set of attributes. The system, on receiving the query and attributes, authenticates the user and forwards the query to the triple policy enforcement module. This module uses access policies to restrict the scope of the query to form a restricted query which accesses only triples which are allowed to be accessed. The restricted query is sent to the triple store for execution and the result is offered back to the user without modification. Rewriting the user query before execution helps to hide triples which violates the policy. Such scenarios happen when there are multiple paths from a source node to a destination node of same length with one path begin accessible to the user while the other path is restricted. Due to inference, other triple may be accessible to form the result and the system may make the mistake of considering them as the triples accessible to the user. Also, rewriting provides an opportunity to optimize the query. We describe each module in more detail in the subsections below.

3.1 Authentication

In general, authentication can be performed by any of different approaches, including OAuth[4], WebID[5] and OpenID[6], each of which aims to support sharing of data across different web sites. In addition, [27] uses SPARQL ASK queries with a set of user attributes to authenticate a user. However, a SPARQL ASK based verification introduces some delay in the over all query execution time [10]. OWL based authentication, like ROWLBAC [28], can be used to determine authentication of the user with special requirements like dynamic and static separation of duty. The choice of an authentication mechanism varies depending on application requirements.

3.2 Policy Representation and Enforcement

In this section we represent and describe access control policies, and an algorithm to enforce access policies to restrict scope of submitted user query. Each access control policy has four components: (1) attributes, (2) actions, (3) requested triple patterns and (4) preconditions. The first component, "attributes (*Attrib*)", describes the set of attributes required to determine if a policy can be applied to a given user query, and thereby helps to filter out policies from a given set of policies. The second component, "action (*Act*)", is used to compare the operation requested by the user and the operation permitted to be performed by the user. The third component, "requested triple Pattern (*ReqTrpPtrn*)", is used to match filtered policies with the triple patterns present in the user query. The final component, "precondition (*PreCond*)", is a set of one or more conditions that must be satisfied in order to access the triple pattern present in "ReqTrpPtrn".

[4] `http://oauth.net/2/`
[5] `http://www.w3.org/wiki/WebID`
[6] `http://openid.net/`

I. Motivating example: To better understand the representation of a policy, consider an example of a scenario based on collaboration and review.

> "A lab member can either be a primary author or a co-author of research papers. Each research paper can be categorized as (1) a submitted paper, (2) a published paper or (3) an ongoing paper. Ongoing papers are subject to proofreading and can receive comments from the co-authors as part of multiple proofreading. Submitted papers are reviewed by reviewers, and reviewers provide their reviews. Reviews are read by the primary author of the paper."

An RDFS diagram for such a scenario with possible classes and properties is shown in Figure 2. For each directed edge, the source is the domain and destination is the range of the relation represented by the label. For simplicity, consider the system to have users as lab members, and their designation in the lab as their attribute. With respect to the scenario mentioned above, a few of the access control requirements can be as follows.

1. Allow a lab members to list her submitted papers on which she is the primary author or a co-author (Policies 1 and 2).
2. Everyone is allowed to see the list of lab members who are either a primary author or a co-author of a published papers (Policies 3 and 4).
3. Allow a lab member to list all her ongoing papers on which she is either the primary author or a co-author (Policies 5 and 6).
4. Allow a lab member to read reviews of a submitted paper if she is its primary author (represented Policy 7).
5. Allow lab members to list the comments on an ongoing papers for which they are the primary author (Policy 8).

Figure 3, shows the representation of each of these policies. Here "Attrib:ALL" indicate that the policy can be enforced on all the queries. For simplicity reasons, we chose above mentioned policy representation and plan to compare different policy representation language as a part of our future work.

II. Access Policies Enforcement: The Access Policy Enforcement module restricts the scope of a submitted user query by rewriting the user query such that only authorized triples are accessed. Algorithm 1 shows the procedure to enforce access policies. In order to restrict the scope, applicable policies are selected from the set of policies by matching user attributes with the attributes mentioned in the policies. Each of the selected policies is used to establish a match between the triple patterns present in the user query and the requested triple pattern of the policy. A match exists if the value (blank node, literal or URI) for each of the entity (subject, predicate, object) present in the user triple pattern is same as the value in the requested triple pattern or if there exists a variable in requested triple pattern at the same position as that of the entity in triple pattern of the user query.

This mapping aligns variable to variable and value to either variable or value. For every match, a restricted triple pattern is generated, in which the values

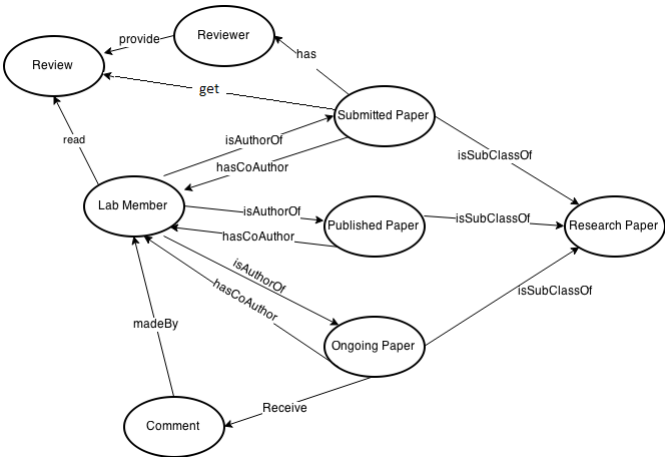

Fig. 2. An RDFS graph of a triple store representing collaboration and review scenario.

present in the user triple pattern are "placed" in the requested triple pattern of the policy, and similar changes are made to the triples present in the precondition of the policy. The restricted triple pattern along with the modified preconditions are combined to form a group pattern and is used to replace the triple pattern present in the user query. The process is repeated for each triple pattern present in the user query. By default, the access is restricted when no matching triple pattern is found from the set of all applicable policies.

3.3 Effect of query rewriting

Since Algorithm 1 restricts the scope of the query on-the-fly, it provides the capability to support an open environment where users are allowed to submit ad hoc queries. However, the execution of a restricted query can be computationally expensive depending on the number of triple patterns and size of the triple store. For example, the restricted form of a query with a triple pattern $\{?s\ ?p\ ?o\}$ will be a set of all possible policies as subqueries combined in a group pattern with "UNION" keyword. Such query rewriting may raise serious issues on the scalability of the approach and would tempt the practitioners to choose Named Graph [11] based approaches to provide access control. However, answering a user query using named graph based approaches requires materialized named graphs, which comes with the cost of simultaneous updates and storage space[12], and it becomes difficult to apply the approach for large triple stores.

On the other hand, using a non-materialized named graph as opposed to materialized one in answering a user query requires a mapping, unfolding and combining of the triple patterns from all the applicable named graphs [12], making it equally computationally expensive. Moreover, the named graph based techniques can introduce redundancies among named graphs as the same set of triple patterns may be used to form the named graphs. For example, a named

#Policy 1 - Allow lab member to list all submittedpaper where he/she is primary author

```
Attrib : ALL
Act : READ
ReqTrpPtrn :
        <USER-NAME> cr:isAuthorOf ?SP
PreCond :
        ?SP rdf:type cr:SubmissionPaper
        <USER-NAME> rdf:type cr:LabMember
```

#Policy 2 - Allow lab member to list all submitted paper where he/she is co-author

```
Attrib : ALL
Act : READ
ReqTrpPtrn :
        ?SP cr:hasCoAuthor <USER-NAME>
PreCond:
        ?SP cr:hasCoAuthor <USER-NAME>
        <USER-NAME> rdf:type cr:LabMember
```

#Policy 3 - Allow everyone to list primary author of published paper

```
Attrib : ALL
Act : READ
ReqTrpPtrn :
        ?X cr:isAuthorOf ?P
PreCond:
        ?P rdf:type cr:PublishedPaper
        ?X rdf:type cr:LabMember
```

#Policy 4 - Allow everyone to list co-author of published paper

```
Attrib : ALL
Act : READ
ReqTrpPtrn :
        ?P cr:hasCoAuthor ?LM
PreCond:
        ?P rdf:type cr:PublishedPaper
        ?LM rdf:type cr:LabMember
```

#Policy 5 - Allow lab member to list all ongoing paper where he/she is primary author

```
Attrib : ALL
Act : READ
ReqTrpPtrn :
        <USER-NAME> cr:isAuthorOf ?OP
PreCond:
        <USER-NAME> rdf:type cr:LabMember
        ?OP rdf:type cr:OngoingPaper
```

#Policy 6 - Allow lab member to list all ongoing paper where he/she is co-author

```
Attrib : ALL
Act : READ
ReqTrpPtrn :
        ?OP cr:hasCoAuthor <USER-NAME>
PreCond:
        <USER-NAME> rdf:type cr:LabMember
        ?OP rdf:type cr:OngoingPaper
```

#Policy 7 - Allow lab member to read review for submitted paper where he/she is primary author

```
Attrib : ALL
Act : READ
ReqTrpPtrn :
        <USER-NAME> cr:read ?r
PreCond:
        ?SP cr:get ?r
        ?r rdf:type cr:Review
        ?SP rdf:type cr:SubmittedPaper
        <USER-NAME> cr:isAuthorOf ?SP
        <USER-NAME> rdf:type cr:LabMember
```

#Policy 8 - Allow lab member to list comments on ongoing paper where he/she is primary author

```
Attrib : ALL
Act : READ
ReqTrpPtrn :
        ?OP cr:receive ?comment
PreCond:
        ?OP rdf:type cr:OngoingPaper
        ?comment rdf:type cr:Comment
        <USER-NAME> cr:isAuthorOf ?OP
        <USER-NAME> rdf:type cr:LabMember
```

Fig. 3. Policies defined using the RDFS graph shown in Figure 2

graph for a person's friend-of-a-friend may contain the conditions of the named graph for its friend [12]. Such redundancies are removed by defining a policy on each edge and rewriting queries before executing them.

```
Data: userQuery, accessPolicies, userAttribute
Result: Restricted user query
begin
    triplePattern ⟵ EXTRACT_TRIPLES(userQuery);
    rstTriplePatterns ⟵ ∅;
    for triple ∈ triplePattern do
        for policy ∈ accessPolicies do
            if policy.attribute == userAttribute OR policy.attribute == ALL
            then
                allowedTriple ⟵ policy.Triple;
                if MATCH(triple, allowedTriple) then
                    rstSub ⟵ triple.Subject;
                    rstPred ⟵ triple.Predicate;
                    rstObj ⟵ triple.Object;
                    rstTriple ⟵ rstSub + rstPred + rstObj
                    preCond ⟵ policy.pre_condition;
                    rstPreCond ⟵
                    replaceEntity(preCond, rstSub, rstPred, rstObj);
                end
                rstTrpPtrn ⟵ createGroupPattern(rstTriple, rstPreCond)
                rstTriplePatterns.add(rstTrpPtrn)
            end
        end
    end
    if rstTriplePatterns == ∅ then
        return AccessRestricted.
    end
    return createGroupPattern(rstTriplePatterns)
end
```

Algorithm 1: Policy enforcement algorithm

4 Experiments and Evaluation

In this section, we explore the feasibility and behavior of our approach for queries with one or more triple patterns. All experiments were performed on a computer running Fedora 17 with a swap memory of 1028 MB on 4GB RAM with i5-2430M CPU of 2.40 GHz and two cores. The Apache Jena framework was used to store data and process user queries[7]. For all the experiments we used the default Fuseki server settings.

4.1 Dataset

In order to determine the feasibility and overall effect of our approach on query execution time, we generated (i) RDF datasets, (ii) access control policies and (3) query datasets.

[7] `http://apache.mirrors.lucidnetworks.net/jena/binaries/apache-jena-fuseki-2.0.0.zip`

RDF dataset generation involved two steps. First, one large dataset was generated using a LUBM [29] script with default value for seed and index. Secondly, smaller datasets were generated in incremental manner from the larger one, such that the smaller ones are included in all the other bigger datasets. This approach to data generation simplifies the query generation process, as the answers to a query generated from smallest dataset will guarantee answers for all other dataset. For access control policies, we listed 24 empirical policies and produced them in the format shown in Section 3.2, based on the schema of LUBM generated dataset. For queries, two sets of 50 queries each were created. One set had one triple pattern chosen at random and the other set had queries with two triple patterns, both randomly chosen in a manner ensuring that the query remained connected. Both query datasets where generated independently, such that no query is associated with any other query. The smallest dataset used to generate queries had 100,545 triples. Algorithm 2 was used to generated query dataset.

The query generation algorithm produces an exact number of SPARQL queries with exact number of triple patterns for any triple store. In order to generate a query, a pattern template is first chosen at random. Each template has three character each representing subject, predicate and object, with the character 'I' and 'V' representing an instance and variable, respectively. The decision of filling the position with instance or variable is made at random. Each position is checked for instance from subject to object. If an instance is found at the subject position, a query is crafted by augmenting an additional pattern that has variables in all three positions with previously chosen old triple patterns and the query is executed on the server. An instance is picked at random from the result sent by the server. If no instance was available, then another pattern was chosen at random and the process repeated until an instance is obtained to fill the template. However, if a variable is found in the subject position, it is left unchanged. Similarly the values for the predicate and object are filled whenever an instance is encountered at those positions.

For the queries with more than one triple pattern, the first triple pattern was formed by the procedure described above. For additional triples, a pattern template was selected at random. If "I" was found in subject position, a triple with all three variable is formed and value was filled at random. If "V" is encountered in subject position then either previous subject or object variables were used to fill the position. Depending on the value of subject, the same triple pattern is augmented with the previous triple pattern to form a query and is used to fill predicate and object, if not filled with variable. For example, say the first filled template for triple pattern is "?var1 worksFor ?var2" and second templates selected was "I V V" then to fill the subject of second template, a query with following where clause "{ $?var1$ $worksFor$ $?var2$. $?var2$ $?var3$ $?var4$}" is created and executed on the server to pick the value from the result obtained. To keep query connected, the choice of the variable at subject and object position for pattern other than first pattern is determined at random from previously

```
Data: numQueries, trpPtrnCount
Result: SPARQL query
begin
    query ⟵ ∅;
    oldTriplePattern ⟵ ∅;
    newTrpPattern ⟵ ∅;
    ptrnTemplates ⟵ {IIV, IVI, IVV, VII, VIV, VVI};
    i ⟵ 0;
    varCount ⟵ 0;
    for i < trpPtrnCount do
        rndPattern ⟵ getRndPtrn(ptrnTemplates);
        if rndPattern.position[0] == 'I' then
            oldTrpPattern ⟵ oldTrpPattern.add({?S?P?O});
            subInstQuery ⟵ generate_SPARQLQuery(oldTrpPattern);
            result ⟵ EXECUTE_QUERY(subInstQuery);
            sub ⟵ pickRndInstance(result);
            oldTrpPattern ⟵ oldTrpPattern.remove({?S  ?P  ?O});
        end
        else
            sub ⟵ pickRndVar(oldTrpPattern);
            varCount ⟵ varCount + 1;
        end
        if rndPattern.position[1] == 'I' then
            oldTrpPattern ⟵ oldTrpPattern.add({sub  ?P  ?O});
            predInstQuery ⟵ generate_SPARQLQuery(oldTrpPattern);
            result ⟵ EXECUTE_QUERY(subInstQuery);
            pred ⟵ pickRndInstance(result);
            oldTrpPattern.remove({rndSub  ?P  ?O});
        end
        else
            pred ⟵ pickRndVar(oldTrpPattern)
            varCount ⟵ varCount + 1;
        end
        if rndPattern.position[2] == 'I' then
            oldTrpPattern ⟵ oldTrpPattern.add({sub  pred  ?O});
            objInstQuery ⟵ generate_SPARQLQuery(oldTrpPattern);
            result ⟵ EXECUTE_QUERY(subInstQuery);
            obj ⟵ pickRndInstance(result);
            oldTrpPattern ⟵ oldTrpPattern.remove({sub  pred  ?O});
        end
        else
            obj ⟵ pickRndVar(oldTrpPattern);
            varCount ⟵ varCount + 1;
        end
        oldTrpPattern ⟵ oldTrpPattern.add({sub  pred  obj});
        i ⟵ i + 1
    end
    return generate_SPARQLQuery(oldTrpPattern)
end
```

Algorithm 2: Query generation algorithm. Position 0,1 and 2 indicate subject, predicate and object, respectively.

filled triples (like ?var2). Such filled templates are combined to form one query and the same process is repeated until desired number of queries are generated.

4.2 Results

We had two goals in the experiment: (1) to understand the behavior of our approach for a variety of single triple patterns and (2) to understand the behavior with random queries of varying number of triple patterns. Each of the queries were restricted using a set of access control policies with same set of attributes. A total of 20 out of 24 policies were matched based on the set of attribute used for the experiment.

To characterize the behavior of our approach for a single triple pattern, we executed queries with one triple pattern, such that it can hold at least one variable and combinations of the triple patterns are covered. The average execution time of each triple pattern is shown in Table 1. As evident from Table 1, the pattern with all three variables took the most time to be executed followed by the pattern with an instance at subject and predicate position followed by a variable at object position. For patterns like "I I V", the execution time for few triple patterns can be significantly effected for a particular dataset, as it depends on the random data generated by the script, the internal indexing of Fuseki server, and the policy representation. Figure 4 represents the comparison when the query was executed with and without access policies, receptively.

	I I V	**I V I**	**I V V**	**V I I**	**V I V**	**V V I**	**V V V**
With RST	0.2725	0.0081	0.0110	0.0040	0.0665	0.0089	6.9128
Without RST	0.0201	0.0003	0.0263	0.0029	0.0345	0.0178	4.3929

Table 1. The average execution time of each triple pattern for five datasets, where "RST" is the restriction applied to the query.

In order to understand the behavior with the queries of varying number of triple patterns, we generated two datasets, TP1 and TP2, which are sets of queries with one and two triple patterns, respectively. We cleared cache memory to release pages, dentries and inodes before each query was executed[8]. As shown in Figure 5 and 6, the execution time of the queries increases with the size of the dataset. The non-uniform increase from first dataset to other datasets is dependent on randomly generated data, internal indexing of Fuseki server and representation pattern of access control policies. However such an effect is not visible when queries with two triple patterns are executed. There is a remarkable increase in the overall execution time for the queries with two triple patterns because of empty subqueries, which are formed when policies are applied to the

[8] Command to clear cache was "$sync; echo 3 | sudo tee /proc/sys/vm/drop\_caches$"

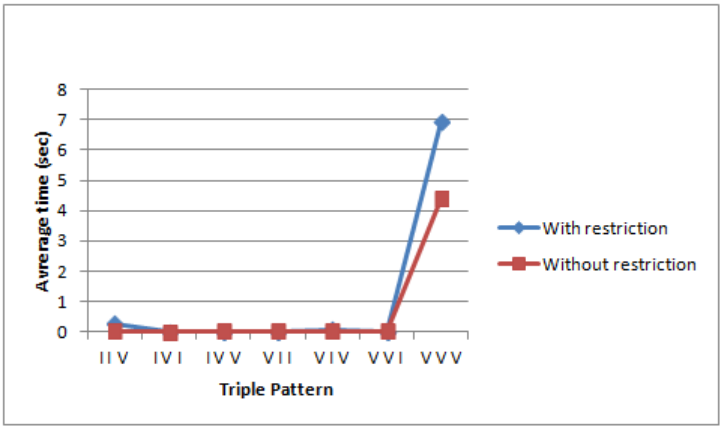

Fig. 4. Average execution time of each query pattern executed over five dataset.

query. Hence with the increase in the number of policies and the number of triple patterns in query, execution time of the query will increase significantly. As the part of our future work we plan to explore optimization techniques to reduce overall execution time.

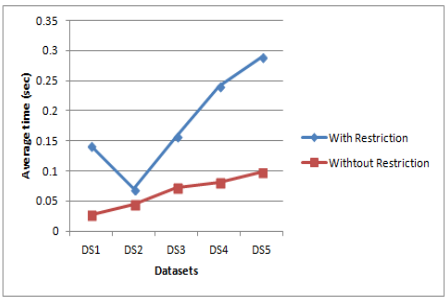

Fig. 5. Average execution time of 50 queries with one triple executed over the datasets with increasing number of triples.

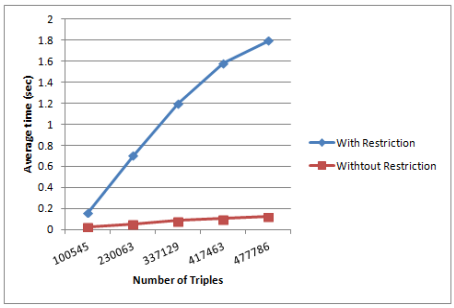

Fig. 6. Average execution time of 50 queries with two triples executed over the datasets with increasing number of triples.

5 Conclusion and Future Work

We described a framework to provide attribute-based fine-grained access control by representing the policies in a triple-based format and enforcing the access policies by rewriting a user's query. We demonstrated the feasibility of our approach

and the effect of enforcing access control polices on five datasets of different size by executing queries of varying number of triple patterns. As a part of initial analysis with a constant set of attributes and policies we found that (1) query execution time increases with increase in the size of the triple store and (2) query execution time significantly increases with the number of triple pattern.

For future work, we plan to extend our approach to work for scenarios with a reasonably large number of user attributes and policies. We also plan to explore the possibilities of optimizing the overall query execution time, since our approach selects applicable policies and create subqueries to restrict the scope of a user query. The number of such subqueries is significantly effected by number of policies present to provide access control. In addition, we will provide a formalization of our approach to demonstrate the completeness of the policy enforcement algorithm. Moreover, as each of the approaches based on named graph, description logic, rule and triple pattern provide different level of security and privacy, we plan to compare such techniques with potential merit and demerit and its effect on overall query execution time.

Acknowledgement. This work was supported by award #1228198 from the U.S. National Science Foundation.

References

1. Kagal, L., Finin, T., Joshi, A.: A policy based approach to security for the semantic web. In: 2nd Int. Semantic Web Conf. Volume 2870., Springer (2003) 402–418
2. Reddivari, P., Finin, T., Joshi, A.: Policy-Based Access Control for an RDF Store. In: IJCAI Workshop on Semantic Web for Collaborative Knowledge Acquisition. (January 2007)
3. Gabillon, A., Letouzey, L.: A view based access control model for sparql. In: Network and System Security (NSS), 2010 4th International Conference on, IEEE (2010) 105–112
4. Kolovski, V., Hendler, J., Parsia, B.: Analyzing web access control policies. In: 16th Int. Conf. on World Wide Web, ACM (2007) 677–686
5. Dietzold, S., Auer, S.: Access control on rdf triple stores from a semantic wiki perspective. In: Workshop on Scripting for the Semantic Web, 3rd European Semantic Web Conf. (2006)
6. Kirrane, S., Abdelrahman, A., Mileo, A., Decker, S.: Secure manipulation of linked data. In: 12th Int. Semantic Web Conf. Springer (2013) 248–263
7. W3C: Web access control. http://www.w3.org/wiki/WebAccessControl
8. Kirrane, S.: Linked Data with Access Control. PhD thesis, National University of Ireland, Galway (2015)
9. Muhleisen, H., Kost, M., Freytag, J.C.: Swrl-based access policies for linked data. Procs of SPOT (2010) 80
10. Costabello, L., Villata, S., Rocha, O.R., Gandon, F.: Access control for http operations on linked data. In: The Semantic Web: Semantics and Big Data. Springer (2013) 185–199
11. Carroll, J.J., Bizer, C., Hayes, P., Stickler, P.: Named graphs, provenance and trust. In: Proceedings of the 14th international conference on World Wide Web, ACM (2005) 613–622

12. Le, W., Duan, S., Kementsietsidis, A., Li, F., Wang, M.: Rewriting queries on sparql views. In: 20th Int. Conf. on World wide web, ACM (2011) 655–664
13. Baader, F.: The description logic handbook: theory, implementation, and applications. Cambridge university press (2003)
14. Priebe, T., Dobmeier, W., Kamprath, N.: Supporting attribute-based access control with ontologies. In: First Int. Conf on Availability, Reliability and Security, IEEE (2006) 8–pp
15. Priebe, T., Dobmeier, W., Schläger, C., Kamprath, N.: Supporting attribute-based access control in authorization and authentication infrastructures with ontologies. Journal of Software **2**(1) (2007) 27–38
16. Knechtel, M., Stuckenschmidt, H.: Query-based access control for ontologies. In: Web Reasoning and Rule Systems. Springer (2010) 73–87
17. Flouris, G., Fundulaki, I., Michou, M., Antoniou, G.: Controlling access to rdf graphs. In: Future Internet-FIS 2010. Springer (2010) 107–117
18. Rachapalli, J., Khadilkar, V., Kantarcioglu, M., Thuraisingham, B.: Redact: a framework for sanitizing rdf data. In: 22nd Int. Conf. on World Wide Web companion. (2013) 157–158
19. Abel, F., De Coi, J.L., Henze, N., Koesling, A.W., Krause, D., Olmedilla, D.: Enabling advanced and context-dependent access control in RDF stores. Springer (2007)
20. Chen, W., Stuckenschmidt, H.: A model-driven approach to enable access control for ontologies. In: Wirtschaftsinformatik (1). (2009) 663–672
21. Knechtel, M., Stuckenschmidt, H.: Query-based access control for ontologies. In: Web Reasoning and Rule Systems. Springer (2010) 73–87
22. Franzoni, S., Mazzoleni, P., Valtolina, S., Bertino, E.: Towards a fine-grained access control model and mechanisms for semantic databases. In: Web Services, 2007. ICWS 2007. IEEE International Conference on, IEEE (2007) 993–1000
23. Haase, P., Broekstra, J., Eberhart, A., Volz, R.: A comparison of rdf query languages. In: The Semantic Web–ISWC 2004. Springer (2004) 502–517
24. Hutt, K.: A comparison of rdf query languages. In: Proc. of 21th Computer Science Seminar, Hartfort, Connecticut. (2005) 1–7
25. Oulmakhzoune, S., Cuppens-Boulahia, N., Cuppens, F., Morucci, S.: fquery: Sparql query rewriting to enforce data confidentiality. In: Data and Applications Security and Privacy XXIV. Springer (2010) 146–161
26. Li, J., Cheung, W.K.: Query rewriting for access control on semantic web. In: Secure Data Management. Springer (2008) 151–168
27. Costabello, L., Villata, S., Delaforge, N., Gandon, F.: Ubiquitous access control for SPARQL endpoints: Lessons learned and future challenges. In: 21st Int. Conf. companion on World Wide Web, ACM (2012) 487–488
28. Finin, T., Joshi, A., Kagal, L., Niu, J., Sandhu, R., Winsborough, W., Thuraisingham, B.: Rowlbac: Representing role based access control in owl. In: 13th ACM Symposium on Access Control Models and Technologies. (2008) 73–82
29. Guo, Y., Pan, Z., Heflin, J.: Lubm: A benchmark for owl knowledge base systems. Web Semantics: Science, Services and Agents on the World Wide Web **3**(2) (2005) 158–182