Temporal Authorization Graphs: Pros, Cons and Limits

Riste Stojanov[1(✉)], Ognen Popovski[2], Milos Jovanovik[1,3], Eftim Zdravevski[1], Petre Lameski[1], and Dimitar Trajanov[1]

[1] Faculty of Computer Science and Engineering, Ss. Cyril and Methodius University in Skopje, Skopje, Macedonia
{riste.stojanov,milos.jovanovik,eftim.zdravevski, petre.lameski,dimitar.trajanov}@finki.ukim.mk
[2] Netcetera, Skopje, Macedonia
[3] OpenLink Software, Burlington, UK

Abstract. As more private data is entering the web, defining authorization about its access is crucial for privacy protection. This paper proposes a policy language that leverages SPARQL expressiveness and popularity for flexible access control management and enforces the protection using temporal graphs. The temporal graphs are created during the authentication phase and are cached for further usage. They enable design-time policy testing and debugging, which is necessary for correctness guarantee.

The security never comes with convenience, and this paper examines the environments in which the temporal graphs are suitable. Based on the evaluation results, an approximated function is defined for suitability determination based on the expected temporal graph size.

Keywords: Authorization · Temporal authorization graphs · Policy language · Semantic access control

1 Introduction

The expansion of the information technologies have produced vast amount of data that are stored in various systems and represent almost every aspect of our professional and private life. The authorization systems are the guardian of these data and regulate its access only to the granted requesters. The distributed environments in which the data is stored introduce many challenges for the authorization systems. The authorization is usually declared with policies that are enforced by an access control module implementation. The standards for policy definition, such as XACML [9], depend on the underlaying data model, and the policies are usually separately for each of the sub-systems. The separate authorization definition is mainly due to the lack of integration in multi-domain scenarios. There are multiple solutions for authentication in distributed environments, such as single-sign-on services [2], WebID [19,23] and OAuth [11], and there are frameworks that enable their integration and combination [18].

I. M. Pires et al. (Eds.): GOODTECHS 2021, LNICST 401, pp. 105–120, 2021.
https://doi.org/10.1007/978-3-030-91421-9_9

The semantic web [3] initiatives have provided solutions for the problems of different data representation in the various systems and have defined standards for bridging this gap. The linked data initiative [4] provides linking the same concepts with different representations from the various systems. However, even though these technologies solve most of the integration issues among various systems, there is no clear authorization solution that stands out and will bring them closer to the enterprise and personal applications.

The environment analyzed in this paper can be described through Definition 1 and Definition 2, using a data centric approach for authorization. The intent $\mathbb{I}$ is used to describe the set of facts presented by the requester or its agent application. It contains all the information necessary for the system to decide whether it will be further processed or not. It usually provides some evidence [14] about the requester, its environment and the intended action. It is also a common practice to include the software agent parameters as evidence, since it submits the intent on behalf of the requester. The system presented in this paper is not responsible for intent construction $\mathbb{I}$ and only provides interface for accepting it. However, it observes each intent change in order to determine the available resources for that state.

Definition 1. ***Protecting data*** ($\mathbb{D}$) *is a set of statements and resources that the authorization system should protect.*

Definition 2. ***Intent*** ($\mathbb{I}$) *is a set of statements and resources that define the requester intent together with its environment.*

The authorization systems usually operate in order to enforce a given set of client requirements. The clients define these requirements in a natural language and should be modeled in a suitable way for the system. An example requirement that an authorization system should model and enforce is shown in Requirement 1. It relates to both the data $\mathbb{D}$ and the intent $\mathbb{I}$, using the relations among them. The requirements are designed around the assumption that there is an implicit intent present. A most general abstraction can be that the requirement define a permission or prohibition of a certain interaction with a subset of the data for a given intent. It also may be observed as a set of rules that constrain an interaction using the connections among the intent and the data.

Requirement 1. *The professors can manage their active courses' grades from their faculty's network.*

This particular requirement permits a managing interaction with a grades, which courses are connected with some professor. Which particular grades are modeled with this requirement will be known when the intent will be present. Only the intents that contain a requester, an agent address and a manage action are applicable for this requirement. When suitable intent is present, the requirement can be materialized and the corresponding grades can be determined.

The requirements are represented as policies in the authorization systems, and the policy language and formalism defines its flexibility, understandability

and maintainability. The flexibility describes the ability to transform the natural language requirements into the policies. If some requirement can not be modeled as a policy, than the system is not flexible enough for that kind of requirements. The time required for the administrators to learn and practice the policy language is referred to as understandability. The maintainability of the authorization system correlates to the human effort required to configure and maintain it. The transformation of the natural language requirements into policies, and their correctness assurance occupies most of the maintenance effort.

In order to rank better in respect with these features, the policy language should be based on well adopted and widely spread technologies and standards. It should be as close as possible with the natural language expressiveness, enabling flexibility to define multiple complex relations among the data and the intent. For better maintainability, the policy language should provide close one-to-one requirement to policy transformation, and ability to test it for correctness and consistency.

2 Related Work

The authorization policies are the formalized requirements that are enforced during the authorization.

The enforcement process is usually implemented differently in each system, but there are three main enforcement patterns that may be detected: **Resource** protection is the most commonly used pattern, where the system controls the access to each of the resources based on the authorization policies. This pattern is most widely used due to its simplicity [18]. However, the downside of this pattern is that it is coarse grained. Another pattern that is being used is with **creating authorized data set** for each user, implemented by constructing a graph that composed only of the permitted data [6,8,16,20,21]. The trade-off of the implementation simplicity of this pattern is the extra time required for graph construction per user login [13,16]. **Query Rewriting** enforcement is the most complex pattern that can be used [20,22]. It adds authorization constructs to each query that is executed in the system, such that only the permitted resources can be obtained. This pattern uses complex query rewriting algorithms that must be extensively tested for correctness [13].

Generalized policy format is described in [13] as:

$$< Subject, Resource, AccessRight > \tag{1}$$

The *Subject*[1] defines for **whom** the policy will be used i.e. the agent or the user that is interacting with the system. The *Resource* defines **what** is protected by the policy, and the *Access Right* defines whether certain action is **allowed or denied** by the policy. In other words, the **access right** defines whether

[1] In this paper we will use the term requester instead of subject, since it beater describes the actor that is interacting with the system.

the policy *permits* or *denies* interaction with the **resource** on behalf of the **requester** in a given context (referred to as condition expressiveness in [14]).

The semantic authorization systems correctness is poorly analyzed in most of the literature, with exception of [13], where the query rewriting is tested for correctness against a temporal graph containing only the permitted resources. However, the possible errors in the policy design and definition processes are not considered in this paper. We have addressed this issue in our previous work [20].

For simpler policy definition, many system allow both permit and deny policies. However, in this case conflicts may arise [5,7,14]. There are various ways to solve the conflicts, such as: default behavior [1,7,20], meta-policies [12], priorities [15,20], and detection and prevention [16,20,24].

The formalization in (1) is not sufficient, since it does not encounter contextual evidences and the relation between the requester description with the under-laying data. The access control models, on the other hand, are focused on separate elements from this policy format and none of them model the complete authorization environment. Additionally, only few systems enable flexibility for connecting the request's attributes with the protected data.

Policies have an essential role in data science applications to mitigate privacy and ethical concerns. For example, during feature engineering processes and when evaluating information value and feature importance of different data [26], also needs to consider whether that data is suitable to be used for that particular application in the first place. Therefore, different applications, such as in churn prediction, may be affected by the various policies in place [25]. Likewise, the computational requirements may affect the scalability of the whole cloud-based solution [10].

3 Policy Format

Definition 3 provides a formalization of the requirements described previously. The activation function α is responsible for testing and applying the intent's data that is implicitly assumed in the requirement. It is executed whenever the intent is changed and filters out the policies that are not suitable for the current state. It also replaces the implicit variables with concrete values form the intent for the suitable policies. The function φ filters the data that is protected by the policy, and ϵ defines whether that data is allowed or denied.

Definition 3. ***Policy*** *is a tuple of* $\langle \alpha, \varphi, \epsilon, \rho \rangle$ *that defines the condition* $\alpha(\mathbb{I} \cup \mathbb{D})$ *that an intent* $\mathbb{I}$ *should satisfy in relation to the protected data* $\mathbb{D}$*, so that interaction* $\epsilon \in \{allow, deny\}$ *will be enforced with the result data* $\mathbb{R} = \varphi(\mathbb{I} \cup \mathbb{D})$*. The element* ρ *is a priority that is used for conflict resolution,* α *stands for policy activation condition and* φ *represents a partial data filtering function.*

3.1 Policy Combination

The policy combination is important for two main reasons: (1) breaking down a complex authorization into simpler rules and (2) conflict resolution. Definition 4 gives a formal definition of a conflict.

Definition 4. *Two policies* $P_1 = \langle \alpha_1, \varphi_1, \epsilon_1, \rho_1 \rangle$ *and* $P_1 = \langle \alpha_2, \varphi_2, \epsilon_2, \rho_2 \rangle$ *are in conflict if:*

- $\epsilon_1 \neq \epsilon_2$,
- *there exist intent for which they are both active,*
- $\Phi_\cap \neq \emptyset$,

where $\Phi_\cap = \Phi_1 \cap \Phi_2$ *and* $\Phi_i = \varphi_i(\mathbb{D} \cup \mathbb{I}), i \in [1, 2]$.

The policy combination is discussed in [7,17], and it suggests that the policies[2] should be combined with $\bigcup \varphi^+ \setminus \bigcup \varphi^-$. Even though this method provides policy combination, it is not flexible for conflict resolution, since the deny policies are always at a higher priority. The most common conflict resolution approaches include: (1) meta-policies, (2) priority, and (3) harmonization. The harmonization approach (3) requires disjoint policies that should be provided by the administrator of the system. In (1) rules for conflict resolutions are defined, which are with higher priority than the other policies, and this makes it a special case of (2).

The priority approach (2) is the most flexible, since it is similar to the way people solve the conflicts when they occur. It is much to define which rule is more important. This is leveraged with the ρ parameter in Definition 3. The Definition 5 defines an operator for combining an ordered set of policies.

Definition 5. ***Policy result combination*** *is a non-commutative operator* $\odot$ *such that:*

$$\langle \epsilon_1, \varphi_1 \rangle \odot \langle \epsilon_2, \varphi_2 \rangle = \begin{cases} \langle \epsilon_1, \varphi_1 \cup \varphi_2 \rangle, \epsilon_1 = \epsilon_2 \\ \langle \epsilon_1, \varphi_1 \setminus \varphi_2 \rangle, \epsilon_1 \neq \epsilon_2 \end{cases}$$

The operator $\odot$ is able to produce different output for different policy priority assignments. For example, if there are policies that allow the data Φ_1, Φ_2, and deny the Φ_3 part of the data, so that $\Phi_\cap = \Phi_1 \cap \Phi_2 \cap \Phi_3$ and $\Phi_\cap \neq \emptyset$. If the policies are activated for same intent , then they are in conflict and one of their 6 possible orderings can be chosen. Here are three example orderings together with the protected data results:

$$\begin{aligned} \rho_1 < \rho_2 < \rho_3 &\Rightarrow \langle \epsilon_+, (\Phi_1 \cup \Phi_2) \setminus \Phi_3 \rangle \\ \rho_1 < \rho_3 < \rho_2 &\Rightarrow \langle \epsilon_+, (\Phi_1 \setminus \Phi_3) \cup \Phi_2 \rangle \\ \rho_3 < \rho_2 < \rho_1 &\Rightarrow \langle \epsilon_-, (\Phi_3 \setminus \Phi_2) \setminus \Phi_1 \rangle \end{aligned}$$

3.2 Policy Language

The policy language defines the level of flexibility, understandability and maintainability. In this paper, the policies are defined in RDF format that enables actual representation of the policy elements described in Definition 3. The RDF

[2] In this description the partial data filter function φ has superscript $+$ or $-$ if it is part of a policy with enforcement method ϵ_+ and ϵ_-, correspondingly.

and SPARQL are combined together in order to bridge the understandability gap, and provide better flexibility and maintainability. Even though SPARQL can be difficult to learn, it is chosen for representation of the activation function (in the policy ontology it is p:intent_binding) and the partial result filtering function (p:protected_data) in order to provide flexibility for requirement modeling. Since the data is stored in semantic format, its query language SPARQL provides greater flexibility for data selection using various patterns. This policy format also enables easier requirement transformation into policies, with approximately one policy per requirement, which improves the system maintainability.

Example 1 shows the policy representation for the requirement Requirement 1. It is with low priority of 1. The prefix *p:*[3] is a lightweight policy ontology that describes the policy language of the system, while *int:*[4] is a basic ontology that define the most common intent classes and properties. Example 1 shows the classes *int:Requester* and *int:Agent* which are used to define who the requester is and its agent definition, in this case containing the requesting IP address. This way the policy language can model the requirement that are context dependent, using these queries and the dynamic nature of the intent.

Example 1.
```
_p a p:Policy;
 p:intent_binding 'select ?s ?a ?n where {
   ?s a int:Requester. ?a a acl:Read.
   ?ag a int:Agent. ?ag int:address ?ip. ?ip int:network ?n}';
 p:protected_data 'construct { ?g ?p ?o } where {
   ?g a univ:Grade. ?g univ:for_course ?c.
   ?c univ:has_professor ?s. ?s univ:works_at ?f.
   ?f univ:has_network ?n. ?g ?p ?o}';
 p:enforce 'allow';
 p:priority 1.0^^xsd:double.
```

4 Enforcement Architecture

The system presented in this paper relies on the policies defined in the format and language presented in Sect. 3.2. The enforcement process always starts with a requester's intent. The requester can choose to provide multiple pieces of evidence in the intent, among which can be its identity, additional attributes about him/her, the environment in which it operates, the action he/she intends to invoke, and some additional action parameters. The intent is represented as a separate semantic graph. The data contained in this graph is not controlled by the authorization system presented here. It only provides interface for intent provisioning.

The authorization system intercepts the intent and builds a temporal graph that corresponds to the data available for that intent. This process is illustrated

[3] http://github.com/ristes/univ-datasets/ont/policy.owl.
[4] http://github.com/ristes/univ-datasets/ont/intent.owl.

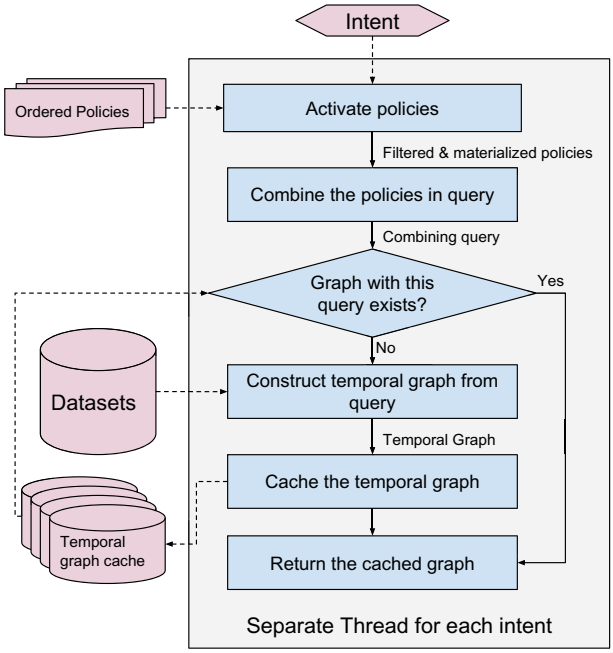

Fig. 1. Temporal graph maintenance

in Fig. 1. It is invoked every time the intent is changed. In this temporal graph creation, only the policies for protecting read operations are considered.

As Fig. 1 shows, the intent is used for policy activation. This process activates the policies using the query provided in the property *p:intent_binding*, which is defined in the policy language. This query is executed against the intent graph (from now on referred to as $\mathbb{I}$), and the resulting variable bindings are used to rewrite the *p:protected_data* query. The rewriting process replaces the variables that are mentioned in the both queries. If the *p:intent_binding* query does not return results, the policy is filtered out as inadequate.

The active policies with rewritten partial data filtering query are then combined using the $\odot$ operator described in Sect. 3.1. The only change made in the implementation is that if the lowest priority policy is denying access, a new implicit allow policy for all data is inserted with even lower priority, in order to obtain a temporal graph which always holds the permitted data. The policy combination is implemented using the Jena library[5]. As described in Sect. 3.2, each of these queries are in the form *construct* T *where* OP, where T is the triple pattern that describes which data will be select in the temporal graph, and OP defines the condition that should be meet by these triples. The system presented

[5] http://jena.apache.org.

here, first rewrites all these queries so that all of them has *T=?s ?p ?o*[6]. If the triple from some of these queries contains a term[7] then the OP is extended with *FILTER (?var=term)*, where *?var* is the variable used at that position in the triple. Next, their OP conditions are combined using the SPARQL *UNION* and *MINUS* operators that correspond to ∪ and \ operations from Definition 5. After this step, a combining query is obtained, which has the form *construct ?s ?p ?o where CombinedOP*.

Once this query is created, the system first check if it is different from the version from the previous intent. If the current intent does not change the temporal graph construction query, the old temporal graph is returned, and otherwise the temporal graph is recreated, cached and then returned for further usage.

Once the temporal graph is created, every read operation is executed against it. The read operations include the operations for fetching resources or triples, listing their properties and executing SELECT, ASK and CONSTRUCT SPARQL queries. This way, all operations operate over the permitted data only.

4.1 Conflict Detection

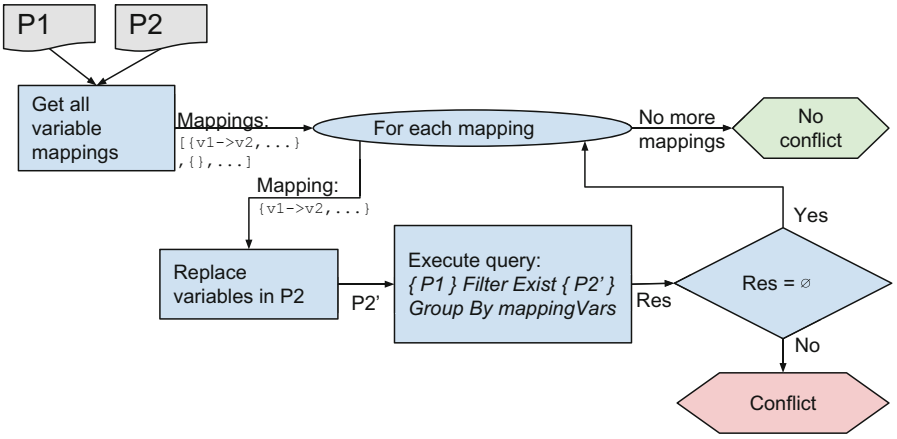

Fig. 2. Conflict detection

Definition 4 defines when two policies are in conflict. The temporal graph implementation enables conflicts to be detected automatically, with execution of the partial data selection queries of each pair of policies with different enforcement method. This is illustrated in Fig. 2, where at the beginning all variables from the policies' activation queries are mapped (all mappings are generated). Then,

[6] The variable names ?s, ?p and ?o are chosen for convenience, while in the implementation their names are randomly generated.

[7] The term unifies the IRI and literal elements.

for each mapping the variables in P2 are rewritten, and the partial data queries are combined together with FILTER EXIST operation. Group by element is added for all mapped variable in order to find a results that are returned for that particular variable combination. When this query returns results, a conflict is detected. This way the administrator can be alerted about the conflict, and can solve them using the policy priorities.

5 Evaluation

As it was previously described, the main overhead that is imposed by this authorization enforcement system is the temporal graph management described in Fig. 1. The policy activation and combination steps are carried out completely in memory, while the graph construction is carried out by the underlaying storage engine, which may uses multiple I/O operations with the disk. Because of this, the main focus in the evaluation was to determine the factors and their influence to the graph creation time.

Multiple datasets were generated and stored using the Jena TDB semantic storage. The dataset generators and evaluation code is based on the Jena API and can be easily extended for other storage engines. Multiple queries with different features were generated, and each of them was executed 40 times, 20 of witch were warm-up.

24 datasets were generated for the evaluation process. The data was generated following university ontology[8]. The dataset size can be expressed with this formula: $|DS_i| = |DS_{i-1}| + |generateGrades(100 * (i\%6 + 1) * 10^{i/6}, i)|$ where $i \in [1..24]$ and DS_0 contains only one faculty, one study program and 4 technical staff users. The *generateGrades* function takes an argument which defines the number of grades that should be generated as a first argument, and the identifier for the course for which this grades will be generated as a second argument. It first generates the course, and then each grade is assigned to that course and to a newly generated student resource.

The fixed number of grades for each course is leveraged for query construction, so that it is possible to determine the query processing time correlation with the dataset size and the number of returned results. The template of these query is shown in Example 2. Another variant to determine the time is evaluated by just adding *FILTER (?v > 6)*. The *<courseIri>* part is different in each evaluation query, and this enables obtaining different number of results.

Example 2.
```
CONSTRUCT { ?g ?p ?o } WHERE {
  ?g univ:for_course <courseIri>. ?g univ:grade_value ?v. ?g ?p ?o
}
```

Since the temporal graph is created with a query composed of multiple SPARQL UNION and MINUS elements, few query variants were made to explore

[8] http://github.com/ristes/univ-datasets/ont/univ.owl.

the dependency from the number of these constructs. These queries are combining the WHERE part shown in Example 2 for a different *courseIri* replacements. All the queries used for evaluation, together with the results are available in the generator project repository[9].

5.1 Evaluation Results

In Fig. 3 part of the query results are shown. Each line represents the dependency of the query processing time from the number of obtained results for some of the generated datasets. All of the queries tested here are composed of a basic graph pattern without filters. It shows that for this type of queries, the processing time depends only from the number of selected results, and not from the dataset size. The queries with the filter appended to them also does not depend on the dataset size, but they took more time to return the same number of results.

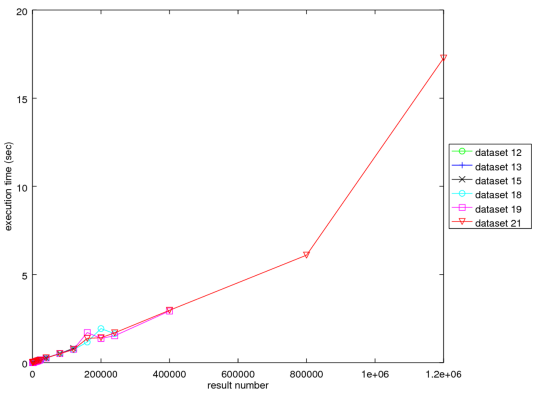

Fig. 3. Temporal graph maintenance

The execution of the queries containing union and minus clauses revealed that the union operation queries does not depend on the dataset size and the number of union expressions, but only on the selected results. On the other hand, the MINUS operation increases the time for query processing depending on the total returned plus the total removed results (the one selected in the minus clauses).

In terms of performance, the enforcement approach with temporal authorization graphs is inferior when executed once for every intended action [7,13,16]. This is why this paper investigates the graph construction performances in order to defines the limits in which this approach is suitable.

The suitability limits are calculated using the following approximate expectations as inputs:

[9] http://github.com/ristes/univ-datasets.

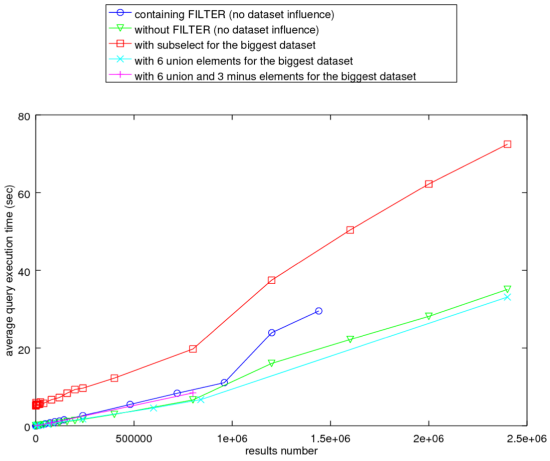

Fig. 4. Different query types comparison

– $|\varphi(p)|$: Average expected protected triples
– *icr*: Expected intent change rate (changes/minute)

The results presented here show the correlations between the processing time and the query types, resulting data size and the dataset size. Since in the most of the cases the dataset size does not have significant impact, the expected processing time function $f(|\varphi(p)|)$ is interpolated from the presented results, averaged for all query types. The average is added since it is expected that multiple policies of various types will be activated, and the average best fits this diversity. An important note is that the function f does not calculate the actual execution time, but it is intended to find the approximate expected processing time based on the expected query results.

The Fig. 4 shows the time required for a different types of queries based on the results they select. This figure shows that the queries containing *FILTER* or *MINUS* expressions are slightly slower than the other one. The reason for this is because the TDB engine has to process all suitable results that match the triple patterns, and then filters out part of those results, leading that these variants of queries require more processing time per result. Our analysis shows that the performance in general depend on the number of processed triples, not on the returned one. This way of observation shows that it is enough to execute all basic graph patterns from all policies combined with union, and for this particular engine (Jena TDB) the results will depend on the number of returned results.

During the experiments all the resulting Jena models were serialized as bytes, and they gave an average of 80 bytes per stored triple. Even though this value depends on the type of the data being stored in the dataset, it can be used to estimate the required memory for temporal graph storing through the system.

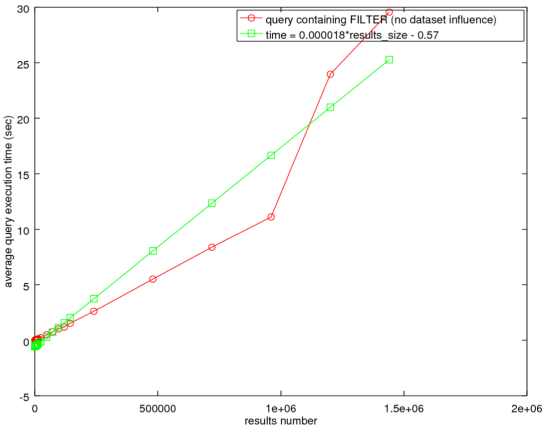

Fig. 5. Linear approximation function

The linear approximation function f shown in (2) can be used to give fast expectation about the query execution time, where *results_size* represents the number of expected results from all policy basic graph pattern combined. Even though this approximation gives lower values for result sizes above 1 M triples, as Fig. 5 shows, the time for this amount of data is more that 10 s which in our opinion exceeds the reasonable limits for login time, and thus the approximation function is no further fitted.

$$f : time = 1.8 * 10^{-5} * results\_size - 0.57 \tag{2}$$

6 Discussion

The policy language presented in this paper provides flexibility for capturing most of the client requirements. The policy combination is designed to follow the requirement process, since it is easier to order the policies by their priority. The *Intent* enables using all available evidences as a semantic statements and their incorporation within the policies. The policies enable combination of the intent's data with the protecting data in the activation and filtering phases, which matches the flexibility in the natural language used in the requirements. This way the policy language can model the requirement that are context dependent, using these queries and the dynamic nature of the intent. The activation query, and its results combination with the partial data filtering enables flexible selection of the requesters for which the policy should be applied. This is even more flexible than the ABAC authentication systems, since more than just property values can be selected. The policy language defined in this paper can select the relevant requester using its contextual evidences, its properties and its connections with the rest of the data, which provides great flexibility in the requirement transformation process. The protection granularity enables protection of resources, triples, quads and graphs., actions and aggregate operations.

This range of protection elements makes this policy language unique, especially in terms of the aggregate operations, which often are requested as exceptions of the standard rules. Such example that infers the average grade for each student in the dataset is shown in Example 3. This policy allows access to the student's average grade for all professors, with a high priority that is likely to be added to the temporal graph.

Example 3.
```
_p1 a p:Policy;
  p:intent_binding 'select ?r ?a where {
   ?r a int:Requester. ?a a acl:Read
  }';
  p:partial_data 'construct { ?st univ:has_average ?avg }
  where {
   ?r a univ:User. ?c univ:has_professor ?r.
   { select (avg(?v) as ?avg) ?st
    where {
     ?g univ:for_student ?st. ?g univ:grade_value ?v
    } group by ?st
   } }';
  p:enforce 'allow';
  p:priority 150.0^^xsd:double.
```

The policy language is chosen to be close to the natural language, and using tools like sentence dependency graph can lower the ambiguity in the process of policy design. The use of a SPARQL for policy activation and partial data selection provides a number of policies close to that of the separate requirement scenarios. The popularity of the technologies used in the policy language will lower the time required to learn it, and the large number of resources will make the policy format more understandable.

The temporal graph can be used in the policy design phase for testing. The administrator can use simulated intents to test the correctness of the policies. This provides early policy inconsistency detection. Also, when the temporal graph is used, there is no way of inferring some data with multiple query probes. For instance, if the authorization system forbids the grades of the students, but allows querying data with filter by the grade value, the grades of the students may be induced with multiple query executions. This can not be done when temporal graphs are used.

6.1 Use Case

In order to demonstrate how the results form this paper can be used in real life scenario. Lets assume that a faculty has 100 professors that can see only the grades for the courses they have lectured. For simplicity, let every course have 100 students. In worst case scenario, if all the professors are employed for 30 years, and in every semester they have held 3 courses, their temporal graph will contain in total 30 years * 2 semesters/year * 3 courses/semester * 100 students/course * 1 grade/student * 4 statements/grade = 72000 statements for the grades, which

gives something less that 7.2 MB for the grades. Even though many students overlap over the courses, lets assume that all the students are different and this gives as additional $\tilde{1}0$ MB, or roughly around 20 MB per professor temporal graph. This graph will contain in total around 200000 statements.

Given the approximation formula (2), the expected time for each professor temporal graph creation will be $1.8 * 10^{-5} * 2 * 10^5 - 0.57 = 3.03$ seconds approximately in worst case. Since the grades are not frequently changed and the professor environment does not influence to the permitted data for the professor, there wont be many temporal graph re-creations, which makes the caching time acceptable, especially given the implicit security provided by the temporal graph that contains only the permitted data.

The memory requirement for the system will be 2 GB of RAM, if all of the professors are using the system in parallel only for the temporal graphs, which now days is the memory that the mobile phones have.

However, the temporal graphs are not always suitable. One such case can be a scenario that contains temporal policies, which will induce many re-creation of the temporal graphs, so no matter how small they are, there will be a lot of overhead.

7 Conclusion

The temporal authorization graphs by their nature provide convenience with the implicit security they provide. Additionally, it is easy to test the policies in the design phase using this approach, leading to higher correctness of the authorization system. Even though their creation may introduce significant performance deterioration, the caching mechanism proposed in this paper overcomes this issue, when there are no frequent changes in the system.

The policy formalism and language presented in this paper enable flexible requirement to policy transformation. The use of the SPARQL in the policy language helps in this process, by allowing selection of arbitrary peaces of data, and following the natural language of the requirements. This way the maintenance of the system is improved, because each requirement is represented with one or few policies, while the temporal graph simplifies their testing.

Finally, this paper shows that there are use cases for which the temporal authorization graphs introduce benefits, and the approximated formula (2) provides a tool that simplifies this process for the Jena TDB storage. During the evaluation process few datasets were created, together with evaluation scenarios, which can be used to fit this formula for other semantic data storage engines.

References

1. 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. In: Aberer, K., et al. (eds.) The Semantic Web, ASWC/ISWC -2007. LNCS, vol. 4825, pp. 1–14. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-76298-0_1

2. Armando, A., Carbone, R., Compagna, L., Cuellar, J., Tobarra. L.: Formal analysis of saml 2.0 web browser single sign-on: breaking the saml-based single sign-on for google apps. In: Proceedings of the 6th ACM Workshop on Formal Methods in Security Engineering, pp 1–10. ACM (2008)
3. Berners-Lee, T., Hendler, J., Lassila, O., et al.: The semantic web. Sci. Am. **284**(5), 28–37 (2001)
4. Bizer, C., Heath, T., Berners-Lee. T.: Linked data-the story so far. In: Semantic Services, Interoperability and Web Applications: Emerging Concepts, pp. 205–227 (2009)
5. Costabello, L., Villata, S., Rodriguez Rocha, O., Gandon, F.: Access control for HTTP operations on linked data. In: Cimiano, P., Corcho, O., Presutti, V., Hollink, L., Rudolph, S. (eds.) The Semantic Web: Semantics and Big Data, ESWC 2013. LNCS, vol. 7882, pp. 185–199. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38288-8_13
6. Dietzold, S., Auer, S.: Access control on RDF triple stores from a semantic wiki perspective. In: ESWC Workshop on Scripting for the Semantic Web. Citeseer (2006)
7. Flouris, G., Fundulaki, I., Michou, M., Antoniou, G.: Controlling access to RDF graphs. In: Berre, A.J., Gómez-Pérez, A., Tutschku, Kurt, Fensel, D. (eds.) Future Internet - FIS 2010, FIS 2010. LNCS, vol. 6369, pp. 107–117. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15877-3_12
8. Franzoni, S., Mazzoleni, P., Valtolina, S., Bertino, E.: Towards a fine-grained access control model and mechanisms for semantic databases. In: IEEE International Conference on Web Services (ICWS 2007), pp. 993–1000. IEEE (2007)
9. Godik, S., Anderson, A., Parducci, B., Humenn, P., Vajjhala. S.: Oasis extensible access control 2 markup language (xacml) 3. Technical report, OASIS (2002)
10. Grzegorowski, M., Zdravevski, E., Janusz, A., Lameski, P., Apanowicz, C., Slezak, D.: Cost optimization for big data workloads based on dynamic scheduling and cluster-size tuning. Big Data Res. **25**, 100203 (2021)
11. Hardt, D.: The OAuth 2.0 authorization framework (2012)
12. Kagal, L., Finin, T., Joshi, A.: A policy language for a pervasive computing environment. In: Policies for Distributed Systems and Networks, 2003. Proceedings. POLICY 2003. IEEE 4th International Workshop on, pp. 63–74. IEEE (2003)
13. Kirrane, S.: Linked data with access control. Ph.D. Thesis (2015)
14. Kirrane, S., Mileo, A., Decker, S.: Access control and the resource description framework: a survey. Seman. Web **8**(2), 311–352 (2017)
15. Kolovski, V., Hendler, J., Parsia, B.: Analyzing web access control policies. In: Proceedings of the 16th international conference on World Wide Web, pp. 677–686. ACM (2007)
16. Muhleisen, H., Kost, M., Freytag, J.-C.: SWRL-based access policies for linked data. Procs of SPOT, **80** (2010)
17. Oulmakhzoune, S., Cuppens-Boulahia, N., Cuppens, F., Morucci, S.: fQuery: SPARQL query rewriting to enforce data confidentiality. In: Foresti, S., Jajodia, S. (eds.) Data and Applications Security and Privacy XXIV, DBSec 2010. LNCS, vol. 6166, pp. 146–161. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-13739-6_10
18. Scarioni, C.: Pro Spring Security. Apress, New York City (2013)
19. Sporny, M., Inkster, T., Story, H., Harbulot, B., Bachmann-Gmür, R.: Webid 1.0: Web identification and discovery. Editor's draft, W3C (2011)
20. Stojanov, R., Gramatikov, S., Mishkovski, I., Trajanov, D.: Linked data authorization platform. IEEE Access **6**, 1189–1213 (2017)

21. Stojanov, R., Gramatikov, S., Popovski, O., Trajanov, D.: Semantic-driven secured data access in distributed IoT systems. In: 2018 26th Telecommunications Forum (TELFOR), pp. 420–425. IEEE (2018)
22. Stojanov, R., Jovanovik, M.: Authorization proxy for SPARQL endpoints. In: Trajanov, D., Bakeva, V. (eds.) ICT Innovations 2017. CCIS, vol. 778, pp. 205–218. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67597-8_20
23. Story, H., Harbulot, B., Jacobi, I., Jones, M.: FOAF+ SSl: restful authentication for the social web. In: Proceedings of the First Workshop on Trust and Privacy on the Social and Semantic Web (SPOT2009) (2009)
24. Toninelli, A., Montanari, R., Kagal, L., Lassila, O.: Proteus: a semantic context-aware adaptive policy model. In: Eighth IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY'07), pp. 129–140. IEEE (2007)
25. Zdravevski, E., Lameski, P., Apanowicz, C., Ślęzak, D.: From big data to business analytics: the case study of churn prediction. Appl. Soft Comput. **90**, 106164 (2020)
26. Zdravevski, E., Lameski, P., Kulakov, A., Filiposka, S., Trajanov, D., Jakimovski, B.: Parallel computation of information gain using Hadoop and MapReduce. In: 2015 Federated Conference on Computer Science and Information Systems (FedCSIS), pp. 181–192 (2015)