A Platform for Assessing the Efficiency of Distributed Access Enforcement in Role Based Access Control (RBAC) and its Validation

by

Marko Komlenović

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2011

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

We consider the distributed access enforcement problem for Role-Based Access Control (RBAC) systems. Such enforcement has become important with RBAC's increasing adoption, and the proliferation of data that needs to be protected. We provide a platform for assessing candidates for access enforcement in a distributed architecture for enforcement. The platform provides the ability to encode data structures and algorithms for enforcement, and to measure time-, space- and administrative efficiency. To validate our platform, we use it to compare the state of the art in enforcement, CPOL [6], with two other approaches, the directed graph and the access matrix [9, 10]. We consider encodings of RBAC sessions in each, and propose and justify a benchmark for the assessment. We conclude with the somewhat surprising observation that CPOL is not necessarily the most efficient approach for access enforcement in distributed RBAC deployments.

Acknowledgements

This thesis was submitted in December 2010 to the University of Waterloo, Canada, for the degree of MASc in the faculty of Electrical and Computer Engineering.

I would like to thank all the people from University of Waterloo who helped me along the way of my graduation.

First and foremost, I acknowledge my professor and supervisor Dr Tripunitara Mahesh, who always had a good advice throughout our many constructive discussions, gave me support and always pushed me to reach up my potential.

I would like to thank my colleague T.Zitouni, for his contribution on measurements part.

I would also like to thank readers of this thesis for their help and suggestions.

Table of Contents

List of Tables

List of Figures

1. Introduction

Access control deals with the provision of regulated accesses to resources by principals. It is one of the most important aspects of security. In Figure 1, we show how access control is enforced. A user wishes to perform read and write operations on a file. The user makes a request that is mediated by an entity called a reference monitor. The reference monitor consults an access control policy to make its decision. An access control policy specifies the resources to which a user has access. The reference monitor either allows or denies the particular action.

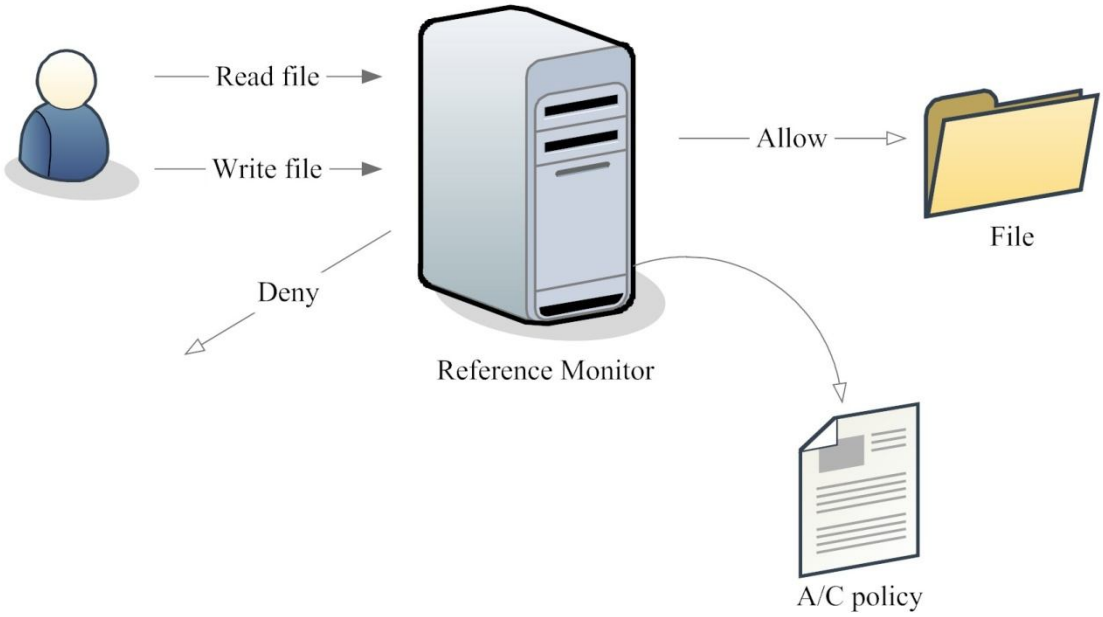


Figure 1 - A Reference Monitor and its use for access enforcement. The user attempts to read and write a file. The reference monitor mediates both attempts and after consulting the access control policy, allows him to read the file, but not write it.

A syntax for access control policies is Role-Based Access Control (RBAC) [2, 3]. RBAC is becoming the de-facto standard for access control in enterprise settings. In RBAC, rather than assigning a user directly to permissions, we assign a user to roles, and the roles to permissions. Also, the roles are associated with one another in a partial ordering called a role-hierarchy. An example of an RBAC policy is shown in Figure 2.

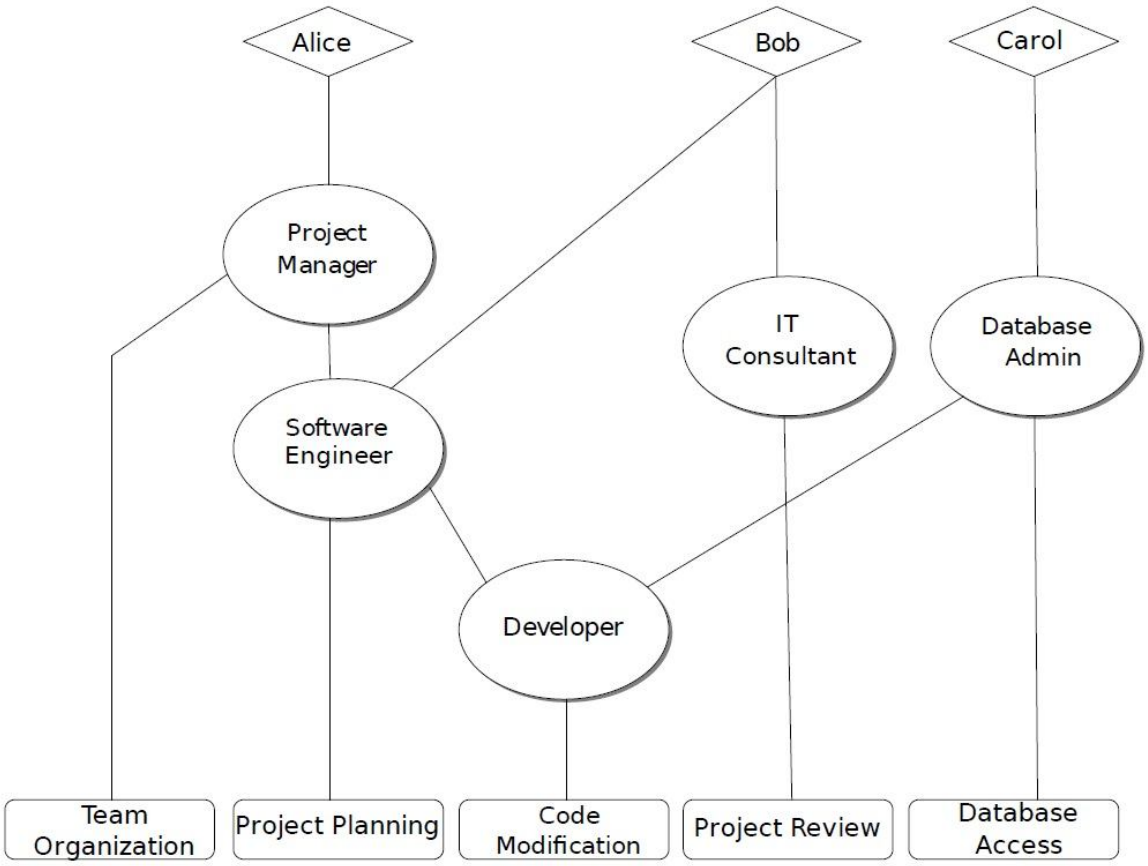


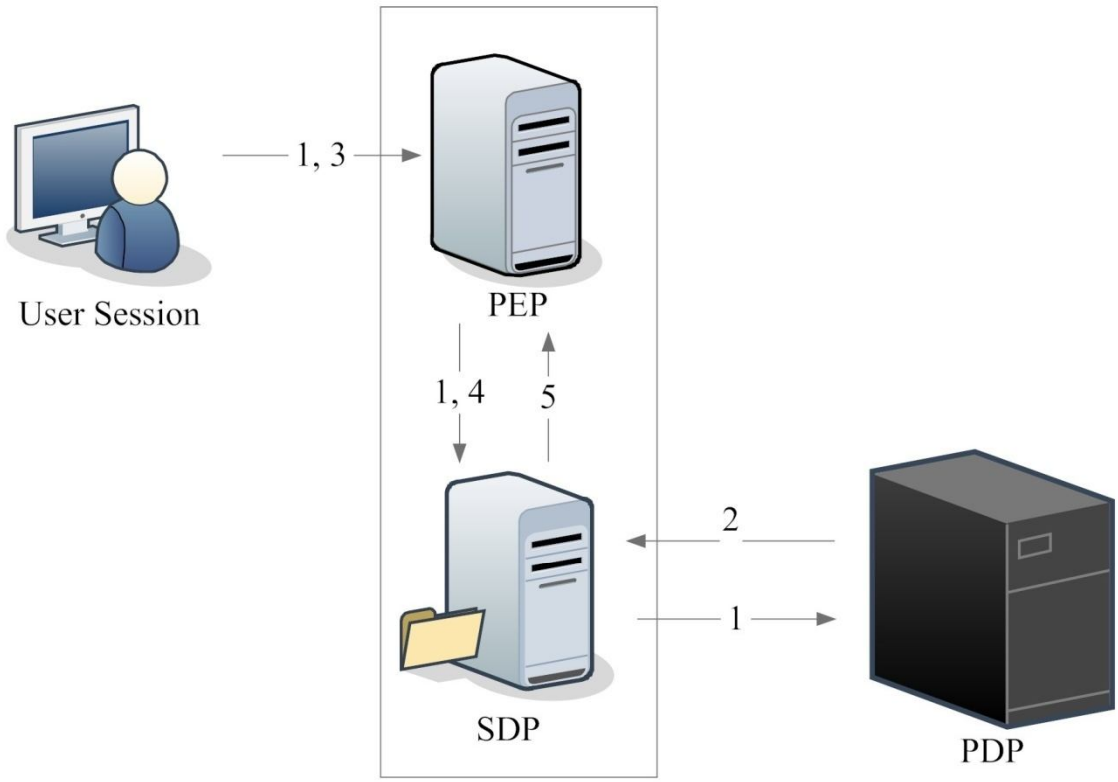
Figure 2 - An Example of RBAC policy. Users are shown in diamonds, roles in ovals and permissions in rectangles. Edges represent user-role, role-role and role-permission assignments. In the example, the user Alice is assigned to the role Project Manager and is therefore authorized to the permission Team Organization. She is also authorized to the role Developer, and therefore to Code Modification.

.

We consider access enforcement in the context of RBAC. In RBAC, a user exercises permissions in sessions. A session is associated with a set of roles to which the user is authorized in the RBAC configuration. A user can create multiple sessions. In the example in Figure 2, users *Alice* and *Bob* may activate sessions s_a and s_b respectively. *Alice* may associate session s_a with the role Software Engineer, which authorizes s_a to the permissions Project Planning and Code Modification. *Bob* may associate s_b with the roles Software Engineer and IT Consultant, which authorizes s_b to the permissions Project Planning, Code Modification and Project Review.

Modern enterprises generate and archive large amounts of data. Such data needs to be protected by access control systems. The proliferation of data requires access control systems to

scale to tens of thousands of resources and permissions [1]. The time-efficiency of access enforcement is an important consideration for RBAC systems. The size of RBAC polices can be large (tens of thousands of permissions and users), and this can impact time-efficiency. For time-efficiency, we may distribute access enforcement across several reference monitors. With such an approach, a single, monolithic reference monitor is no longer a performance bottleneck. Wei et al. [4] have proposed an architecture for such distributed enforcement (see Figure 3).



1. Session initiation request
2. Access enforcement structure
3. Access request
4. Validated/translated access request
5. Access decision

Figure 3 - An architecture, for distributed access-enforcement in RBAC, and an associated flow. The PDP is a centralized entity at which the RBAC policy is maintained. Enforcement is performed at a PEP. The PEP is aided by an SDP. The SDP can be seen as a cache of a portion of the RBAC configuration from PDP.

In the architecture, the Policy Decision Point (PDP) is a centralized entity at which the RBAC policy is maintained. Enforcement is performed at Policy Enforcement Points (PEPs). PEPs are aided by Secondary Decision Points (SDPs). An SDP can be seen as a cache of a portion of the RBAC configuration from the PDP. In Figure 3, we show a typical chronological flow of events. In Step 1, a user activates a session at a PEP/SDP. In RBAC, users exercise permissions in sessions. A session is associated with a set of roles to which the user is authorized in the RBAC configuration. In the example in Figure 2, users Alice and Bob may activate sessions s_a and s_b respectively.

The request to activate a session propagates to the PDP, which makes the decision on whether it is allowed. If it is, in Step 2, the PDP communicates a data structure to the SDP that the latter uses in Steps 3, 4 and 5 to make decisions on access requests that pertain to that session, that are communicated to it by the PEP.

In adopting the architecture from Figure 3, a question that arises is: what are the data structure and associated algorithms we should use in an SDP, so that an access check is fast? In answering this question, we also need to consider other aspects that may be traded-off to achieve time-efficiency. Two such aspects are:

- **Space efficiency** — this relates to the space that a particular data structure takes at the SDP. Space and time efficiency can be at odds; this is the classical time-space trade-off.
- **Administrative efficiency** — with this, we ask whether a particular data structure at the SDP lends itself to easy administration in the propagation of administrative changes that are made at the PDP, to the SDP. We quantify this as the time it takes to update the SDP when an administrative change is made to the RBAC configuration at the PDP.

We provide a platform for assessing candidates for access enforcement in the architecture shown in Figure 3. The platform provides the ability to encode data structures and algorithms that may be used for enforcement, and to measure the time-, space- and administrative efficiency.

To validate our platform, we use it to assess three approaches for the data structure at the SDP. They are: directed graph, access matrix [9, 10] and CPOL [6]. Apart from validating our platform, our intent with our assessment is to compare CPOL with two other approaches that are

natural candidates. CPOL is an approach to distributed access enforcement which, to our knowledge, is the state of the art from the standpoint of time-efficiency. The directed graph is a natural candidate as an RBAC policy can be perceived as a directed graph. The access matrix is a canonical and intuitively appealing representation for an access control policy. Consequently, we consider it a natural candidate as the data structure at an SDP.

A challenge in conducting an empirical assessment is the lack of a meaningful benchmark. The establishment of meaningful benchmarks is seen as an important milestone in several settings in computing. We propose and adopt a benchmark in our work (see Chapter 4). Our objective is for what we propose to serve as a macro-benchmark [7] — one that has RBAC policies and session profiles that are realistic.

In summary, our contribution is a platform for assessing approaches for distributed access enforcement in RBAC. Also, we validate its utility by assessing three approaches, one of which is the state of the art for access enforcement, but has not been previously used in the context of RBAC. The remainder of the thesis is organized as follows. In the next chapter, we discuss related work. In Chapter 3, we describe our architecture and the three approaches that we access. In Chapter 4, we describe our benchmark, and present our validation. We conclude in Chapter 5, with a rating of each of the three approaches for time-, space- and administrative efficiency.

Portions of this work have been accepted to appear in a peer-reviewed publication [14].

2. Related Work

There is large amount of research in distributed access control, and in distributed RBAC in particular. However, there is relatively little work on efficient access enforcement in these contexts. To our knowledge, CPOL [6] is the state of the art in access enforcement in distributed settings. CPOL employs caching and a structure called an AccessToken that is application-specific to speed-up access enforcement. The work on CPOL points out also that simply using database querying does not suffice for fast access enforcement. Our work is close also to those of Wei et al. [4], Tripunitara and Carbunar [5] and Liu et al. [8], that address the access enforcement problem in RBAC. Wei et al. [4] propose the architecture that we adopt in this paper (see Figure 3). In that context, they propose authorization recycling which is one of the approaches that we assess. Liu et al. [8] propose a technique that they call transformations for access checking in RBAC. We see a transformation as encoding RBAC in an access matrix; it is one of the approaches that we assess.

3. Design

As we state in Chapter 1, our architecture for distributed access-enforcement in RBAC (see Figure 3), has two components. One is the PDP which the central repository of the RBAC policy. We store the policy as a directed graph at the PDP. The other component is an SDP. In our architecture, one or more SDPs may be associated with a PDP. In the following sections, we discuss our design of the PDP and SDP and rationalize it.

3.1. Policy Decision Point (PDP)

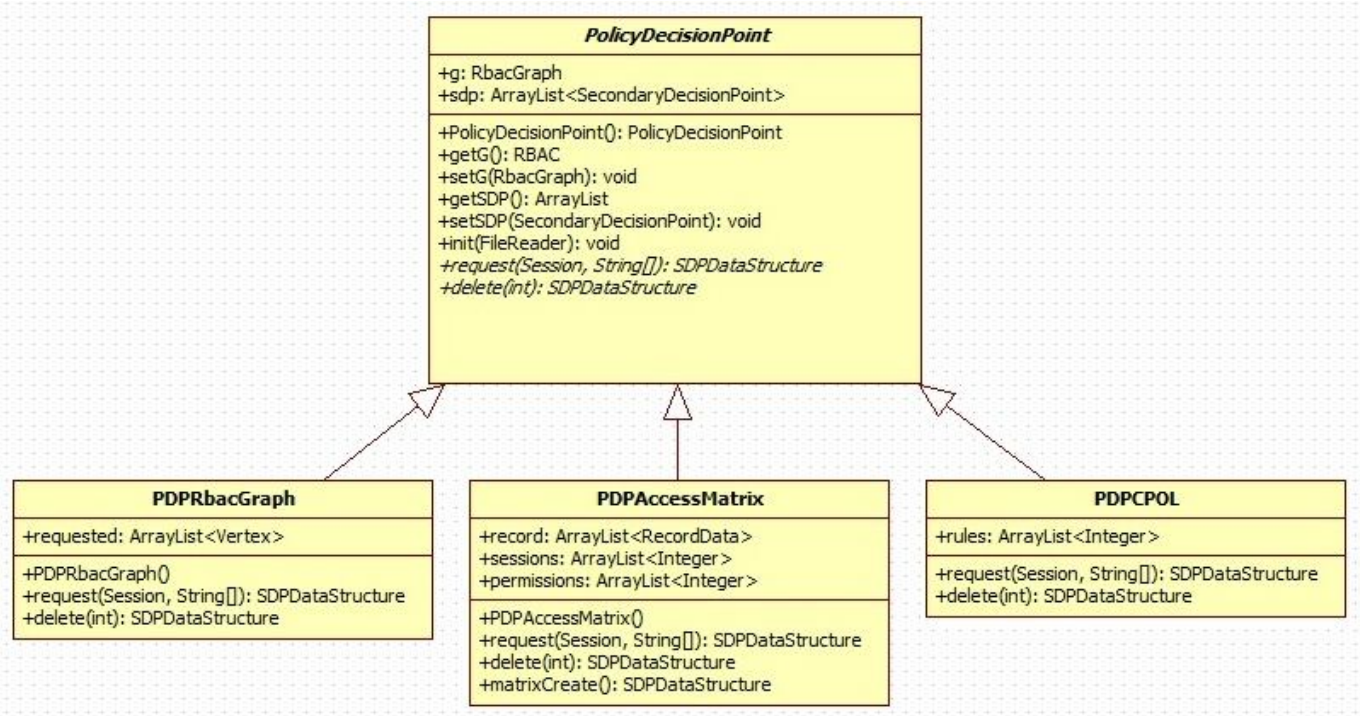


Figure 4 - PDP class diagram

The PDP has the following two attributes.

```
public static RbacGraph g;

public ArrayList<SecondaryDecisionPoint> sdp;
```

One attribute is of type RbacGraph, which is our implementation of a directed graph as an adjacency list. The other is an array list of SDPs that are associated with this PDP. The methods associated with a PDP are as follows.

```
public void init(FileReader fReader) throws IOException{...}
```

We use `init()` to initialize the PDP with an RBAC policy from an input file.

```
public abstract SDPDataStructure request(Session s, String[] roles);
public abstract SDPDataStructure delete(int session_id);
```

3.1.1. The RBAC Policy at a PDP

As we mention in Section 3.1.1, the RBAC policy at a PDP is maintained as a directed graph.

There are customarily two options for representing a directed graph [28]. One is as an adjacency matrix, and the other is as an adjacency list. We have chosen the adjacency list, as that lends itself to easier administration of an RBAC policy. Administration comprises changes to an RBAC policy, such as the addition and removal of users, the assignment and revocation of users to roles, and roles to other roles and permissions. The addition of a user-role relationship, for example, translates to the addition of an entry to a linked list in the adjacency list representation for a directed graph. Similarly, the removal of a role-permission relationship translates to the removal of an entry from a linked list. The reason why we do not care about time of checking the association between two vertices, is because we can assume that the PDP is of less importance for us, and that is run on powerful computer system, preferably fast with a lot of storage. When we say that the PDP is of less importance for us, we want to emphasise that the problem we are dealing with is more associated with the SDP than the PDP. The access checking itself is done at

the SDP. Only time we go to the PDP, is when we do not have that particular session on the SDP so we need to retrieve data from the PDP. In this case we only recalculate new structure that is to be sent back to the SDP.

`RbacGraph` is a graph implemented as an adjacency list. It has array list of vertices. It is a direct graph with a particular structure – it is acyclic, and its vertices can be partitioned into three sets [14]. Vertices are: User-Vertex, Role-Vertex and Permission-Vertex. There are some constraints on edges between those sets, and they are explained later (Section 3.2.1).

`RbacGraph` class implements `RBAC` class. `RBAC` class extends `SDPDataStructure`, which is an empty interface. The reason behind this is a need to have the same structure signature at any SDP. Having this implemented this way, we can implement its daughter classes in more elegant manner.

`RbacGraph` class looks as follows.

```
public class RbacGraph implements RBAC {
      public ArrayList<Vertex> vertices;
      public HashMap<String, UserVertex> users;
...
}
```

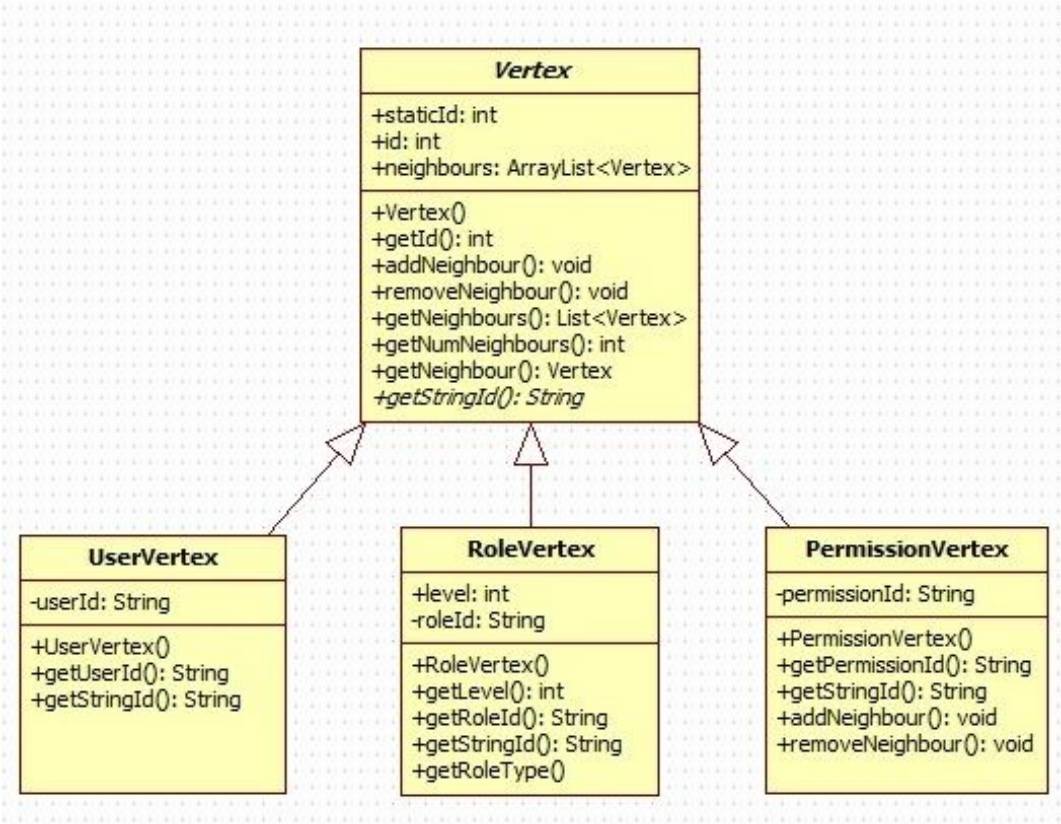


Figure 5 - Vertex class diagram

In order to explain `RbacGraph` we have to go to a more basic unit that graph is made of. That unit is a vertex. Therefore we have to take a look at the Structures package which contains the Vertex class. List of classes of Structures package is as follows.

`Vertex.java`

`UserVertex.java`

`RoleVertex.java`

`PermissionVertex.java`

`RBAC.java`

Vertex class As we can clearly see from the Figure 7, `Vertex` is an abstract class. All of the `UserVetrex`, `RoleVertex` and `PremissionVertex` extend `Vertex` class.

Apart from identification argument, `Vertex` has an array list of other vertices, which are also of the `Vertex` type. This way we know which vertex is adjacent to which one. Since the graph is

directed and acyclic, by its definition we only have edges from top to bottom, meaning that no role can point to user. Other restrictions/conditions are as follows.

- A role can point to another role only if it is of lower level of hierarchy
- A role can point to permissions
- A user can point to roles
- A user can point to permissions

Apart from usual methods (constructor, setters and getters methods) Vertex class includes some other methods. We only emphasise the importance of `addNeighbour()` and `removeNeighbour()` mehods, which we use to add a new neighbour to the adjacency list of neighbours, or delete a neighbour vertex from the list. Both methods take Vertex as an input parameter, as a neighbour which is to be added or removed from array list of neighbours. Method `getNeighbours()` will return a `List<Vertex>` of all the neighbours for particular vertex.

Classes that extend `Vertex` class have additional argument, which is string representation of id. `RoleVertex` class also includes integer argument `level`, which is used to represent a level in the graph (hierarchy) of the particular role. As we state in conditions, role can only point to another role if that other role has lower level of hierarchy; in this case that is argument level. `PermissionVertex` class has one distinction from other two classes that extend `Vertex` class. When we try to add or remove a neighbour to a permission vertex, method throws an exception since this is not valid scenario according to our conditions from above.

Class interfaces of all four classes are shown in appendix A.

Now that we explained all the components of graph we can move to explaining the graph structure and its implementation. We also explain the implementation of the SDP and the algorithms and data structures we use at the SDP.

3.2. Secondary Decision Point (SDP)

We show the SDP's class diagram in Figure 5. The attributes that we associate with an SDP are as follows:

```
public ArrayList<Session> sessions;
protected static int sdpId;
public int id;
public SDPDataStructure g;
public PolicyDecisionPoint pdp;
```

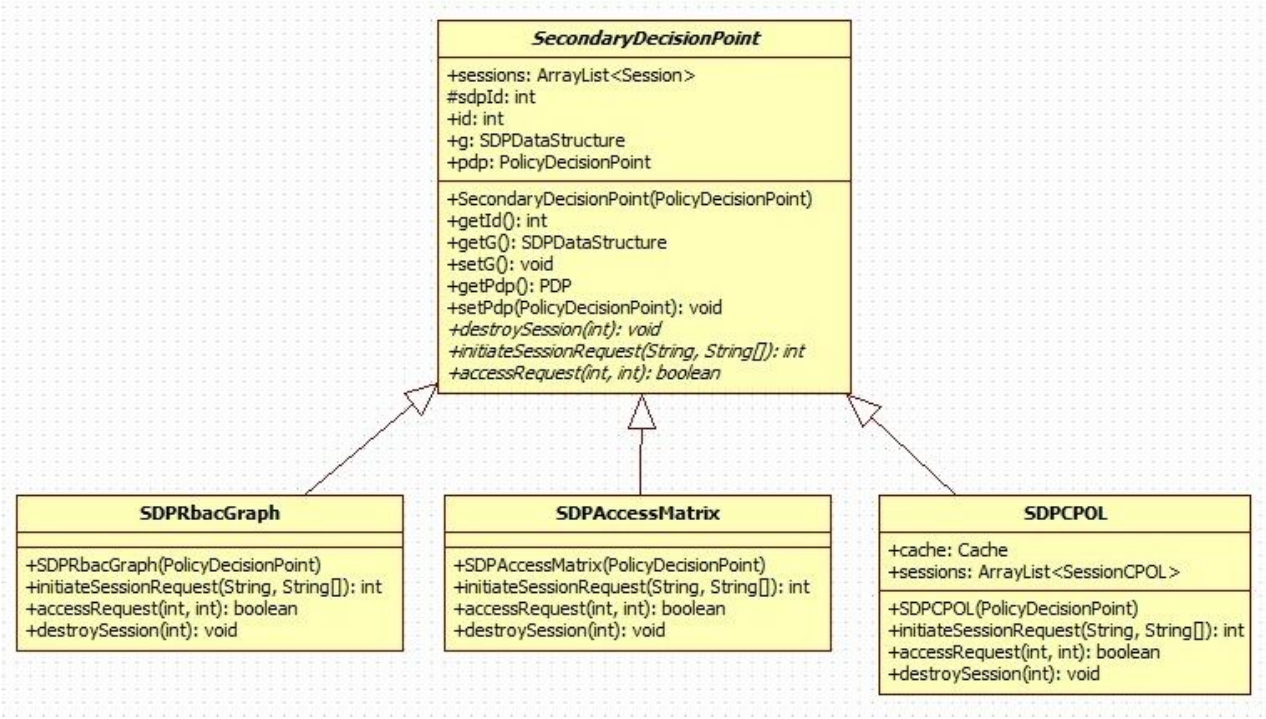


Figure 6 - SDP class diagram

Apart from identifiers, each SDP has a number of sessions that are associated with it. It also includes the data structure that is specific for its implementation and the PDP with which it is associated. The methods with which an SDP is associated are as follows.

```
public abstract void destroySession(int sessionId);
public abstract int initiateSessionRequest(String userId, String[] roles);
public abstract boolean accessRequest(int sessionId, int permissionId);
```

The communication between the PDP and the SDP occurs through `initiateSessionRequest()` and `destroySession()`. As with the PDP, the SDP's methods are abstract. They are concretized by a particular data structure that we use at the SDP. A user invokes `initiateSessionRequest()` to initiate a new session. A user issues `accessRequest()` when he wants to exercise a permission within a session. A user invokes `destroySession()` to delete a session.

Each of the SDP's array list of sessions is of type Session. It contains information about the user that initiated the session, an identifier for it, and array list of permissions that he wants to access. Permissions in RBAC are opaque [2]. Therefore, we have chosen to represent permissions as strings. Class diagram of Session class is shown below.

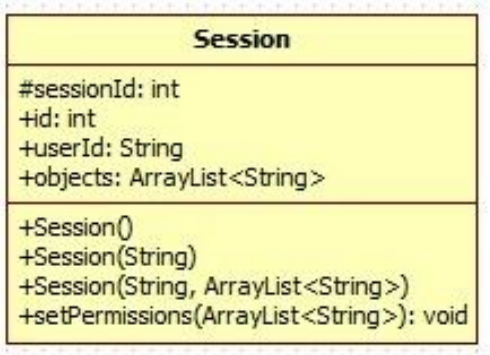


Figure 7 - Session class diagram

We have concretized the SDP with three different data structures. This is part of our validation for the abstraction that we discuss in Section 3.1.1. In the following sections, we discuss each such data structure, and the manner in which we extend the abstract SDP class for the data structure.

3.2.1. Directed graph

As we mention in the previous section our way of implementing directed graph is as an array list. We choose this particular implementation since the cost of updating the graph, which includes both deletion and adding an edge or vertex, is less than in matrix version of implementation. We leverage our implementation for a directed graph that we discuss in Section 3.1.1. We explain the process of getting the graph structure that results at the SDP from our example sessions s_a and s_b (Chapter 1, Figure 2).

When a session is activated, the PDP communicates to the SDP (Step 2 of Figure 3). Let $\hat{G}$ be the complete RBAC configuration at the PDP perceived as a directed graph. Let $S = \{s_1, \dots, s_n\}$ be the set of sessions that are active at a PEP, and $R_i = \{r_1, \dots, r_{m_i}\}$ be the set of roles that are associated with the session s_i. Let $P_i = \{p_1, \dots, p_k\}$ be the set of permissions that are reachable in $\hat{G}$ from the roles in $\bigcup_i R_i$. Then the vertices of $\hat{G}$ are $S \cup P \cup R_1 \cup \dots \cup R_n$. The edges of G are $\{\langle s_i, r_j \rangle : r_j \in R_i\} \cup E(I)$ where E(I) is the set of edges of the subgraph I of $\hat{G}$ that is induced by the vertices in $P \cup R_1 \cup \dots \cup R_n$. That is, G is similar to a subgraph of $\hat{G}$, except with sessions in place of users, and the edges induced by the vertices that are relevant to the sessions.

The access $\langle s, p \rangle$ is allowed if and only if the vertex p is reachable from s in G. We represent G as an adjacency list, which is a standard representation of a graph [11]. As an example, consider the sessions s_a and s_b from Section 1 for the RBAC policy in Figure 1. The session s_a is activated by Alice and is associated with the role Software Engineer. The session s_b is activated by Bob and is associated with the roles Software Engineer and IT Consultant. The resultant directed graph at the SDP is shown in Figure 8.

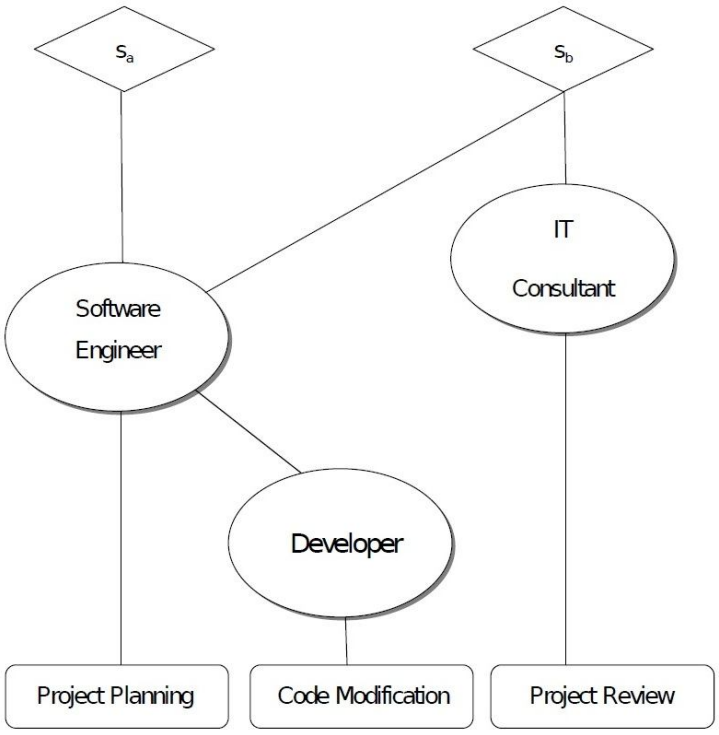


Figure 8 - The directed graph for our example sessions s_a and s_b that are discussed in the text, for the RBAC policy in Chapter 1, Figure 2.

SDPRbacGraph and PDPRbacGraph As we can see from the PDP and the SDP class diagrams (Figures 4 and 5), `PDPRbacGraph` and `SDPRbacGraph` are specializations of the PDP and the SDP classes. They inherit all the parent's class methods and arguments. These specializations are necessary since we could have three different implementations of the PDP and the SDP.

In next paragraphs we discuss implementation of the SDP where we use `RbacGraph` structure as its way of storing data.Class `SDPRbacGraph` is a daughter class to the `SDP` class (see Figure 5). Class `PDPRbacGraph` is a daughter class to the `SDP` class (see Figure 4). One of the modifications is additional argument, which contains the copy of the graph structure that is at the SDP.

```
public ArrayList<Vertex> requested;
```

Following the flow from Figure 3 (see Chapter 1), we have couple of steps in this process.

1. User comes and activates a session at the SDP. User sends initiation request to the SDP with its unique identifier and with a set of roles. Set of roles contains roles' string identifiers. At this step we call following method from the SDP class.

```
public int initiateSessionRequest(String userId, String[] roles);
```

2. This step corresponds to a step 2 from Figure 3. At this point the SDP creates a new session for particular user and propagates a request to the PDP that is assigned to. We accomplish this by calling a method `request()` from PDP.

```
public SDPDataStructure request(Session s, String[] roles);
```

Now it is up to the PDP to check and see if a user is authorized to access roles from the set he sent. If this checking is successful, new structure of graph will be sent back to the SDP. The graph structure has one vertex at the top level, which represents the session, and all the role vertices that user had requested in middle levels. At the bottom level of a graph structure are all the permission vertices that are assigned to those role vertices. In the following paragraph we explain the algorithm for getting a new graph structure.

Firstly we create a new vertex with a session identifier as our top level vertex in the response graph. Secondly, we get induced graph of the user that requested the session initiation. Next step is to go through all roles from the set and check for each of them if the user is assigned to them. If a user is not assigned to at least one of them, we return the NULL value as a response. This means that user has tried unauthorised access, the method returns an appropriate warning message. If a role is indeed assigned to a user, we merge the whole subgraph of that role with a response graph. This previous step is critical one. There is possibility when adding more roles' subgraphs that some of them could duplicate in each other's graphs. However we avoid this case since when we check a new role, we check if it is already in the response graph. Only in the case if a role is not in the response graph, we add it to a array list of vertices of a response graph. At the PDP we update the array list `requested` with newly requested vertices. When we check all the roles, we return the response graph to the SDP.

3. Now, when we create the session and update the graph structure at the SDP, user tries to send access request. User sends a request to access some permission within its session.

```
public boolean accessRequest(int sessionId, int permissionId);
```

This whole process is done at the SDP level so we do not communicate with the PDP. As we can see from `accessRequest()` method's signature, user sends `sessionId` and `permissionId`. We check whether the particular session is authorized to perform a particular session in a following way. First we fetch the vertex that has `sessionId` as its identifier (if there is such, if not we return a warning message). Secondly, we go through vertex's subgraph trying to find a permission vertex with has matching identifier with a given input argument. If we manage to accomplish all this, we return a TRUE value. This indicates that access request is approved. In any other case, whether the session is inexistent or permission is inexistent, we return a FALSE value indicating that access request is denied. If we take Figure 3 in consideration this step corresponds to steps 3, 4 and 5 from Figure 3.

4. The session at the SDP can be destroyed or invalidated. This could have probably been implemented as a time dependent feature, but for the testing purposes we have made our implementation to look as follows.
When we call a `destroySession()` method from the SDP we do following. The SDP automatically invokes delete method from the PDP. At the PDP the idea is to delete the session vertex and all other vertices from its subgraph. The only condition when we delete a vertex is that it is not assigned to any other vertex from the rest of the graph. We do not operate with the graph structure from the PDP but rather with array list requested which contains duplicate of the SDP's graph structure. When we complete deletion, we send a new graph structure to the SDP as a response. The SDP will replace its graph structure with new one. Signatures of the `delete()` and `destroySession()` methods are shown below.

```
public void destroySession(int sessionId);
```

```
public SDPDataStructure delete(int sessionId);
```

3.2.2. Access Matrix

The access matrix [9, 10] is yet another very natural candidate to be used as a structure at the SDP. It is canonical and intuitively appealing representation for an access control configuration [14]. The encoding of RBAC session in an access matrix is straightforward. Rows are indexed by sessions and columns by permissions. Our entity in a matrix is a bit. For example if a session *i* is assigned to certain permission *j*, than field of the matrix *[i, j]* is going to have value of bit 1, in other case it is going to be 0. In Table 1, We show the access matrix that results at the SDP from our example sessions s_a and s_b (Figure 2)**.**

	Project Planning	Code Modification	Project Review
s_a	1	1	0
s_b	1	1	1

Table 1 - Access matrix for our example from Chapter 1, for the RBAC policy in Figure 2.

Encoding of RBAC in Access Matrix `RbacMatrix` class implements `RBAC`, it has three attributes. First one is a matrix and another two are the dimension of matrix (corresponding to number of rows and columns). Interface of the RbacMatrix class is shown below.

```
public class RbacMatrix implements RBAC {

    public String [][] matrix;
    int M, N;

    public RbacMatrix();
    public RbacMatrix(int M, int N) {...}
    public RbacMatrix(int M) {...}
```

```
        public boolean isPair(int a, int b) {...}
}
```

First constructor is an empty constructor, second one initializes matrix of *MxN* dimension with all default values (zeros). Third constructor initializes matrix of *MxM* dimension. In this last case both dimensions are the same, however we only have one input parameter. Method `isPair()` is used so we can check whether a particular session has permission to access the permission (since we said that rows stand for sessions and columns stand for permissions). The value that a method returns, depends on a field *[a, b]* of the matrix, from given input parameters *a* and *b*. If that field is within the range of the matrix, we check its value. If the value is 1 (one), method returns TRUE, otherwise FALSE.

SDPAccessMatrix and PDPAccessMatrix The communication between the SDP and the PDP stays the same as mentioned before (see Section 3.1.1). The only difference is the way we implement methods at the PDP level since the returning data structure is now access matrix. Methods at the SDP are the same, except the implementation of the access request method. In this method we use previously mentioned `isPair()` method (Section 3.3.1.) from `RbacMatrix` class which simply checks whether certain session is allowed to perform particular permission. The methods of the `SDPAccessMatrix` are shown below.

```
public int initiateSessionRequest(String userId, String[] roles);
public boolean accessRequest(int sessionId, int permissionId);
public void destroySession(int sessionId);
```

All methods' signatures are the same as before. However when it comes to the implementation of methods at the PDP, there are quite a few changes forth of mentioning. The interface of the `PDPAccessMatrix` class looks like this.

```
public class PDPAccessMatrix extends PolicyDecisionPoint {

        public ArrayList<RecordData> record;
        public ArrayList<Integer> sessions;
        public ArrayList<Integer> permissions;
```

```
    public PDPAccessMatrix() {...}
    public SDPDataStructure request(Session s, String[] roles) {...}
    public SDPDataStructure delete(int sessionId) {...}
    public SDPDataStructure matrixCreate() {...}
}
```

We can notice that in this specialization of the PDP we have fields than we did not have in previous one (Section 3.2.3.). In order to keep track which session issues request for which set of permissions, we introduce array list of `RecordData`. We implement `RecordData` as a separate class. An instance of that class is a session's integer identifier and an array list of permissions assigned to that session. The general idea is to make process of reconstructing a matrix as easy as possible. Apart from argument `record`, `PDPAccessMatrix` class has two more arguments, both of them are implemented as array lists. Argument `sessions,` we use to store all the sessions that have requested initiation at the SDP level. Argument `permissions`, we used to store all the permissions the sessions have requested to access to. It is worth of mentioning that this array list has property of a set, there is no duplicating of permissions in it. The methods of `PDPAccessMatrix` class have the same signatures but the implementation is different.

When we send the initiation request from the SDP, we expect a new data structure to be returned from the PDP. We assume that initiation request is allowed at the SDP level. We go through the graph at the PDP and we automatically create new a new instance or `RecordData` as we collect all the permissions that user has requested for. Having that and the new session's identifier, we add a new instance of `RecordData` to an array list. At the same time we populate two other array lists, making sure that we do not end up having duplicates in any one of them. After all this is done we create and return new matrix to the SDP. We create a new using method `matrixCreate()`. This method picks the permission with largest integer identifier and the session with largest identifier assigning them to the dimensions of the matrix. It can be noticed that doing that we might end up with half empty matrix, e.g. permissions that have not been even requested but their integer identifier is smaller than the one we picked. However this is space/time trade off that is inevitable in matrix case. When we create an empty matrix, the next step is to go through an array list of `RecordData` and match the sessions with its permissions. This way we fill in the matrix with its values. For example, if an element of an array list has a

session identifier with value of *M*, and the list of permissions with values *{i, i+1, i+2, ..., n}*, then we set the following fields in a new matrix to value 1; *[M, i], [M, i+1], [M, i+2], ..., [M, n]*.

When it comes to deleting a session, the SDP calls for the `delete()` method at the PDP. There are two cases in this.

1. Session we try to delete is the last row in the matrix
2. Session we try to delete is not the last row in the matrix

In both cases we delete particular session from both `record` and `sessions` array list. However in the first we case reduce matrix's row count by one and then recreate new matrix which we send as back to the SDP. Reconstructing a new matrix in the second case is done fast. The ordering of the columns and rows does not change, we simply delete the last element from the `sessions` and `record` array list, no reordering of the columns' and rows' indexes is necessary. In the second case after updating array list structures we return a NULL value. Returning NULL value signals to the SDP that it can just do invalidation of that session's row. All the values for that row will be set to 0 (zero). This way we do not reconstruct a matrix, but we will have unused space. We could do this in the first case as well, without going to the PDP and reconstructing a new matrix.

It is worth to mention that in this approach we are facing space/time trade off. We can see that access checking is constant, but the price we have to pay is that we have to store potentially very big matrix structure at the SDP.

3.2.3. CPOL

CPOL [6] is an approach to distributed access enforcement that has been proposed in the context of trust management. In trust management, the configuration (or policy) is distributed as well. Also, the syntax of polices is different from RBAC. Consequently, we need to provide an encoding of RBAC sessions in CPOL.

In CPOL, an AccessToken is used to determine whether access should be granted or not. In the original design [6], an AccessToken is opaque – its structure is specific to an application. A policy comprises Rules; each Rule contains an AccessToken. To check whether an access

should be approved, we need to check the set of Rules and whether any of them contains particular AccessToken that will grant an access.

Encoding of RBAC in CPOL Our encoding of RBAC in CPOL is as follows; we argue that this is the most natural encoding. We implement the SDP as a CPOL Cache [14]. Implementing a cache is one of the crucial things for CPOL performance, and that is why we want this to be as good as possible and close to original implementation in C++. Cache contains an array list of cache entries (CacheEntry.java). Cache entry contains CacheKey, an AccessToken and condition (which is Boolean type). We represent CacheKey as an integer sessionId. AccessToken is a set of permissions to which session is authorized [14]. Our study of the original CPOL implementation suggests that a manner in which a set of permissions should be implemented is of big importance for CPOL performances. We used Java library java.util.set to represent a set of permissions. In Appendix B we show all the classes within Cpol package. Figure 9 shows class diagram of Cpol package.

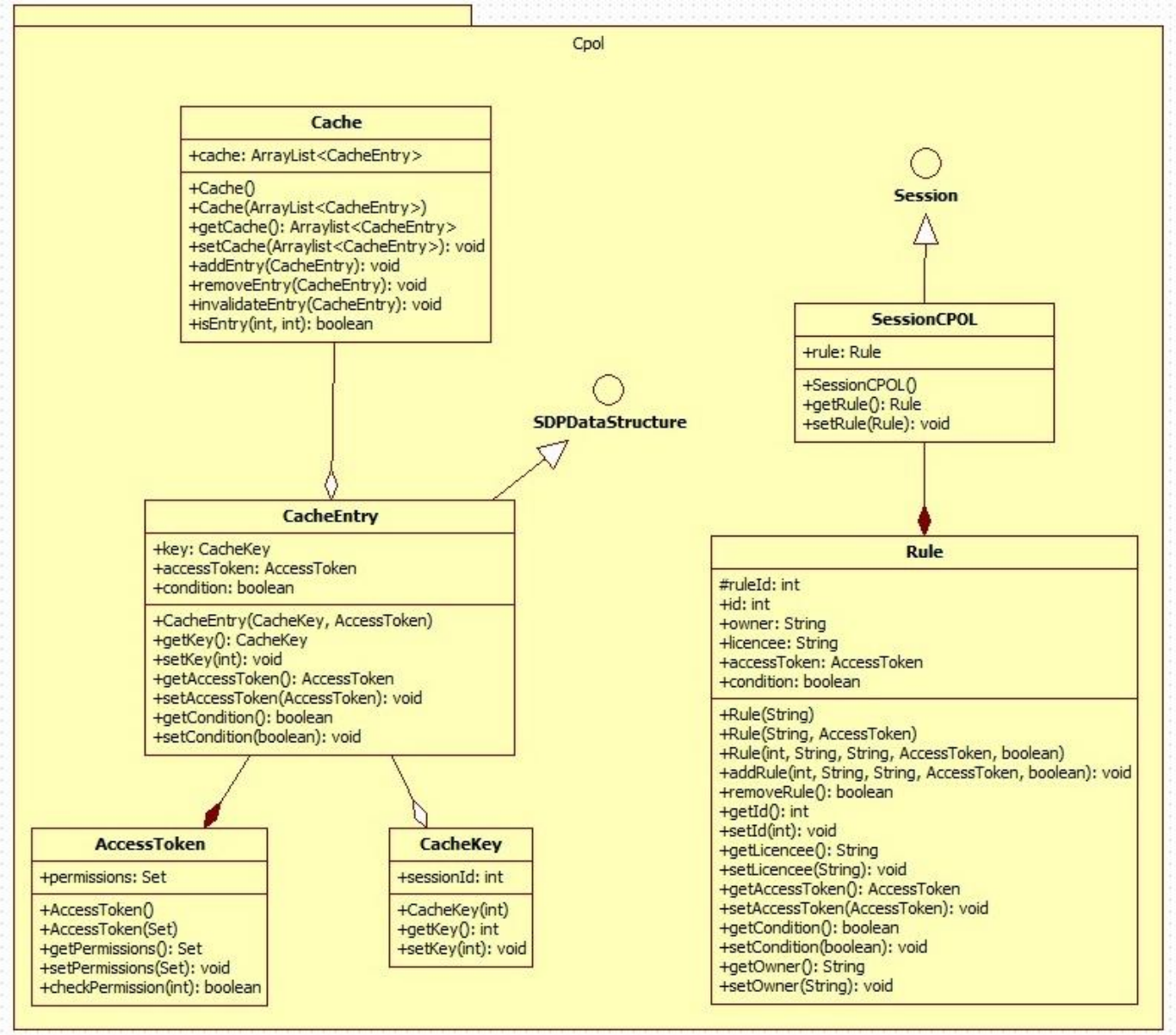


Figure 9 - Cpol package class diagram

Class `Rule.java` is basically mapping of a Session as it is in other implementations (Sections 3.2.1. and 3.2.2.). To simplify this mapping we introduced a new class `SessionCPOL`, which extends `Session` class. As an only field, `SessionCPOL` class, contains Rule. We can now see the mapping with the Session more clearly and more logical. When implementing each class, we take care that it becomes worthy duplicate of its original implementation. `Rule` contains following fields; owner, licencee, accesstoken and condition.

Owner, which is type of String, is unused in our process of evaluation, and since it is used only by a system itself, we say that system is the owner. In our implementation we use licencee to represent userId. Licencee is a type of String. Since we map Rule to the Session, the only natural thing is for licencee to be userId. AccessToken is type of AccessToken, which is described above. Field condition is type of Boolean, and it is used to say whether a particular Rule is valid or not.

In our example of the sessions s_a and s_b from the RBAC policy (see Chapter 1, Figure 2), we have two CPOL Rules, $Rule_a$ and $Rule_b$, which contain $AccessToken_a$ and $AccessToken_b$ respectively. $AccessToken_a$ is {Project Planning, Code Modification}. $AccessToken_b$ is {Projec Planning, Code Modification, Project Review}. The cache has keys s_a and s_b, for the two access tokens. [14]

SDPCPOL and PDPCPOL — The communication between the SDP and the PDP stays the same as mentioned before (see Section 3.1.1). Signatures of the SDP's methods stay unchanged as before, but their implementation defers from previous ones as expected. We show interface of `SDPCPOL` below.

```
public class SDPCPOL extends SecondaryDecisionPoint {

      public Cache cache;
      public ArrayList<SessionCPOL> sessions;

      public SDPCPOL(PolicyDecisionPoint pdp) {...}
      public int initiateSessionRequest(String userId, String[] roles);
      public boolean accessRequest(int sessionId, int permissionId);
      public void destroySession(int sessionId);

}
```

As mentioned before this instance of the SDP is implemented as a cache. At the SDP level we store Cache which has multiple CacheEntries and an array list of sessions, which are type of SessionCPOL. This list has actually properties of set, since there could not be two same Rules, therefore nor could exist two same `SessionCPOL`. This is exactly what we had mentioned in

previous section; in order to check whether a particular access should be approved, we check the set of Rules to determine if some contains an AccessToken. The other way of checking this is to go through a Cache, which is a keyed table, and as soon as we hit the requested entry we check whether an AccessToken within contains particular permissionId. This is however the way we do it in our implementation since the access request when issued has two parameters; sessionId and permissionId. When a session is to be deleted; we do this in two steps; first step is to invalidate entry with the key that is equal to sessionId, and the second step is to remove the rule that has been created for particular session. The way we did step three is very intuitive, we simply iterate through list of `SessionCPOL` and when we find a match we remove it from a list. At this point no communication with the PDP is required. However since we also keep copy of all the Rules at the PDP, we will invoke delete method from the PDP, so that the PDP updates its list as well. When a session initiation request is issued, we do next steps. First step is to create new instances of Rule and `SessionCPOL` of input parameters of the method. After this is completed we send a request to the PDP with new `SessionCPOL` and set of roles, as parameters. If all is in order (there is no violation of access of any kind), the PDP should return a new CacheEntry to the SDP. Newly created instance of Rule is being updated with AccessToken from new CacheEntry, and its condition is set to TRUE. After doing so, Rule is being assigned to a new instance of SessionCPOL. Finally both new CacheEntry and `SessionCPOL` are added to their lists, and so session initiation is being successful.

At the `PDPCPOL`, we have two methods; request and delete. First one is invoked by the SDP when there is an initiation of a session, second one when a session is to be deleted. As mentioned before, we keep array list of Rules at the SDP level as well. However instead of having the whole Rule structure we keep only its id at this level. When the user is identified, and when we make sure that all the roles he requested, he can access to, we will form a list of permissions. From a input parameter we extract a Rule's id which will be placed in a list of Rules at the PDP. After all this is done successfully, we create new AccessToken, CacheKey (key itself is Rule's id) and new CacheEntry, that is to be returned as a response to the SDP. If anything goes wrong, NULL value will be returned. Second method is delete method, which we invoke by the SDP when we want to delete a session. This method is mapping to a `removeRule()` method from original Cpol implementation [6]. Since Session is mapped to a Rule, now sessionId parameter is actually id of a Rule we want to remove from a list.

In our implementation of CPOL, we have adhered closely to the original implementation. In a Chapter 4, we discuss why we have based our assessment on a new implementation. Our implementation compares in performance to the original (see Chapter 4.). [14]

4. Validation

4.1. Benchmark

In this chapter we explain how using existing architecture that we developed, we can evaluate different implementations of access enforcement in RBAC. Doing this we wish to determine whether our platform is sound and independent of any possible data structures that might be used at the SDP level. Firstly we would like to describe the benchmark we used. Benchmark has two components: RBAC polices [29] and session profiles [14]. We have designed and implemented programs to generate data sets for the benchmark [14]. The programs are written in Java. Each one of them takes certain arguments that correspond to the categorizations we discuss in next sections.

4.1.1. RBAC Profiles

The RBAC policies that comprise our benchmark are from prior research in RBAC, and experience with RBAC deployments that have been documented in books and the research literature. We present a summary in Table 2. We categorize RBAC policies along the following axes.

Source	# Users	# Roles	# Permissions	RH Depth	RH Model	Connectivity
Literature	500-999 1000-1999 2000-2999 3000-3999 5000-6000 10000-40000	10-200 200 100 200-250 200 120-1300	10-3000 1000-3500 100-2000 1500-11000 1500-2000 100-11000	1 - 5	Stanford Hybrid Core	Constant (range) to roles, Constant (range) to permissions, Distribution (e.g., Zipf, uniform)
Synthetic	40001-400000 400001-1600000	1600-16000 16000-64000	1500-2000 1500-11000			

Table 2 - Categorization of RBAC polices in the benchmark.

Source We have two sources, "Literature", and "Synthetic". By Literature, we mean that we have directly acquired particular kinds of policies from literature that documents research and experience with RBAC. Our sources for these can be classified into three.

1. Top —down design of RBAC polices [3, 15, 16, 17]
2. Role mining and engineering [18, 19, 20, 21, 22, 23, 24, 25]
3. Evaluation of approaches to access-enforcement [4, 5, 8]

We also have created some new kinds of policies based on policies from the literature. We call these Synthetic policies.

Number of users, roles and permissions The numbers of users, roles and permissions are typically co-dependant in RBAC policies from the literature. In Table 2, we show the number of users, and the corresponding numbers of roles and permissions for policies from the literature, and for Synthetic policies. We point out that roles do not grow, for example, linearly, with users, but more as a step function.

We point out also that the number of permissions range from a fraction of the number of users, to a somewhat significant multiple. The reason for this range is that RBAC is deployed in one of two contexts. One is for high-level policies in which permissions are abstract. Another is at a much lower level, in which resources that are protected are individual files or email messages; in such systems, there can be a considerable number of permissions. (It is common for a permission to be a pair $\langle o, r \rangle$, where o is the object or resource that is protected, and r is a privilege or right. However, this is not the only encoding as a permission that is meaningful; see, for example, the work of Crampton [26].)

For our Synthetic policies, we consider numbers for typical enterprises that we have not already considered under Literature. The number of employees of an enterprise can be up to 1.6 million [12]. If such enterprises deploy RBAC, we anticipate that they will want to model each employee as an RBAC user. For such policies, we anticipate that the number of roles will be in the same proportion to the number of users as for the largest range for users from the literature. We do not anticipate that the number of permissions will increase significantly. Consequently, we adopt for permissions similar numbers as the largest ranges from the literature.

Role hierarchy (RH) and connectivity There are three categories we consider for the structure of RBAC policies. As Table 2 indicates, these are RH Depth, RH Model and Connectivity. By RH Depth, we mean the maximum path-length from a role to a permission. In our survey of the literature, the RH Depth does not exceed 5.

We consider two RH Models, Stanford and Hybrid. In the Stanford model [3], roles are layered, and a role at layer *i* directly inherits roles only in layer *i + 1*, and is inherited directly only by roles in layer *i - 1* (or by users, for the topmost layer of roles). The Stanford model arises in the top-down design of RBAC policies. Realizing the Stanford model in an enterprise typically results in 4 or 5 layers of roles [3]. The hybrid model arises in both the top-down design of RBAC policies and in role mining. In the hybrid model, the role hierarchy is some partial ordering, and not layered as in the Stanford model. A special case of the two models is when there is no role-role relationship. This is called Core RBAC and arises in role mining [8, 21].

4.1.2. Session profiles

There is some prior work which has datasets on session profiles [5, 8, 13]. We augment those datasets with our own. We categorized session profiles into two; activation and access checks; Table 3 [14, 29].

Activation	Access checks

• Intra-session
  ○ Number of roles
  ○ Number of permissions
  ○ Nature of roles
  ○ Nature of permissions
• Inter-session
  ○ Number of sessions
  ○ Arrival rate
	• Number
	• Nature

Table 3- Session profile categories in our benchmark.

Under activation category, we consider attributes associated with activation of session. For intra-session we have four arguments. Number of roles represents how many roles we would like to activate. This could be a fixed number or a range. Number of permissions is number of how many permissions we want to access. When we talk about nature of roles, there is quite a few ways this one is used. For example we may specify that only roles that are directly assigned to user are activated. Another would be that only roles that activate same sets of permissions are activated. This is also referred as separation-of-duty [3]. Nature of permissions gives us option to choose whether to activate only permissions user is assigned to, or rather any permission from the system.

For inter-session we have two attributes. Number of session represents exactly number of sessions we would like to activate. The other one, arrival rate, is the kind of arrival we would like our session activations to show up in the system. We consider bursty and uniform arrival rate. By bursty arrival, we mean that session activations are interspersed with relatively long "quiet" periods in which we have no session activations. In between those activations, we have access checks for the existing sessions [14]. In uniform session arrivals, session activations are uniformly interspersed with access checks [14]. We conclude that bursty arrivals are more likely sessions directly used by humans, and on the other side, uniform arrivals would be possible for automated processes which activate the sessions.

Our second category is relates to access checks. Here we have two attributes: the *number* and the *nature* of access checks. First one represents the actual number of sessions we would like to activate. Under nature of access checks, we characterize the permissions for which access checks needs to be made. For example one way is to perform all the access checks to all of permissions, until we reach the number of access check. The other way is to perform access checks only to permissions that are allowed for a user.

4.2. Evaluation and methodology

In next section we will talk about performance of three different approaches against our platform. We consider time, space and administrative efficiency.

4.2.1. Methodology

Meaningful empirical assessment is a significant challenge in computing. For Java programs, non-determinism in making empirical observations can result from various factors, such as dynamic compilation and garbage collection. The methodology we adopt overcomes such non-determinism and is statistically rigorous. It is based on the work of Georges et al.[27].

Java programs run within an instance of a Virtual Machine (VM). We collect the average time across multiple VM invocations, as there can be variation across such invocations. Within a VM invocation, we need to avoid skew from the effects of starting up the VM and reach what is called steady-state [27]. For each VM invocation, we determine the number of benchmark iterations that we need to perform by finding at least k consecutive steady-state values for which the coefficient of variation (CoV) is less than some preset value (we have chosen 2%). The value of k starts at some value (4, in our case) and increases so long as the CoV decreases, upto the threshold. We record the mean of the k values for each VM invocation. Our final benchmark time is the mean across all VM invocations.

To minimize the effects from garbage collection, we keep the heap size constant across VM invocations. Apart from the mean, we also compute confidence intervals. Our objective is for the confidence intervals to not overlap, as then, with a certain confidence (95%, in our case), we can assert that the two values are statistically distinct. All the values we report and graph in this paper are statistically distinct from other values.

We have conducted our experiments on an isolated Intel dual core E8400 PC that runs at 3 GHz, has 3.5 Gbytes of RAM and runs the Ubuntu Linux operating system. Our Java version is 1.6.0_18, and we run the OpenJDK Runtime Environment.

4.2.2. Evaluation

Time efficiency For time efficiency we consider both inter-, and intra-session profiles. We have two inter-session attributes: the number of sessions, and the arrival rate. In Table 4 and Figures 10, 11 and 12, we present our results for time efficiency, with the inter-session attributes as parameters. We also consider an intra-session attribute, the nature of RH. We discuss the results

that pertain to that in the next section. In each dataset we have 2500 users, each authorized to different numbers of roles and permissions. We have 100 roles in total, and 100 permissions. Our objective is to understand the behaviour of each approach as the two inter-session attributes change. Consequently, we consider from 2 through 15 sessions, and both bursty and uniform arrivals for the sessions. There are several observations we make from our results.

Arrival rates We observe from Table 4 that none of the approaches is impacted by the session arrival rate (burst vs. uniform).

Number of sessions The graphs in Figures 10, 11 and 12 show the impact of the number of sessions on each of three approaches. We observe that all three approaches are resilient to an increase in the number of sessions from the standpoint of time efficiency. That is, access check time does not necessarily grow with the number of sessions. We expect this to be the case, so long as the PEP/SDP is not stressed by adding too many sessions. None of the approaches has an access check algorithm whose time-complexity is parameterized by the number of sessions.

It is not our objective to stress a PEP/SDP by considering large numbers of sessions. Indeed, the number of sessions a PEP/SDP can support without significant impact on its performance depends on its resources such as hardware. Our objective is gain broader insights into the three approaches, notwithstanding the resources available to a PEP/SDP, assuming some realistic model of computation (the "Random-Access Machine" model, for example [11]).

Efficiency The access matrix is very time-efficient; in our tests, an access check takes less than 1 μs. This is unsurprising as an access check is done in constant time with minimal additional overhead. CPOL is only slightly less efficient; for this particular dataset, we can perceive the number of permissions to which a session is authorized as constant. Consequently, the manner in which a CPOL AccessToken is realized does not impact time-efficiency. The directed graph is highly efficient for Core RBAC. This is because a path from a session vertex to a permission vertex is exactly 2; consequently, it is highly efficient when we have only up to a few hundred roles. We study the impact on time efficiency from intra-session attributes (e.g., a large number of permissions in a session) in the next paragraphs.

Jitter By jitter, we mean the variation in access check times as the number of sessions changes. We can quantify this as the percentage error in the mean; that is, the ratio of the standard deviation to the mean. We observe from Figures 10, 11 and 12 that this is quite small for the directed graph, access matrix and CPOL. In our datasets, a user is directly assigned to the same number of roles across each of the Stanford, Hybrid and Core policies. Consequently, there is more heterogeneity in the roles that a user may activate in the Stanford policy than in the other two.

		Directed graph	Access matrix	CPOL
Bursty	Stanford	32.70	0.79	2.14
	Hybrid	9.41	0.80	3.12
	Core	5.17	0.74	2.87
Uniform	Stanford	29.47	0.62	1.50
	Hybrid	8.45	0.62	1.44
	Core	5.93	0.60	1.51

Table 4 - Average access check times in µs with the inter-session attributes, and one intra-session attribute (nature of RH), as parameters.

We have studied the impact of intra-session attributes on time-efficiency. We vary three parameters in our experiments in this context: the number of roles per session, the number of permissions per session and the nature of RH (Stanford, Hybrid and Core). Figures 10, 11 and 12 shows the impact of the last attribute on time efficiency, and Figures 13, 14 shows average access check times in μs for Core RBAC, for which the number of roles and permissions range from small (10) to large (10,000). Such numbers are consistent with Table 2.

Role hierarchy Table 4 and the graphs in Figures 10, 11 and 12 show the impact of Stanford vs. Hybrid vs. Core as the choice for RH. Only for the directed graph do we see an impact from the choice of RH. For the directed graph, a deeper RH results in an increased access check time as we need to traverse a longer path from a session vertex to a permission vertex. This is reflective of our dataset — a user is directly assigned to the same number of roles for all three of the

Stanford, Hybrid and Core RBAC policies. However, in the Stanford policy, he is authorized to more roles as a result of the deep RH.

Scalability We observe from Figure 13 that the directed graph scales poorly as we increase the number of roles. The reason is that access checking for the directed graph is vertex reachability, which is linear in the size of the graph. For the access matrix and CPOL, the time for access checking is independent of the number of roles in a session. In this respect, they scale well with the number of roles.

The access matrix and CPOL scale well also with the number of permissions, as Figure 14 indicates. This is somewhat surprising as an AccessToken in CPOL is linear in the number of permissions in a session. As we mention earlier, it is crucial to the time efficiency of CPOL that this encoding be efficient. Notwithstanding this, up to 10,000 permissions, these issues appear to have no tangible impact on the time efficiency of these approaches. The directed graph fares poorly in this context as well. This is because the adjacency list approach often requires a linear search to find a vertex (permission, in this case).

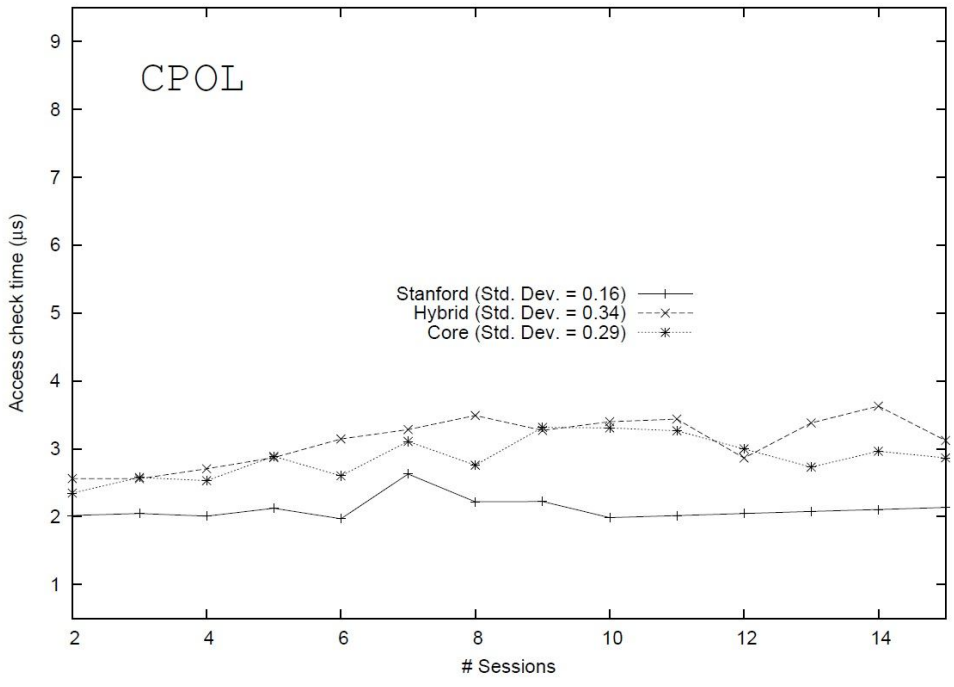


Figure 10 - Average access check time in µs and the corresponding standard deviation for CPOL.

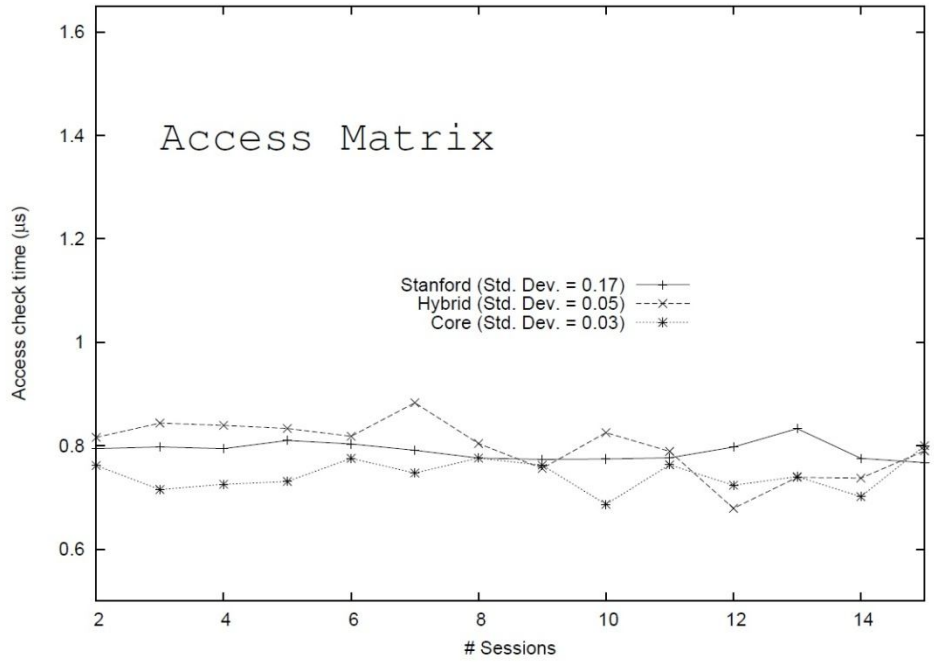


Figure 11 - Average access check time in µs and the corresponding standard deviation for access matrix.

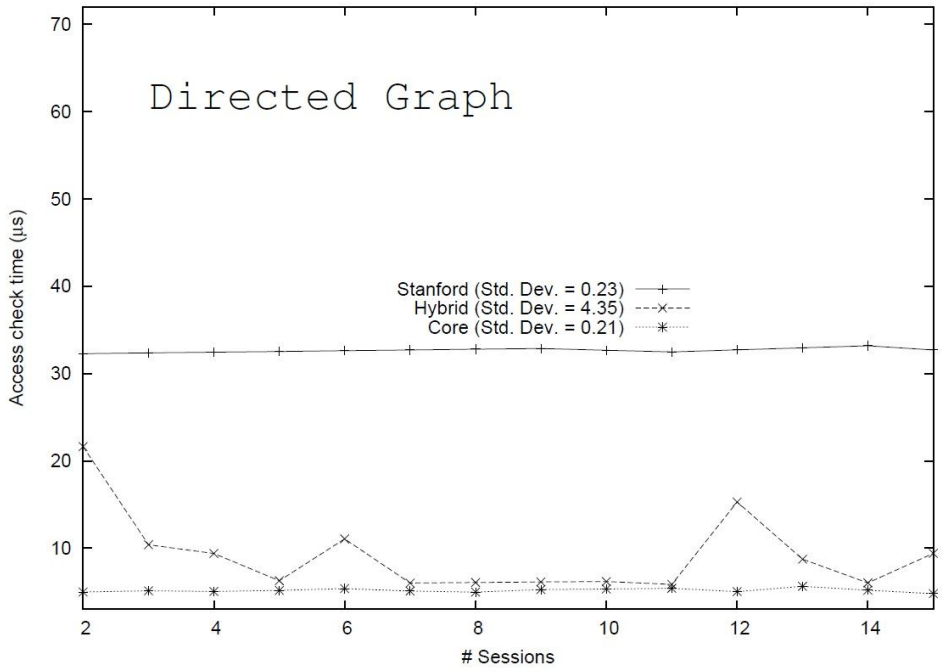


Figure 12 - Average access check time in µs and the corresponding standard deviation for directed graph.

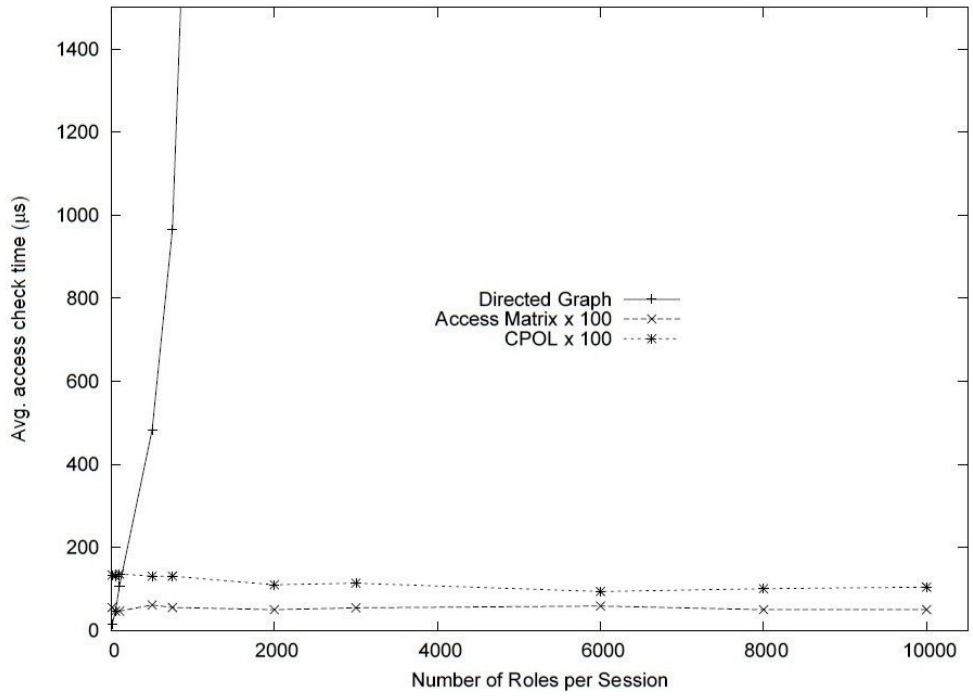


Figure 13 – Time efficiency for small (10) to large (10,000) numbers of roles in a session.

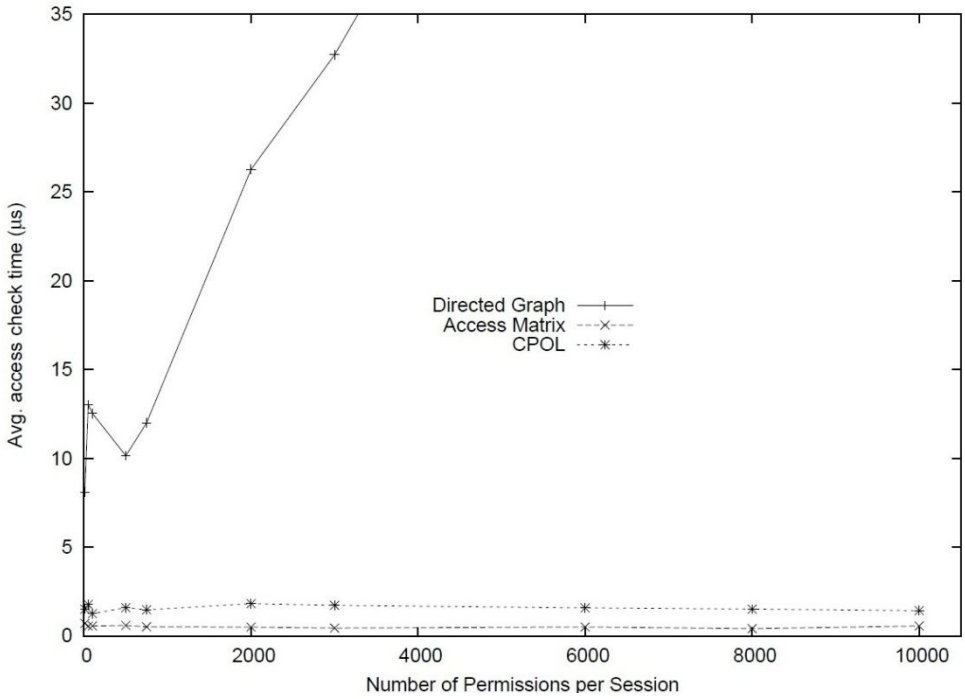


Figure 14 - Time efficiency for small (10) to large (10,000) numbers of permissions in a session.

Space efficiency In this section, we analyze the space-efficiency of the three approaches. We base our assessment on what we have observed from our implementations, and an analysis of the data structures. The space needed for a directed graph grows linearly with the number of sessions. In the worst-case, it can also grow linearly with the number of permissions and roles per session. However, on average, the size of the directed graph is constant in the number of permissions and roles. This is because we expect roles and permissions to be shared by several sessions.

The access matrix is highly space inefficient. The reason is that it grows quadratically with the number of sessions and the number of permissions to which any session is authorized. CPOL is linear in the number of sessions. It is linear also in the number of permissions per session, and therefore not as space efficient as the directed graph. It is agnostic to the number of roles in a session. In Figures 15, we present graphs that capture the above discussion. The graphs have been generated based on our implementations. The reason the access matrix is highly space-efficient for small numbers of sessions is that it is a bit matrix. However, as the number of sessions and permissions per sessions grow, its (quadratic) growth quickly negates the fact that each entry in the matrix is only a bit.

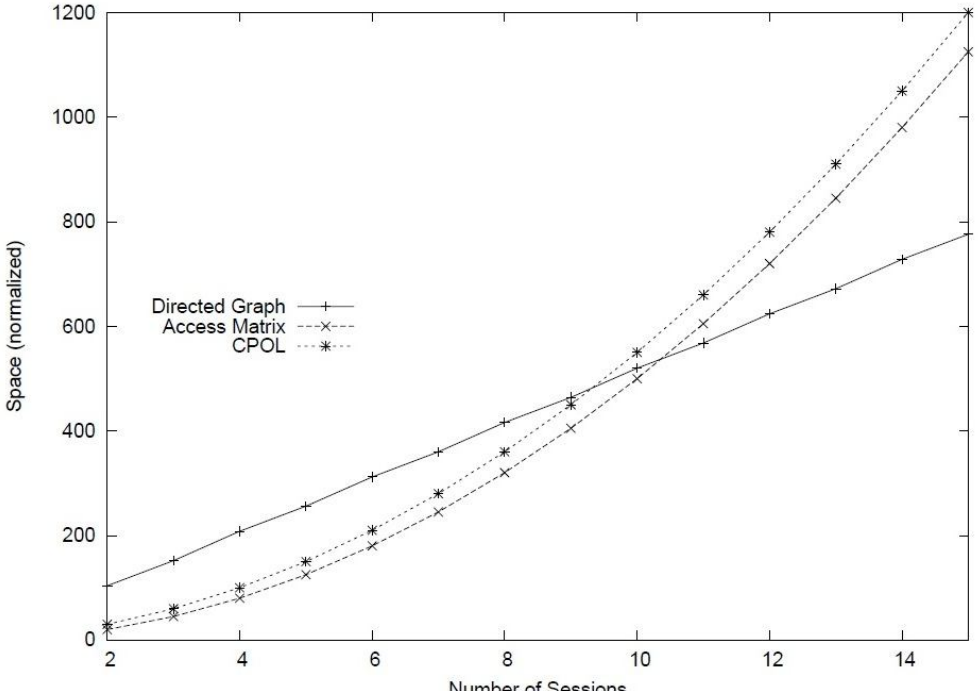


Figure 15 – The space efficiency of our approaches. In our data set that we used to generate this graph, the number of roles and permissions grows by a constant factor per session. We show that access matrix and CPOL are space inefficient, while directed graph is space efficient.

Administrative efficiency An administrative change is the addition or deletion of a user-role, role-role or permission-role relationship in an RBAC policy. The addition of a user at the PDP has no impact on an SDP. However, the removal of a user may impact an SDP, as that user's sessions need to be removed. This impact is linear in the number of sessions in the worst case for the directed graph, quadratic in the number of sessions and permissions in the worst case for the access matrix and linear in the number of sessions in the worst case for CPOL.

The addition or removal of a permission can impact an SDP. The impact is constant-time for the directed graph, linear in the worst case in the number of sessions for the access matrix and linear in the number of sessions for CPOL. The addition or removal of a role can authorize or forbid a session, respectively, to several permissions. We can infer the impact on the three approaches from our discussions on permissions.

In Table 5, we show the results of a proportional mix of administrative changes. The research literature on RBAC administration has focussed mostly on user-role changes, presumably because these are the most frequent in real-world deployments. We assume that 75%

of the changes are to user-role relationships. We conjecture that permission-role changes are the next most frequent (20%) and changes to roles are infrequent (5%). In our experiments, sessions overlap with one another in terms of permissions and roles to a constant factor.

	Directed graph	Access matrix	CPOL
100	13.45	2934.00	321.75
200	22.20	9003.60	1644.00
300	39.15	1741.05	5439.30
400	45.80	3748.80	5053.00
500	38.50	25097.25	12567.25
600	87.30	20488.20	3492.00
700	108.85	18676.70	1737.05
800	142.00	33686.40	7352.00
900	151.65	17543.25	17145.00
1000	158.50	31068.00	6800.00

Table 5 - The administrative overhead on a Core RBAC policy. We assume a proportion of 75% changes to user-role relationships, 20% to role-permission relationships, and 5% to role-role relationships. The number of sessions is 1000, and every user has at least one session.

From the results from Table 6, we can see that the size of RBAC plays important role. For example, if the deployment is small in the size of the RBAC policy (e.g., only up to 100's of roles and permissions), then the access matrix is the best choice. However, if the deployment gets larger, then space and time efficiency gives poor results for access matrix. CPOL has the same scenario. While it performs good in terms of access checking times, it fails when it comes to administration and space efficiency considerations. If there is a need for balance reasonable space and access check time with ease of administration, then the directed graph is a good choice.

We have assessed the three approaches to distributed access enforcement in RBAC. Our approach is empirical, and we have proposed and used a benchmark as the basis. Based on our quantitative results, we are able to provide guidance on the best approach from among the three for particular RBAC deployments.

		Directed graph	Access matrix	CPOL
Time	Inter-session	fair	good	good
	Intra-session	poor	good	good
Space		fair	poor	poor
Admin		good	poor	poor

Table 6 - Our rating of "good", "fair", "poor" for each approach that we assess. While we argue that these ratings follow from our quantitative observation, they are somewhat subjective.

5. Conclusion

We have designed and implemented a platform for assessing distributed access enforcement in RBAC. We have validated it by using it to assess three approaches, one of which is CPOL [6], the state of the art in distributed access enforcement. Our assessment has provided a somewhat surprising result that CPOL is not necessarily the best choice for access enforcement in settings that are typical for RBAC.

In future work, we plan to explore the use of our platform for assessing approaches to access enforcement in contexts other then RBAC, such as trust management. Also, we plan to explore new trust models between the PDP and the SDP. We may perceive the PDP as outsourcing access enforcement to the SDP, but not fully trusting the SDP to, for example, keep the access state secret.

Bibliography

[1] Personal Communication, Open Text Corporation, Aug. 2010.

[2] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman, *"Role-based access control models," IEEE Computer*, vol. 29, pp. 38-47, February 1996.

[3] D. F. Ferraiolo, D. R. Kuhn, and R. Chandramouli, *Role-Based Access Control*. Artech House, Apr. 2003.

[4] Q. Wei, J. Crampton, K. Beznosov, and M. Ripeanu, "Authorization Recycling in RBAC Systems," in *Proceedings of the 13th ACM Symposium on Access Control, Models and Technologies (SACMAT'08)*, pp. 63-72, 2008.

[5] M. Tripunitara and B. Carbunar, "Efficient Access Enforcement in Distributed Role-Based Access Control (RBAC) Deployments," in *Proceedings of the 14th ACM Symposium on Access Control, Models and Technologies (SACMAT'09)*, pp. 155-164, 2009.

[6] K. Borders, X. Zhao, and A. Prakash, "Cpol: High-performance policy evaluation," in *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS'05)*, pp. 147-157, 2005.

[7] S. Wilson and J. Kesselman, Java *Platform Performance: Strategies and Tactics*. Prentice Hall, May 2000.

[8] Y. Liu, C. Wang, M. Gorbovitski, T. Rothamel, Y. Cheng, Y. Zhao, and J. Zhang., "Core role-based access control: Efficient implementations by transformations," *PEPM'06: Proceedings of the 2006 ACM SIGPLAN symposium on Partial Evaluation and semantics-based Program Manipulation*, pp. 112-120, May 2006.

[9] G. S. Graham and P. J. Denning, "Protection — principles and practice," in *Proceedings of the AFIPS Spring Joint Computer Conference*, vol. 40, pp. 417-429, AFIPS Press, May 16-18 1972.

[10] M. A. Harrison, W. L. Ruzzo, and J. D. Ullman, "Protection in operating systems," *Communications of the ACM*, vol. 19, pp. 461-471, Aug. 1976.

[11] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. The MIT Press, 3 ed., Sept. 2009.

[12] "Global 500." Fortune Magazine, 2010. Available from http://money.cnn.com/magazines/fortune/global500/2010/.

[13] Q. Yao, A. An, E. Terzi, and X. Huang, "Finding and analyzing database user sessions," *Proceedings of the 10th International Conference on Database Systems for Advanced Applications (DASFAA)*, 2005.

[14] M. Komlenovic, M. Tripunitara, T. Zitouni, "An Empirical Assessment of Approaches to Distributed Enforcement in Role-Based Access Control (RBAC)," *accepted to appear, ACM Conference on Data and Application Security and Privacy (CODASPY)*, Feb. 2011. (12 pages)

[15] A. Kern, M. Kuhlmann, A. Schaad, and J. Mo_ett, "Observations on the role life-cycle in the context of enterprise security management," *7th ACM Symposium on Access Control Models and Technologies*, June 2002.

[16] A. Schaad, J. Mo_ett, and J. Jacob., "The role-based access control system of a european bank: A case study and discussion," *proceedings of ACM Symposisum on Access Control Models and Technologies*, pp. 3-9, May 2001.

[17] A. Kern, "Advanced features for enterprise-wide role-based access control," *Proceedings of the 18th Annual Computer Security Applications Conference*, pp. 333-343, December 2002.

[18] D. Zhang, K. Ramamohanarao, S. Versteeg, and R. Zhang., "Rolevat: Visual assessment of practical need for role based access control," *ACSAC*, pp. 13-22, 2009.

[19] J. Vaidya, V. Atluri, and J. Warner, "Roleminer: mining roles using subset enumeration," *Proceedings of the 13th ACM conference on Computer and communications security (CCS'06)*, pp. 144-153, 2006.

[20] D. Zhang, K. Ramamohanarao, T. Ebringer, and T. Yann, "Permission set mining: Discovering practical and useful roles," ACSAC'08: *Proceedings of the 2008 Annual Computer Security Applications Conference*, pp. 247-256, 2008.

[21] I. Molloy, N. Li, T. Li, Z. Mao, Q. Wang, and J. Lobo, "Evaluating role mining algorithms," *Proc. ACM Symposium on Access Control Models and Technologies (SACMAT)*, pp. 95-104, 2009.

[22] C. Blundo and S. Cimato, "A simple role mining algorithm," Proceedings of the 2010 ACM Symposium on Applied Computing, pp. 1958-1962, 2010.

[23] M. Frank, A. Streich, D. Basin, and J. Buhmann, "A probabilistic approach to hybrid role mining," *Proc. 16th ACM conference on Computer and Communications Security (CCS)*, pp. 101-111, 2009.

[24] I. Molloy, H. Chen, T. Li, Q. Wang, N. Li, E. Bertino, S. Calo, and J. Lobo, "Mining roles with semantic meanings," *Proc. ACM Symposium on Access Control Models and Technologies (SACMAT)*, 2008.

[25] M. Jafari, A. Chinaei, K. Barker, and M. Fathian, "Role mining in access history logs," *Journal of Information Assurance and Security*, 2009.

[26] J. Crampton, "On permissions, inheritance and role hierarchies," in *Proceedings of the Tenth ACM Conference on Computer and Communications Security (CCS-10)*, pp. 27-31, ACM Press, Oct. 2003.

[27] A. Georges, D. Buytaert, and L. Eeckhout, "Statistically rigorous java performance evaluation," *Proceedings of OOPSLA'07*, pp. 57-76, May 2007.

[28] Cormen, Thomas H., Leiserson, Charles E., Rivest, Ronald L. and Stein, Clifford (2001). *Introduction to Algorithms* (2nd ed.). MIT Press and McGraw-Hill. ISBN 0-262-53196-8.

[29] T. Zitouni, "A Statistically Rigorous Evaluation of the Cascade Bloom Filter for Distributed Access Enforcement in Role Based Access Control (RBAC) Systems." MASc Thesis, University of Waterloo, December 2010.

Appendix A

```
package Structures;

public abstract class Vertex {

	public static int staticId;

	public int id;

	public ArrayList<Vertex> neighbours;

	public Vertex() {...}

	public int getId() {...}

	public void addNeighbour(Vertex v) {...}

	public void removeNeighbour(Vertex v) {...}

	public List<Vertex> getNeighbours() {...}

	public int getNumNeighbours() {...}

	public Vertex getNeighbour(int i) {...}

	public abstract String getStringId();

}

public class UserVertex extends Vertex {

	private String userId;

	public String getUserId() {...}

	public String getStringId() {...}

	public UserVertex(String userId) {...}

}

public class RoleVertex extends Vertex {

	public int level;
```

```
        private String roleId;

        public RoleVertex(String roleId) {...}

        public int getLevel() {...}

        public void setLevel(int level) {...}

        public String getRoleId() {...}

        public String getStringId() {...}

        public String getRoleType() {...}

}

public class PermissionVertex extends Vertex {

        private String permissionId;

        public String getPermissionId(){...}

        public String getStringId(){...}

        public PermissionVertex(String permissionId) {...}

        public void addNeighbour(Vertex v) {...}

        public void removeNeighbour(Vertex v) {...}

}
```

Appendix B

```
package Cpol;

public class SessionCPOL extends Session {

	public Rule rule;

	public SessionCPOL(Rule rule) {...}

	public Rule getRule() {...}

	public void setRule(Rule rule) {...}

}

public class Rule {

	public static int ruleId;

	public int id;

	public String owner;

	public String licencee;

	public AccessToken accesstoken;

	public boolean condition;

	public Rule(String sessionId) {...}

	public Rule(String sessionId, AccessToken A) {...}

	public Rule(int id, String owner, String licence, AccessToken
accessToken, boolean condition) {...}

	public void AddRule(int requester, String owner, String licence,
AccessToken accessToken, boolean condition) {...}

	public boolean removeRule() {...}

	public int getId() {...}
```

```
	public void setId(int id) {...}

	public String getLicencee() {...}

	public void setLicencee(String licencee) {...}

	public AccessToken getAccesstoken() {...}

	public void setAccesstoken(AccessToken accessToken) {...}

	public boolean getCondition() {...}

	public void setCondition(boolean condition) {...}

	public String getOwner() {...}

	public void setOwner(String owner) {...}
}


public class Cache {

	public ArrayList<CacheEntry> cache;

	public Cache() {...}

	public Cache(ArrayList<CacheEntry> cache) {...}

	public ArrayList<CacheEntry> getCache() {...}

	public void setCache(ArrayList<CacheEntry> cache) {...}

	public void addEntry(CacheEntry entry) {...}

	public void removeEntry(CacheEntry entry) {...}

	public void invalidateEntry(int sessionId) {...}

	public boolean isEntry(int sessionId, int permissionId) {...}
}
public class CacheEntry implements SDPDataStructure {

	public CacheKey key;
```

```
	public AccessToken accessToken;
	public boolean condition;
	public CacheEntry(CacheKey key, AccessToken accesToken) {...}
	public CacheKey getKey() {...}
	public void setKey(int key) {...}
	public AccessToken getA() {...}
	public void setA(AccessToken accessToken) {...}
	public boolean getCondition() {...}
	public void setCondition(boolean condition) {...}

}

public class CacheKey {
	public int sessionId;
	public CacheKey(int key) {...}
	public int getKey() {...}
	public void setKey(int key) {...}
}
public class AccessToken {
	public Set permissions;
	public AccessToken() {...}
	public AccessToken(Set permissions) {...}
	public Set getPermissions() {...}
	public void setPermissions(Set permissions) {...}
```

```
    public boolean checkPermission(int permissionId) {...}

    public void add(AccessToken accessToken) {...}

}
```