Twesha Mitra
CS 1699
Project 2

To run this application, simply run "./run.sh <policy file>" or the following:

```
javac  -cp  .:json-simple-1.1.1.jar  Enforce.java  Policy.java
User.java Rule.java Resource.java

java -cp .:json-simple-1.1.1.jar Enforce <policy file>
```

Since I have a .jar file for JSON parsing, it needs to be included in the classpath when compiling and running, hence the script. The user must specify a JSON policy file that is in the current directory. If you are going to run one of the provided policy files, I would suggest policy1.json as it is easier to verify than the others.

Task W0:

This system implements an attribute based access control model. Consider that this system is built for a company with departments, positions, projects, and resources. Such attributes make up the ABAC model that has been implemented. The model support indirection, delegation, attribute inference, and attribute intersection.

Task W1:

The policy file that the user has to write must be in JSON format. There is only one file with users, resources, accesses that provide the rules, and delegations. So, the user must include a file with all users, subjects, and accesses and the ordering is important. First, one must specify all the users, then resources, then rules, and finally, any delegations necessary. The objects for each should be in the following format (with actual values instead of empty strings):

```
[{
                "user":" ",
                department":" ",
                "position":" ",
                "project":" ",
        },
        {
                "resource":" ",
                "project":" ",
        },
        {
                "access":" ",
        },
        {
                "delegateAuthAttr":" "
}]
```

The user object defines attributes of a user. Assuming this system is built for a company to use, the attributes that are used to determine access are user name/ID, position, project, and department. The user identification can be stored in any format. A company could store the user's names, EIDS, or SSN as long as they are in string format. The resource object defines a resource's attributes. For example, there could be a file that is a license that belongs to a specific project. Access and delegation objects allow the user to create rules, attribute intersections, indirection, delegation, and attribute inference. These features are discussed later in this paper.

The "access" attribute provides the rule. It uses attribute inference, attribute intersection, and indirection to assign permissions. For example, consider that a user types in the following:

{"access": "file1/read<-position/developer&&project/project1"}

This can be translated to "All developers who are working on project1 can read file1". In this scenario, the way permissions are granted is based on position and project attributes of a user. This single rule encapsulates indirection, attribute inference, and attribute intersections. By creating such a rule, we can assign a large set of permissions at once. Instead of having to go through all the users and assigning them permissions, the system simple assigns all developers with specific attributes. Additionally, the rule allows for attribute intersection by allowing access to a resource if a user holds a certain position and is working on a certain project. This, in turn allows for attribute inference. We can infer from the rule that if a user holds the specified position and is working on the project that, he/she can access that file. Delegation of attribute authority is also supported in the system. If a delegation is specified, the 'delegator' access rights will be granted to the 'delegatee' if they are both working on the same project. In English, you could say this as "Manager _managerX can assign permissions to developers in Project _projectX".

Task W3:

Indirection is automatically supported when the user writing the policy file creates a rule using the "access" value. With this syntax, the user can easily assign permissions to a large group of people. Consider the following example:

{"access": "file1745/write<-position/projectManager&&project/project1"}

This would imply that the system should grant write access to any project managers of project1. Let's say another resource called "file100" is added to project1 and all developers working on project1 should be able to read it. Then, instead of going through and assigning permissions to each user, the administrator writing the file can create the following:

{"access":"file100/read<-position/developer&&project/project1"}

Now, all developers who are currently working on project1 can read this new file that was added and it only took one line to assign those permissions. The generic syntax would look like this:

{"access":"<resource>/<action><-position/<positionValue>&&project/<projectVal>"}

Task W4:

Delegation is supported through “delegateAuthAttr”. An admin can write a delegation rule that states that a manager on a certain project can assign permissions to his/her developers who are working on that specific project. For example, consider the following:

{“delegationAuthAttr”:”project2/projectManager<-project2/manager”}

This implies that a manager of project2 can assign its own permissions to project managers on project2. This makes it easy for example, for a new employee to get access to documents they need. Their manager would simply write this delegation and they would get this access. Another example that is in one of the policies is:

{“delegationAuthAttr”:”project1/developer<-project1/manager”}

The generic syntax to write one of these is:
{“delegationAuthAttr”:<project>/<position><-<project>/<position>”}, where the projects must match. Otherwise, the delegation will not occur.

Task W5:

The two other access control features that are supported are attribute intersection and attribute inference. Attribute inference allows us to infer access if a user has a certain attribute. For example, we can have a rule that states that, if you are a developer, you can access file1. Since, this system grants access based on position and project attributes, attribute intersection is used to make sure access is only granted if a user has both attributes. For example, consider the following:

{“access”: “payroll_document/read”<-department/humanRes&&position/specialist”}

In plain English, this would read “Only a specialist in the human resources department can read the payroll document”. This would also imply that, if you are a specialist in the human resources department, we can infer that you can read the payroll document. Such a rule is not yet supported in this system, but I hope to build upon this as I have already added these attributes to my model. One rule that is used in this system is:

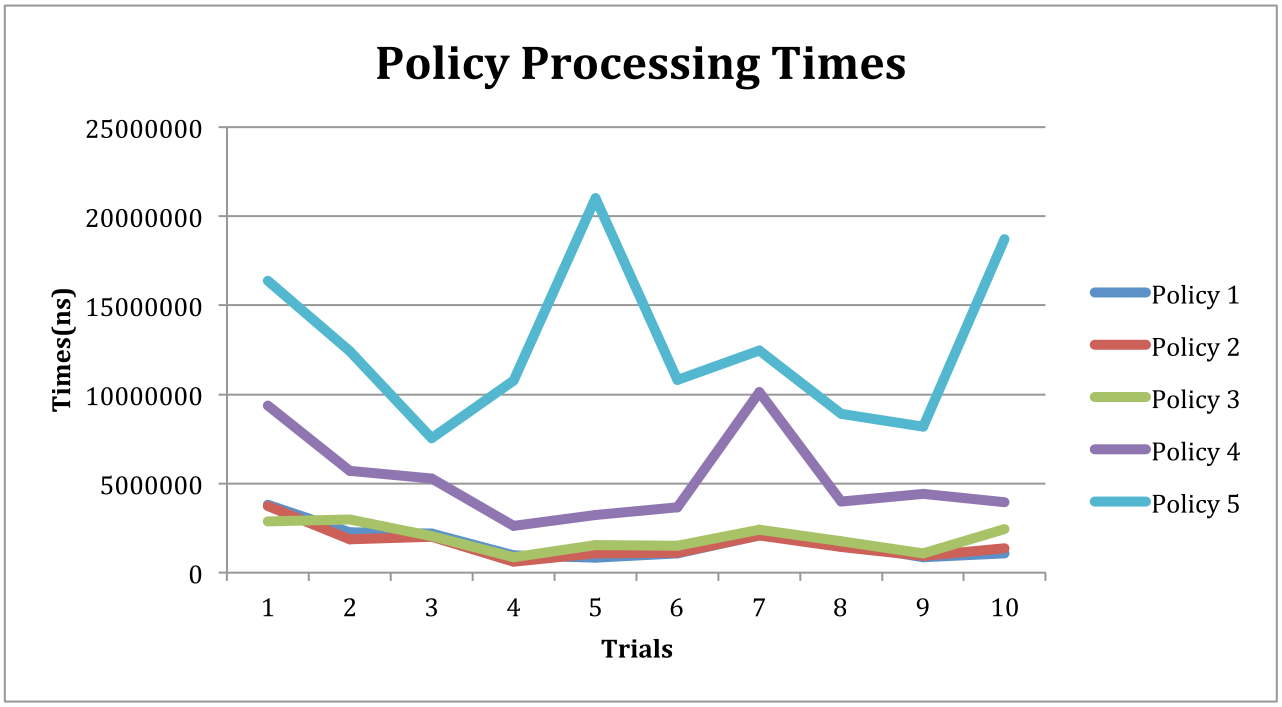
{“access”:”file1/write<-position/manager&&project/project1”}

We can translate this to plain English by saying “A manager of project1 can write to file1”. This access is granted only if the person making the request is a manager AND is on project1. This displays the attribute intersection. Additionally, from this example we can say that if a user is a manager and he/she is on project, then he/she can write to file1 and this displays attribute inference. These two features arose naturally from this ABAC model and the syntax allows the administrator to achieve both in one line. Again, the syntax is:

{“access”:”<resource>/<action><-position/<positionValue>&&project/<projectVal>”}

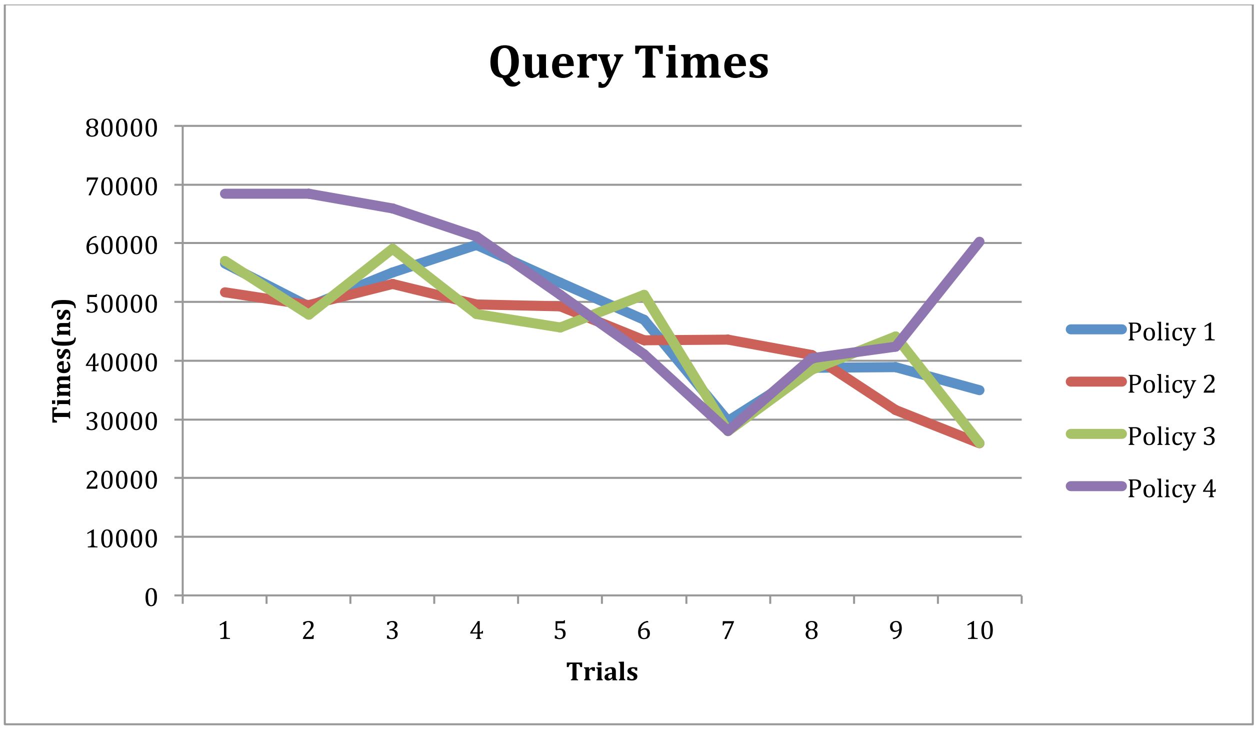
Task W7 : Policy File Processing Times

Trials	Policy 1	Policy 2	Policy 3	Policy 4	Policy 5
1	3798153	3756345	2878643	9392890	16384571
2	2268656	1886406	11187388	5712419	12419678
3	2183262	2029747	2055774	5290753	7540773
4	974848	611767	853801	2618560	10781876
5	840826	1091318	1543802	3231534	20998008
6	1089748	1099808	1515642	3657908	10823906
7	2082332	2090555	2396596	10123597	12461831
8	1638056	1436710	1769768	3986450	8925852
9	851133	940138	1075292	4432609	8202303
10	1092091	1354758	2449629	3964217	18714800



Task W7: Query Processing Times

Trial	Policy 1	Policy 2	Policy 3	Policy 4
1	56,535	51,676	57,045	68,403
2	49,261	49,466	47,835	68,457
3	54,984	53,111	59,037	65,920
4	59,664	49,600	47,904	61,174
5	53,277	49,270	45,624	51,160
6	46,992	43,463	51,194	41,044
7	29,708	43,591	28,023	28,033
8	38,800	40,949	38,413	40,449
9	38,939	31,617	44,124	42,383
10	35,013	25,889	25,955	60,259



These graphs relay a lot of information about this system. I constructed 5 policies of different sizes to see the effect of size on processing. I also varied some complexity in these files. As the graph titles “Policy Processing Times” shows, the time is took for the system to process policy5.json was considerably higher than that of other policies. For all 10 trials, the processing time for policy 5 was higher. The graph helps us see the trend and the table is provided in case one needs to look at the exact numbers. All other policies were parsed much faster than Policy 5. This is probably due to the fact that I am searching through arrays before I add a rule, a user, etc. These are O(n) operations that slow down the processing of a policy. These times were measured in nanoseconds because the processing was being done almost instantly and I need to get as small a unit as possible. In an enterprise situation, in which a company could have thousands of users, millions of documents, and several hundred thousand permissions, this could definitely take a while, but the graph titled “Query Times” shows the benefit of this system.

As one can see on the graph, access queries took about the same amount of time throughout the trials. This probably stems from the fact that I use hashmaps in my lookup that allow for O(1) runtimes. There is no need to look through a very large ACL matrix to check access. So, there is tradeoff with this system. It is evident that incredibly large policy files such as ones that large enterprises use could take a long time to process. But, on the other hand, due to the efficient lookup schema, access queries take a small amount of time to execute.

In all, attribute based access control allows us to use pre existing attributes of users to grant them access. Companies already store such attributes about their employees, so this information can be fed into this ABAC model to make access control decisions. It is a powerful model that allows for fast queries. In the future, I would like to work towards speeding up the processing of the policy file and adding on more attributes of users into the model.