ENTERPRISE FILE-SHARING SYSTEM WITH LIGHTWEIGHT ATTRIBUTE-BASED ACCESS CONTROL

Zhi XIONG[1], Ting GUO[2]*, Changsheng ZHU[3], Weihong CAI[4], Lingru CAI[5]

Attribute-based access control (ABAC) bases on attributes to define access rules and relies on them to make authorization decisions. The existing ABAC schemes have two deficiencies: low efficiency in rule execution and high difficulty in rule writing. We propose a lightweight ABAC scheme. It uses Python logical expression to describe access rule and uses the eval function to execute rule. We also design some mechanisms to simplify rule writing. Test results show that our rule can overcome the above two deficiencies. Based on the ABAC scheme and Samba, we build an enterprise file-sharing system and present its access control effect.

Keywords: attribute-based access control; file-sharing; lightweight; access rule

1. Introduction

An enterprise has a large quantity of documents, material, software and other files that need to be shared between its employees. It is inconvenient to share these files through Email or removable storage devices, so it is necessary to build an enterprise file-sharing system. An enterprise usually has many employees and therefore the system has many users. Consequently, for security reasons, the system must control access to file resources. Furthermore, the system must be easy to use. Specifically, it would be preferable if users can access the shared files online and can use the system without installing additional client software and without an advanced level of expertise.

Currently, mainstream file-sharing methods for an enterprise are: FTP, NFS, Samba [1], and cloud storage. However, FTP is not convenient, because users must first download files from file server then read them, and first modify files then upload them to file server. NFS and Samba can solve the problem, and they allow users to access files online. Thereinto, NFS client needs to be

[1] Department of Computer Science, Shantou University, Shantou, Guangdong, China, e-mail: zxiong@stu.edu.cn
[2]* Department of Computer Science, Shantou University, Shantou, Guangdong, China, Corresponding author, e-mail: yb_yb163@163.com
[3] Research Division, Shantou University, Shantou, Guangdong, China, e-mail: cszhu@stu.edu.cn
[4] Department of Computer Science, Shantou University, Shantou, Guangdong, China, e-mail: whcai@stu.edu.cn
[5] Department of Computer Science, Shantou University, Shantou, Guangdong, China, e-mail: lrcai@stu.edu.cn

Unix/Linux, but Samba client can be Windows. Especially, Windows users can access the shared files in Samba through network neighborhood but need not to install any client software. So, Samba is more suitable for an enterprise to share files. Besides, many Internet operators provide enterprise cloud storage service, also known as enterprise network disk. Despite the low cost of using enterprise cloud storage, it has potential risk of data breaches because critical enterprise data is placed on the operator's servers.

In the aspect of access control, Samba only supplies very simple control scheme and lacks flexibility. Some products of enterprise cloud storage provide basic access control function, but their control schemes mainly all adopt DAC (Discretionary Access Control) or RBAC (Role-Based Access Control) model. However, the two models cannot cope with the sharp increase of users and resources and cannot supply fine-grained access control. ABAC (Attribute-Based Access Control) [2] is a distinct access control model because it controls access to resource by evaluating access rules against the attributes of the entities (such as subject, resource and environment). Especially, if identity, role and resource security level are also abstracted as entity attributes, ABAC model is able to enforce traditional IBAC (Identity-Based Access Control), RBAC and MAC (Mandatory Access Control) models, respectively. Because of this flexibility, ABAC provides an ideal access control scheme for open network environment. If applying ABAC to an enterprise file-sharing system, the access rules must meet the following requirements: (i) every user operation needs access decision, so rule parsing and execution cost must be small; (ii) the rules may be written by normal users, so they must be easy and convenient to write; (iii) to achieve fine-grained access control, the rules must have strong expressivity; and (iv) adding or deleting user attributes (user attributes may be defined by administrator) should not need to change system code, so rule description should be loosely coupled with system code. Aimed at these requirements, we propose a lightweight ABAC scheme. Based on the ABAC scheme and combined with Samba and other open-source software, we build an enterprise file-sharing system. The system is not only safe, but also easy to use. The rest of this paper is organized as follows. The related works are introduced in Section 2. Section 3 gives system design. In Section 4, we propose the lightweight ABAC scheme. Section 5 introduces system implementation. Examples and tests are given in Section 6. Finally, we conclude in Section 7.

2. Related Works

In ABAC, the description, expressivity and execution efficiency of access rules are three key issues. XACML (eXtensible Access Control Markup Language) is an attribute-based access control policy language and processing model [3]. Many researches [4, 5, 6, 7] about ABAC all focus on XACML.

XACML uses XML (eXtensible Markup Language) to describe requests and rules. However, it is very costly to generate and parse XML. In XACML, a rule may trigger a set of rules, and these rules may give contradictory decisions, so combining algorithm is used to arrive at a final access decision, and many works [8, 9] study on it. This also introduces overhead. Furthermore, XACML uses complex XML tags to describe rules, so it is hard for normal users to write rules.

Attribute-based encryption [10] is a type of public-key encryption in which the secret key of a user and the ciphertext are dependent on attributes. In such a system, the decryption of a ciphertext is possible only if the set of attributes of the user key matches the attributes of the ciphertext [10]. So, attribute-based encryption can be deemed as a kind of ABAC scheme. However, the expressivity of its rule is very weak, because the rule just can express an entity with or without some attributes but cannot compare an attribute with a number or string. Rule engine [11] is a software system that executes some business rules in a practical runtime environment. It enables enterprise policies and operational decisions to be defined, tested, executed and maintained independently from application code [11]. With the aid of rule engine, we can describe access control rules [12, 13]. However, rule engine is not designed specifically for access control, but it is a very complicated system and its running overhead is excessive.

Some works [14, 15] themselves define the form of access rules, and realize the parsing and execution of access rules. It is a significant amount of work. They do not evaluate the execution efficiency of their schemes. ABAC is a powerful access control model, but suffers from a few drawbacks, such as lower decision efficiency and rule explosion [16]. Many works [5, 16, 17] introduce role to ABAC and propose attribute- and role-based access control to solve these problems. In this paper, we propose a lightweight ABAC scheme which overcomes the two deficiencies of the existing ABAC schemes: low efficiency in rule execution and high difficulty in rule writing. Moreover, its rule has strong expressivity. Test results demonstrate the feasibility of our scheme.

3. System Architecture Design

The architecture of our system is given in Fig. 1. Specifically,

(i) Enterprise files are stored in MooseFS [18], which is a distributed file system. User attributes and access rules are stored in MongoDB [19].

(ii) File-sharing system uses Linux server. We create a directory “/mnt/mfs” in the server, and mount MooseFS to the directory by FUSE (Filesystem in USErspace). In Samba, we share the directory.

(iii) Windows users can access the shared directory through network neighborhood, and Linux users can access the shared directory through smbmount. Users can manage user attributes and access rules by web browsers.

(iv) File access, as well as attribute and rule management, must get permission from access controller. The access controller makes decision according to access rules.

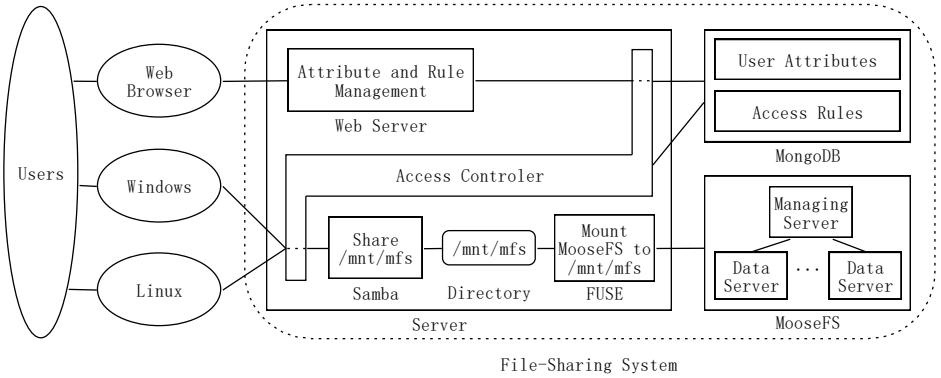

Fig. 1. Architecture of enterprise file-sharing system

4. Lightweight ABAC Scheme

4.1. Entity and Entity Attribute

In our scheme, access rule allows the use of three entities: subject, resource and environment. Subject denotes user. Resource is the file or directory being accessed, and each resource corresponds to a unique path. Environment denotes the context of the access, such as user IP address and access time.

Different types of enterprises need different user attributes, so our system allow the enterprise to self-define subject attributes. The system has an administrator user (admin), who has the highest permission level. Subject attributes are defined by the administrator, and he is also responsible for maintaining the attribute values of every user account. Resource attributes and environment attributes are defined by the system. Resource attributes include owner and security level. Environment attributes include user IP address, access date and access time.

4.2. Permission Types

Access rules are made by the administrator or normal users. Excessive permissions will make rule-making complex and prone to error, so we just define three permissions: read, write and manage. Table 1 gives the corresponding operations of each permission. The write permission for a directory is meaningless, but we allow a file/directory inherits the permission from its parent directory (see Section 4.6), so the write permission for a directory is still useful.

4.3. Rule Description

Each permission of each resource corresponds to an access rule. We use a logical expression to describe an access rule. A rule may consist of subject,

resources and environment attributes, arithmetic, logical, relational and set operators, numeric and string constants, and so on. The execution result of a rule is a boolean value, and *True* denotes permission while *False* denotes denial.

Table 1

The corresponding operations of each permission

Resource	Read	Write	Manage
Directory	List the files and directories within the directory	\	Rename or delete the file/directory. Modify the attributes or access rules of the file/directory.
File	Read the file	Write the file	

In an access rule, an entity (namely subject, resource or environment) is denoted by a dictionary variable, and an entity attribute is denoted by an item in the dictionary. The key of item corresponds to attribute name, and the value of item corresponds to attribute value. The reason why we use dictionary but not class (the rule engine Drools uses class, see Section 6.1) to denote entity is that the structure of a dictionary needs not to be defined in advance, but a class needs. This way, we can add or delete some entity attributes without touching any system code. We also provide that the subject, resource and environment are represented by the variable *S*, *R* and *E*, respectively. For example,

(S['Username']==R['Owner']) or (E['UserIP']=='192.168.1.111')

and

(S['Title'] in ['Professor', 'Associate Professor']) and (R['SecurityLevel']<=2)

are two legitimate access rules.

4.4. Common Functions

In access rules, we can use the built-in functions of the programming language, e.g., *round* and *min*. In order to further enhance rule's expressivity, the system also defines some common functions, and allow users to use these functions in the rules. For example, *RegExpMatch* and *WeekDay* are two functions defined by the system, *RegExpMatch* is regular expression matching function, and *WeekDay* is used to get the day of the week. Then,

(RegExpMatch(E['UserIP'], '^192\.168\.1\.')) and (WeekDay(E['Date'])==5)

is a valid rule.

4.5. Rule Call

The intention of rule call is to facilitate the reuse and writing of rules. We can store some frequently-used rules as callee rules, and call (namely contain) them in the access rules of resource. This is somewhat similar to procedure call. Each callee rule has a name, and we provide that a callee rule is called by using "{#RuleName#}". For example, there are two callee rules:

OwnerAceess: S['Username']==R['Owner'],

StaitcIP: RegExpMatch(E['UserIP'], '^192\.168\.1\.[1-9][0-9]$'),

then

{#OwnerAceess#} and {#StaitcIP#}

is a valid rule.

As mentioned in Section 2, many works introduce role to ABAC so as to overcome the disadvantages of ABAC. The function of role is easy to implement through rule call. For example, the callee rule "*CSStaff: S['Department']=='Computer'*" can be used to denote the role of "the staff in computer science department", and we can contain "*{#CSStaff#}*" in a rule to authorize the role of "the staff in computer science department".

4.6. Rule Inheritance and Reference

Enterprise files are usually organized into a tree-like hierarchical structure. Moreover, we have the following considerations: (i) The permissions of a resource (directory or file), most of the time, inherit the permissions from its parent resource (directory); (ii) If one has read permission to a resource, he usually has to have read permission to its parent resource; (iii) If one has write permission to a resource, he usually also has write permission to its sub resources; (iv) If one has write permission to a resource, he usually has to have read permission to the resource; (v) Manage permission is similar to write permission.

Hence, the system defines three fields for each permission: inherit, reference (read permission does not need this field) and rule. Field inherit and reference hold values of the boolean type, and field rule holds a value of the string type. For each permission of a resource, the relationship between the combination of its field values and its final access rule is given in Table 2.

Table 2

The final access rule of each permission

Permission	Field			Final access rule
	Inherit	Reference	Rule	
Read	True	\	Empty	The final access rule of read permission of its parent resource (by default)
	True	\	Not empty	(The final access rule of read permission of its parent resource) and (the value of rule)
	False	\	Empty	True
	False	\	Not empty	The value of rule
Write and Manage	True		Empty	The final access rule of write/manage permission of its parent resource (by default)
	True		Not empty	(The final access rule of write/manage permission of its parent resource) or (the value of rule)
	False	True		The final access rule of read permission of itself
	False	False	Empty	True
	False	False	Not empty	The value of rule

The inheritance and reference mechanisms of access rules enable the rules of the parent resource to be reused by the sub resources and enable the rule of read permission to be reused by write and manage permission. It can not only significantly reduce the storage space of rules, but also greatly reduce the workload of rule-writing. In addition, for a permission, the case of "the value of field inherit is *True* and the value of field rule is empty" is viewed as the default case, under which we need not store its access rule. Because the case occurs frequently, it can save a large amount of storage space.

By the way, each user operation just corresponds to a final access rule, that is, each user operation just triggers one rule. So, unlike XACML, we do not need combining algorithm to deal with rule conflict.

5. System Implementation

5.1. Rule Description Language and Rule Execution

We use Python logical expression to describe access rule. Due to the flexibility and powerful expressivity of the Python language, its logical expression can easily describe complicated access control rule. Moreover, as long as one masters the basic syntax of Python, he can write access rules.

The *eval* function is used to execute an access rule. Since the rules are written by users, we must prevent them to execute malicious code in the rules, for example using "*__import__ ("os").system(command)*" to delete or modify system files. Therefore, we do not allow "*__import__*" to appear in rules. Further, we limit the variables and functions that can be used in the rules. The *eval(expression[, globals[, locals]])* function takes two extra arguments to allow us to do this. In our scheme, the permissible variables are *S*, *R* and *E*, and the permissible functions are Python built-in functions and the common functions supplied by the system, such as *RegExpMatch* and *WeekDay*.

5.2. Storage of Entity Attributes and Access Rules

Environment attributes need not be stored, and they are dynamically generated. Subject attributes and resource attributes are stored in MongoDB, which is an open-source document database. In MongoDB, a collection is similar to a table in relational databases, and a record in collection is a document which is a data structure composed of field and value pairs.

Resource attributes and access rules are stored together in the same collection, and a resource may correspond to a document in the collection. For example, the document for the root of the shared directory is:

```
{
   Path: "/", Owner: "admin", SecurityLevel: 3,
   Rules: {
      read: {inherit: False, rule: "S['Username']=='admin')"},
```

```
        write: {inherit: False, reference: True},
        manage: {inherit: False, reference: True},
    }
}
```

5.3. Access Controller

Samba is written in the C language and access rules are Python logical expression, so we need a cross-language service development framework to implement access controller. The Apache Thrift [20] is such a software framework, and it supports many languages, including C, PHP and Python. By using Thrift, we only need to define service interface in a .thrift file, then compile the file to source code by Thrift compiler. The generated code can be used to easily build RPC (Remote Procedure Call) clients and servers that communicate seamlessly and efficiently across programming languages. So we use Thrift framework to develop access controller.

In the service interface of the access controller, we just need to define one service method that returns a boolean value. The service interface is as follows:

```
service AccessControl {
   bool CheckPermission(1: string username, 2: string userip,
                        3: string resourcepath, 4: string permission)
}
```

The *CheckPermission* method checks permission by calling the recursive method *Decision(resourcepath, permission, S, E)*. The two methods are written in Python.

5.4. Permission Check in Samba

In our system, the version of Samba is 4.4.9. In the *NTSTATUS smbd_check_access_rights(struct connection_struct *conn, const struct smb_filename *smb_fname, bool use_privs, uint32_t access_mask)* function of smbd/open.c file, we add some code to check permission. Specifically, we call the *CheckPermission* method in the *AccessControl* service interface via Thrift. It's worth mentioning that we can distinguish user operation according to *access_mask* and some rights bits, such as *SEC_FILE_READ_DATA*, *SEC_FILE_WRITE_DATA*, *SEC_DIR_LIST*, and *SEC_STD_DELETE*. If the returned value is *False*, the *smbd_check_access_rights* function returns *NT_STATUS_ACCESS_DENIED* denying this access. In addition, our permission check does not affect the permission check done by Samba itself.

6. Examples and Tests

In this section, we first, combined with practical examples, compare our ABAC scheme with two popular ABAC schemes, namely XACML and rule engine, then we present the access control effect of our system. Note that,

attribute-based encryption is also a kind of ABAC scheme, but its expressivity of rule is very weak (see Section 2), so we do not compare our scheme with it.

6.1. Rule Writing

Suppose we want to describe the rule: user's username equals resource's owner and user's IP address matches a regular expression.

In our scheme, the rule can be described as follows:

```
(S['Username']==R['Owner']) and
    (RegExpMatch(E['UserIP'], '^192\.168\.1\.[1-9][0-9]$'))
```

In XACML, the main code of the policy file that describes the rule is as follows. It can be seen that the rule-writing in XACML is very difficult.

```
<Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:and">
  <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
    <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:string-one-and-only">
      <SubjectAttributeDesignator AttributeId="Username"
          DataType="http://www.w3.org/2001/XMLSchema#string" />
    </Apply>
    <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:string-one-and-only">
      <ResourceAttributeDesignator AttributeId="Owner"
          DataType="http://www.w3.org/2001/XMLSchema#string" />
    </Apply>
  </Apply>
  <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:string-regexp-match">
    … For the limitation of space, this part is omitted.
  </Apply>
</Apply>
```

Drools [21] is an open-source rule engine in Java. If describing the rule in Drools, the main code of the rule file is as follows. We can see that it is easy to describe rules in Drools. However, Drools uses class to denote entity. The structure of a class needs to be defined in advance, so when we add or delete some entity attributes, we have to change system code.

```
when
  $S:Subject()
  $R:Resource(owner == $S.username)
  $E:Environment(userIP matches "^192.168.1.[1-9][0-9]$")
then
  drools.getWorkingMemory().setGlobal("Result", new Boolean(true));
```

The three schemes all not only allow us to use various types of variables, various types of operators, regular expression, etc., in the rules, but also allow us to self-define functions, so the expressivity of their rules is very strong. However, the rule-writing in our scheme is more concise and clear.

6.2. Rule Execution Efficiency

We also use the three schemes to describe the following rule: user's position is manager and resource's security level is no more than 2. This rule is denoted as Rule 2, and the rule in Section 6.1 is denoted as Rule 1. We test the execution time of the two rules in the three schemes on a PC, which runs Windows 7 64bit with Intel Core i5-3470S CPU and 4G memory. The implementation of XACML we used is Sunxacml [22], which is open-source and written in the Java language. The version of Sunxacml, Drools and Java is 2.0, 6.5.0. Finally, and 1.7.0_80, respectively. In our system, the version of Python is 3.6.0. Table 3 gives the test results. When we test the rule execution time in Sunxacml and Drools, the rules are stored in files. However, the time of reading rule file is just about 1ms, so it can be ignored.

Table 3

Rule execution time

Rule	Execution time (ms)		
	Sunxacml	Drools	Our scheme
Rule 1	417	1531	0.03
Rule 2	413	1492	0.02

The rule execution in Sunxacml and Drools is costly, because they need to parse complex rule. In contrast, our scheme needs not parse complex rule (it is very easy to handle rule call), but executes the rule directly, so the execution cost is very small. Specially, Drools is not designed specifically for access control, but is a powerful hybrid reasoning system, so it is very complicated and costly.

6.3. Access Control Effect

We test the access control effect of our system on a Windows 7 client. First of all, it is important to note that we can access the shared files through network neighborhood on Windows systems but need not to install any client software. Fig. 2 to 4 give the test results.

Fig. 2. Login

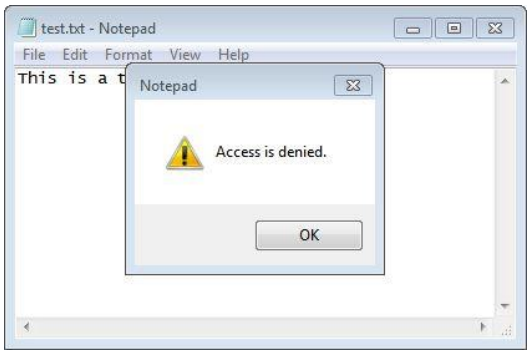

Fig. 3. Have no permission to write a file

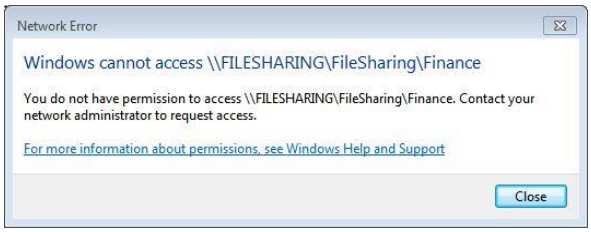

Fig. 4. Have no permission to read a file/directory

Fig. 2 is the login dialog box shown by the OS (Operating System). The login validation is done by Samba, but not our system. If we have no permission to write a file, when we have modified the file and try to save it, the OS will pop up a "Save as" dialog box. If we try to overwrite the original file forcibly, the OS will refuse us to write the file and give the dialog box as Fig. 3 shows. If we have no permission to read a file or directory, but try to read it, the OS will deny our operation and pop up the dialog box as Fig. 4 shows.

For the limitation of space, we do not present more test screenshots. To sum up, our system can enforce access control for the read, write, delete, rename, and so on operations of files and directories

7. Conclusions

In this paper, aimed at the requirements of enterprise file-sharing system, we propose a lightweight ABAC scheme. It uses Python logical expressions to describe access rules, and directly uses the *eval* function to execute rules. Common function mechanism is used to enhance rule's expressivity. We also design some mechanisms to simplify and facilitate the writing of rules, including rule call, inheritance and reference. Based on the ABAC scheme, Samba and other open-source software, we build an enterprise file-sharing system. We give the implementation method of access controller and permission check. The test results show that, in our scheme, the access rule not only has strong expressivity and small execution cost, but also is more concise and easy to write. The test results also show that our system can exactly enforce access control according to access rules. In conclusion, our system is not only safe, but also easy to use.

R E F E R E N C E S

[1] The Samba Team. Samba. https://www.samba.org/.

[2] *V.C. Hu, D. Ferraiolo, R. Kuhn, A. Schnitzer, K. Sandlin, R. Miller and K. Scarfone*. Guide to Attribute Based Access Control (ABAC) Definition and Considerations. NIST Special Publication 800-162, NIST, 2014. http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-162.pdf.

[3] *R. Cover*. Extensible Access Control Markup Language (XACML), 12 December 2009, http://xml.coverpages.org/xacml.html.

[4] *S.M. Park and S.M. Chung*. Privacy-Preserving Attribute-Based Access Control for Grid

Computing. International Journal of Grid and Utility Computing, 2014, **vol. 5**, no. 4, pp. 286-296.

[5] *X. Jin*, *R. Sandhu* and *R. Krishnan*. RABAC: Role-Centric Attribute-Based Access Control. In Proceedings of International Conference on Mathematical Methods, Models, and Architectures for Computer Network Security, 2012, Lecture Notes in Computer Science, **vol. 7531**, pp. 84-96.

[6] *M. Hüffmeyer* and *U. Schreier*. Formal Comparison of an Attribute Based Access Control Language for RESTful Services with XACML. In Proceedings of the 21st ACM on Symposium on Access Control Models and Technologies, 2016, pp. 171-178.

[7] *I. Ray*, *T.C. Ong*, *I. Ray* and *M.G. Kahn*. Applying Attribute Based Access Control for Privacy Preserving Health Data Disclosure. In Procceedings of 3rd IEEE EMBS International Conference on Biomedical and Health Informatics, 2016, pp. 1-4.

[8] *D. Xu*, *N. Shen* and *Y. Zhang*. Detecting Incorrect Uses of Combining Algorithms in XACML 3.0 Policies. International Journal of Software Engineering and Knowledge Engineering, 2015, **vol. 25**, no. 09n10, pp. 1551-1571.

[9] *J. Crampton* and *C. Williams*. On Completeness in Languages for Attribute-Based Aaccess Control. In Proceedings of ACM Symposium on Access Control Models and Technologies, 2016, pp. 149-160.

[10] *V. Goyal*, *O. Pandey*, *A. Sahai* and *B. Waters*. Attribute-Based Encryption for Fine-Grained Access Control of Encrypted Data. In Proceedings of ACM Conference on Computer and Communications Security, 2006, pp. 89-98.

[11] Wikipedia. Business Rules Engine. https://en.wikipedia.org/wiki/Business_rules_engine.

[12] *M. Yu*, *X. Ding*, *X. Wang* and *Y. Gong*. The Design of Intelligent Access Control Systems Based on Jess. In Proceedings of International Conference on Advances in Computer Science, Environment, Ecoinformatics, and Education, 2011, pp. 57-62.

[13] *Z. Xiong*, *J. Xu*, *G. Wang*, *J. Li* and *W. Cai*. UCON Application Model Based on Role and Rule-Engine. Computer Engineering and Design, 2013, **vol. 34**, no. 3, pp. 831-836.

[14] *L. Zhang*, *X. Wang*, *W. Dou* and *D. Liu*. Acess Control Method Based on Fuzzy ECA Rules for Pervasive Computing Environments. Computer Science, 2013, **vol. 40**, no. 2, pp. 78-83.

[15] *J. Zhong* and *S. Hou*. Attribute-Based Universal Access Control Framework in Open Network Environment. Journal of Computer Applications, 2010, **vol. 30**, no. 10, pp. 2362-2365, 2640.

[16] *H. Xiong*, *X. Chen*, *X. Fei* and *H. Gui*. Attribute and RBAC-Based Hybrid Access Control Model. Application Research of Computers, 2016, **vol. 33**, no. 7, pp. 2162-2169.

[17] *V. Varadharajan*, *A. Amid* and *S. Rai*. Policy Based Role Centric Attribute Based Access Control Model Policy RC-ABAC. In Proceedings of International Conference on Computing and Network Communications, 2015, pp. 427-432.

[18] Core Technology, Inc. MooseFS. https://moosefs.com/.

[19] MongoDB, Inc. MongoDB. http://www.mongodb.org/.

[20] Apache Software Foundation. Apache Thrift. http://thrift.apache.org/.

[21] Red Hat, Inc. Drools. https://www.drools.org/.

[22] SourceForge. Sun's XACML Implementation. http://sunxacml.sourceforge.net/.