Optimizing Access Control Performance for the Cloud

Slim Trabelsi[1], Adrien Ecuyer[2], Paul Cervera Y Alvarez[3] and Francesco Di Cerbo[1]
[1]*SAP Labs France, Mougins, France*
[2]*Open Systems, Zurich, Switzerland*
[3]*Amadeus, Sophia-Antipolis, France*

Keywords: Cloud, Access Control, Performance, Caching, Scalability, Policy, Security.

Abstract: Cloud computing is synonym for high performance computing. It offers a very scalable infrastructure for the deployment of an arbitrarily high number of systems and services and to manage them without impacts on their performance. As for traditional systems, also such a wide distributed infrastructure needs to fulfil basic security requirements, like to restrict access to its resources, thus requiring authorization and access control mechanisms. Cloud providers still rely on traditional authorization and access control systems, however in some critical cases such solutions can lead to performance issues. The more complex is the access control structure (many authorization levels, many users and resources to protect); the slower is the enforcement of access control policies. In this paper we present a performance study on these traditional access control mechanisms like XACML, which computes the overhead generated by the authorizations checking process in extreme usage conditions. Therefore, we propose a new approach to make access control systems more scalable and suitable for cloud computing high performance requirements. This approach is based on a high speed caching access control tree that accelerates the decision making process without impacting on the consistency of the rules. Finally, by comparing the performance test results obtained by our solution to a traditional XACML access control system, we demonstrate that the ACT in-memory approach is more suitable for Cloud infrastructures by offering a scalable and high speed AC solution.

1 INTRODUCTION

Cloud computing offers a high performance computing infrastructure, combined with a huge amount of resources (like storage, database, network, servers, memory, virtual machines, etc.). Resources are made available to customers with a strong guarantee on the high quality of the computing performance. Scalability, availability and accessibility are among the key pillars of a cloud service or platform; naturally, these characteristics must not be sacrificed at the altar of security. Security comprises many aspects and functionalities, and access control represents one of the most common concepts which are mandatory for any production system. Access control (AC) is a mechanism in charge of restricting and filtering the access to system resources (data objects, Personal Identifiable Information PII, services, platforms, or infrastructure). Specific authorizations (also called permissions) are assigned to system users; when a user requests access to a resource, an AC engine is in charge of checking if the requesting user's permissions are compliant with the set of AC rules associated to that resource. The AC rules can be expressed using very basic lists called ACL, or using a more sophisticated and structured representation called policy. The AC rules described via policies are largely adopted in complex systems, those that are available to many potential users and that offer different resources. The main advantage of using AC policies is the possibility to manage and maintain a huge number of complex rules. Compared to the basic ACLs, AC policies require a reasoning engine able to explore process and decide on the applicability of AC rules against access requests. Cloud computing instances rely on AC systems configured via AC policies that express AC rules targeting a complex structure of user set and resources (Popa, 2010). The more significant is the density of the user and resource model; the more complex is the structure of the policies. A complex AC policy requires an additional computing and reasoning effort to the AC engine in order to interpret its rules and enforce the decision. Some studies (Antonios, 2011) and (Tang, 2012) pointed out the new requirements in terms of AC models introduced by cloud computing the difficulties of traditional authorization systems to handle very complex policies for cloud computing systems, and in some critical cases this can lead to performance issues in the resource access process. In this paper we conduct a performance study on a XACML (OASIS, 2013) AC engine for a cloud infrastructure. XACML, stands for eXtensible Access Control Markup Language, is an OASIS XML policy standard for AC. Although if XACML is not initially designed for the Cloud, a deployment was proposed in (Reddy, 2012). Our performance study consists of stress tests on the AC enforcement engine, requiring it to process a huge number of access requests for a wide range of resources protected by complex policies. The objective of this study is to indicate the limitation of traditional AC engines when they are deployed in cloud platforms. Subsequently, we propose two approaches to optimize the performance of the considered AC enforcement engine and make it more suitable to cloud platforms. Both solutions are based on a pre-caching authorization tree; this method consists of storing all AC rules in a tree data structure, in order to optimize the policy exploration operation. This access tree can be implemented in two ways: Using a traditional DB (loaded in memory), or using a structured hash table system (also loaded in memory). We stress the importance of the in-memory deployment, in order to take advantage of the "unlimited" resources offered by the Cloud infrastructures.

This paper is organized as follows: In section two we survey the state of the art on the different performance studies dealing with performances of AC systems, in section three we describe the deployment of the XACML engine in the cloud as a reference study basis, in section four we detail our access control tree based solutions, in the section five we compare the different approaches through performance tests and try to identify the most scalable approach.

2 RELATED WORK

Optimizing performances of AC enforcement engines is an issue that appeared and was partially addressed before the emergence of the Cloud computing paradigm. It is interesting to review these studies to learn about their approaches and try to map them to the new requirements introduced by the cloud infrastructures. An AC mechanism has its deployment and operational costs, for instance when implementing its structure in very large scale storage systems, one has to face a trade-off among performance, availability and security (Leung, 2007). Balancing this trade-off is particularly challenging in the storage cloud environment, as providers must implement an efficient access control system that scales elastically and meets the high availability requirements of the cloud. The bottleneck caused by the lack of cloud-adapted AC enforcement mechanisms can be exploited for DoS attacks to break a service or a system like in the case described in (Niu, 2009).

2.1 Performance Issues for Access Control Enforcement in Traditional Systems

One of the most interesting approaches that inspired our solution is the XEngine. In (Hwang, 2011), and (Daly, 2011) the XEngine is presented as a fast and scalable evaluation engine for XACML. This work proposes to transform a XACML policy into a different data structure; this method converts the hierarchical structure of the XACML policy into a flat structure and then transforms any set of multiple combining algorithms into a first-applicable algorithm section. This work is inspired by the decision diagram approach where the decision is made based on a tree data structure. This work addresses the problem of performing fast XACML evaluation; however their model is quite static especially with respect to the proposed tree structure that systematically uses the subject as root of the AC tree. Such model lacks of efficiency especially when many resources are requested at the same time. In our solution, we proposed several enhancements to this approach in order to make it more flexible and efficient for any type of queries.

(Squicciarini, 2011) proposed an alternative approach called reordering and clustering policy rules. Their work proposes a way to optimize policies based on statistics to reorder rules. As it consists of a reordering of rules, only the deny-overrides and permit-overrides combining algorithm are taken in consideration for the optimization. Their solution works well if the amount of requests per requesters and the type of requester are not dynamic. In the case where they are dynamic the reordering is not efficient, and a clustering solution is proposed. The clustering is based on a per subject basis. In our previous example, this leads to the creation of two views, one for each subject, student and faculty. However, Squicciardini et al consider only two

XACML combining algorithms.

2.2 Performance Issues for Access Control Enforcement in Cloud Systems

An initial study was proposed by (Punithasurya, 2012) to compare different AC models; they tried to compare different parameters including the performance aspect. This evaluation of such parameters was not really motivated by a technical analysis. Therefore, it is hard to interpret their results and compare the different AC models with respect to performance aspects.

Well known cloud providers like Amazon S3 or Microsoft Windows Azure Storage, has a simple method for granting access to third parties using ACLs, provided that they are registered as S3 users or groups (they do not implement complex AC rules). Objects can be made public by granting access to the Anonymous group but there is no way to selectively grant access to principals outside of the S3 domain. The main limitation of S3 ACLs is the restriction to list 100 principals. Instead of proposing a scalable solution, Amazon S3 proposed to split the ACLs into small independent clusters. EMC Atmos Online opted for the same approach with a more hierarchical structure between the different cluster groups.

The caching solution was also proposed for cloud computing systems (Reeja, 2012). The approach described in that paper implements two policy decision points (PDP), one for new access requests and another one for access requests that were already processed by the primary PDP.

(Harnik, 2011) conducted an analysis on the ACL based mechanism for cloud computing in order to point out their weaknesses and to propose their AC mechanism. It is based on a capability-based system that allows the integration of existing AC solutions thus leading to hybrid architectures for AC systems. Such hybrid systems combine the benefits of capability-based models with other commonly used mechanisms such as ACLs or RBAC. On the other hand this study points out the weakness of ACL used by many major cloud providers like Amazon and Microsoft.

3 DEPLOYING XACML IN THE CLOUD

In the previous section we explored the different access control system deployed in the current cloud infrastructures. We observed that most of the solutions are based on basic ACLs and not really adapted to the Cloud requirements. We explain in this section how the XACML is deployed in a Cloud platform, and how to optimise the enforcement process using access control trees. We explain how our new solution takes benefit and enhance the tree based structures proposed by other studies in the literature.

3.1 XACML Integration

XACML is a declarative AC policy language implemented in XML and a processing model describing how to evaluate authorization requests according to the rules defines in policies. It allows for the definition of AC but also usage control rules through obligations. This language is very expressive and can be used to define a lot of different kind of policies. The choice of this language was due to the completeness of its expressivity for access control rules. We can define many AC models (like RBAC, ABAC, UBAC, etc.) with this language. It has a good flexibility for defining rule conditions.

We propose the XACML engine architecture depicted in Figure 1 as an integration architecture to the Cloud infrastructure.

- Cloud User interface: UI layers provided by the cloud as a service or as a platform feature. This cloud layer offers an IDM function to manage the user identity and authentication features. Through this UI the user can access to the different services and resources offered by the cloud. The identity (or role) of the user and her access request are collected at this level in order to be exploited by the PEP (Policy Enforcement Point) of the XACML engine
- XACML Engine: it is deployed in the cloud a service or as platform functionality (in this paper we chose to deploy it as a service). It offers the traditional functionality defined by the XACML specifications: PAP (Policy Administration Point) to manage the policy repository, PDP (Policy Decision Point) that evaluates the user request, and the authorizations contained in the policies, PIP (Policy Information Point) that provides external information about the user profile, and the Context Handler that coordinates all the previous components.
- Cloud Resources: we connected the resource manager of the XACML engine to the different resources provided by the cloud infrastructure

(database, services, VMs, etc.) in order to associate a user request with the requested resources. These resources are only accessible if the PDP evaluates positively the user request.

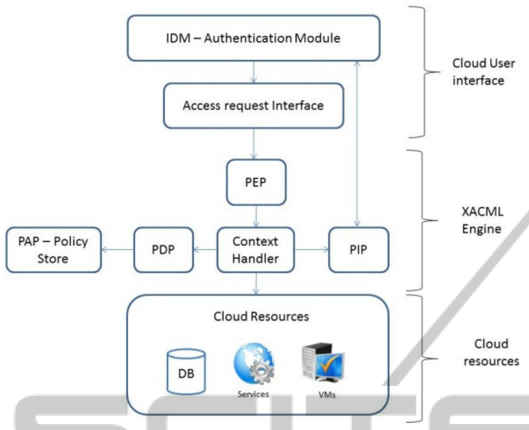

Figure 1: XACML Architecture for the Cloud.

3.2 Cloud Platform

The XACML AC engine that we used and modified in this paper is deployed as a service on a PaaS. We chose SAP Hana Cloud since it can be seen as SaaS and PaaS. SAP Hana Cloud platform also offers services like connectivity, document storing, identity, persistence and mail. But the most important selling argument offered by SAP Hana Cloud is that the platform is running over an in-memory database (Plattner, 2009), which fits perfectly with our in-memory based solution, described in details in the next sections.

4 OPTIMIZED ACCESS CONTROL ENFORCEMENT

The AC policy evaluation and enforcement can be complex and lengthy tasks to perform, impacted by the complexity of rules. In cloud computing, this issue is combined with thousands of users requesting simultaneously access to resources. This leads to an overhead during the authorization checking process. Therefore, it is important to implement a solution for speeding up the policy-based authorization checking process. Such solution should scale properly in order to handle multiple concurrent requests.

4.1 Access Control Trees

The main issue with XACML is the complexity due to its high expressivity. In order to optimize the evaluation computation process, we decided to implement a solution based on AC trees (ACT). The ACT is a concept based on aggregation. The use of pre-processing to aggregate privacy policies into the ACT will allow performance efficient AC check on mass of data. We can represent the aggregate AC information into a tree. The tree data structure has two key advantages for us: first, it provides a simple view of the AC structure; secondly, it facilitates the application of hashing techniques on a tree for efficient data search functions. This hashing technique will allow us to use an emerging database class (NoSQL) in order to improve further the already good performance of an ACT implementation in a relational database.

To map the AC rules into a tree structure we adopted the model proposed by (Bascou, 2002) and adapt it to the XACML policy schema. In this model, only accessible data objects are available. If the access to an object is denied, it will not appear. Following this model, our ACT contains only permitted data objects. If the data object cannot be accessed it will simply not appear. For this reason we call this tree the “Permit Tree” (PT). The AC tree or PT can be represented as follows:

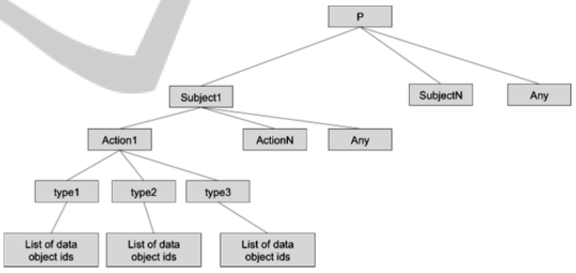

Figure 2: Access Control Tree (Permission Tree).

The PT (the same procedure can be applied for the Deny rules thus obtaining a Deny Tree) represented in Figure 2 is structured in three main levels: The first level contains the list of authorized subjects (or users, or roles, etc.) declared in the XACML policy repository. The ANY subject ID is used for objects that are accessible to all users with no restrictions. The second level represents the different actions or operations that can be executed on the data. If the list of actions is undefined, there is also an element *Any (Action)*. The third level represents the different types for data objects. Subsequently, the fourth layer contains a list of accessible data object IDs. The layer order (in the example subject, action and resource) can change according to system requirements. For example the first level can be the ID of the object. In that case the selection is made on the object to be accessed, for which one gets the list of authorized users.

4.2 Tree Management Algorithms

In this section we describe the tree management algorithms, i.e. how the AC tree is built, maintained and explored.

4.2.1 Insertion Algorithm

This algorithm describes how one can add a new data object associated with its XACML rule. In other words, this means to insert data object references in the ACT corresponding to the XACML policy associated to the data object.

Tree Input Parameters
The input for the insertion or "insert" algorithm is a data object with a policy associated to it. The policy is seen as a rule set that can be composed by 4 rule types; they are listed in Table 1. One of them is specific, while others are not specific. The first rule type is a specific rule and can be easily mapped in the ACT tree. The others (2-4) are not specific. Not specific rules can be recognized by the "any" keyword.

Table 1: List of rule types.

Rule Type	Specific rule?
1.< *subject; action; decision* >	specific
2.< *subject; **any**; decision* >	non-specific
3.< ***any**; action; decision* >	non-specific
*4.< **any**; **any**; decision* >	non-specific

Tree Insertion Algorithm
The insertion algorithm consists of a body and two functions. The body loops over each rule and check if the rule is specific or not. If it is not specific, the algorithm creates all specific rules corresponding to it, by means of the first function *getAllSpecificRules()*. Then for all specific rules the algorithm executes the *handleSpecificRule()* function. The *getAllSpecificRules()* function receives as input a non-specific rule which is (specific type). According to the non-specific rule type, all corresponding specific rules will be created.

The *handleSpecificRule()* focuses on what to do with a specific rule. First it checks if the rule was already handled. Then it checks if the subject of the rule already exists in PT. If it is not the case, the subject is created in the tree and any sub-tree of the PT is copied. If a specific subject can access any data object ID in the "any" action sub-tree (rule type 2 in Table 1), the algorithm firstly extracts all action sub-trees then checks if the rule decision is permit or deny. If it is permit, the algorithm adds data object ID to the subject sub-tree for the corresponding action. At the end the algorithm inserts the rule in the handled rule set in order to ensure the consistency of the PT.

Insertion Algorithm

```
Input: <R1,R2,…,Rn> as P,PII, current tree as T
handeledRules={} //set of rules of type <subject,action>
for each rule R in P do
  if R is of type <subject,action,permit> OR <subject,any decision> then
     if T.subject not exist then
       Add subject in T.subjects
       Copy actions from T.any into T.subject
     end if
   endif
   if R is of type <subject ,action,decision> then
     if decision == permit then
       Add PII in T.subject.action
     endif
     Add <subject, action> in handledRules
   else if R is of type <subject, any ,decision> then
     actionList = T.decision.subject.actions
     for each action A in actionList do
       if <subject, A> not in hanledRules then
         if decision == permit then
           add PII in T.subject.A
         endif
         Add <subject, A> in handledRules
       endif
     endfor
   else if R is of type <any,action,decision> then
     Subjectlist = T.decision.subjects
     for each subject S in subjectList do
       if <S, action> not in handledRules then
         if decision == permit then
           Add PII in T.S.actions
         endif
         Add <S, Actions> in handledRules
       Endif
     Enfor
   else if R is of type <any, any, decision>  then
     subjectList = T.decision.subjects
     for each subject S in subjectList do
       actionList = T.decision.S.actions
       for each action A in actionList do
         if <S, A> not in handledRules then
           if decision == permit then
             add PII in T.S.A
           endif
           add <S, A> in handledRules
         endif
       endfor
     endfor
   endif
endfor
return T
```

4.2.2 Request Algorithm

The request algorithm is used to explore the AC tree in an optimal way and retrieve the authorizations related to a particular rule. This algorithm is executed when a user requests access to a resource.

Tree Input Parameters
The request input takes the form: < Subject; Action; Resource Type >. If one created a tree where the level 1 node is a subject, then it is possible to start the exploration using the input parameter Subject.

Tree Request Algorithm
If the subject exists in the tree then a deep exploration of the path is needed to find the data object IDs related to it. If the subject does not exist, the access to the resource is denied.

Request Algorithm

```
Input: <subject, action> as R, current tree as PT
If R.subject not exist in PT then
  Return set of data objects ids resulting form PT.any.action
Else
  Return set of Data object ids resulting form PT.subject.action
endif
```

5 IMPLEMENTING THE ACT FOR THE CLOUD

Compared to the XEngine solution, we propose implementations specially designed for the Cloud platforms and the in-memory capabilities offered by such platforms. The proposed ACT can be stored in two ways: in a relational database or in structured hash tables. Both are stored in-memory (using the in-memory Hana DB). Each of the solutions has its own advantages and drawbacks, detailed in the following performance analysis section.

5.1 Database Implementation

In this implementation, the ACTs (Permit Tree and Deny Tree) are stored in a relational database, therefore any request is performed in the database using SQL queries. The two ACTs are stored in the same table: *ACT(uniqueId, subject, name)*

This table is composed by three columns:

- UniqueId: representing the IDs of data objects.
- Subject: representing the user ID or role specified in the Policy.
- Name: the value of the data object (ex: john.doe@example.com).

The primary key of this table is composed of the three columns. The composed primary key ensures that a tuple is unique in the table and so in the ACTs as well. Additionally, the primary key also creates indexes on the three columns that will accelerate the queries performed on the table.

Each request type is translated into a SQL statement:

- Verify the access on a given data object by a given subject:

```
SELECT uniqueId FROM ACT WHERE Subject =
? AND Name = ?, If the query returns an
unique Id, then the access can be granted
```

- Retrieve the list of data objects that a given Subject can access

```
SELECT uniqueId FROM ACT WHERE subject =
?,the query returns a set of Data objects
IDs
```

- Retrieve the list of data objects from a given category (Name) that a given subject can access

```
SELECT uniqueId FROM ACT WHERE subject =
? AND name = ?, the query returns a set
of data objects ids
```

- Retrieve the list of subjects for a given data object

```
SELECT subject FROM ACT WHERE uniqueId =
?, the query returns a set of Subjects
```

5.2 Hash based Implementation

In this implementation, we proposed to store the access control trees into in-memory hash tables and persisted in a relational database. The persistence of ACTs is managed like in the previous implementation. The tree data structure fits well with the use of hash tables. This implementation consists of two different trees, *ACTbyData* and *ACTbySubject* with their own data structures.

The *ACTbyData* is implemented using the Java structure *HashTable*. It maps keys to values. In our case we mapped the Data IDs to a set of delegates. The set of delegates is represented using the java structure *HashSet*, which implements a set back by a hash table. We mapped the delegates to a nested *HashTable*. This nested *HashTable* maps the Data names to sets of Data IDs. This latter set is represented using the Java structure *HashSet*.

Therefore, we propose the following two data structures:

- `ACTbyPii: Hashtable<Long, HashSet<String>>`
- `ACTbySubject: Hashtable<String, Hashtable<String, HashSet<Long>>>`

The performance of *Hashtables* and *Hashsets* depends on two parameters: the initial capacity and the load factor. The former defines the number of buckets that are created at the creation of the hash tables. The load factor is a measure representing how full the hash table is allowed to get before its capacity is automatically increased. It is a Boolean value.

6 PERFORMANCE STUDY

In this section we evaluate the implementation of the ACT in the use case. The setup for the evaluation is the following. It is a PC with an Intel(R) Core(TM) i5-2500 CPU @ 3.30 GHz. It has 8.00 GB of RAM and is running Windows 7. The SAP HANA Cloud Platform where we deployed our application offers an elastic package with a limit of 12 GB of RAM. For the ACT we performed the tests on a traditional XACML engine HERAS-AF, the Cloud platform. We developed two different ACT implementations, one using a Derby DB loaded in-memory and the hash version using Redis.

6.1 Comparing the Three AC Implementation Versions

The basic testing scenario is a sequential query of 2000 different subject requesting 2000 different data objects (in our scenario PIIs representing e-mail addresses, and names). Every request is made by a single user that asks to access everything and only gets one object at the end. Every PII is constrained by a policy permitting one subject to access.

Both of the XACML engines are deployed as a service on the SAP Hana Cloud platform and installed on top of a database containing PIIs. Access requests are generated then sent to the engines as a HTTP REST requests.

At 2000 data object, the Derby version is 2428x faster than the HERAS one, and the hybrid version is even 16998x faster. The difference is important since the HERAS based solution has to explore a complex structure of XML and objects representing the policy rules in order to match the applicable policy with the context of the request. This is more expensive in terms of processing time than a DB SQL query execution or a Hash function call.

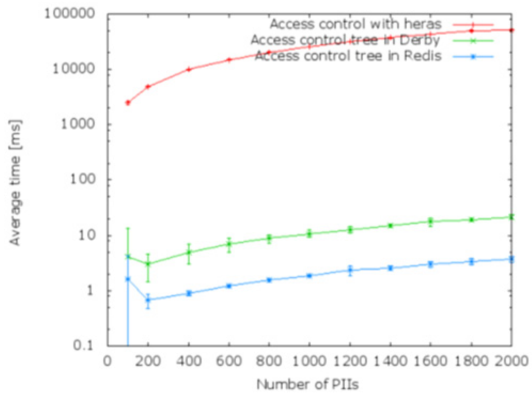

Figure 3: Comparing ACT solution with the traditional XACML HERAS engine.

6.2 Comparing the in-DB and the in-Hash ACT Solutions

The processing time of the HERAS solution gets worse with the increase of the system population (data objects + users). We decided then to focus our performance study on the comparison between the two proposed ACT implementations.

We run the same tests with a set of 6000 data objects accessible for 6000 different subjects. At 6000 data objects available the in-hash implementation of the ACT is 6x faster than the DB one. Furthermore the in-hash version scales better. Its performances are more stable as we can see with the standard deviation which is more important for the DB solution.

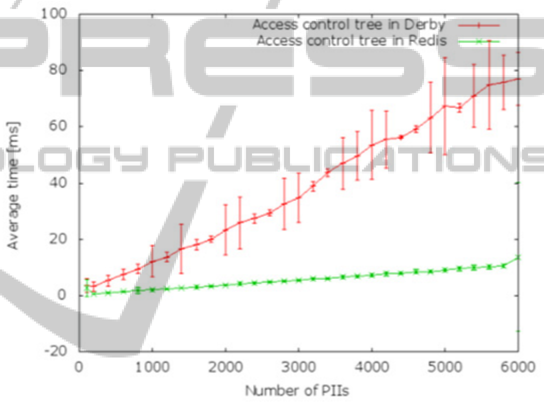

Figure 2: ACT in DB Vs. In-hash with 100% of the data objects accessible

7 CONCLUSION

The performance requirement for AC systems on the cloud is currently neglected by most of the cloud providers. Most of the cloud services are relying on single central AC systems implementing ACLs and in charge of handling all the access requests coming from all the cloud users. This issue can end up with serious performance problems especially when the cloud systems become complex. AC systems may become the main bottleneck disrupting the high speed computing capabilities of cloud servers. In our paper we evaluated a XACML engine for a cloud platform and demonstrated the limits of such policy based AC engine when access requests become huge. We proposed an AC tree caching system that can be implemented in parallel to the traditional AC systems in order to accelerate AC decisions. For this solution we proposed two implementations: one based on a relational database, and another one based on a structured hash table system. Both solutions are stored in-memory. We tested these two solutions against a traditional XACML-based solution for the cloud. The performance discrepancy between the traditional AC system and our ACT based solutions is very important. Especially the hash based ACT seems the more scalable and the more adapted to cloud platforms.

ACKNOWLEDGEMENTS

This paper is done in the context of the PPP Fi-Ware project and the FP7 EU COCO Cloud Project.

REFERENCES

G. Antonios, “Towards new access control models for Cloud computing systems”, *PhD report University of Macedonia, Department of Applied Informatics*

Z. Tang, J. Wei, A. Sallam, K. Li, R. Li, “ A New RBAC Based Access Control Model for Cloud Computing”, *7th International Conference, GPC 2012*, Hong Kong, China, May 11-13, 2012. Proceedings, pp 279-288

OASIS. Extensible Access Control Markup Language (xacml).https://www.oasis-open.org/committees/xacml

J. J. Bascou L. Gallon A. Gabillon, M. Munier and E. Bruno, « An access control model for tree data structures”. In *ISC '02 Proceedings of the 5th International Conf. on Information Security*, 2002.

A. W. Leung, E. L. Miller, and S. Jones. Scalable security for petascale parallel file systems. In SC ’07: Proceedings of the 2007 *ACM/IEEE conference on Supercomputing*, pages 1–12, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-764-3. doi: http://doi.acm.org/10.1145/1362622.1362644.

Z. Niu, H. Jiang, K. Zhou, T. Yang, and W. Yan. Identification and authentication in large-scale storage systems. *Networking, Architecture, and Storage, International Conference on,* 0:421–427, 2009.

J. Hwang A.X. Liu, F. Chen and T. Xie. Designing fast and scalable xacml policy evaluation engines. IEEE Transactions on Computers, Dec 2011.

A. Squicciarini S. Maruf, M. Shehab and S. Sundareswaran. Adaptive reordering and clustering based framework for efficient xacml policy evaluation. *IEEE Transactions on Services Computing, Oct-D*ec 2011.

J. Daly J. Brown and A. Gregory. The xengine policy decision point for xacml 3.0. Computer security *project in Department of Computer Sciences at the Michigan State University*, 26 Oct 2011.

Popa, Lucian, Minlan Yu, Steven Y. Ko, Sylvia Ratnasamy, and Ion Stoica. "CloudPolice: taking access control out of the network." In Proceedings of the 9th ACM SIGCOMM *Workshop on Hot Topics in Networks, p. 7. ACM,* 2010.

Punithasurya K and Jeba Priya S. Article: Analysis of Different Access Control Mechanism in Cloud. International Journal of *Applied Information Systems 4(2):34-39, September 2012. Published by Foundation of Computer Science*, New York, USA.

C.K. K. Reddy, P.R Anisha, K.S. Reddy, S.S. Reddy, “Third Party Data Protection Applied To Cloud and Xacml Implementation in the Hadoop Environment With Sparql”, IOSR Journal of Computer Engineering (IOSRJCE) ISSN: 2278 - 0661 Volume 2, Issue 1 (July - Aug. 2012), PP 39 – 46

Reeja S L, “Role Based Access Control Mechanism in Cloud Computing using co-operative secondary authorization Recycling Method”, 2012. *International Journal of Emerging Technology and Advanced Engineering* Website: www.ijetae.com (ISSN pp. 2250-2459, Volume 2, Issue 10, October 2012)

Amazon Simple Storage Service (Amazon S3). Amazon, b. http://aws.amazon.com/s3/.

Windows Azure Platform. Microsoft, a. http:// www.microsoft.com/windowsazure/windowsazure/.

Atmos Online Programmer’s Guide. EMC, a. https:// community.emc.com/docs/DOC-3481, accessed Jan 12, 2010.

D. Harnik, E. K. Kolodner, S. Ronen, J. Sataran, A. Shulman-Peleg, S. Tal,”Secure Access Mechanism for Cloud Storage”, Journal of Scalable Computing: Practice and Experience Volume 12, Number 3, pp. 317–336. http://www.scpe.org

HERAS-AF XACML - University of Applied Sciences Rapperswil. http://www.herasaf.org/

SAP HANA Cloud Portal http://scn.sap.com/ community/hana-cloud-portal

Apach Derby http://db.apache.org/derby/

Redis Database http://redis.io/

Plattner, H. "A common database approach for OLTP and OLAP using an in-memory column database." *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. ACM, 2009.