IMPLEMENTATION OF CRYPTOGRAPHICALLY ENFORCED RBAC

Valentin Ghiță[1], Sergiu Costea[2], Nicolae Țăpuș[3]

RBAC (Role-based access control) is an efficient method for sharing objects between different groups of users. If RBAC is implemented using ABE (Attribute Based Encryption), then the resulting system enforces access control indirectly, through cryptography. We propose a new multi-user system based on Cryptographically Enforced RBAC. To improve performance, our system combines existing work on Cryptographically Enforced RBAC with symmetric cryptography. From the best of our knowledge, we are the first to implement and experimentally evaluate the feasibility of such a system, which was previously only analyzed theoretically. We describe the architecture of our system, its implementation and evaluate performance. Our solution can be used to implement secure storage in clouds where the service provider is untrusted.

Keywords: access control, RBAC, ABE, secure storage

1. Introduction

Users often want to be certain that their stored data is confidential, with others unable to read their files. This is commonly achieved using encryption, where each user generates a key and encrypts their files before writing them to disk. However, confidentiality becomes more difficult to enforce in shared company networks and cloud storage, where multiple users need access to the same files. Classic access control methods [1, 2] describe how to enforce restrictions in complex systems with many users. However, if someone gets direct access to the storage device itself (either through software configuration errors or physical access), they compromise the security of the entire access control system [3]. Additionally, the storage service provider might be untrustworthy (for example, in the case of cloud services). Users must either trust the service provider to not disclose their data, or store their data encrypted with a secret decryption key. However, if they want to share the data with other users, the system scales poorly, because they must use a different key for each group of users they shared a file with.

RBAC (Role-Based Access Control) [4] provides better access control by allowing a user to be part of multiple roles (which are like groups) and roles to have access to multiple resources. This access control method is suited for

[1] Faculty Automatic Control and Computer Science, University POLITEHNICA of Bucharest, e-mail: sergiu.costea@cs.pub.ro
[2] Faculty Automatic Control and Computer Science, University POLITEHNICA of Bucharest
[3] Faculty Automatic Control and Computer Science, University POLITEHNICA of Bucharest

organizations, where each user has one or more roles, and each role has different permissions. Ferrara et al. described a new method of implementing RBAC [5], called Cryptographically Enforced RBAC (Crypto RBAC), using a new cryptographic method called Attribute Based Encryption [6]. Using this method, the access control operations also enforce encryption of files. Crypto RBAC can be used as an access control method in shared storage providers as it ensures both confidentiality, through encryption, and a better access control model than classic Linux systems. However, prior work on Crypto RBAC was only theoretical, with the most relevant performance evaluations relying on simulations and estimations [7].

We introduce a new Crypto RBAC system that offers a standard RBAC interface and exposes a simplified storage interface to users. We describe the architecture of our system and the implementation of our working prototype. While currently a proof of concept, our system can be used as the foundation for creating cloud-based secure storages which scale to large numbers of users and complex access control requirements. Our system exposes an API which closely follows the cryptographic primitives described in [5], and thus achieves strong security guarantees. We only look at the security of storage access; the issue of secure key distribution is orthogonal to our work and can be easily implemented in practice with user passwords and TLS.

We also evaluate the performance of the system. As we describe later, Crypto RBAC is not suitable for systems where the resources (files) change often, because *write* operations have a high cost. Instead, it is useful when the resources are modified rarely.

Our contribution is twofold. First, we implement Crypto RBAC as a standalone library which developers can use to quickly build interfaces to storages, while having strong security guarantees for access control. To improve performance, we combine Crypto RBAC with a symmetric encryption scheme; this allows us to run efficient symmetric encryption and decryption operations on stored data (which can contain very large files), while using expensive ABE encryption only on small keys. Second, we evaluate the security and performance of our implementation. Our paper is organized as follows. In Section 2 we give a brief overview of the main access control models. In Section 3 we include the background on the main technologies used in our implementation, while Sections 4 and 5 describe the system architecture and implementation. Section 6 illustrates a working example of our system.

2. Related Work

Multiple models for access to resources have been proposed in addition to RBAC, including Discretionary Access Control (DAC) and Mandatory Access Control (MAC) [1, 2]. DAC states that users having some rights over an object

can delegate that right to others. It is useful in scenarios with few users and where resources have a specific owner; for example, Linux systems use DAC for access control. MAC is more rigid and states that a user can use an object only if they have specific rights for that exact object; it is used in security-critical scenarios such as military applications or operating systems with a focus on security (e.g. SELinux [8]).

The flexibility of RBAC [4] makes it suitable for large scale organizations, and studies have shown that most organizations opt for this method of access control. However, classical RBAC implementations lack formal proofs of security and are commonly implemented using both trusted resource managers and storage. Recent work by Ferrara et al. [5] proposed to implement RBAC using strong cryptography, where each resource is encrypted using ABE keys. The privilege to read resources is replaced by the ability to decrypt them. Their work includes formal proofs which show that it is unfeasible for an adversary to retrieve information they should not have access to. While the work of Ferrara et al. is mostly theoretical, newer papers [9] have evaluated the performance of Crypto RBAC. Their estimations suggest that implementations based on ABE should be efficient enough to be used in production environments; however, their estimates are based on simulations and not an actual implementation.

3. Access Control Models

Role Based Access Control (RBAC) is a method used to decide which users have access to various resources. It is best suited for organizations where users are organized in roles and each role has a set of permissions to access resources.

In a system using RBAC, each transaction is made through a system manager. The manager implements the following functions:

- Create and delete new roles, users and resources;
- Assign users to roles
- Withdraw roles from users
- Grant and revoke permissions for roles to resources;
- Checks permissions for read and write operations when users try to access resources.

The manager holds the state of the entire system, including the associations between users and roles and between roles and permissions. As a simplification, we assume in this paper that the roles list is static and cannot be modified after the system is created. Therefore, we do not describe the operations that involve changes to the roles list.

Each RBAC system, regardless the implementation, must include functions for the operations described above:

`Init()`. Initializes the state of the whole system; the lists of users and resources, the user – role assignment table and the role – resource permission assignments table are set to empty. The (static) roles list is initialized with the predefined list.

`AddUser(name)`. If a user having the same name does not exist, then the user is added to the list of users. Note that the new user has no permissions yet.

`AddResource(name, contents)`. If the resource having the same name does not exist yet, a new resource is created with the given contents. Note that no role has access to the resource yet.

`AssignUser(user, role)`. Assigns the given user to the given role. After this step, the user has access to all the resources specified in the role – resource permission assignments table.

`GrantPerm(role, resource)`. Grants permission for the given role to the given resource. After this operation, the resource can be accessed by all the users in the given role.

`DeassignUser(user, role)`. Withdraws the given user from the given role.

`DelUser(user)`. Delete the given user. Before actually deleting the user, the manager must first withdraw it from all the roles he was assigned to.

`RevokePerm(role, resource)`. Revokes permission to the given resource for the given role.

`DelResource(resource)`. Deletes the given resource. Prior to deletion, the manager must revoke the permissions of all the roles that had access to the deleted resource.

`Write(resource, user, contents)`. Writes the given contents in the given file, on behalf of the given user. The operation succeeds only if the user has access to the file (e.g. is member of at least one role which has access to the file).

`Read(resource, user)`. Reads the given resource on behalf of the given user. The operation succeeds only if the user has access to the file

Attribute Based Encryption (ABE) is a cryptographic primitive that redefines the public-key cryptography approach. In a standard public-key system, the user public key is generally a random key that must satisfy a set of properties to match the private key. In ABE, the public key of a user contains strings, known as user attributes.

In ABE, plain text is encrypted for a set of attributes. The generated ciphertext can only be decrypted by users having attributes that match the ones under which the text was encrypted. The keys, in this case, are issued by a trusted

entity. As we will use ABE together with RBAC, the trusted entity will be the RBAC manager. ABE defines four functions [6]:

`Setup()`. Initializes the ABE algorithm with a random state, providing a global private key (master key – `MK`) and a set of public parameters – `PK`.

`KeyGen(attributes, MK, PK)`. Generates a user private key based on his public attributes. The user can decrypt the resources encrypted under the given attributes.

`Encrypt(plainText, attributes, PK)`. Encrypts the given plain text under the given attributes. The function outputs a ciphertext that can be decrypted by private keys generated for the given attributes.

`Decrypt(ciphertext, privateKey, PK)`. Decrypts the given ciphertext using the given user private key. The operation succeeds only if the attributes under which the ciphertext was encrypted match the attributed for which the private key was generated.

Advanced implementations of ABE allow defining access policies to specify the way key attributes are matched with ciphertext attributes. An access policy is an expression that is applied to a set of attributes and returns *true* of *false*. As the access policy is an expression, it is convenient to represent it as a tree, called tree access structure. Figure 1 shows an example of an access policy represented as a tree structure that specifies that the attributes set must contain the Teacher attribute or both the Student and Lab assistant attributes. Depending on where the access policy is stored, Attribute Based Encryption can be either Key-Policy ABE or Ciphertext-Policy ABE.

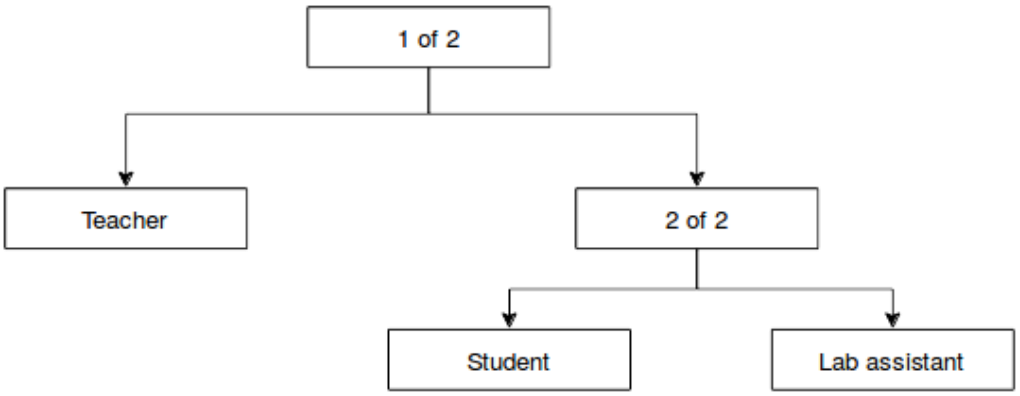

Fig. 1. Example of a tree access structure

In Key-Policy ABE (KP-ABE), the access policy structure is attached to user private keys and the ciphertext is computed using a list of attributes. When a private key tries to decrypt a ciphertext, the attributes of the ciphertext must match the access policy of the ciphertext.

In Ciphertext-Policy ABE (CP-ABE), the access policy is stored in the ciphertext. This is a more natural model, because at the decryption the attributes under which the private key was generated must match the access policy for the ciphertext.

4. CryptoRBAC System Architecture

We implemented Cryptographically Enforced RBAC using Attribute Based Encryption methods. The core of our architecture consists of a trusted Manager which handles user keys, encryption, and decryption. Additionally, the Manager translates generic read and write requests from clients to the appropriate storage API calls (e.g. local storage or remote storage in the cloud).

We make the following correlations between the entities in RBAC and ABE:

- The roles in RBAC become the attributes in ABE. We do this by defining attributes with role names. A user in the system has as attributes in ABE the roles he is a part of in RBAC.
- The resources in RBAC are the ciphertext resulting from ABE operations. The contents written to resources are plain texts.
- The users in RBAC are given ABE private keys.

The system is designed as an intermediary between the users and the storage provider. The users make all the read and write operations through the manager.

The system is best suited for organizations. A system administrator with access to the trusted Manager must create the users and assign them to corresponding roles.

Users can either access the storage through the Manager, or read data directly; when direct access is desired, users can receive read access to the entire storage space. However, since privacy is enforced through encryption, their private key will only allow them to decrypt the contents specified in the RBAC policy. The storage provider is considered untrusted, if the private keys are not stored on the same storage. The manager has access to every user's data. Also, all the communication between the users and the manager is made on secure channels, for example by using Transport Layer Security [10]. Figure 2 shows the overall design of the implemented system. The users can be both on the same machine as the manager, or access the manager remotely, through the Internet. Also, as the storage can be invisible for the users, the system can use multiple storage systems (both local and remote).

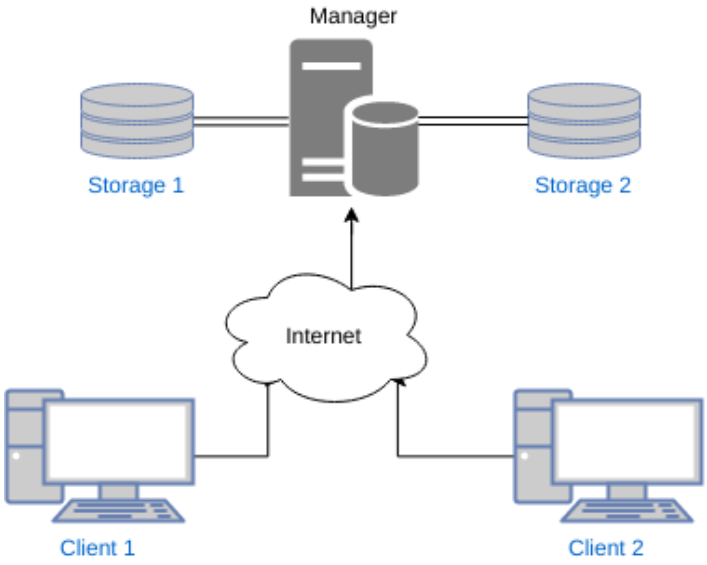

Fig. 2. System architecture.

The user's private keys are regenerated each time the user is assigned or withdrawn from a role or when a role the user is part of is updated. This leads to a key freshness problem: the keys can be changed even when the users are not online. The stale, out of date, keys on offline clients can no longer decrypt any content.

One possibility is to store the keys entirely on the system manager. However, this means that all read operations must pass through the Manager. Additionally, for each read the manager must first find the private key corresponding to the user. Both these issues lead to poor scalability. Another issue relates to security. If all the keys are stored on the manager, if it is compromised, then all user data is compromised. To avoid storing private keys for long periods of time on the Manager and to avoid stale key issues, we use a mixed approach: the keys are stored on the manager until the users are online. Whenever users join the system they first check to see if there are newer keys available; if true, they download the new keys and can access the storage directly. Because our implementation matches the algorithm descriptions in [6], security immediately follows; only users whose roles allow access to a specific file can read that file. In our case, a read attempt from a user without valid access rights causes the library to output an error. For efficiency, we do not encrypt and decrypt files using ABE directly. Instead, we use ABE to secure AES keys, which are in turn used to encrypt and decrypt the files. The security properties of the Cryptographic RBAC

algorithms guarantee that the AES keys can only be obtained by valid users (i.e., those that have been granted the correct access rights).

5. Implementation

The system is implemented in C, using the open-source ABE library *libbswabe2*. The library handles all the ABE operations and exposes the ABE interface described in Section 3.

The library is an implementation of CP-ABE, meaning that an access policy must be created for each encrypted file. As the attributes in the system are user roles, the access policy tells that the private key attributes must match one of the access policy attributes. As the ABE operations are slower than AES, the library only encrypts and decrypts a random key that will be used for AES [11]. A ciphertext generated by the ABE library contains the AES key encrypted using ABE under some attributes and the plaintext encrypted using AES with the key described before. To decrypt the ciphertext, the library first decrypts the key using ABE, if the private key matches the policy and the plain text results after a normal AES decryption. When the set of roles that have access to a file changes, the file must be re-encrypted using the attributes corresponding to the new set of roles. This also happens when a user is withdrawn from a role. Because the user keys are stored locally, the manager must generate a new attribute for the role and re-encrypt all the files the role has access to using the new attribute.

As the changes in the organization structure (roles, roles permission and user-role assignments) involve reencryption of files, Crypto RBAC is not well-suited for such systems. Reads are fast, so the system can be used efficiently for systems where the reads are frequent, but the changes are not.

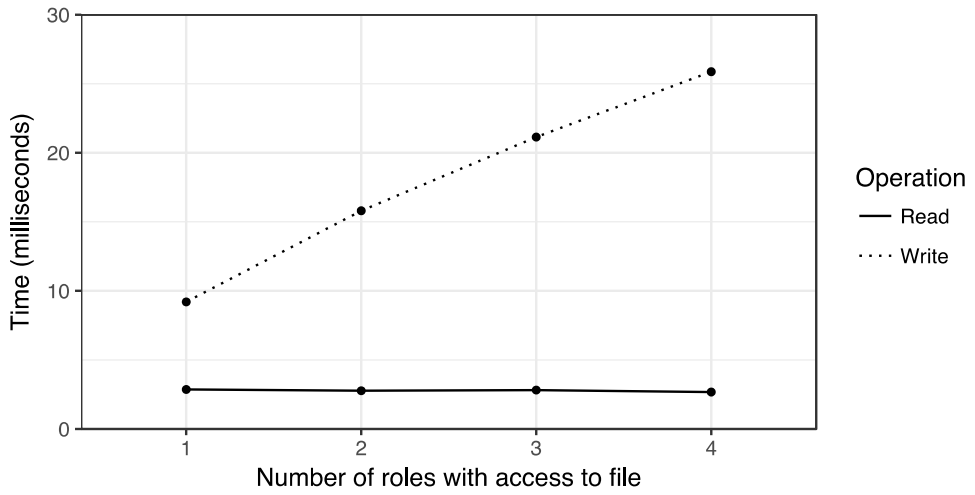

Fig 3. Performance of read and write operations for a 1024-byte file owned by different numbers of roles.

6. Performance Evaluation

We implemented a system in which access control is enforced indirectly, using cryptographic methods. This means that there are no explicit checks for permissions when a user tries to decrypt a resource. The system takes the user's private key and the encrypted resource and tries to decrypt the resource using the key. If the decryption process is successful, then the user has access to file, in the RBAC sense. If the decryption of an encrypted resource using a user private key successful, it means that the key contains attributes that satisfy the policy under which the resource was encrypted using ABE. As attributes correspond to user roles, it means that the user has the necessary roles to access the resource. The implementation resulted in an experimental system that can execute all the RBAC commands described in Section 2. We evaluated performance by measuring the average duration of library functions. We assume real systems will rarely modify access controls, so *assign* and *grant* operations will be much rarer than *reads* and *writes*. Therefore, we only include results for *reads* and *writes*. We ran the experiments on a 64-bit Ubuntu machine with a 2.7GHz Intel Core i5 CPU with hardware support for AES operations. Figure 3 shows the duration of read and write operations for 1024-byte files. Because the performance of *writes* depends on the number of roles that have access to the written file, we varied the number of roles from 1 to 4. To measure the performance of the algorithms, the tests were run without parallelization, and performed single *reads* or *writes*. Performance for *reads* remained constant, with 2.7ms on average for one read. However, performance for *writes* rapidly degrades with an increase in the number of users, ranging from 9.2ms when a single role had file access, to 25.87ms for 4 roles. For the latter, our library serves 36 write requests per second using a single core, which is insufficient for large systems. Most of the time was spent in ABE code, with AES and filesystem operations negligible in comparison. The results suggest that the library is usable in small scale systems where the number of writes is lower than the number of reads (e.g., small organizations). However, for systems with many roles and write operations the solution is not currently feasible. More efficient ABE schemes and implementations are necessary in this case.

7. Conclusions

We described the implementation of a Cryptographic RBAC system, which is used to enforce access control policies indirectly, through cryptography. The implemented system is a functional prototype of Crypto RBAC that can run all the RBAC operations in the Section 2, hiding the implementation and the underlying storage to the users. For this system to work, the manager must be trusted by the users. The system can work with untrusted or shared storage providers, where other users can read the stored files, if the users keep their

private keys private. To minimize the risk of revealing all the user data when the manager is compromised, we proposed a hybrid key management method where the manager keeps generated user private keys until the users are online (when the manager pushes the keys to users).

We evaluated experimentally the performance of our implementation, and showed that it is efficient enough for systems containing a small number of users or few *write* operations. However, performance rapidly degrades with an increase in the number of users. For such cases, further work on optimizing the ABE algorithms is needed.

REFERENCES

[1]. *S. Osborn, R. Sandhu, and Q. Munawer*, Configuring role-based access control to enforce mandatory and discretionary access control policies. ACM Transactions on Information and System Security (TISSEC) 3, no. 2 (2000): 85-106.

[2]. *R. Sandhu, and Q. Munawer*. How to do discretionary access control using roles. In Proceedings of the third ACM workshop on Role-based access control, pp. 47-54. ACM, 1998.

[3]. *D. Harnik, B. Pinkas, and A. Shulman-Peleg*, Side channels in cloud services: Deduplication in cloud storage. IEEE Security & Privacy 8, no. 6: 40-47, 2010.

[4]. *D. Ferraiolo, J. Cugini, and D. R. Kuhn*, Role-based access control (RBAC): Features and motivations. In Proceedings of 11th annual computer security application conference, pp. 241-48. 1995.

[5]. *A. L. Ferrara, G. Fuchsbauer, and B. Warinschi*, Cryptographically enforced RBAC, 26th Computer Security Foundations Symposium, IEEE, 2013.

[6]. *V. Goyal, O. Pandey, A. Sahai, and B. Waters*, Attribute-based encryption for fine-grained access control of encrypted data, In Proceedings of the 13th ACM conference on Computer and communications security, pp. 89-98. ACM, 2006.

[7]. *W. C. Garrison, A. Shull, S. Myers, and A. J. Lee*, On the practicality of cryptographically enforcing dynamic access control policies in the cloud. In IEEE Symposium on Security and Privacy (SP), 2016.

[8]. *P. Loscocco*, Integrating flexible support for security policies into the Linux operating system. In Proceedings of the FREENIX Track: USENIX Annual Technical Conference, p. 29. The Association, 2001.

[9]. *W. C. Garrison III, A. Shull, S. Myers, and A. J. Lee*, On the Practicality of Cryptographically Enforcing Dynamic Access Control Policies in the Cloud, IEEE Symposium on Security and Privacy, 2016.

[10]. *T. Dierks*, The transport layer security (TLS) protocol version 1.2, RFC 5246, 2008.

[11]. *J. Daemen, and V. Rijmen*, AES proposal: Rijndael, 1999.