Cryptographically Enforced Role-Based Access Control for NoSQL Distributed Databases

Yossif Shalabi, Ehud Gudes

Cryptographically Enforced Role-Based Access Control for NoSQL Distributed Databases

Yossif Shalabi and Ehud Gudes

Ben-Gurion University, Beer-Sheva, 84105, Israel,
shalabiyossif@gmail.com, ehud@cs.bgu.ac.il

Abstract. The support for Role-Based Access Control (RBAC) using cryptography for NOSQL distributed databases is investigated. Cassandra is a NoSQL DBMS that efficiently supports very large databases, but provides rather simple security measures (an agent having physical access to a Cassandra cluster is usually assumed to have access to all data therein). Support for RBAC had been added almost as an afterthought, with the Node Coordinator having to mediate all requests to read and write data, in order to ensure that only the requests allowed by the Access Control Policy (ACP) are allowed through.
In this paper, we propose a model and protocols for cryptographic enforcement of an ACP in a cassandra like system, which would ease the load on the Node Coordinator, thereby taking the bottleneck out of the existing security implementation. We allow any client to read the data from any storage node(s) – provided that only the clients whom the ACP grants access to a datum, would hold the encryption keys that enable these clients to decrypt the data.

1 Introduction

Security has been a notable weakness in almost every NoSQL database, a fact that was highlighted in a 2012 InformationWeek special report entitled *"Why NoSQL Equals No Security."*[2] Since the inception of NoSQL databases, their whole point was seen as guaranteeing rapid, unfettered access to big data. Thus, naturally, enforcing access control was seen by the Big Data community mainly as a hindrance in the way of fast data access: At the same time, NoSQL databases are now used by big financial institutions, healthcare companies, government services, and even by millitary intelligence – so, these systems must be able to handle sensitive data, providing the necessary security guarantees. *"And yet, the NoSQL ecosystem is woefully behind in incorporating even basic security,"*[2] This shortcoming affects all aspects of data security: users authentication, access control, transport-level security of inter-node communication, etc, so, an ever-growing deployment of NoSQL systems can subject them to many attacks which are likely to catch these systems entirely unprepared.

Recognizing this situation, a major focus on security had been put in Cassandra since version 3.0, even though the security subsystem is not enabled by default. Cassandra's built-in authorization module does not use encryption, and

instead enforces the Access Control Policy (ACP) by relying on *security monitors*, i.e. privileged components that handle a client's requests for access [1, 14]. At the same time, Cassandra 3.0 introduced data encryption, applicable to whole tables. The encryption keys are stored on the server, and are not directly related to an ACP; the same keys may be used for all encrypted tables, irrespective of the access permissions for these tables. Data encryption in Cassandra is designed to protect the data from an attacker bypassing the system's security monitor, e.g. by getting a root access to one of the nodes, or by physically stealing the disks from one of the nodes; the assumption is that the server would only decrypt data as part of a granted access. None of the existing modules for Cassandra, however, implement *cryptographic access control*, which would allow all read requests unconditionally, and use the centralized security monitors to mediate only the write requests. The benefit of such access control system for a distributed database is to avoid the bottleneck of the security monitor when most requests are to read data – which is indeed the case for most NoSQL deployments.

In this paper we present a model and protocols for enforcing cryptographically based RBAC using Cassandra. Cassandra was chosen as the platform for our proof-of-concept implementation owing to the following reasons:

- It is a popular NoSQL database. It is used by many companies, including Facebook and Twitter, and recently Cassandra is also being used by Cisco and Platform64 for personalized television streaming.
- It is open-source and well-documented; – although, as a mature industrial-quality DBMS, its code is very sophisticated and not easy to modify;
- "Out of the box", it includes at least the basic support for RBAC, whereas less mature open-source NoSQL DBMSs, such as Druid or Voldemort, which are considerably simpler and therefore easier to modify, don't include any support for any sort of access control, so that implementing RBAC in these DBMSs would require a major redesign of their core.

In this paper we suggest to enforce cryptography based RBAC using Predicate encryption [9] and adapt it especially to a distributed architecture like that of NoSQL Cassandra. The main contribution of the paper lies in the novelty of the protocol and the detailed description of its proposed implementation.

The rest of this paper is organized as follows. Section 2 provides a more detailed background on cryptographic access control and the various access control systems suggested for cloud storage, including a survey of the related work. Section 3 discusses the proposed protocol and its implementation. Section 4 concludes the paper and outlines future research.

2 Background and Related Work

Whereas the classic schemes for private-key and public-key encryption were concerned with privacy of *transmission*, with a predefined recipient or set of recipients that must hold the relevant encryption key to be able to decrypt the message, – *attribute-based encryption* (ABE) [6] extends it to privacy of *data*

storage, where different users may have access to different subsets of the stored data. It would be inefficient to store multiple copies of the data, one copy for each user that must have access to the data, each copy encrypted with its user's personal key; furthermore, it would be inconvenient for each user to keep a separate key for each file that he's authorized to access. Instead, ABE operates on a set of *attributes* held by the users; each datum would be encrypted only once, with the encryption key derived from the ACP, expressed as a conjunction of attribute equalities or attribute ranges that a user must satisfy to be able to decrypt the data. *Predicate encryption* (PE)[9] generalizes ABE by allowing policy expressions consisting of conjunctions, disjunctions, and more complicated equations on the users' attributes. A user can decrypt data only if the data's *access predicate,* evaluated on the user's attribute(s), is logically true.

An alternative approach[7] based on *symmetric-key encryption* (e.g. DES), is to store each user's set of keys, known as the user's *key-chain*, in the file system, encrypted with the user's *master key*. To access a file, the user will have to first read his key-chain file, and to decrypt the relevant data key with his master key. Such a scheme is, in fact, used in most modern web browsers and personal operating systems. Another option for a symmetric-key-based system is to store the keys corresponding to each file together with the file, in a *keys record*; once again, each data key is encrypted with the corresponding user's master key. The advantage of this option over the first one is that when deleting a file, all corresponding keys are deleted together with the file.

The conventional cloud systems impose certain trade-offs[5]: the users can either get advanced functionality, but will have to trust the *cloud service provider* (CSP), giving it unrestricted access to the data; or, conversely, they can use advanced security systems, withholding the encryption keys from the CSP, – but will then get limited functionality and/or performance, as the CSP cannot perform any local data processing. To help strike a balance between privacy and performance, one may use ABE, and split the ACP into two layers: the *inner encryption layer* (IEL), with keys unknown to the CSP, ensures that the data remain protected from the CSP; and then the CSP itself applies the *outer encryption layer* (OEL) on top. If an ACP update conforms to the access restrictions set by the IEL, then the affected data may be re-encrypted in the cloud, eliminating the need for the data owner to download and to re-upload the data.

The benefits of such *two-layer encryption* (TLE)[11] depend entirely on the decomposition of the ACPs into sub-ACPs for the IEL and the OEL: the more of the future ACP updates are restricted to the OEL sub-ACPs, the better. This dynamic aspect of the two layers policy, i.e. setting it up in anticipation of future ACP updates, is more fully addressed in a TLE scheme when the ACP is an *access control matrix,*[15] instead of a set of predicates. Our own proposed scheme is based on such a policy, named `Delta_SEL`, in which the initial ACP is translated into the IEL, and the OEL is initially empty. An example of an ACP update in a `Delta_SEL` system is as follows: suppose user u_1 has access permissions for files f_1 and f_2, so that the IEL encrypts both with a key $k_{1,2}$, known to u_1, and another user u_2 is granted access to f_1. This is handled by

granting u_2 the IEL key $k_{1,2}$, and at the same time, encrypting f_2 in the OEL with a new key k'_2, issued only to u_1. Revocation is handled in a similar way: if u_1 is now revoked access to f_1, a new OEL key k''_1 is issued to u_2, and f_1 is re-encrypted with k''_1. A major advantage of this TLE scheme is that the IEL keys never change, and therefore ACP updates never require the expensive retransmission and re-encryption by the data owner.

Another relevant paper presents a formal model for analyzing cRBAC (cryptographically enforced RBAC) systems[3]. A subsequent paper[4] describes an automated tool for verification of role reachability in RBAC systems by converting them into formally-verifiable imperative programs, and then applying a combination of abstract approximation and precise simulation of ACP updates, both of them operating heuristically and non-deterministically. The authors provide a formal analysis of their approach, however, they do not analyze any specific cRBAC protocols, especially implemented in the context of a distributed system like Cassandra. We present such detailed protocols in the next section, and the longer version of this paper includes a semi-formal analysis. Another recent paper by Garrison et. al [8] discusses the use of ABE/IBE cryptography for enforcing cloud based policies like RBAC, but does not address a distributed architecture like that of Cassandra which we focus on.

The main goal of our own work is implementing cRBAC in a distributed cloud environment using Predicate encryption PE. The scheme we're using for PE[9] allows attributes from $\mathbb{Z}_N^n$, for some $N = p \cdot q \cdot r$, where p, q, r are three distinct primes; and equality predicates on inner (dot) products of the attribute vectors. Alternatively, the scheme allows scalar attributes from $\mathbb{Z}_N$, and equality predicates on polynomials over $\mathbb{Z}_N$. A "dual" of the latter construction allows the attributes to be polynomials, and the predicates to correspond to evaluation at a fixed point. Access predicates expressed as DNF or CNF formulae are easily convertible into the polynomial form. A detailed discussion of predicate encryption is out of scope for this paper.We just depict the main steps involved:

Setup: The setup algorithm takes a security parameter and outputs a public key PK and a secret master key MSK

Key Derivation: Key derivation takes as input the master secret key and a vector of predicate values (X1,X2,...Xn) and outputs a secret key SK associated with this vector

Encryption: Encrypt takes as input the public key PK, a set of attribute values (Y1,Y2,...Yn) and the message M and generates an encrypted message M'

Decryption: Decrypt takes the encrypted message M', the secret key SK and the list of attributes values Y. The decryption succeeds only if the scalar product of the two vectors X and Y is zero

This encryption scheme for polynomial predicates can be extended to boolean formulae, as follows: $(x = a_1) \vee (x = a_2)$ is converted into the polynomial predicate $(x - a_1)(x - a_2) = 0$; $(x_1 = b_1) \wedge (x_2 = b_2)$ is converted into the polynomial predicate $r \cdot (x_1 - b_1) + (x_2 - b_2)$, for a random $r \in \mathbb{Z}_N$. These ideas extend to more complex combinations of disjunctions and conjunctions,

meaning that the predicate encryption scheme can handle arbitrary CNF or DNF formulae.

Cassandra is a distributed storage system for managing very large amounts of structured data spread out across many commodity servers, while providing highly available service with no single point of failure [10]. Cassandra aims to run on top of an infrastructure of hundreds of nodes. In Cassandra, columns are grouped together into sets called column families, which may be nested. The rows are dynamically partitioned over a set of storage nodes in the cluster, using an order preserving hash function on the row ID. The output range of the hash function is treated as a ring, and each storage node is assigned a random position on the ring. The hash based distribution is done by the node coordinator while each storage node is responsible for its share of the range of records. The location of a row (record) can be computed easily by the node coordinator. and the read or write of a record is directed to one of the storage nodes containing it. The support for RBAC was added in recent releases of Cassandra[14]. In Cassandra syntax, users are represented as roles that have a permission to log in, i.e. a user is a special kind of role. Cassandra roles are first-class database objects, and so they have permissions defined on themselves, too: a role may be assigned permissions (ALTER, DROP, GRANT, REVOKE) on other roles, or equally on itself. As was mentioned, there is no connection between Cassandra's support for RBAC and Cassandra's support for encryption, which is a major motivation to this paper.

3 Proposed Design

In our design, we apply the ideas from cryptographically based RBAC[3], Predicate encryption[9], the two layers encrypton of [5], and Cassandra specific distributed architecture [10]. The main principles of the design is that the ACP is implemented by encryption at the storage node level. Thus once the location of the record is determined, the information requested is sent directly to the client by the node storage manager, and the client can decrypt it only if it has the correct decryption key. The new cRBAC component supply the built-in data encryption module with the relevant encryption keys, based on the user identity. Nodes send the requested data directly to the client, and do not require the coordinator to relay the data; this avoids a possible bottleneck in data throughput. The role of the coordinator in handling read requests is limited to finding a node which has a copy of the data, and referring the client to that node. This is performed in the same way as normally, by maintaining a $(key \rightarrow nodes)$ mapping; our changes to the access control do not affect this in any way. Another principle employed by our scheme is the ability to verify written records by any client. This will be detailed later. In summary the main principles of our model are:

1. Read can be performed by any client, but the content is meaningful only for a client authorized by the ACP
2. Write can be performed by any client, but only ACP authorized users will have a valid signature on the written record.

3. Any client can verify the validity of a written record even if it cannot decrypt its content

3.1 Read/Write Access

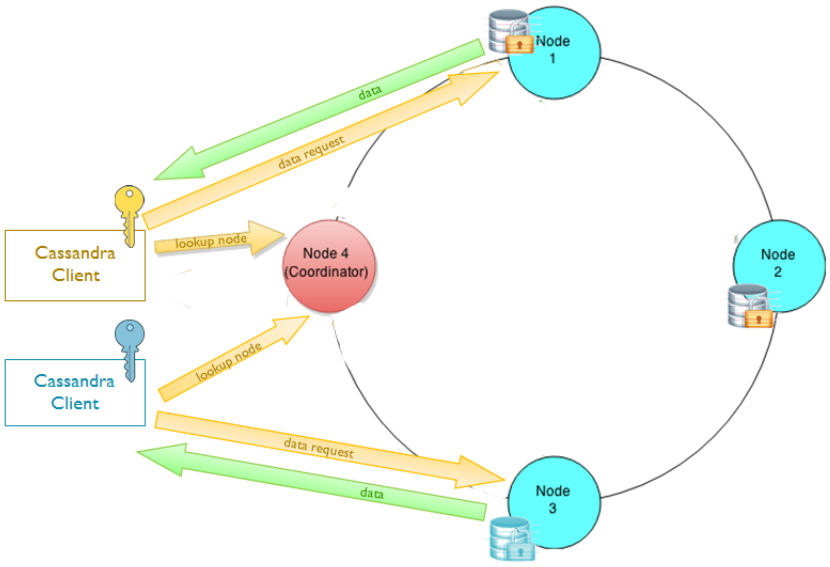

Fig. 3.1. Handling read requests

Handling read requests is illustrated in Figure 3.1. We use the cRBAC predicate encryption scheme described earlier. We employ a combination of a *master key*, from which multiple *encryption keys* and *decryption keys* are derived according to the ACP. The encryption keys (one per data file) are used to encrypt the data initially, and the same encryption keys are distributed to the clients who have write access to the corresponding files. The decryption keys (one per role) are distributed to all clients according to their role: each client's decryption key allows him to decrypt the data that he has read access to, according to the ACP.

Handling of write requests, in a way that does not require the coordinator to mediate each request, is slightly more complicated. Our proposal relies on the fact that most cloud storage systems (including Cassandra) only support appending to existing data, but not deleting or overwriting them. In our cRBAC-based system, any user may append new records, by sending them directly to the storage node, and only involving the coordinator for looking up the relevant node for his key value(s). Each written record must include the writer's encrypted

signature; and a subsequent reader, upon receiving the data from a storage node, needs to perform a little extra work in order to verify the signature of the record, and discard any invalid records. The signature is simply an encrypted hash of the record data; all that matters is that any client can easily tell whether it's valid or not. The key derivation for encrypting these signatures is done similarly as for the read access keys, but distributed in a different way: the decryption keys (one per data file) are known to all readers (who can use them for verification), and the encryption keys (one per role) are private to each writer.

Example In this example, there are three roles A, B, C and three files X, Y, Z. The access control policy is shown below in matrix form:

	X	Y	Z
A	rw		
B		rw	
C	r	rw	rw

Initially, the server generates a master key mk and random numbers a, b, c. Then, the server derives encryption keys

$$\begin{aligned} k_X &= derive_e(mk, (a \cdot c, -a - c, 1)), \\ k_Y &= derive_e(mk, (b \cdot c, -b - c, 1)), \\ k_Z &= derive_e(mk, (c, -1, 0)) \end{aligned}$$

for the three files, and decryption keys

$$\begin{aligned} k_A &= derive_d(mk, (1, a, a^2)), \\ k_B &= derive_d(mk, (1, b, b^2)), \\ k_C &= derive_d(mk, (1, c, c^2)) \end{aligned}$$

for the three roles. Each client gets a decryption key according to his role. Predicate encryption [9] guarantees that

$$decrypt(k_A, encrypt(k_X, X)) = decrypt(k_C, encrypt(k_X, X)) = X,$$

and so on for the other two files; it also guarantees that $decrypt(k_B, encrypt(k_X, X))$ is a random bit string that does not reveal any information about the contents of X; and so are $decrypt(k_A, encrypt(k_Y, Y))$, $decrypt(k_A, encrypt(k_Z, Z))$, and $decrypt(k_B, encrypt(k_Z, Z))$. This is because

$$\begin{aligned} (1, a, a^2) \cdot (a \cdot c, -a - c, 1) = (1, c, c^2) \cdot (a \cdot c, -a - c, 1) = 0, while \\ (1, a, a^2) \cdot (b \cdot c, -b - c, 1) \neq 0, \\ (1, b, b^2) \cdot (a \cdot c, -a - c, 1) \neq 0, \\ (1, a, a^2) \cdot (c, -1, 0) \neq 0, \\ (1, b, b^2) \cdot (c, -1, 0) \neq 0. \end{aligned}$$

For the purpose of write access control, the server generates a separate master key wk, and derives encryption keys

$$w_A = derive_e(mk, (1, a, a^2)),$$
$$w_B = derive_e(mk, (1, b, b^2)),$$
$$w_C = derive_e(mk, (1, c, c^2))$$

for the three roles, and decryption keys

$$w_X = derive_d(wk, (a, -1, 0)),$$
$$w_Y = derive_d(wk, (b \cdot c, -b - c, 1)),$$
$$w_Z = derive_d(wk, (c, -1, 0))$$

for the three files. Each client gets an encryption key according to his role, as well as a complete set of decryption keys (for write validation). Predicate encryption guarantees that $decrypt(w_X, encrypt(w_A, hash(X))) = hash(X)$, and therefore any client can, by using the publicly known w_X, validate that X had indeed been written and signed by A. Furthermore, predicate encryption guarantees that $decrypt(w_X, encrypt(w_C, hash(X))) \neq hash(X)$, and therefore an attempt by C to write and sign X will be detected by a subsequent reader, and ignored. These guarantees can be demonstrated in the same way as before, by computing the dot-products of derivation vectors for the keys.

The data are stored (possibly by different nodes) in the following manner, with the encrypted data of each file preceded by the writer's signature:

$encrypt(w_A, hash(X))$	$encrypt(k_X, X)$
$encrypt(w_?, hash(Y))$	$encrypt(k_Y, Y)$
$encrypt(w_C, hash(Z))$	$encrypt(k_Z, Z)$

A has $k_A, w_A, k_X, w_X, w_Y, w_Z$;
B has $k_B, w_B, k_Y, w_X, w_Y, w_Z$;
C has $k_C, w_C, k_Y, k_Z, w_X, w_Y, w_Z$.

Additional notes:

- In the illustration above, $w_?$ could be either w_B or w_C: when several roles have write access to a file, the signature does not disclose which one of them performed the write – only that the write was valid. There is a privacy problem here in case there is only one writer to a file. This will be dealt with in future work.
- A client who does not have read access to a file cannot validate whether it had been written legally or not, since the validation requires knowing the hash of the file contents. Every client needs validation keys for all files that he can read; validation keys for files that he cannot read are useless to him, but there's no harm and no waste in distributing all validation keys to all users, and this is simpler to manage.

– A signature cannot be "reused" for a file with different contents, or for a different file; however, it can be "reused", by a client without write access to a file, to revert the file to its earlier valid version. If this is undesirable, then a version number or a timestamp must be stored inside the file contents.
– The computational overhead for the cryptographic operations does not directly depend on the number of files or roles (each file has either one or two layers of encryption[11, 15]), and this overhead is expected to remain far below the data transmission latency. When there are many files or many roles, the task of traversing a user's key-chain and looking up the correct key for a file may appear computationally demanding; but in fact, storing these in a sufficiently big hash table, indexed either by file ID, user ID or any functional equivalent will make looking up a key a very efficient operation, with effectively a constant time complexity.

3.2 Write Access Issues

In the proposed system, there is no central authority to decide, for each write request, whether to allow or to deny it; instead, the clients themselves decide, for each written record, whether it had been written legally. This is similar to how the Blockchain[12] operates – anybody can issue a write, but any invalid writes will be ignored by the readers. Blockchain is a "distributed ledger" which keeps a verifiably immutable (but appendable) record of all transactions since the system's initialization, with the integrity preserved via a "proof of work", via signatures by trusted parties, or by other means. Originally implemented in 2008 for the Bitcoin cryptocurrency, the blockchain is public and permissionless, allowing any user to submit new blocks.

There are, however, some difficulties associated with cryptographic access control on write. One such difficulty is the coherence of access policy updates: as there is no central authority to handle all write requests and policy updates, it's impossible to tell which "happened" earlier, a revocation of a write permission, or a write request relying on the same permission. In the simplest (and most secure) case, the readers verify the signatures using their present signature validation keys (SVKs) – discarding all records whose writers don't currently have the write access, whether or not they had write access at the time the records were written. A more complicated implementation would have to keep track of historic SVKs, and correlate each signature with an SVK which could be in effect at the time of the writing.

A further difficulty with cryptographic access control on write is a possibility for a DoS attack, where a user with no write access appends excessive amounts of invalid records, running the system out of storage capacity. However, the traditional security model for NoSQL databases[2] had always assumed that the cluster operates deep in the back-end of a high-load system, and is not exposed to any external agents which could be malicious; and that some of the security can be sacrificed in order to improve the cluster performance. This is also the reason why the NoSQL databases have adopted the append-only data model, where an excessive amount of data updates can run the system out of storage capacity.

Even though readers may report the occurrence of invalid writes to the nodes coordinator, this is of no help against a DoS attack, since the written records (whether valid or invalid) are anonymous, and therefore, it's impossible to track down and block a client which writes excessive amounts of records (whether valid or invalid). Once again, this is not a defect in our proposed system, but a deliberate design decision upon which the NoSQL DBMSs are built.

3.3 ACP Updates

ACP updates require the data to be re-encrypted with new keys. We want to minimize the coordinator's involvement in this; therefore, the re-encryption should be performed locally to the data, by the nodes. This requires the new cRBAC component to plug into the node daemon as well, which seems to be rather uncommon for an authorizer implementation, and may require some adaptation of the core Cassandra node daemon to accept the plug-in. The biggest part of the cRBAC component, however, would run at the coordinator server to implement the keys management, i.e. to handle the changes to the ACP by generating new encryption keys and issuing them to the relevant users, as well as instructing the nodes to re-encrypt the affected data with the new keys. It is important to note that reencryption is done mostly at the **second level of encryption**, and the content itself is not re-encrypted.

Examples Suppose that in the example above, C is revoked access to Y:

	X	Y	Z
A	rw		
B		rw	
C	r	~~rw~~	rw

For this, the server derives a new key $k'_Y = derive_e(mk, (b, -1, 0))$, and orders over-encryption of Y with the new key:

$encrypt(w_A, hash(X))$	$encrypt(k_X, X)$
$encrypt(w_?, hash(Y))$	$encrypt(k'_Y, encrypt(k_Y, Y))$
$encrypt(w_C, hash(Z))$	$encrypt(k_Z, Z)$

$(1, c, c^2) \cdot (b, -1, 0) \neq 0$, and therefore C can no longer read Y:
Since $decrypt(k_C, encrypt(k'_Y, encrypt(k_Y, Y)))$ results with a random bit string that does not reveal any information about the contents of $encrypt(k_Y, Y)$ – which, in turn, C would still be able to decrypt.

To make sure C's signature on Y will no longer be accepted, the server derives a new decryption key $w'_Y = derive_d(wk, (b, -1, 0))$, and distributes it to all clients, so that:

A has $k_A, w_A, k_X, w_X, (w_Y), w'_Y, w_Z$;
B has $k_B, w_B, (k_Y), k'_Y, w_X, (w_Y), w'_Y, w_Z$;
C has $k_C, w_C, (k_Y), k_Z, w_X, (w_Y), w'_Y, w_Z$.

(the keys which are no longer of any use to the holding client are parenthesized).

The existing signatures remain valid: e.g., if Y had been written and signed by B, then

$$decrypt(w'_Y, encrypt(w_B, hash(Y))) = \\ = decrypt(w_Y, encrypt(w_B, hash(Y))) = hash(Y)$$

even though the decryption key had been updated; this is because

$$(b, -1, 0) \cdot (1, b, b^2) = 0.$$

However, a signature made by C will now be detected as invalid:

$$decrypt(w'_Y, encrypt(w_C, hash(Y)) \neq hash(Y)$$

Now suppose that A is granted read/write access to Y:

	X	Y	Z
A	rw	rw	
B		rw	
C	r		rw

In this case, no over-encryption is necessary: since access of A is strictly wider than the access of B, the server simply hands over B's keys to A, so that:

A has $k_A, k_B, w_A, w_B, k_X, k'_Y, w_X, (w_Y), w'_Y, w_Z$;
B has $k_B, w_B, (k_Y), k'_Y, w_X, (w_Y), w'_Y, w_Z$;
C has $k_C, w_C, (k_Y), k_Z, w_X, (w_Y), w'_Y, w_Z$.

The effect is that A acquires all of the permissions that B had. However, if B is subsequently granted new permissions that A does not have, then B will need to be issued a new set of private keys (k'_B, w'_B). For example, suppose that B is granted read/write access to Z:

	X	Y	Z
A	rw	rw	
B		rw	rw
C	r		rw

The server now has to generate new random numbers b' and c', and derive the new keys for B and C:

$$k'_B = derive_d(mk, (1, b', b'^2)),$$
$$k'_C = derive_d(mk, (1, c', c'^2)),$$
$$w'_B = derive_e(mk, (1, b', b'^2)),$$
$$w'_C = derive_e(mk, (1, c', c'^2)).$$

Then the server derives new decryption keys

$$w''_Y = derive_d(wk, (a \cdot b', -a - b', 1)),$$
$$w'_Z = derive_d(wk, (b' \cdot c', -b' - c', 1)),$$

–and distributes them to the clients; next, the server derives a new encryption key for X:

$$k'_X = derive_e(mk, (a \cdot c', -a - c', 1)),$$

–and orders over-encryption of X with the new key:[1]

$encrypt(w_A, hash(X))$	$encrypt(k'_X, encrypt(k_X, X))$
$encrypt(w_?, hash(Y))$	$encrypt(k'_Y, encrypt(k_Y, Y))$
$encrypt(w_C, hash(Z))$	$encrypt(k_Z, Z)$

$(1, b', b'^2) \cdot (b' \cdot c, -b' - c, 1) = 0$, but $(1, b', b'^2) \cdot (c, -1, 0) \neq 0$; therefore, B can read $encrypt(k_Y, Y)$ with his new key k'_B, but cannot yet read Y. To complete the grant of read access to Y, the server hands over k_C to B; this is secure because the only files that k_C enables decrypting are X and Z, and X has been over-encrypted with k'_X to protect it from B. Finally, the server sends k'_X to A because it has the write access. After the grant is complete:

A has $k_A, k_B, w_A, (w_B, k_X), k'_X, k'_Y, w_X, (w_Y, w'_Y), w''_Y, (w_Z), w'_Z$;
B has $k_B, k'_B, k_C, (w_B), w'_B, (k_Y), k'_Y, w_X, (w_Y, w'_Y), w''_Y, (w_Z), w'_Z$;
C has $k_C, k'_C, (w_C), w'_C, (k_Y, k_Z), w_X, (w_Y, w'_Y), w''_Y, (w_Z), w'_Z$.

Note that A no longer needs w_B, as the updated w''_Y allows him to sign his writes to Y using his own encryption key w_A. Also note that k'_X had to be derived using the new value of c' in order to protect X from B, who had been given k_C as part of granting read access for Z.

Management of Keys As shown in the examples above, each user has to keep a large set of keys:

- To *read* a file, a user needs his own key to decrypt the data, and also one key per file he has read access to, to validate the writers' signatures;
- To *write* a file, a user needs one key per file he has write access to, to encrypt the data; and also his own key to create a valid signature.

[1] It may appear wasteful that granting B read access for Z requires a re-encryption of X, and possibly of many other files that C has read access to. Local re-encryption of Z by the storage node (decryption with k_C and encryption with a new key) would be the most efficient alternative, but it would have compromised the privacy of Z by disclosing its plain-text to the storage node; therefore, this is not an option for our proposed system. Therefore, B has to be given k_C, since the encryption with k_Z can only be decrypted using k_C; and if X is not re-encrypted, then B would be able to read X using k_C. This overhead (switching from one-layer encryption to TLE) may only happen once for each file; and the possible alternatives (either downloading Z, decrypting it, encrypting it with a new key, and uploading it again; or double-encrypting all files at system initialization time, paying all of the possible overhead upfront) are in fact much less efficient.

Managing so many keys may be inconvenient to the user. To facilitate the keys management, we can use any of the techniques from [7], such as key-chain files, keys records attached to the files, or hierarchical keys management. For our implementation, we chose the key-chain technique, since it involves less overhead or knowledge on behalf of the ACP manager.

3.4 Formal Description of the Protocol

To simplify the description, we're presenting here a variant of our system where all files are double-encrypted at system initialization time (`Full_SEL`), instead of incrementally as part of the ACP updates (`Delta_SEL`)[15].

- Involved parties:
  1. data owner / ACP manager MGR
  2. user belonging to ROLE (the client)
  3. nodes coordinator CTR
  4. data storage node NODE
- Note that key management and distribution is done by the MGR, while second layer encryption is done by the CTR. The CTR never has to relay any data transmission.

1. **Setup**:
   (a) MGR derives (from the ACP) the initial encryption keys
   (b) MGR encrypts the initial data
   (c) MGR issues an "initialize" request to CTR, and receives the ID(s) of NODE(s) where to upload the initial data
   (d) MGR uploads the (encrypted) initial data to NODE(s)
   (e) MGR distributes the encryption keys to ROLEs, according to the ACP
2. **Read**:
   (a) ROLE issues a "lookup" request to CTR, and receives the ID of the NODE where the data is stored
   (b) ROLE downloads the encrypted data from NODE
   (c) For each record, starting from the latest:
      i. ROLE validates the signature item on the record
      ii. If the record's signature is valid, ROLE proceeds to (d)
      iii. Otherwise, ROLE proceeds to the next record read
   (d) ROLE decrypts the data
3. **Write**:
   (a) ROLE encrypts the data with his *read* key
   (b) ROLE signs the data with his *write* key
   (c) ROLE issues a "lookup" request to CTR, and receives the ID of NODE where the data is stored
   (d) ROLE uploads the encrypted data to NODE
4. **Grant read access for X to ROLE U**

(a) If U's role key k_U had been shared with another ROLE V, who does not have read access to X, then
   i. MGR derives a new role key k'_U and sends it to U
   ii. MGR derives a new encryption key k'_X, forming the derivation vector as described earlier, based on the identities of all ROLEs (including U's new identity) which are to have read access to X
   iii. MGR proceeds to step (d) as described below

(b) Otherwise, if another ROLE W already has read access to X, and U already has read access to all data that W has read access to, then
   – no re-encryption is necessary: instead, MGR hands over W's role key k_W to U

(c) Otherwise, [i.e. if k_U had not been shared with any ROLE V which does not have read access to X, and there's no such ROLE W which already has read access to X, and for which U already has read access to all data that W has read access to,] MGR derives a new encryption key k'_X, forming the derivation vector as described earlier, based on the identities of all ROLEs (including U) which are to have read access to X

(d) MGR distributes k'_X to all ROLEs which have write access to X

(e) If U's role key k_U does not allow decrypting the 1st layer encryption on X (i.e. if U didn't have read access to X at setup time, and had not been given a role key of a ROLE which had read access to X at setup time), then
   i. MGR chooses any role key $k_{U'}$ which allows decrypting the 1st layer encryption on X (i.e. U' had read access to X at setup time),
   ii. MGR hands over $k_{U'}$ to U

(f) MGR issues an "update ACP" request to CTR, passing k'_X as part of the request

(g) CTR handles the request by relaying k'_X to NODE(s) where X is stored

(h) NODE(s) CTR re-encrypt the data locally, using k'_X for the 2nd layer encryption

5. **Revoke read access from X**

(a) MGR derives a new encryption key k'_X, forming the derivation vector as described earlier, based on the identities of all ROLEs which still have read access to X

(b) MGR distributes k'_X to all ROLEs which have write access to X

(c) MGR issues an "update ACP" request to CTR, passing k'_X as part of the request

(d) CTR handles the request by relaying k'_X to NODE(s) where X is stored

(e) NODE(s) CTR re-encrypt the data locally, using k'_X for the 2nd layer encryption

6. **Grant write access for X to ROLE U**

(a) MGR sends the encryption key k_X to U

(b) If U's role key w_U had been shared with another ROLE V, who does not have write access to X, then
   i. MGR derives a new encryption key w'_U and sends it to U

ii. MGR derives a new decryption key w'_X, forming the derivation vector as described earlier, based on the identities of all ROLEs which are to have write access to X
iii. MGR distributes w'_X to all ROLEs which have read access to X

(c) Otherwise, if another ROLE W already has write access to X, and U already has write access to all data that W has write access to, then
- there's no need to derive new keys: instead, MGR hands over W's encryption key w_W to U

(d) Otherwise, MGR derives a new decryption key w'_X, forming the derivation vector as described earlier, based on the identities of all ROLEs (including U) which are to have write access to X
(e) MGR distributes w'_X to all ROLEs which have read access to X
(f) No re-encryption is necessary

7. **Revoke write access from X**
(a) MGR derives a new decryption key w'_X, forming the derivation vector as described earlier, based on the identities of all ROLEs which still have write access to X
(b) MGR distributes w'_X to all ROLEs which have read access to X
(c) No re-encryption is necessary
8. **Granting ROLE membership**
(a) MGR sends the new ROLE member all keys held by other ROLE members
9. **Revoking ROLE membership**
(a) MGR re-issues keys for all remaining ROLE members
(b) MGR orders re-encryption of all data accessible by the remaining ROLE members
(c) MGR updates its local representation of the ACP, which for each user lists the effective access permissions, with all role memberships expanded
(d) MGR handles the changes in effective access permissions, as detailed above

To prove correctness of the protocols above, one has to show formally that every ACP change results only with the intended Read or Write permissions. This will be included in the longer version of this paper.

4 Conclusions

This paper proposes a scheme for cryptographic enforcement of RBAC, using Cassandra for the proof-of-concept implementation. It combines several pre-existing techniques and algorithms, such as: Predicate encryption, Second level encryption and Cassandra's distributed architecture, for providing a flexible and efficient scheme to apply cRBAC for ACP enforcement in NoSQL databases. It presents a formal description of the resulting protocol, and presents examples of its operation. Currently we are implementing the protocols as part of a real-life Cassandra database. Results of this experimental evaluation will be reported in the future. We also plan to investigate further some of the issues mentioned, such as Privacy or DDOS attacks.

References

[1] DataStax: Securing Cassandra. https://docs.datastax.com/en/cassandra/3.0/cassandra/configuration/secureIntro.html (2015)

[2] Davis, M.A.: Why NoSQL equals NoSecurity. InformationWeek (2012)

[3] Ferrara, A.L., Fuchsbauer, G., Warinschi, B.: Cryptographically enforced RBAC. In: Computer Security Foundations Symposium (CSF), 2013 IEEE 26th. pp. 115–129. IEEE (2013)

[4] Ferrara, A.L., Madhusudan, P., Nguyen, T.L., Parlato, G.: VAC – verifier of administrative role-based access control policies. In: International Conference on Computer Aided Verification. pp. 184–191. Springer (2014)

[5] Foresti, S.: Data security and privacy in the cloud. In: 29th Annual IFIP WG 11.3 Working Conference on Data and Applications Security and Privacy (2015)

[6] Goyal, V., Pandey, O., Sahai, A., Waters, B.: Attribute-based encryption for fine-grained access control of encrypted data. In: Proceedings of the 13th ACM Conference on Computer and Communications Security, CCS 2006, Alexandria, VA, USA, Ioctober 30 - November 3, 2006. pp. 89–98 (2006)

[7] Gudes, E.: The design of a cryptography based secure file system. IEEE Transactions on Software Engineering (5), 411–420 (1980)

[8] III, W.C.G., Shull, A., Myers, S., Lee, A.J.: On the practicality of cryptographically enforcing dynamic access control policies in the cloud. In: IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016. pp. 819–838 (2016)

[9] Katz, J., Sahai, A., Waters, B.: Predicate encryption supporting disjunctions, polynomial equations, and inner products. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 146–162. Springer (2008)

[10] Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. Operating Systems Review 44(2), 35–40 (2010)

[11] Nabeel, M., Bertino, E.: Privacy preserving delegated access control in public clouds. IEEE Transactions on Knowledge and Data Engineering 26(9), 2268–2280 (2014)

[12] Pilkington, M.: Blockchain technology: principles and applications. Research Handbook on Digital Transformations (2015)

[13] MIT CSAIL Computer Systems Security Group: Crypto tutorial. http://css.csail.mit.edu/security-seminar/cryptoslides.ppt (2010)

[14] Tunnicliffe, S.: Role based access control in Cassandra. http://www.datastax.com/dev/blog/role-based-access-control-in-cassandra (2015)

[15] Vimercati, S.D.C.D., Foresti, S., Jajodia, S., Paraboschi, S., Samarati, P.: Encryption policies for regulating access to outsourced data. ACM Transactions on Database Systems (TODS) 35(2), 12 (2010)