# Cryptographically Enforced RBAC

Anna Lisa Ferrara[1], Georg Fuchsbauer[2], and Bogdan Warinschi[1]

[1] University of Bristol, UK,
anna.lisa.ferrara@bristol.ac.uk,bogdan@cs.bris.ac.uk
[2] Institute of Science and Technology, Austria
georg.fuchsbauer@ist.ac.at

**Abstract.** Cryptographic access control promises to offer easily distributed trust and broader applicability, while reducing reliance on low-level online monitors. Traditional implementations of cryptographic access control rely on simple cryptographic primitives whereas recent endeavors employ primitives with richer functionality and security guarantees. Worryingly, few of the existing cryptographic access-control schemes come with precise guarantees, the gap between the policy specification and the implementation being analyzed only informally, if at all.
In this paper we begin addressing this shortcoming. Unlike prior work that targeted ad-hoc policy specification, we look at the well-established Role-Based Access Control (RBAC) model, as used in a typical file system. In short, we provide a precise syntax for a computational version of RBAC, offer rigorous definitions for cryptographic policy enforcement of a large class of RBAC security policies, and demonstrate that an implementation based on attribute-based encryption meets our security notions.
We view our main contribution as being at the conceptual level. Although we work with RBAC for concreteness, our general methodology could guide future research for uses of cryptography in other access-control models.

## 1 Introduction

Traditional enforcement of access-control policies relies on low-level mechanisms like operating-system kernels and on-line monitors [5]. An appealing alternative that had received quite a bit of attention uses cryptography [17, 20, 23]. An example that illustrates the general idea behind this approach is access control through cryptographic sealing [17]. Here, a principal encrypts a file under some symmetric key and then restricts access by ensuring that only appropriate parties can get hold of the key. Among other benefits, cryptographic implementations extend the applicability of access controls to more settings, leverage already existing setups (e.g. PKIs) for fast deployment, and offer more flexible distribution of trust through mechanisms like threshold cryptography and/or multi-party computation (see [16] for a discussion of appropriate application scenarios).

Since monitors grant only policy-compliant requests, in monitor-based implementations correct enforcement of policies holds by design. Unfortunately, such a direct link cannot be established for cryptographically enforced access control. Policies are enforced indirectly through an associated key-management strategy. Furthermore, cryptographic guarantees are probabilistic (an adversary can always guess keys with non-zero probability), while policies are usually designed with an absolute semantics in mind (some party does/does not have access to some object). This observation illustrates the gap between the type of semantics usually associated to policies and typical cryptographic guarantees, which is unfortunately present in existent research on cryptographic access control. For simple policies implemented with simple primitives the lack of formal guarantees is perhaps justified since a) the translation between policies and the associated key management is simple and b) the security of the primitives is well-understood. When policies are complex or rely on primitives with advanced features, the absence of rigorously established guarantees should be worrisome. Cryptographic definitions, even those that rely on straightforward intuition should be used with great care as they tend to be subtle, lead to strange interactions, and are fragile in the presence of real-world attacks like adaptive corruption [13, 8]. Furthermore, even if the constructions rely on

clear intuition, a rigorous security proof may prove trickier than expected, may not be possible, or worse, may reveal overlooked attacks. Finally, note that such proofs hinge on cryptographic definitions of secure enforcement of security policies, which are almost never provided (see below for two notable exceptions).

### Overview of Our Results

In this paper we start to address what we perceive to be a pervasive lack of computational security guarantees in the area of cryptographically enforced access control. Our results are set in the context of the popular role-based access control model (RBAC) [2, 14, 25], as applied to a file system.

In RBAC, users (from some set of users $U$) are allocated one or more roles (from some set of roles $R$). In turn, each role has associated a set of permissions (from some set of permissions $P$). In core RBAC the underlying semantics is that a user $u$ has permission $p$ if there exists a role $r$ such that $u$ was assigned to $r$ and $r$ has permission $p$. The RBAC model allows for dynamic changes to the state of the system where users, roles and permissions are added and deleted; role membership may also change dynamically. A more complex variant of RBAC is obtained by considering a role hierarchy as a partial-order relation on $R$ (hierarchical RBAC [2]). In this case, it is demanded that higher-level roles subsume permissions owned by lower-level roles. For the sake of simplicity, in the rest of the paper we will only consider core RBAC. This is not a limitation, since any hierarchical RBAC policy can be easily transformed into an equivalent core RBAC policy.

The first ingredient of our framework is an execution model which in turn relies on a fixed (but general) system architecture. We consider a set of users with unrestricted read access to the file system. Each file is identified with a set of RBAC permissions: a user should be able to read the content of the file if he has one of the permissions. A cryptographic RBAC system is defined by the algorithms executed by parties when engaging in the different actions stipulated in the RBAC model (e.g. adding/deleting users and permissions). In general, each such action could be implemented via a multi-party computation between all parties and the manager. We model a scaled-down version where each RBAC management command is a non-interactive protocol: the manager locally executes an algorithm and then (privately) sends information to users who then locally update their state. Our model can be easily generalized and it is sufficient for the analyses of the concrete implementations we consider. These details fix a general syntax of a cryptographic RBAC implementation (henceforth we refer to such implementations as cRBAC).

We are interested in executions of cRBAC systems that are driven by some arbitrary adversary. The adversary can corrupt users (but not the manager) and can demand that the manager execute arbitrary RBAC commands, in arbitrary order, and with arbitrary parameters. For example, an adversary may ask the manager to add arbitrary users to the system, create new permissions, assign users to roles, etc. Intuitively, what is desired from any good implementation is that the adversary should only gain access, via the users he corrupted, to those files which he can trivially access following the sequence of actions that it dictates. For example, if the adversary demands that a corrupt user $u$ be assigned to role $r$, and that this role has permission $p$ then $u$ should clearly have access to the file to which $p$ corresponds. Conversely, if none of the users that the adversary controls has permission $p$ then a file corresponding to $p$ should be secret for the adversary. We formulate this property via an indistinguishability-based cryptographic game, and call a cRBAC implementation that satisfies it *secure*.

Next, we identify a class of RBAC policies for which we would like to define secure cryptographic enforcement. The policies express restrictions on what permissions users may have, subject to certain constraints. The class is large enough to express policies like "separation of duties", and "privilege escalation".

We provide an interpretation for such policies being satisfied as follows. First, we use a standard symbolic semantics for the execution of RBAC systems to define a notion of symbolic satisfaction for policies. Then we consider executions of an adversary with a manager that enforces the policy in a symbolic sense: the manager keeps track of the commands executed so far and only executes a new command if it does not lead to a policy violation (again, in the symbolic sense). Notice that this is a minimal requirement for the policy to hold. Computational satisfaction demands that in such executions, the access restrictions expressed by the policy enforced by the manager hold, in a computational sense.

We then prove a theorem that establishes a relation between secure cRBAC implementations and computational policy enforcement: in such implementations any policy from the class that we identified

is satisfied under the assumption that the manager follows the policy. A different interpretation of the theorem is that a secure cRBAC implementation provides a form of computational soundness [10]: policies that are satisfied symbolically are also satisfied computationally.

Finally, we demonstrate the usability of our framework by applying it to a cRBAC implementation based on attribute-based encryption [24, 18], a primitive designed with applications to access control in mind[3]. Our somewhat technical security proof confirms the applicability of this type of encryption to access control and also illustrates well an important point we made above: even constructions that rely on obvious intuition hide significant challenges that are revealed only following a rigorous proof.

We conclude by remarking that we view the central contribution of this paper as being mainly conceptual. We identify a problem which seems to have gone almost unnoticed (exception make [3] and [19], who address this problem for more ad-hoc policy languages or with more demanding definitions based on universal composability (UC) [9], respectively), and present a recipe that could be useful for follow-up work on computational guarantees of cryptographically enforced access control.

## 2 Role-Based Access Control

In this section we recall some details regarding role-based access control (RBAC) [2, 14, 25]. This model has emerged as a simple and effective access-control mechanism for large organizations. It simplifies policy specification and the management of user rights using a two-tier management: it groups users into roles and assigns permissions to each role. A permission represents an approval to perform an operation on an object, i.e., a permission is an object-action pair. Specifically, a user $u$ has permission $p$ if there exists a role $r$ such that $u$ is assigned to $r$ and $r$ has permission $p$.

The RBAC model allows for dynamic changes where users, roles, and permissions can be added and deleted, and role membership may also change. Since the structure of any organization is usually stable, in the rest of the paper we will consider the set of roles fixed. However, we allow users and permissions to be added and deleted, and any changes to user-role and permission-role assignments, which typically change quite frequently (e.g. employees moving across departments, reassignment of duties, etc.).

Next, we formally describe an RBAC system $\mathcal{S}$ over a set of roles $R$ as a state-transition system. Let $\mathcal{U}$, $\mathcal{P}$ be unbounded sets of users and permissions, respectively. A *state* of $\mathcal{S}$ is a tuple $(U, P, UA, PA)$, where $U$ and $P$ are finite subsets of $\mathcal{U}$ and $\mathcal{P}$, respectively. The relation $UA \subseteq U \times R$ is the *user-role assignment* relation, and $PA \subseteq P \times R$ is the *permission-role assignment* relation. A pair $(u, r) \in UA$ means that user $u$ belongs to role $r$, and $(p, r) \in PA$ means that role $r$ has permission $p$. A user $u$ is authorized for permission $p$ if there exists a role $r \in R$ such that $(u, r) \in UA$ and $(p, r) \in PA$.

We assume that the *initial* state is equal to $s_0 = (U_0, P_0, UA_0, PA_0)$.

Figure 1 shows a toy example state of an RBAC system representing a health-care facility. The example is an instantiation of the RBAC case study available at [1].

Let `RULES` be the set of administrative commands. Given two states $s = (U, P, UA, PA)$ and $s' = (U', P', UA', PA')$, there is a *transition* from $s$ to $s'$ with command $q \in$ `RULES` (denoted as $s \xrightarrow{q}_{\mathcal{S}} s'$) if one of the following conditions holds:

[AddUser(u)] The user $u$ belongs to $\mathcal{U} \setminus U$, $U' = U \cup \{u\}$, $P' = P$, $UA' = UA$, and $PA' = PA$;

[DeleteUser(u)] The user $u$ belongs to $U$, $U' = U \setminus \{u\}$, $P' = P$, $UA' = UA \setminus \{(u, r) \in UA \mid r \in R\}$ and $PA' = PA$;

[AddObject(p)] The permission $p$ belongs to $\mathcal{P} \setminus P$, $U' = U$, $P' = P \cup \{p\}$, $UA' = UA$, and $PA' = PA$;

[DeleteObject(p)] The permission $p$ belongs to $P$, $U' = U$, $P' = P \setminus \{p\}$, $UA' = UA$ and $PA' = PA \setminus \{(p, r) \in PA \mid r \in R\}$;

[AssignUser(u,r)] The user $u$ and role $r$ belong respectively to $U$ and $R$, $U' = U$, $P' = P$, $UA' = UA \cup \{(u, r)\}$, and $PA' = PA$;

[3] There are different variants of attribute-based encryption (ABE), such as *ciphertext-policy* ABE and *key-policy* ABE, both of which can be used for our implementation.

**Roles:**

$Doctor$; $PrimaryDoctor$; $Nurse$; $Patient$; $Receptionist$; ...

**Permissions:**

$p_1 = (read, MedicalRecord)$;
$p_2 = (read, ListOfDoctors)$;
$p_3 = (read, ListOfAppointments)$;
...

**Users:**

$Mary$; $Jim$; $Luke$; $Evelin$; ...

**UA:**

$(Mary, Receptionist)$;
$(Jim, Doctor), (Jim, PrimaryDoctor)$;
$(Luke, Patient)$;
$(Evelin, Doctor)$;
...

**PA:**

$(p_1, Doctor)$;
$(p_1, PrimaryDoctor)$;
$(p_2, Patient)$;
$(p_3, Receptionist)$;
...

**Fig. 1.** RBAC Toy Example.

[DeassignUser(u,r)] The user $u$ and role $r$ belong respectively to $U$ and $R$, $U' = U$, $P' = P$, $UA' = UA \setminus \{(u, r)\}$, and $PA' = PA$;

[GrantPermission(p,r)] The permission $p$ and role $r$ belong respectively to $P$ and $R$, $U' = U$, $P' = P$, $UA' = UA$, and $PA' = PA \cup \{(p, r)\}$;

[RevokePermission(p,r)] The permission $p$ and role $r$ belong respectively to $P$ and $R$, $U' = U$, $P' = P$, $UA' = UA$, and $PA' = PA \setminus \{(p, r)\}$;

An execution trace of an RBAC system $s_0 \xrightarrow{q_0}_{\mathcal{S}} s_1 \xrightarrow{q_1}_{\mathcal{S}} \ldots \xrightarrow{q_{n-1}}_{\mathcal{S}} s$ is defined as usual.

### Security Policies of RBAC Systems

The design of an RBAC system includes the specification of some security properties that must hold during the entire evolution of the system. Such properties define *security policies* which express limits on the access of users to permissions (e.g. users belonging to role $r$ cannot be granted some permission associated to role $\tilde{r}$). Examples of security properties of interest are the following [22]:

**Separation of Duty:** models conflict of interest. There is a separation of duty constraint between two roles $r$ and $\tilde{r}$ when each user is forbidden to simultaneously belong to both roles;

**Privilege Escalation:** models the requirement that lower-rank users cannot gain access to resources meant for a higher rank. There is a privilege-escalation constraint from role $r$ to role $\tilde{r}$ when each user in $r$ cannot acquire any permission associated to $\tilde{r}$.

*Example 1.* Examples of separation of duties and privilege escalation in the health-care facility from [1] are respectively: 1) a user cannot be assigned both roles *Receptionist* and *Doctor* (e.g. to avoid fraud by

preventing the user to falsely claim to treat a patient and billing the insurance company); 2) a patient cannot have privileges of his own primary doctor.

To formally define such policies, we introduce a predicate $\mathsf{HasAccess}$, which reflects when a user $u$ symbolically has access to a permission $p$. A user $u$ has some permission $p$ if $u$ has been assigned a role that has the permission:

$$\mathsf{HasAccess}(u,p) \quad \Leftrightarrow \quad \exists r \in R : (u,r) \in UA \ \wedge \ (p,r) \in PA \ . \quad (1)$$

The next definition identifies a class of RBAC security policies for which we will define (symbolic and) cryptographic enforcement.

**Definition 1 (Security Policy).** *Given an RBAC system $\mathcal{S}$ over a set of roles $R$, a* Security Policy $\Phi$ *is a formula of the following form:*

$$\forall u \in U \ \forall p \in P : \mathsf{Cond}(u,p) \ \Rightarrow \ \neg\mathsf{HasAccess}(u,p) \ ,$$

*where $\mathsf{Cond}(u,p)$ is a predicate over a user $u \in U$ and a permission $p \in P$. We restrict $\mathsf{Cond}(u,p)$ to predicates that can be evaluated over an RBAC state; satisfaction of a policy in an RBAC state and its extension to traces are defined in the obvious way.*

*A security policy $\Phi$ is thus completely determined by the predicate* $\mathsf{Cond}$.

The class of security policies identified by *Security-Policy* formulas captures both separation of duties and privilege escalation security constraints. Privilege escalation from role $r$ to role $\tilde{r}$ requires that any user who is assigned role $r$ ($(u,r) \in UA$) does not have access to any permission $p$ assigned to role $\tilde{r}$ ($(p,\tilde{r}) \in PA$) and can be formulated as:

$$\forall u \in U \ \forall p \in P \quad : \quad [(u,r) \in UA \ \wedge \ (p,\tilde{r}) \in PA)] \quad \Rightarrow \quad \neg\mathsf{HasAccess}(u,p) \ .$$

Separation of duties between roles $r$ and $\tilde{r}$ formalizes that no user $u$, assigned to role $r$ should have permissions associated to $\tilde{r}$, and the same should hold for $r$ and $\tilde{r}$ swapped. This can be formulated as:

$$\forall u \in U \ \forall p \in P : \big[(u,r) \in UA \wedge (p,\tilde{r}) \in PA] \vee [(u,\tilde{r}) \in UA \wedge (p,r) \in PA\big] \ \Rightarrow \ \neg\mathsf{HasAccess}(u,p) \ .$$

As an example, let $(U, P, UA, PA)$ be the RBAC state of Figure 1 and consider the security policies of Example 1. It is easy to see that the predicate $\mathsf{Cond}$ holds for both security policies. Indeed, Mary belongs to role *Receptionist* but does not belong to role *Doctor*. On the other hand, Jim and Evelin are doctors but not receptionists. Also, Luke is a patient but he does not belong to role *PrimaryDoctor*. Moreover, both security-policy formulas hold. Indeed, it holds that $\neg\mathsf{HasAccess}(Mary, p_1)$, $\neg\mathsf{HasAccess}(Jim, p_3)$, $\neg\mathsf{HasAccess}(Evelin, p_3)$ and $\neg\mathsf{HasAccess}(Luke, p_1)$.

**Multiple Policies** Without loss of generality we can restrict ourselves to a single policy, as a set of policies defined by $\mathsf{Cond}_1, \ldots, \mathsf{Cond}_n$ is simultaneously satisfied if and only if the policy defined by

$$\mathsf{Cond}_{1,\ldots,n}(u,p) \ :\Leftrightarrow \ \mathsf{Cond}_1(u,p) \vee \ldots \vee \mathsf{Cond}_n(u,p)$$

is satisfied. This is because

$$\bigwedge_{i=1}^{n} \big[\forall u \in U \ \forall p \in P : \mathsf{Cond}_i(u,p) \Rightarrow \neg\mathsf{HasAccess}(u,p)\big]$$

is logically equivalent to

$$\forall u \in U \ \forall p \in P \ : \ \big[\mathsf{Cond}_1(u,p) \ \vee \ \ldots \ \vee \ \mathsf{Cond}_n(u,p)\big] \ \Rightarrow \ \neg\mathsf{HasAccess}(u,p) \ .$$

## 3 Cryptographic RBAC

In this section we introduce our notion of cryptographic RBAC used to protect access to a file system.

### 3.1 Syntax

We consider a system setup where a manager interacts with a set of users. The manager is in charge of executing RBAC commands and monitors write access (but not read access!) to files. In our system we identify a permission $p \in P$ with read access to a file. By a slight abuse of notation we identify the file with $p$.

At any point the global state of a cRBAC system $\mathcal{CRBAC}$ is given by the local states of the manager ($st_M$) and of each user $u$ ($st[u]$). Moreover, there is a file system $FS$, which is publicly accessible (formally, $FS$ is a bitstring, in implementations $FS$ would be an array of encrypted files, each file corresponding to a permission $p \in P$). $\mathcal{CRBAC}$ is defined by the following algorithms: Init, AddUser, DelUser, AddObject, DelObject, AssignUser, DeassignUser, GrantPerm, RevokePerm, Update, Write and Read.

The initialization procedure Init takes as input the security parameter $\lambda$ and a set of roles $R$ for the RBAC and outputs the initial states of the manager and an initial state for the file system $FS$. The remaining algorithms (with the exception of Write and Read) implement the commands of an RBAC system, which were introduced in Section 2. Each algorithm takes as input the local state of the manager $st_M$, the file system $FS$ and an additional parameter (such as the user $u$ to be added for AddUser). In executions these algorithms are run by the system manager and they output the updated file system, a possibly updated state for the manager, and a message $msg_u$ for every user $u$. The message $msg_u$ (when non-empty) is sent to user $u$, who then runs Update on her state $st[u]$ and $msg_u$ to obtain an updated state $st[u]$. In effect, we model the implementation of RBAC commands as non-interactive multi-party computation where after some local computation the manager sends messages to all of the parties who update their local states.

We model the idea that parties do not have write access to files by only giving the manager the possibility to write to the file system. Algorithm Write takes as input a filename $p$ and a content $m$ to be written to $p$. Upon execution, Write outputs an updated file system $FS$. Finally, Read is an algorithm that users can use to read the file system. It takes as input a user $u$'s state $st[u]$, the file system $FS$ and a filename $p$ and retrieves the content of $p$ (if $u$ has access to $p$—see the correctness definition in the next section).

### 3.2 Correctness

Intuitively, a cRBAC system $\mathcal{CRBAC}$ is *correct* if every user who according to the symbolic state of the system has access to a permission $p$ can obtain the content of $p$. More precisely, let $(st_M, FS)$ be the output of Init. Now consider an arbitrary sequence of executions of algorithms of $\mathcal{CRBAC}$, which induces a symbolic RBAC state $(U, P, UA, PA)$ by applying the RBAC commands corresponding to the algorithm calls (we ignore the read and write commands as these are not defined in the symbolic world). Consider the values $st_M, FS, \{st[u]\}_{u \in U}$ after the executions. Then whenever some content $m$ was the last one to be written to an object $p$ via Write and whenever in the symbolic RBAC we have $\mathsf{HasAccess}(u, p)$ for some user $u \in U$ then $\mathsf{Read}(st[u], FS, p)$ should output $m$.

As usual in cryptography, we formalize this by a game between a challenger and a polynomial-time adversary $\mathcal{A}$. The latter has access to oracles to change the RBAC state and write content to objects/files.

The challenger, which simulates the manager and the users, keeps the state $(U, P, UA, PA)$ of the symbolic RBAC. For every RBAC command defined in Section 2, there is an oracle that executes the intended RBAC command. The execution is as follows: when invoked, each oracle first modifies the symbolic RBAC state accordingly and then executes the corresponding computational $\mathcal{CRBAC}$ algorithm and simulates the updates of user states. The oracle WRITE executes Write.

The challenger keeps a table $T$ and each time WRITE is called on $(p, m)$, it sets $T[p] \leftarrow m$. In the end the adversary outputs a pair $(u^*, p^*)$ and loses if $\mathsf{HasAccess}(u^*, p^*) = 0$. Otherwise, the challenger runs $m^* \leftarrow \mathsf{Read}(st[u^*], p^*, FS)$ and the adversary wins if $m^* \neq T[p^*]$.

$\mathcal{CRBAC}$ satisfies *correctness* if for any polynomial-time adversary the probability of winning the above game is 0.

### 3.3 Security

Intuitively, a cryptographic implementation of an RBAC system is *secure* if whenever a user $u$ does not have access to an object $p$, i.e. $\mathsf{HasAccess}(u, p) = 0$, then the user should not be able to deduce anything about the content of $p$ from the file system $FS$.

Following cryptographic conventions, we formalize this via an indistinguishability-based definition: an adversary $\mathcal{A}$, who can impersonate users, chooses two messages $m_0$ and $m_1$, one of which is randomly selected and written to an object $p$ of $\mathcal{A}$'s choice; the adversary must then determine which of the two messages it was.

More precisely, we define a game that involves an adversary $\mathcal{A}$ interacting with the manager of a cRBAC implementation. The game selects a random bit $b \leftarrow \{0, 1\}$, which the adversary must guess. The adversary can ask the manager to execute any of the cRBAC commands. We assume private channels between the manager and the users. To model impersonation of users, we let the adversary *corrupt* users: $\mathcal{A}$ obtains the state of the user it corrupts and from then on receives all the messages sent to that user by the manager. Moreover, the adversary can ask for a *challenge* $(m_0, m_1)$ for an object $p$. In response to this query the experiments runs $\mathsf{Write}(p, m_b)$.

The experiment prevents the adversary from winning trivially by making a corrupt user have access to an object with a challenge content. (Otherwise, the adversary could simply use $\mathsf{Read}$ to read the message and thereby determine the bit $b$.)

We define the following experiment, which maintains a symbolic representation of the RBAC as it evolves through the adversary's queries: $(U, P, UA, PA)$ with $U \subseteq \mathcal{U}$, $P \subseteq \mathcal{P}$, $UA \subseteq U \times R$ and $PA \subseteq P \times R$. In addition to this, the experiment maintains two lists: $Cr \subseteq \mathcal{U}$ is the list of users which the adversary has corrupted and $Ch \subseteq \mathcal{P}$ is the list of permissions in which the adversary has asked to be put a challenge. We use these lists to characterize and prevent trivial wins by the adversary, e.g. by corrupting a user that can access a file where a challenge has been written.

**Definition 2.** *We define the security of a cryptographic RBAC through the following experiment:*

$\underline{\mathbf{Exp}^{ind}_{\mathcal{CRBAC},\mathcal{A}}(\lambda)}$
$b \leftarrow\!\!\$\ \{0, 1\}$; $Cr, Ch \leftarrow \emptyset$
$(st_M, FS, \{st[u]\}_{u \in \mathcal{U}}) \leftarrow\!\!\$\ \mathsf{Init}(1^\lambda, R)$
$b' \leftarrow\!\!\$\ \mathcal{A}(1^\lambda, FS : \mathcal{O})$
*Return* $(b' = b)$

*The oracles $\mathcal{O}$ to which the adversary has access are specified in Figure 2 (and discussed below). We say that $\mathcal{CRBAC}$ is* secure *if for all probabilistic polynomial-time adversaries $\mathcal{A}$, we have*

$$\mathbf{Adv}^{ind}_{\mathcal{CRBAC},\mathcal{A}}(\lambda) := \left| \Pr[\mathbf{Exp}^{ind}_{\mathcal{CRBAC},\mathcal{A}}(\lambda) \to \mathtt{true}] - \tfrac{1}{2} \right|$$

*is negligible in $\lambda$.*

Using the first eight oracles, $\mathcal{A}$ can make the manager execute RBAC commands (each one of which first checks whether its execution would let $\mathcal{A}$ win trivially, in which case it returns $\perp$). The oracles run the corresponding $\mathcal{CRBAC}$ algorithm and update the state of the RBAC system accordingly; using the messages $\{msg_u\}_{u \in U}$ output by the algorithm, the oracle then updates the state of each honest user and sends messages for the corrupt users to $\mathcal{A}$.

The oracle CorruptU lets the adversary take over a user: it sends to $\mathcal{A}$ the local state of that user and adds him to the list $Cr$ of corrupt users. Write simply executes $\mathsf{Write}$ and Challenge writes a challenge into an object, which is then added to the list $Ch$.

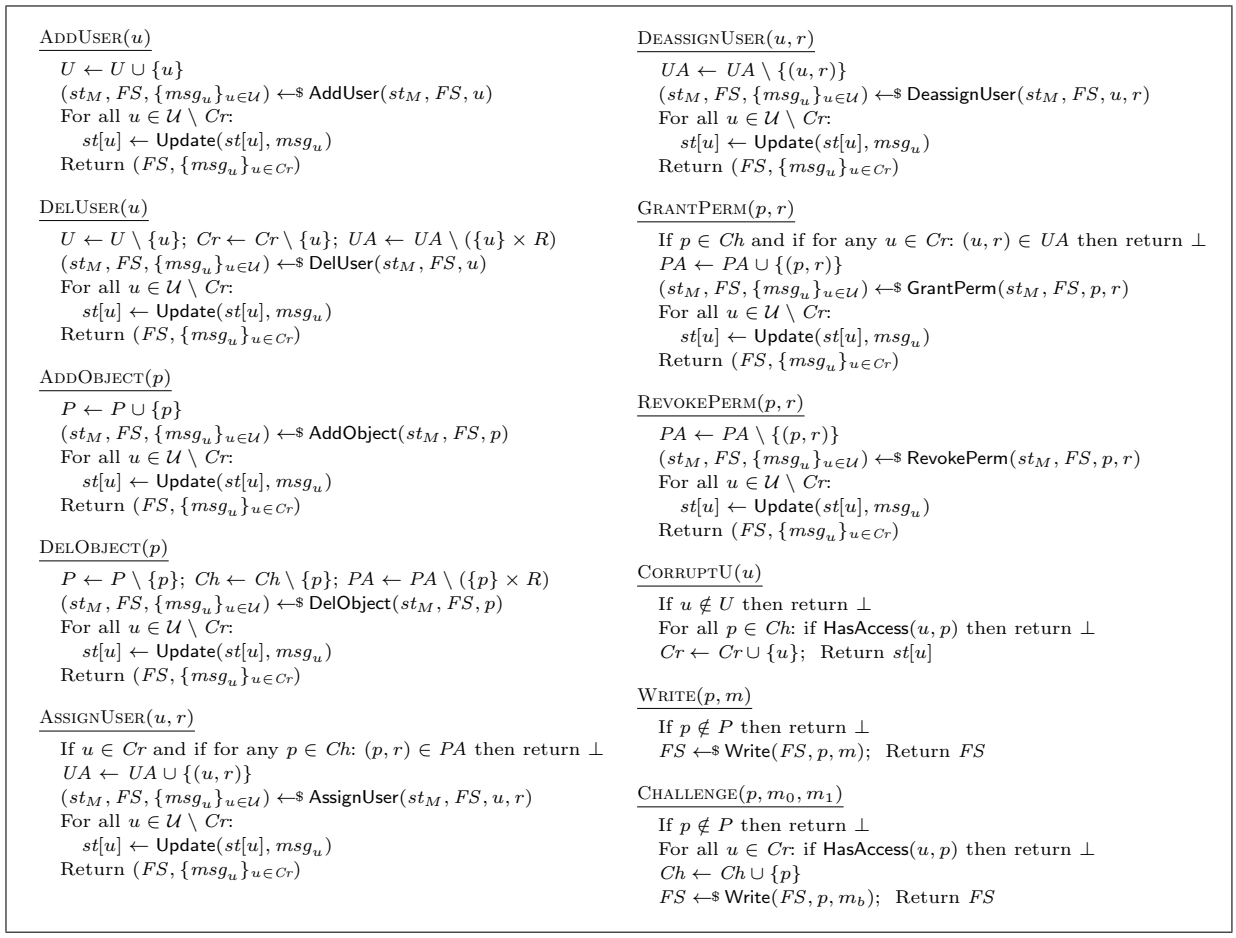


**Fig. 2.** Oracles for $\mathbf{Exp}^{\text{ind}}_{\mathcal{CRBAC},\mathcal{A}}$

Note that if $p \in Ch$ then even after calling WRITE on $p$ it still remains on the list $Ch$, as a corrupt user must still not have access to it: otherwise, the adversary could store what was in $p$ before the WRITE call and later apply Read to it.

Moreover, note that we do not provide the adversary with an oracle for Read for the corrupted users, as the adversary holds their secret keys and has access to $FS$, and could thus run Read on its own.

## 4 Secure Policy Enforcement

In this section we define what it means for a cryptographic RBAC to enforce a policy, as defined in Section 2. A security policy $\Phi$ is defined by a condition Cond and requires that for any user $u$ and any object $p$, when Cond is satisfied by $u$ and $p$ then $\mathsf{HasAccess}(u, p) = 0$. That is, symbolically, the user $u$ should not have access to $p$, in the sense that for all $r \in R : (u, r) \notin UA \vee (p, r) \notin PA$.

A cryptographic RBAC implementation enforces the policy if it guarantees that whenever $\mathsf{Cond}(u, p) = 1$ then user $u$ should also not have access to $p$ "in reality". We define computational access via an indistinguishability game and define secure policy enforcement by demanding that whenever Cond is satisfied for a user $u$ and a permission $p$ then no computational adversary impersonating user $u$ can decide which of two messages of his choice is the content of $p$.

We formalize this via the following game: An adversary impersonates user $u$ and can interact with the manager played by the challenger, who ensures that at any point the symbolic RBAC satisfies $\Phi$. This

models the necessary assumption that the manager enforces policy $\Phi$ symbolically. The adversary can query the challenger to execute any RBAC command and to write content via the $\mathsf{Write}$ command.

At some point the adversary outputs two challenge messages $m_0$ and $m_1$, of which one is randomly selected and written to $p$. The adversary wins the game if at this point we have $\mathsf{Cond}(u,p) = 1$ (and thus $u$ should not have access to $p$ symbolically) and nevertheless the adversary guesses which message was written to $p$. If the probability of the adversary guessing correctly is essentially $\frac{1}{2}$ then we say that the system enforces the security policy.

**Definition 3.** *A cRBAC* enforces a security policy $\Phi$ *of the form*

$$\forall u \in U\ \forall p \in P : \mathsf{Cond}(u,p)\ \Rightarrow\ \neg\mathsf{HasAccess}(u,p)$$

*if for all $u \in U$ and all $p \in P$, we have that for any probabilistic polynomial-time adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ the following is negligible in $\lambda$:*

$$\mathbf{Adv}^{\textit{ind-}\Phi\textit{-}(u,p)}_{\mathcal{CRBAC},\mathcal{A}}(\lambda) := \left|\Pr[\mathbf{Exp}^{\textit{ind-}\Phi\textit{-}(u,p)}_{\mathcal{CRBAC},\mathcal{A}}(\lambda) \to \mathtt{true}] - \tfrac{1}{2}\right|$$

*where for any fixed $(u^*, p^*) \in U \times P$ the experiment $\mathbf{Exp}^{\textit{ind-}\Phi\textit{-}(u^*,p^*)}_{\mathcal{CRBAC},\mathcal{A}}$ is defined as follows:*

$\underline{\mathbf{Exp}^{\textit{ind-}\Phi\textit{-}(u^*,p^*)}_{\mathcal{CRBAC},\mathcal{A}}(\lambda)}$
$b \leftarrow\!\!\$\ \{0,1\}$
$(st_M, FS, \{st[u]\}_{u\in\mathcal{U}}) \leftarrow\!\!\$\ \mathsf{Init}(1^\lambda, R)$
$(m_0, m_1, st_{\mathcal{A}}) \leftarrow\!\!\$\ \mathcal{A}_1(1^\lambda, FS, st[u^*] : \mathcal{O})$
*If* $\mathsf{Cond}(u^*,p^*) = 0$
  $b' \leftarrow\!\!\$\ \{0,1\}$*; return* $(b' = b)$
$FS \leftarrow \mathsf{Write}(FS, p^*, m_b)$
$b' \leftarrow\!\!\$\ \mathcal{A}_2(st_{\mathcal{A}}, FS)$
*Return* $(b' = b)$

The adversary has access to oracles executing RBAC commands and a $\mathsf{Write}$ oracle. As the game represents an RBAC system maintained by a manager enforcing the policy $\Phi$, RBAC commands which would lead to a violation of $\Phi$ are not executed. Each oracle thus first checks whether the RBAC command preserves validity of $\Phi$. If this is not the case, it returns $\perp$. Otherwise, the oracle updates the symbolic state $(U, P, UA, PA)$ and executes the corresponding $\mathcal{CRBAC}$ algorithm. The user messages output by the algorithm are then used to update the states of the honest users, while $msg_{u^*}$ is returned to the adversary, together with the updated file system $FS$. The formal descriptions of $\mathcal{A}$'s oracles $\mathcal{O}$ can be found in Figure 3.[4]

We next show that it is sufficient for a cRBAC to be secure in the sense defined in Section 3, in order to implement any security policy, as defined in Definition 3. The following theorem states that if $\mathcal{CRBAC}$ is secure and the manager symbolically enforces policy $\Phi$ then policy $\Phi$ is satisfied computationally.

**Theorem 1.** *If $\mathcal{CRBAC}$ is a secure cryptographic RBAC then for any security policy $\Phi$, $\mathcal{CRBAC}$ securely enforces $\Phi$.*

**Proof:** Let $\Phi$ be any security policy, $u^* \in \mathcal{U}$ be any user and $p^* \in \mathcal{P}$ be any permission; let $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ be an adversary for $\mathbf{Exp}^{\text{ind-}\Phi\text{-}(u^*,p^*)}_{\mathcal{CRBAC}}$. To prove the theorem, we need to show that $\mathbf{Adv}^{\text{ind-}\Phi\text{-}(u^*,p^*)}_{\mathcal{CRBAC},\mathcal{A}}(\lambda)$ is negligible in $\lambda$. We construct an adversary $\mathcal{B}$ for $\mathbf{Exp}^{\text{ind}}_{\mathcal{CRBAC}}$ which wins this game with the same probability as $\mathcal{A}$.

After receiving $(1^\lambda, FS)$ from its challenger, $\mathcal{B}$ queries CorruptU$(u^*)$ to get $st_{u^*}$ and runs $\mathcal{A}_1$ on $(1^\lambda, FS, st_{u^*})$. $\mathcal{B}$ maintains the symbolic state $(U, P, UA, PA)$ of the RBAC throughout the game. Whenever

[4] For ease of exposition, we introduce an auxiliary routine $\mathsf{UserUpdates}$, which updates all honest user states and returns $(FS, msg_{u^*})$ to the adversary.

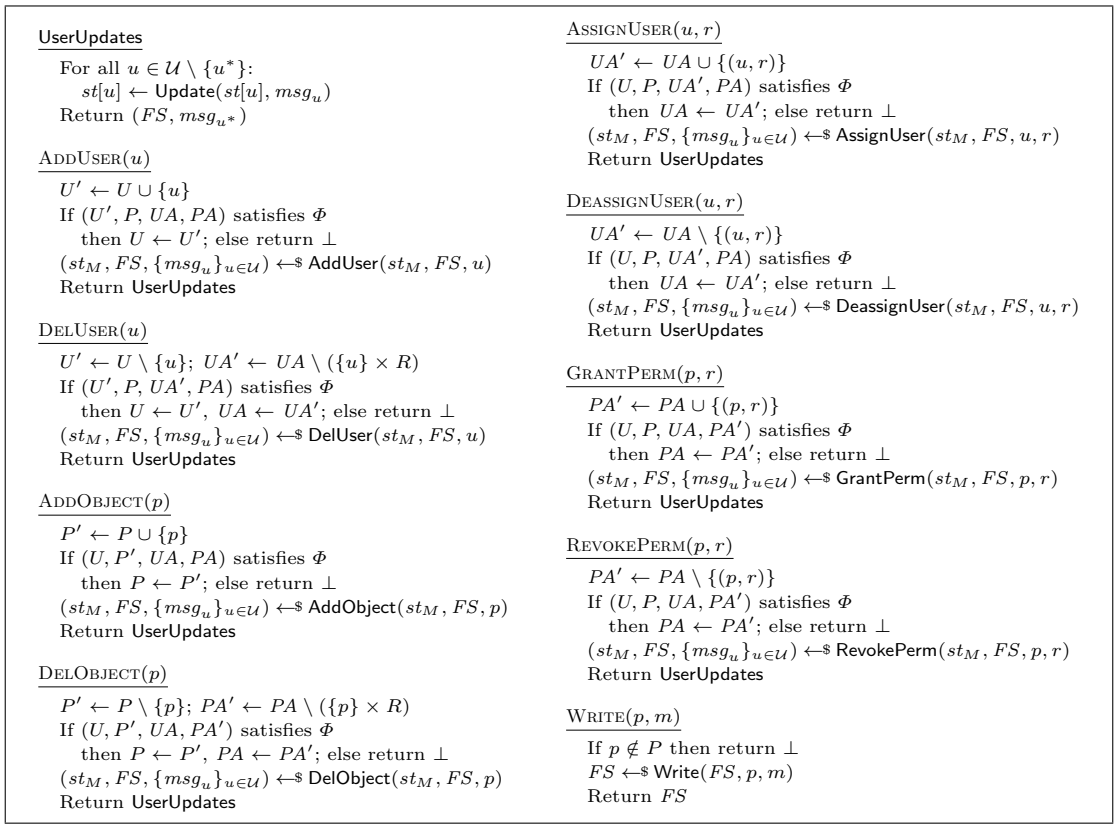


**Fig. 3.** Oracles for $\mathbf{Exp}^{\text{ind-}\Phi\text{-}(u^*,p^*)}_{\mathcal{CRBAC},\mathcal{A}}$

$\mathcal{A}_1$ queries Write, $\mathcal{B}$ relays the query to its challenger and the response to $\mathcal{A}_1$. For any other oracle call, $\mathcal{B}$ first checks whether the call would violate $\Phi$, in which case it returns $\perp$, and otherwise queries its own oracle. More precisely, $\mathcal{B}$ executes the first 3 lines in the description of the $\mathbf{Exp}^{\text{ind-}\Phi\text{-}(u^*,p^*)}_{\mathcal{CRBAC}}$-oracle (see Figure 3); and if it has not returned $\perp$, it makes the corresponding query and forwards the reply to $\mathcal{A}_1$.

Note that $\mathcal{B}$'s oracles never return $\perp$, as in $\mathbf{Exp}^{\text{ind}}_{\mathcal{CRBAC},\mathcal{B}}$, the list $Ch$ remains empty throughout this phase. $\mathcal{B}$ therefore perfectly simulates $\mathbf{Exp}^{\text{ind-}\Phi\text{-}(u^*,p^*)}_{\mathcal{CRBAC}}$ for $\mathcal{A}_1$.

Once $\mathcal{A}_1$ terminates and outputs $(st_{\mathcal{A}}, m_0, m_1)$, $\mathcal{B}$ checks whether $\mathsf{Cond}(u^*, p^*) = 0$ and if so, aborts the simulation and outputs a random bit $b' \leftarrow\$ \{0, 1\}$. Otherwise, $\mathcal{B}$ queries Challenge$(p^*, m_0, m_1)$ and, upon receiving $FS$, runs $\mathcal{A}_2$ on $(st_{\mathcal{A}}, FS)$. When $\mathcal{A}_2$ outputs $b'$, $\mathcal{B}$ outputs the same.

Throughout the game, the state $(U, P, UA, PA)$ which $\mathcal{B}$ maintains (and which always equals the state maintained by $\mathcal{B}$'s challenger) satisfies $\Phi$, because $\mathcal{B}$ does not execute any RBAC command which would lead to a violation of $\Phi$. If $\mathcal{B}$ did not abort then $\mathsf{Cond}(u^*, p^*) = 1$, and since $\Phi$ is satisfied, we have $\neg\mathsf{HasAccess}(u^*, p^*)$.

From this, and since $Cr = \{u^*\}$, we deduce that the call to Challenge$(p^*, m_0, m_1)$ does not return $\perp$. Thus, $\mathcal{B}$ perfectly simulates $\mathbf{Exp}^{\text{ind-}\Phi\text{-}(u^*,p^*)}_{\mathcal{CRBAC}}$ for $\mathcal{A}$ and therefore wins $\mathbf{Exp}^{\text{ind}}_{\mathcal{CRBAC}}$ with the same probability with which $\mathcal{A}$ wins $\mathbf{Exp}^{\text{ind-}\Phi\text{-}(u^*,p^*)}_{\mathcal{CRBAC}}$.

We therefore have $\mathbf{Adv}^{\text{ind-}\Phi\text{-}(u^*,p^*)}_{\mathcal{CRBAC},\mathcal{A}} = \mathbf{Adv}^{\text{ind}}_{\mathcal{CRBAC},\mathcal{B}}$, thus the former is negligible whenever the latter is, which proves the theorem.

Two remarks are in order. The first one is on the fact that the above proof might seem quite straightforward. The security requirement for cRBAC (Definition 2) formalizes that whenever in the symbolic RBAC state we have $\neg$ HasAccess$(u, p)$ then this is also the case computationally: an adversary having corrupted $u$ cannot distinguish challenge messages in $p$, meaning that its content is hidden from the adversary.

It then follows almost immediately that in a cRBAC for a policy that models requirements of users not having permissions (which are precisely the kind of policies we consider), as long as the policy is enforced symbolically, the policy is also enforced computationally.

One then may question if the above result (and the notion of a policy being computationally satisfied) is even worth stating: it is in some sense obvious that policies that are symbolically satisfied are also computationally satisfied (if the cRBAC is secure). We emphatically believe that this is not the case. The main aim of this paper is to present a rigorous approach to the analysis of cryptographic enforcement of access control and a formal definition of computational enforcement of a policy is an important (if not the most important) part of such an approach. Furthermore, although in this paper computational satisfaction of policies is closely related to the notion of a secure cRBAC, in other settings similar relations may not exist (for example, if one considers a larger class of policies than the one considered here). In such settings, proving secure enforcement of policies may need to go a different route.

# 5 Implementation of cRBAC

## 5.1 Predicate Encryption

We choose to describe our implementation of cryptographic RBAC in terms of predicate encryption [21], as this allows us to encompass all of the different kinds of attribute-based encryption (ABE) schemes [24, 18]. In predicate encryption (PE) a ciphertext is associated to an element $y$ from a set $\mathcal{Y}$, whereas keys are associated to elements $x$ from a set $\mathcal{X}$. In a PE scheme for a predicate $\mathit{pred} \subseteq \mathcal{X} \times \mathcal{Y}$, keys for $x$ can decrypt ciphertexts for $y$ if and only if $\mathit{pred}(x, y) = 1$.

For our purpose, having a universe of attributes $A = \{1, \ldots, n_{\max}\}$, we let $\mathcal{X}$ and $\mathcal{Y}$ be the power set of $A$. For $x, y \subseteq A$, the predicate is defined as $\mathit{pred}(x, y) = 1$ if and only if $x \cap y \neq \emptyset$. Thus a ciphertext w.r.t. a set of attributes can be decrypted by a key if it shares an attribute with the ciphertext. We call this *predicate encryption for non-disjoint sets* (PE-NDS).

In (key-policy) ABE [18] messages are encrypted w.r.t. a set of attributes; keys, which are issued for a policy over the attributes, can decrypt those ciphertexts whose attributes satisfy the policy. In the dual notion of *ciphertext-policy* (CP) ABE [7] messages are encrypted w.r.t. policies and keys are issued for sets of attributes.

Predicate encryption for non-disjoint sets is a special case of both flavors of ABE: In CP-ABE by defining the policy for a ciphertext w.r.t. a set of attributes $y = \{a_1, \ldots, a_n\}$ as

$$\phi_y \equiv a_1 \vee \ldots \vee a_n \ . \tag{2}$$

Then any key for a set of attributes $x$ satisfies the policy if and only if $x \cap y \neq \emptyset$, that is, if $\mathit{pred}(x, y) = 1$. For key-policy ABE, it suffices to define the policies for the keys analogously.

Thus, any instantiation of CP-ABE and KP-ABE whose policies include disjunctions of predicates (i.e., for all $y \subseteq A$, $\phi_y$ as in (2) is admitted as a policy) immediately yields an instantiation of PE-NDS. We now formalize PE-NDS.

A PE scheme for the predicate $\mathit{pred} \subseteq \mathcal{P}(A) \times \mathcal{P}(A)$,

$$\mathit{pred}(x, y) = 1 \ \Leftrightarrow \ x \cap y \neq \emptyset$$

is a tuple of algorithms $\mathcal{PENDS} = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$. The setup algorithm $\mathsf{Setup}$ on input the security parameter $\lambda$ and the attribute universe $A$ returns a key pair $(\mathit{pk}, \mathit{msk})$, the master public and secret keys. The key-generation algorithm $\mathsf{KeyGen}$ on inputs $\mathit{msk}$ and $x \subseteq A$ returns a secret key $\mathit{sk}_x$ for the set of attributes $x$. The encryption algorithm $\mathsf{Enc}$ on inputs $\mathit{pk}$ and $y \subseteq A$ as well as $m$ returns a ciphertext $c$. The decryption algorithm $\mathsf{Dec}$ on inputs $\mathit{sk}_x$ and a ciphertext $c$ returns a string $m$. A PE-NDS scheme $\mathcal{PENDS}$ is *correct* if for all $\lambda, x, y, m, r$ satisfying $x \cap y \neq \emptyset$, all $(\mathit{pk}, \mathit{msk})$ output by $\mathsf{Setup}(1^\lambda)$ and all $\mathit{sk}_x$ output by $\mathsf{KeyGen}(\mathit{msk}, x)$ it holds that

$$\mathsf{Dec}(\mathit{sk}_x(\mathsf{Enc}(\mathit{pk}, y, m; r)) = m \ .$$

Since when knowing $\mathit{msk}$ one can always derive a key for $x := A$ and then decrypt, we also directly write $\mathsf{Dec}(\mathit{msk}, c)$.

**Security of PE-NDS** Security of our predicate encryption is defined by the following game in which the adversary $\mathcal{A}$ must guess a bit $b$. $\mathcal{A}$ is given the public key and has access to a challenge oracle LR ("left or right"), which on input $(y, m_0, m_1)$ returns an encryption of $m_b$ w.r.t. the set of attributes $y$, and an oracle KeyGen, which returns a secret key for the queried set $x$. The oracles maintain two sets $X, Y \subseteq A$. $X$ contains the union of all values $x \subseteq A$ queried to KeyGen, and $Y \subseteq A$ contains the union of all values $y \subseteq A$ for which the adversary requested challenge ciphertexts. As long as these two sets are disjoint, the adversary cannot win trivially. Otherwise the adversary could ask for a challenge w.r.t. some $y$ and a key for some $x$ with $x \cap y \neq \emptyset$ and could then decrypt the challenge ciphertext.

**Definition 4.** *Define the following game:*

$\underline{\mathbf{Exp}^{ind}_{\mathcal{PENDS},\mathcal{A}}(\lambda)}$
$b \leftarrow\!\!\$\ \{0,1\},\ X, Y \leftarrow \emptyset$
$(pk, msk) \leftarrow\!\!\$\ \mathsf{Setup}(1^\lambda, A)$
$b' \leftarrow\!\!\$\ \mathcal{A}(pk : \mathrm{LR}, \textsc{KeyGen})$
*Return* $(b' = b)$

$\underline{\mathrm{LR}(y, m_0, m_1)}$
*If* $X \cap y \neq \emptyset$ *then return* $\perp$
$Y \leftarrow Y \cup y$
*Return* $c \leftarrow\!\!\$\ \mathsf{Enc}(pk, y, m_b)$

$\underline{\textsc{KeyGen}(x)}$
*If* $Y \cap x \neq \emptyset$ *then return* $\perp$
$X \leftarrow X \cup x$
*Return* $sk_x \leftarrow\!\!\$\ \mathsf{KeyGen}(msk, x)$

*We say that* $\mathcal{PENDS}$ *has* indistinguishability *if for all probabilistic polynomial-time adversaries* $\mathcal{A}$*, we have that*

$$\mathbf{Adv}^{ind}_{\mathcal{PENDS},\mathcal{A}}(\lambda) := \left| \Pr[\mathbf{Exp}^{ind}_{\mathcal{PENDS},\mathcal{A}}(\lambda) \to \mathtt{true}] - \tfrac{1}{2} \right|$$

*is negligible in* $\lambda$*.*

We remark that in this context CPA-security (that is, when the adversary does not have access to a decryption oracle) is sufficient. This is because since in cRBAC the manager monitors write access, in our instantiation it is the manager who produces encryptions (when running Write), meaning the adversary cannot inject maliciously created ciphertexts.

Finally, note that a PE-NDS implemented by any key-policy (or ciphertext-policy) ABE satisfies Definition 4 if the underlying ABE satisfies the standard indistinguishability notion for ABE.

### 5.2 Implementation of cRBAC with PE-NDS

We implement cryptographic RBAC with predicate encryption by associating roles to attributes and permissions to ciphertexts. Users receive keys which correspond to the set of roles $\{r_1, \ldots, r_n\}$ they are associated with. A file is encrypted w.r.t. the set of attributes/roles which have a permission for it. We also deal with revocation (in that users can be deassigned and permissions revoked). This is implemented as follows: We keep a table $RT$ which keeps associations of roles and attributes. Whenever a user $u^*$ is deassigned from a role $r^*$, the role is associated with a new attribute. The files associated with $r^*$ are then re-encrypted under the new attribute, so that the $u^*$'s key will not decrypt them anymore (except of course when $u^*$ still has access to them via other roles). We then reissue keys to all users which are assigned $r^*$.

We now give our cRBAC implementation $\mathcal{CRBAC}[\mathcal{PE}]$ based on a PE-NDS scheme $\mathcal{PE}$ and start with the initialization algorithm. It sets up the PE-NDS for a sufficiently large attribute universe $A = \{1, \ldots, n_{\max}\}$ and initializes the file system $FS$ (for simplicity, we store the PE-NDS public key $pk$ in $FS[0]$). It then initializes the role table $RT$ by associating attributes to all roles sequentially.

$\underline{\mathsf{Init}(1^\lambda, R)}$

$(pk, msk) \leftarrow\!\!\$\ \mathsf{Setup}_{\mathcal{PE}}(1^\lambda, A)$
$FS, RT \leftarrow \emptyset;\ ctr \leftarrow 1$
For all $r \in R$:
  $RT[r] \leftarrow ctr;\ ctr \leftarrow ctr + 1$
$(U, P, UA, PA) \leftarrow (\emptyset, \emptyset, \emptyset, \emptyset)$
$FS[0] \leftarrow pk;\ st_M \leftarrow (msk, RT, ctr, (U, P, UA, PA))$
Return $(st_M, FS, \{st[u]\}_{u \in \mathcal{U}})$

We next specify how we implement RBAC commands. The corresponding algorithms are given $st_M$ (which we assume is always parsed as $(msk, RT, ctr, (U, P, UA, PA))$) and $FS$, and some command-specific input, and output a new manager state $st_M$, the modified file system $FS$ and messages $msg_u$ for all users $u \in U$.

$\underline{\mathsf{AddUser}(st_M, FS, u^*)}$
  $U \leftarrow U \cup \{u^*\}$
  Return $(st_M, FS, \{\emptyset\}_{u \in U})$

$\underline{\mathsf{AddObject}(st_M, FS, p^*)}$
  $P \leftarrow P \cup \{p^*\}$
  Return $(st_M, FS, \{\emptyset\}_{u \in U})$

When receiving a message $msg$, a user runs $\mathsf{Update}$ on her state (which in our implementation is a secret key for $\mathcal{PE}$) and the message.

$\underline{\mathsf{Update}(st[u], msg_u)}$
  If $msg_u \neq \emptyset$ then $st[u] \leftarrow msg_u$
  Return $st[u]$

To assign a role $r^*$ to a user $u^*$, first $(u^*, r^*)$ is added to the user-role assignment relation $UA$. Then the user is given a secret key for the set $x$ of attributes which (via $RT$) currently correspond to the user's new set of roles:

$\underline{\mathsf{AssignUser}(st_M, FS, u^*, r^*)}$
  If $u^* \notin U$ or $r^* \notin R$ or $(u^*, r^*) \in UA$
    Return $(st_M, FS, \{\emptyset\}_{u \in U})$
  $UA \leftarrow UA \cup \{(u^*, r^*)\}$
  $x \leftarrow \{RT[r] \mid (u^*, r) \in UA\}$
  $sk_{u^*} \leftarrow \mathsf{KeyGen}_{\mathcal{PE}}(msk, x);\ msg_{u^*} \leftarrow sk_{u^*}$
  For all $u \in U \setminus \{u^*\}$:
    $msg_u \leftarrow \emptyset$
  Return $(st_M, FS, \{msg_u\}_{u \in U})$

To grant a permission $p^*$ to a role $r^*$, the manager adds $(p^*, r^*)$ to $PA$, and then re-encrypts the content of $FS[p^*]$ under the attributes corresponding to the roles $r$ for which $(p^*, r) \in PA$:

$\underline{\mathsf{GrantPerm}(st_M, FS, p^*, r^*)}$
  If $p^* \notin P$ or $r^* \notin R$ or $(p^*, r^*) \in PA$
    Return $(st_M, FS, \{\emptyset\}_{u \in U})$
  $PA \leftarrow PA \cup \{(p^*, r^*)\}$
  $y \leftarrow \{RT[r] \mid (p^*, r) \in PA\}$
  $FS[p^*] \leftarrow \mathsf{Enc}_{\mathcal{PE}}(FS[0], y, \mathsf{Dec}_{\mathcal{PE}}(msk, FS[p^*]))$
  Return $(st_M, FS, \{\emptyset\}_{u \in U})$

For convenience, we define the following auxiliary algorithm, which assigns a new attribute to a role $r^*$ and updates the user keys and $FS$ accordingly. It writes the current counter value to $RT[r^*]$, then re-encrypts all files associated to $r^*$ and re-issues all user keys associated to $r^*$.

$\underline{\mathsf{RoleUpdate}(st_M, FS, r^*)}$

$RT[r^*] \leftarrow ctr$; $ctr \leftarrow ctr + 1$
For all $p \in P$:
  If $(p, r^*) \in PA$:
    $y \leftarrow \{RT[r] \,|\, (p, r) \in PA\}$
    $FS[p] \leftarrow \mathsf{Enc}_{\mathcal{PE}}(FS[0], y, \mathsf{Dec}_{\mathcal{PE}}(msk, FS[p]))$
For all $u \in U$:
  If $(u, r^*) \in UA$:
    $x \leftarrow \{RT[r] \,|\, (u, r) \in UA\}$
    $sk_u \leftarrow \mathsf{KeyGen}_{\mathcal{PE}}(msk, x)$; $msg_u \leftarrow sk_u$
  Else $msg_u \leftarrow \emptyset$
Return $(st_M, FS, \{msg_u\}_{u \in U})$

Now to deassign a user $u^*$ from a role $r^*$, it suffices to erase $(u^*, r^*)$ from $UA$ and then update the role $r^*$, so the role is associated to a new attribute for which $u^*$ will not have a key anymore. To delete a user $u^*$, we first deassign all its roles and then remove $u^*$ from $U$.

$\underline{\mathsf{DeassignUser}(st_M, FS, u^*, r^*)}$

If $(u^*, r^*) \notin UA$ then return $(st_M, FS, \{\emptyset\}_{u \in U})$
$UA \leftarrow UA \setminus \{(u^*, r^*)\}$
Return $\mathsf{RoleUpdate}(st_M, FS, r^*)$

$\underline{\mathsf{DelUser}(st_M, FS, u^*)}$

If $u^* \notin U$ then return $(st_M, FS, \{\emptyset\}_{u \in U})$
For all $r \in R$:
  If $(u^*, r) \in UA$:
    $(st_M, FS, \{msg_u\}_{u \in U})$
      $\leftarrow \mathsf{DeassignUser}(st_M, FS, u^*, r)$
$U \leftarrow U \setminus \{u^*\}$
Return $(st_M, FS, \{msg_u\}_{u \in U})$

To revoke a permission $p^*$ from role $r^*$, we delete $(p^*, r^*)$ from $PA$ and then re-encrypt the content of $p^*$ under the reduced set of attributes. To delete an object $p^*$, we revoke it from all roles and erase it from $P$.

$\underline{\mathsf{RevokePerm}(st_M, FS, p^*, r^*)}$

If $(p^*, r^*) \notin PA$ then return $(st_M, FS, \{\emptyset\}_{u \in U})$
$PA \leftarrow PA \setminus \{(p^*, r^*)\}$
$y \leftarrow \{RT[r] \,|\, (p^*, r) \in PA\}$
$FS[p^*] \leftarrow \mathsf{Enc}_{\mathcal{PE}}(FS[0], y, \mathsf{Dec}_{\mathcal{PE}}(msk, FS[p^*]))$
Return $\mathsf{RoleUpdate}(st_M, FS, r^*)$

$\underline{\mathsf{DelObject}(st_M, FS, p^*)}$

If $p^* \notin P$ then return $(st_M, FS, \{\emptyset\}_{u \in U})$
For all $r \in R$:
  If $(p^*, r) \in PA$:
    $(st_M, FS, \{msg_u\}_{u \in U})$
      $\leftarrow \mathsf{RevokePerm}(st_M, FS, p^*, r)$
$P \leftarrow P \setminus \{p^*\}$; $FS[p^*] \leftarrow \emptyset$
Return $(st_M, FS, \{\emptyset\}_{u \in U})$

Finally, we give the two algorithms for writing and reading content in the file system. (Recall that $FS[0]$ contains the encryption key.)

$\underline{\mathsf{Write}(FS, p^*, m)}$
$y \leftarrow \{RT[r] \mid (p^*, r) \in PA\}$
$FS[p] \leftarrow \mathsf{Enc}_{\mathcal{PE}}(FS[0], y, m)$
Return $FS$

$\underline{\mathsf{Read}(sk_u, FS, p^*)}$
$m \leftarrow \mathsf{Dec}_{\mathcal{PE}}(sk_u, FS[p^*])$
Return $m$

A few remarks are in order. After deassigning a user, the attribute corresponding to a role has to be renewed, so that future files cannot be decrypted by the deassigned user anymore. It may appear less natural to renew a role's attribute when we revoke a permission, as it seems sufficient to re-encrypt the file without the revoked role's attribute. However, consider a corrupt user $u$ trying to gain access to a file $p$. If $p$ was encrypted as $c$ under a role $r$, which is later revoked (resulting in a ciphertext $c'$), then assigning $u$ to $r$ is allowed, as $\mathsf{HasAccess}(u, p) = 0$ at all times. However, $u$ would be able to read the content of $p$, as after being assigned $r$, she could use the resulting secret key to decrypt, not the current content $c'$ of $p$, but $c$.

This attack is thwarted in our instantiation, as the attribute for $r$ under which $p$ had been encrypted was changed when revoking $r$ from $p$; thus when $u$ is assigned $r$ later, her key will be associated to the new attribute, meaning that she cannot decrypt $c'$.

### 5.3 Security of Our Implementation

**Theorem 2.** *If $\mathcal{PE}$ satisfies indistinguishability then $\mathcal{CRBAC}[\mathcal{PE}]$, as defined above, is a secure implementation, as defined in Definition 2.*

**Proof:** We prove the security of $\mathcal{CRBAC}[\mathcal{PE}]$ by reduction to indistinguishability of $\mathcal{PE}$. Let $\mathcal{A}$ be an adversary against $\mathcal{CRBAC}[\mathcal{PE}]$, which we will from now on denote $\mathcal{CRBAC}$ for simplicity. We construct an adversary $\mathcal{B}$ against $\mathcal{PE}$ such that $\mathbf{Adv}^{\text{ind}}_{\mathcal{PE},\mathcal{B}}(\cdot) = \mathbf{Adv}^{\text{ind}}_{\mathcal{CRBAC},\mathcal{A}}(\cdot)$. Thus if $\mathcal{PE}$ is secure then so is $\mathcal{CRBAC}$.

In $\mathbf{Exp}^{\text{ind}}_{\mathcal{PE}}$, $\mathcal{B}$ is given $pk$ and has access to two oracles LR and KeyGen. $\mathcal{B}$ starts the simulation of $\mathbf{Exp}^{\text{ind}}_{\mathcal{CRBAC},\mathcal{A}}$ by initializing $Cr, Ch \leftarrow \emptyset$. However, $\mathcal{B}$ does not choose a bit $b$, as it simulates $\mathbf{Exp}^{\text{ind}}_{\mathcal{CRBAC},\mathcal{A}}$ with the bit $b$ set to the one its own challenger chose in $\mathbf{Exp}^{\text{ind}}_{\mathcal{PE},\mathcal{B}}$. Next, $\mathcal{B}$ simulates $\mathsf{Init}$, except that instead of running $\mathsf{Setup}_{\mathcal{PE}}$, it uses the received $pk$ and leaves $msk$ blank in $st_M$.

$\mathcal{B}$ then answers $\mathcal{A}$'s oracle calls by running the algorithms of $\mathcal{CRBAC}$ as specified, with three exceptions: $\mathcal{B}$ does not create any user secret keys $sk_u$ for honest users; only when the adversary corrupts a user, $\mathcal{B}$ calls its KeyGen oracle to obtain a key. As $\mathcal{B}$ does not hold the master secret key, it cannot decrypt as required by the subroutine $\mathsf{RoleUpdate}$. It therefore stores all plaintexts in a table $MS$ ("message system", in analogy to $FS$), where it can look them up instead of decrypting.

Finally, when $\mathcal{A}$ queries its Challenge oracle for $(p, m_0, m_1)$, $\mathcal{B}$ forwards the encryption request to its own LR oracle to obtain an encryption of $m_b$. $\mathcal{B}$ simulates thus $\mathbf{Exp}^{\text{ind}}_{\mathcal{CRBAC}}$ for $\mathcal{A}$ with the same bit $b$ that was chosen by $\mathcal{B}$'s own challenger. As $\mathcal{B}$ does not know $b$, it stores both $m_0$ and $m_1$ in $MS$ and calls LR whenever it requires an encryption of $m_b$. Finally, $\mathcal{B}$ outputs the same bit $b'$ that $\mathcal{A}$ outputs.

The more technical part of the proof is to show that $\mathcal{B}$'s queries to LR and KeyGen never return $\perp$, as then $\mathcal{B}$'s simulation of $\mathbf{Exp}^{\text{ind}}_{\mathcal{CRBAC}}$ is perfect and $\mathcal{B}$'s probability of outputting $b' = b$ is the same as $\mathcal{A}$'s.

We formally specify $\mathcal{B}$:

$\underline{\mathcal{B}(pk : \text{LR}, \text{KeyGen})}$
$Cr, Ch \leftarrow \emptyset$
$FS, RT \leftarrow \emptyset$; $ctr \leftarrow 1$
For all $r \in R$:
  $RT[r] \leftarrow ctr$; $ctr \leftarrow ctr + 1$
$(U, P, UA, PA) \leftarrow (\emptyset, \emptyset, \emptyset, \emptyset)$; $FS[0] \leftarrow pk$

$b' \leftarrow\!\!\$\ \mathcal{A}(1^\lambda, FS : \mathcal{O})$
Return $b'$

$\mathcal{B}$ answers $\mathcal{A}$'s oracle calls by following their description from Figure 2, using the implementation specified in Section 5.2—except for the changes discussed above.

$\underline{\textsc{AddUser}(u^*)}$
$U \leftarrow U \cup \{u^*\}$
Return $(FS, \{\emptyset\}_{u \in Cr})$

$\underline{\textsc{AddObject}(p^*)}$
$P \leftarrow P \cup \{p^*\}$
Return $(FS, \{\emptyset\}_{u \in Cr})$

$\underline{\textsc{CorruptU}(u^*)}$
If $u^* \notin U$ then return $\perp$
For all $p \in Ch$: if $\mathsf{HasAccess}(u^*, p)$ then return $\perp$
$Cr \leftarrow Cr \cup \{u^*\}$
$x \leftarrow \{RT[r] \mid (u^*, r) \in UA\}$
$sk_{u^*} \leftarrow \textsc{KeyGen}(x)$
Return $sk_{u^*}$

$\underline{\textsc{AssignUser}(u^*, r^*)}$
If $u^* \in Cr$ and if for any $p \in Ch$: $(p, r^*) \in PA$
then return $\perp$
If $u^* \notin U$ or $r^* \notin R$ or $(u^*, r^*) \in UA$
Return $(FS, \{\emptyset\}_{u \in Cr})$
$UA \leftarrow UA \cup \{(u^*, r^*)\}$
If $u^* \in Cr$:
$x \leftarrow \{RT[r] \mid (u^*, r) \in UA\}$
$sk_{u^*} \leftarrow \textsc{KeyGen}(x)$; $msg_{u^*} \leftarrow sk_{u^*}$
For all $u \in Cr \setminus \{u^*\}$:
$msg_u \leftarrow \emptyset$
Return $(st_M, FS, \{msg_u\}_{u \in Cr})$

$\underline{\textsc{Write}(p^*, m)}$
If $p^* \notin P$ then return $\perp$
$MS \leftarrow (m, \cdot)$
$y \leftarrow \{RT[r] \mid (p^*, r) \in PA\}$
$FS[p^*] \leftarrow \mathsf{Enc}_{\mathcal{PE}}(FS[0], y, m)$
Return $FS$

$\underline{\textsc{Challenge}(p^*, m_0, m_1)}$
If $p^* \notin P$ then return $\perp$
For all $u \in Cr$: If $\mathsf{HasAccess}(u, p^*)$ then return $\perp$
$Ch \leftarrow Ch \cup \{p^*\}$
$MS \leftarrow (m_0, m_1)$
$y \leftarrow \{RT[r] \mid (p^*, r) \in PA\}$
$FS[p^*] \leftarrow \mathrm{LR}(y, m_0, m_1)$
Return $FS$

$\underline{\textsc{GrantPerm}(p^*, r^*)}$
If $p^* \in Ch$ and if for any $u \in Cr$: $(u, r^*) \in UA$
then return $\perp$
If $p^* \notin P$ or $r^* \notin R$ or $(p^*, r^*) \in PA$
Return $(FS, \{\emptyset\}_{u \in Cr})$

$PA \leftarrow PA \cup \{(p^*, r^*)\}$
$y \leftarrow \{RT[r] \,|\, (p^*, r) \in PA\}$
If $p^* \in Ch$:
  $FS[p^*] \leftarrow \mathrm{LR}(y, MS[p^*][0], MS[p^*][1])$
Elseif $FS[p^*] \neq \emptyset$:
  $FS[p^*] \leftarrow \mathsf{Enc}_{\mathcal{PE}}(FS[0], y, MS[p^*][0])$
Return $(FS, \{\emptyset\}_{u \in Cr})$

The next auxiliary algorithm simulates RoleUpdate, except that it only updates the keys of corrupt users. (Recall that $\mathcal{B}$ does not query keys for honest users.)

RoleUpdateCr($r^*$)
$RT[r^*] \leftarrow ctr$; $ctr \leftarrow ctr + 1$
For all $p \in P$:
  If $(p, r^*) \in PA$:
    $y \leftarrow \{RT[r] \,|\, (p, r) \in PA\}$
    If $p \in Ch$:
      $FS[p] \leftarrow \mathrm{LR}(y, MS[p][0], MS[p][1])$
    Elseif $FS[p] \neq \emptyset$:
      $FS[p] \leftarrow \mathsf{Enc}_{\mathcal{PE}}(FS[0], y, MS[p][0])$
For all $u \in Cr$:
  If $(u, r^*) \in UA$:
    $x \leftarrow \{RT[r] \,|\, (u, r) \in UA\}$
    $sk_u \leftarrow \textsc{KeyGen}(x)$; $msg_u \leftarrow sk_u$
  Else $msg_u \leftarrow \emptyset$
Return $(FS, \{msg_u\}_{u \in Cr})$

DeassignUser($u^*, r^*$)
If $(u^*, r^*) \notin UA$ then return $(FS, \{\emptyset\}_{u \in Cr})$
$UA \leftarrow UA \setminus \{(u^*, r^*)\}$
Return RoleUpdateCr($r^*$)

DelUser($u^*$)
If $u^* \notin U$ then return $(FS, \{\emptyset\}_{u \in Cr})$
For all $r \in R$:
  If $(u^*, r) \in UA$:
    $(FS, \{msg_u\}_{u \in Cr}) \leftarrow \textsc{DeassignUser}(u^*, r)$
$U \leftarrow U \setminus \{u^*\}$; $Cr \leftarrow Cr \setminus \{u^*\}$
Return $(FS, \{msg_u\}_{u \in Cr})$

RevokePerm($p^*, r^*$)
If $(p^*, r^*) \notin PA$ then return $(FS, \{\emptyset\}_{u \in Cr})$
$PA \leftarrow PA \setminus \{(p^*, r^*)\}$
$y \leftarrow \{RT[r] \,|\, (p^*, r) \in PA\}$
If $p^* \in Ch$:
  $FS[p^*] \leftarrow \mathrm{LR}(y, MS[p^*][0], MS[p^*][1])$
Elseif $FS[p^*] \neq \emptyset$:
  $FS[p^*] \leftarrow \mathsf{Enc}_{\mathcal{PE}}(FS[0], y, MS[p^*][0])$
Return RoleUpdateCr($r^*$)

DelObject($p^*$)
If $p^* \notin P$ then return $(FS, \{\emptyset\}_{u \in Cr})$
For all $r \in R$:

If $(p^*, r) \in PA$:
$(FS, \{msg_u\}_{u \in Cr}) \leftarrow \textsc{RevokePerm}(p^*, r)$
$P \leftarrow P \setminus \{p^*\}$; $Ch \leftarrow Ch \setminus \{p^*\}$; $FS[p^*] \leftarrow \emptyset$
Return $(FS, \{\emptyset\}_{u \in Cr})$

Below we show the following:

**Lemma 1.** *In the above simulation, $\mathcal{A}$'s oracles* LR *and* KeyGen *never return $\perp$.*

Based on this lemma, we argue that $\mathcal{B}$'s simulation is perfect: this follows, as $\mathcal{B}$ simply implements the oracles by definition, except that it keeps a table of clear messages $MS$ instead of decrypting the ciphertexts (which, by correctness of $\mathcal{PE}$, amounts to the same); it uses its LR oracle to generate encryptions of $m_b$, and generates user keys only when they are corrupted. The latter is perfectly indistinguishable from $\mathcal{A}$'s perspective. It follows thus that $\mathcal{B}$ guesses $b$ correctly whenever $\mathcal{A}$ does, and thus $\mathbf{Adv}^{\text{ind}}_{\mathcal{PE},\mathcal{B}}(\cdot) = \mathbf{Adv}^{\text{ind}}_{\mathcal{CRBAC},\mathcal{A}}(\cdot)$, which completes the proof.

**Proof:**[Proof of Lemma 1] In order to prove that $\mathcal{B}$'s oracles do not return $\perp$, it suffices to show that the following condition is an invariant of the experiment:

$$X \cap Y = \emptyset \ , \tag{C1}$$

where $X$ contains all $x$ queried to KeyGen and $Y$ contains all $y$ queried to LR (these are precisely the sets which $\mathcal{B}$'s challenger keeps in $\mathbf{Exp}^{\text{ind}}_{\mathcal{PE}}$).

We prove by induction over the number of oracle calls in $\mathbf{Exp}^{\text{ind}}_{\mathcal{CRBAC}}$ that (C1) is always satisfied, which proves the lemma. To do so, we simultaneously prove that three more conditions are also satisfied. The first asserts that, symbolically, a corrupt user never has access to a challenge permission:

$$\forall u \in Cr\, \forall p \in Ch : \neg \mathsf{HasAccess}(u, p) \ , \tag{C2}$$

with $\mathsf{HasAccess}(u, p) \Leftrightarrow \exists r \in R : (u, r) \in UA \wedge (p, r) \in PA$. At any point in the game, let $Z := \{RT[r] \mid r \in R\}$ denote the set of current attributes associated to roles. For all $a \in Z$, let $r_a$ denote the role s.t. $RT[r] = a$ (since $\mathsf{RoleUpdateCr}$ increases $ctr$ after assigning its value to an element of $RT$, $r_a$ is unique). We also prove that the following two conditions hold throughout the simulation:

$$a \in X \cap Z \ \Leftrightarrow \ \exists u \in Cr : (u, r_a) \in UA \tag{C3}$$

$$a \in Y \cap Z \ \Leftrightarrow \ \exists p \in Ch : (p, r_a) \in PA \tag{C4}$$

Intuitively, (C3) means that if a currently-in-use attribute is in $X$ then it is because there is a corrupt user who is assigned the corresponding role. Analogously, (C4) means that if a current attribute is on the list $Y$ then it is because its role $r_a$ is associated to a challenge permission.

It is immediate that at the beginning of the game all four conditions are satisfied, as $X = Y = Cr = Ch = \emptyset$.

Below we will show that oracle calls only add elements to $X$ and $Y$ which are also in $Z$. Thus if $X \cap Y = \emptyset$ before the oracle call then to show (C1), it suffices to show that $(X \cap Z) \cap (Y \cap Z) = \emptyset$ afterwards.

If there was an element $a$ in this set then by (C3) and (C4) we would have

$$\exists u \in Cr\, \exists p \in Ch : (u, r_a) \in UA \wedge (p, r_a) \in PA \ ,$$

which contradicts (C2). In order to show (C1) (and thus prove the lemma), it therefore suffices to show (C2), (C3), (C4) and:

$$\text{All elements added to } X \text{ and } Y \text{ are also in } Z. \tag{C5}$$

We now show that whenever conditions (C2), (C3), (C4) and (C5) are satisfied before an oracle call then they still are afterwards.

(C5): The above is easily verified by looking at the implementations of the oracles: LR is always called with an argument $y$ s.t. $y \subseteq \{RT[r] \mid r \in R\} = Z$; likewise, KeyGen is always called with $x \subseteq Z$. Note also that RoleUpdateCr updates the table $RT$ *before* calling the oracles.

(C2): We next show if (C2) is satisfied before an oracle call then so it is afterwards: (C2) states that $\forall u \in Cr\, \forall p \in Ch\, \forall r \in R : (u, r) \notin UA \vee (p, r) \notin PA$. This can only cease to hold if an oracle adds elements to $Cr$, $Ch$, $UA$ or $PA$, respectively. However, the only oracles doing this are CorruptU, Challenge, AssignUser and GrantPerm, respectively, which all first detect whether they would cause a violation of (C2), in which case they return $\perp$.

(C3) and (C4): These two statements are about $X, Y, Cr, Ch$ and $RT$ (the latter defining $Z$). Oracles AddUser, AddObject, Write do not modify any of these sets, thus the conditions hold whenever they held before the call. We analyze the remaining oracles one by one.

- CorruptU: If (C3) holds before the call then it also holds after: $Z$ is not modified and all $a$ which were in $X$ before still correspond to users in $Cr$; for all attributes which were added to $X$, the corresponding user is $u^* \in Cr$. Conversely, if $(u^*, r) \in UA$ then $RT[r]$ is now in $X$. (C4) is not affected, as neither $Y$, $RT$, $Ch$, nor $PA$ are modified.
- AssignUser: Again, (C4) is not affected, as none of the relevant sets change. Validity of (C3) is also maintained: if $u^* \notin Cr$ then $X$ is not modified and $UA$ is only extended by a non-corrupt user, which together do not affect (C3). If $u^* \in Cr$ then (as for CorruptU), (C3) is still valid: for all $a$ which were in $X \cap Z$ before there still exists a corrupt user such that $(u, r_a) \in UA$ and vice versa. Any new $a$ is added to $X$ if and only if $(u^*, r_a) \in UA$.
- Challenge: We show validity of (C4) is maintained: all $a$ which were in $Y \cap Z$ still correspond to their $p \in Ch$, and vice versa; and any new $a$ is added to $Y$ if and only if $(p^*, r_a) \in PA$, so these $a$'s correspond to $p^* \in Ch$. Condition (C3) is not affected, since $X$, $RT$, $Cr$ and $UA$ are unaffected.
- GrantPerm: Again, (C3) is not affected, for the same reasons as above. If $p^* \notin Ch$ then $Y$ is not modified and $PA$ is only extended by a non-challenge policy $p^*$, which together do not affect (C4). If $p^* \in Ch$ then (as for Challenge), (C4) is still valid: for elements which were in $Y \cap Z$ and $Ch$ before the oracle call, both implications of (C4) still hold. Moreover, any $a$ is added to $Y$ if and only if $(p^*, r_a) \in PA$.
- RoleUpdateCr: This subroutine, run on $r^*$, replaces $a_{\text{old}}$ by $a_{\text{new}}$ in $RT[r^*]$. Thus after the call, $a_{\text{old}}$ is not in $Z$ anymore. If for some $p \in Ch$: $(p, r^*) \in PA$, the routine then queries LR, which adds $a_{\text{new}}$ to $Y$; thus (C4) still holds with $a_{\text{old}}$ replaced by $a_{\text{new}}$. Otherwise, $\forall p \in Ch : (p, r^*) \notin PA$, so as (C4) held before the call: $a_{\text{old}} \notin Y \cap Z$. As in this case, LR is not called, we have $a_{\text{new}} \notin Y$, thus (C4) holds as before.

  Completely analogously, (C3) is maintained, since if $a_{\text{old}}$ was in $X \cap Z$ because of some user $u$ then now $a_{\text{new}}$ is in $X \cap Z$ and still associated to the same user via $r^* = r_{a_{\text{new}}}$; and if $a_{\text{old}}$ was not in $X \cap Z$ then $a_{\text{new}}$ is not added to it either.
- DeassignUser: This oracle removes $(u^*, r^*)$ from $UA$ and then calls RoleUpdateCr on $r^*$. If for some $u \in Cr, u \neq u^* : (u, r^*) \in UA$ then (C3) still holds, as $a_{\text{old}}$ is replaced by $a_{\text{new}}$ in $X \cap Z$, which is then associated to $u$. On the other hand, if after the revocation we have $\forall u \in Cr : (u, r^*) \notin UA$ then $a_{\text{new}}$ is not added to $X \cap Z$ and again, (C3) still holds.

  (C4) is not affected by DeassignUser as it only changes $UA$ and RoleUpdateCr preserves validity of (C4).
- DelUser: This oracle first runs DeassignUser for all roles held by the user. We have already shown that this preserves validity of (C3) and (C4). After this we have $\forall r \in R : (u^*, r) \notin UA$. Thus (C3) holds independently of $u^*$, which means that after removing $u^*$ from $U$ (and possibly $Cr$) (C3) still holds. Moreover, (C4) is not affected by this either.
- RevokePerm: If $p^* \notin Ch$ then this does not affect (C4), as $Y$ is not modified and $PA$ is only reduced by a pair not containing a challenge permission. Executing RoleUpdateCr does not affect (C4) either, as shown above. If $p^* \in Ch$ then, by (C4), we have: $\forall r : (p^*, r) \in PA \Rightarrow RT[r] \in Y \cap Z$, so the LR call leaves $Y$ unchanged, so (C4) still holds. Now RoleUpdateCr is called, but with $(p^*, r^*)$ being erased from $PA$. Thus, while $a_{\text{old}}$ gets removed from $Y \cap Z$, $a_{\text{new}}$ only gets added if for some $p \in Ch, p \neq p^*$: $(p, r^*) \in PA$; in this case (C4) holds afterwards, as this $p$ is associated with $a_{\text{new}}$. If after the revocation $\forall p \in Ch : (p, r^*) \notin PA$ then $a_{\text{new}} \notin Y \cap Z$ and (C4) still holds as well.

  (C3) is not affected by RevokePerm, as it only changes $PA$ and RoleUpdateCr preserves validity of (C3), as above.

• DelObject: This oracle first runs RevokePerm for all roles associated to $p^*$, which, as just shown, preserves validity of (C3) and (C4). After this, we have $\forall r : (p^*, r) \notin PA$. Thus (C4) (and (C3)) holds independently of $p^*$; therefore, they still hold after removing $p^*$ from $P$ and $Ch$. This concludes the proof of the lemma and thereby the proof of the theorem.

### 5.4 Further Implementations of PE-NDS

We chose to describe our implementation using a predicate-encryption (PE) scheme for the simple predicate $pred(x, y) \Leftrightarrow x \cap y \neq \emptyset$ (non-disjoint sets (NDS) of attributes), because it closely models the minimal functionality required for our cryptographic RBAC.

To implement PE-NDS, the most natural choice is attribute-based encryption, of which it is a special case. Here we mention other possible implementations of PE-NDS, which in turn would lead to alternative cryptographic RBAC implementations.

**Parallel Encryption** This implementation is the least efficient, but uses only a very basic cryptographic primitive: public-key encryption (PKE). The setup creates a PKE key pair $(pk_a, sk_a)$ for every attribute $a$ in the attribute universe $A$. The master public and secret keys are defined as $pk = \{pk_a\}_{a \in A}$ and $msk = \{sk_a\}_{a \in A}$. $\mathsf{KeyGen}$, on input $msk$ and $x \subseteq A$ returns $sk_x = \{sk_a\}_{a \in x}$ and $\mathsf{Enc}$, on input $(pk, y, m)$ returns a set $\{c_a\}_{a \in y}$, where $c_a$ is a PKE encryption of $m$ under $pk_a$.

This can be made more efficient using a *hybrid-encryption* approach [26], where $\mathsf{Enc}(pk, y, m)$ first chooses a key $K$ for symmetric encryption, then encrypts $m$ under $K$ and uses PKE to encrypt $K$ under every key in $\{pk_a\}_{a \in y}$.

**Broadcast Encryption** Another implementation, situated somehow between ABE and hybrid encryption, uses *broadcast encryption* [15]. In broadcast encryption (BE) there is a set of receivers holding secret keys and a message can be broadcast to any subset of receivers such that only users in the subset can decrypt.

Associating attributes $a \in A$ to receivers in a BE scheme, we get the following implementation of PE-NDS: Running the BE setup returns a public key $pk$, which is used as the public key for the PE-NDS and a set of secret receiver keys $\{sk_a\}_{a \in A}$ which we define as $msk$. As for parallel encryption, key generation is done by selecting the keys $\{sk_a\}_{a \in x}$ which correspond to the set $x$; encryption w.r.t. a set $y$ is broadcast encryption to the corresponding set $y$ of receivers.

Since the goal of BE is to minimize the size of a ciphertext sent to many receivers, this approach might lead to smaller ciphertexts and thus consequently minimizes the size of $FS$ in the resulting implementation of $\mathcal{CRBAC}$. On the downside, the user secret keys are sets of BE receiver secret keys, which in general may be bigger than ABE secret keys.

## 6 Discussion and Future Work

In this paper we use well-established methodologies in modern cryptography to formulate precise syntax and security requirements for cryptographic access control. Besides the definition for secure cryptographic implementations, our results include a security proof for a cryptographic RBAC based on attribute-based encryption and a theorem that shows that in such systems a policy is satisfied (computationally) as long as the manager ensures that the policy is satisfied symbolically.

There are several directions in which our work can be extended. An obvious target is to provide a similar treatment to other variants of RBAC (e.g. hierarchical RBAC [2] and attribute-based RBAC [4, 27]). Much more ambitiously, it would be interesting to cast our results as instances of a more general framework with abstract notions of symbolic and computational access-control enforcement. Flexible mechanisms for policy specification, e.g. through general logics would also be desirable, but one would have to deal with the difficulties associated with formalizing general computational satisfaction of logic formulas [12, 11, 6].

Our analysis identifies predicate encryption for non-disjoint sets as a sufficient primitive for meeting the functionality and security goals of cRBAC. This allows avoiding general attribute-based encryption (or even worse, general predicate-encryption schemes) for the implementation. From this perspective, a rigorous analysis of cryptographic access-control implementations can also serve for identifying and choosing a trade-off between the efficiency, functionality and the security of the primitives and that of the resulting access-control system.

Finally, it would be interesting to understand and formally relate the approach that we take here with the simulation-based approach, as applied to access control by Halevi, Karger and Naor [19].

## References


1. http://www.cs.stonybrook.edu/∼stoller/ccs2007/.
2. American National Standards Institute: ANSI INCITS 359-2004 for Role Based Access Control, 2004.
3. Martín Abadi and Bogdan Warinschi. Security analysis of cryptographically controlled access to XML documents. *J. ACM*, 55(2), 2008.
4. Mohammad A. Al-Kahtani and Ravi S. Sandhu. Induced role hierarchies with attribute-based rbac. In *SACMAT*, pages 142–148, 2003.
5. J.P. Anderson. Computer security technology planning study. Technical Report 73-51, Computer Security technology planning study, 1972.
6. Gergei Bana, Koji Hasebe, and Mitsuhiro Okada. Computational semantics for first-order logical analysis of cryptographic protocols. In *Formal to Practical Security*, pages 33–56, 2009.
7. John Bethencourt, Amit Sahai, and Brent Waters. Ciphertext-policy attribute-based encryption. In *2007 IEEE Symposium on Security and Privacy*, pages 321–334. IEEE Computer Society Press, May 2007.
8. Florian Böhl, Dennis Hofheinz, and Daniel Kraschewski. On definitions of selective opening security. In *Public Key Cryptography*, volume 7293 of *Lecture Notes in Computer Science*, pages 522–539, 2012.
9. Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd Annual Symposium on Foundations of Computer Science*, pages 136–145. IEEE Computer Society Press, October 2001.
10. Véronique Cortier, Steve Kremer, and Bogdan Warinschi. A survey of symbolic methods in computational analysis of cryptographic systems. *J. Autom. Reasoning*, 46(3-4):225–259, 2011.
11. Anupam Datta, Ante Derek, John C. Mitchell, and Arnab Roy. Protocol composition logic (pcl). *Electr. Notes Theor. Comput. Sci.*, 172:311–358, 2007.
12. Anupam Datta, Ante Derek, John C. Mitchell, Vitaly Shmatikov, and Mathieu Turuani. Probabilistic polynomial-time semantics for a protocol security logic. In *ICALP*, pages 16–29, 2005.
13. Cynthia Dwork, Moni Naor, Omer Reingold, and Larry J. Stockmeyer. Magic functions. *J. ACM*, 50(6):852–921, 2003.
14. David Ferraiolo and Richard Kuhn. Role-based access control. In *15th NIST-NCSC National Computer Security Conference*, pages 554–563, 1992.
15. Amos Fiat and Moni Naor. Broadcast encryption. In Douglas R. Stinson, editor, *Advances in Cryptology – CRYPTO'93*, volume 773 of *Lecture Notes in Computer Science*, pages 480–491. Springer, August 1994.
16. Kevin Fu, Seny Kamara, and Yoshi Kohno. Key regression: Enabling efficient key distribution for secure distributed storage. In *NDSS*, 2006.
17. David K. Gifford. Cryptographic sealing for information secrecy and authentication. *Commununications of the ACM*, 25(4):274–286, 1982.
18. Vipul Goyal, Omkant Pandey, Amit Sahai, and Brent Waters. Attribute-based encryption for fine-grained access control of encrypted data. In Ari Juels, Rebecca N. Wright, and Sabrina De Capitani di Vimercati, editors, *ACM CCS 06: 13th Conference on Computer and Communications Security*, pages 89–98. ACM Press, October / November 2006. Available as Cryptology ePrint Archive Report 2006/309.
19. Shai Halevi, Paul A. Karger, and Dalit Naor. Enforcing confinement in distributed storage and a cryptographic model for access control. *IACR Cryptology ePrint Archive*, 2005:169, 2005.
20. Anthony Harrington and Christian Jensen. Cryptographic access control in a distributed file system. In *Proceedings of the eighth ACM symposium on Access control models and technologies*, SACMAT '03, pages 158–165, New York, NY, USA, 2003. ACM.
21. Jonathan Katz, Amit Sahai, and Brent Waters. Predicate encryption supporting disjunctions, polynomial equations, and inner products. In Nigel P. Smart, editor, *Advances in Cryptology – EUROCRYPT 2008*, volume 4965 of *Lecture Notes in Computer Science*, pages 146–162. Springer, April 2008.
22. Ninghui Li and Mahesh V. Tripunitara. Security analysis in role-based access control. In *SACMAT*, pages 126–135, 2004.

23. Gerome Miklau and Dan Suciu. Controlling access to published data using cryptography. In *VLDB 2003: 29th International Conference on Very Large Data Bases*, pages 898–909, 2003.
24. Amit Sahai and Brent R. Waters. Fuzzy identity-based encryption. In Ronald Cramer, editor, *Advances in Cryptology – EUROCRYPT 2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 457–473. Springer, May 2005.
25. Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.
26. Victor Shoup. Using hash functions as a hedge against chosen ciphertext attack. In Bart Preneel, editor, *Advances in Cryptology – EUROCRYPT 2000*, volume 1807 of *Lecture Notes in Computer Science*, pages 275–288. Springer, May 2000.
27. Lingyu Wang, Duminda Wijesekera, and Sushil Jajodia. A logic-based framework for attribute based access control. In *FMSE*, pages 45–55, 2004.