UNIVERSITÀ
DI TRENTO

Department of Information Engineering and Computer Science

Bachelor's Degree in
Computer Science

Final Dissertation

Cryptographic Access Control for Balancing Trust, Protection, and Performance

Supervisor
Prof. Silvio Ranise

Student
Simone Brunello

Co-supervisors
Mr. Stefano Berlato
Dr. Roberto Carbone

Academic year 2022/2023

Contents

Abstract

Numerous companies are currently migrating (or have already migrated) their technological infrastructure along with their data and their customers' data to the cloud for, e.g., enhanced scalability, greater flexibility, and monetary cost savings. However, it is well-known that outsourcing the storage of (possibly sensitive) data to the cloud poses significant challenges to the confidentiality and integrity of the data themselves. Indeed, cloud-hosted data are exposed to a wide array of threats including external attackers, malicious insiders, and honest-but-curious cloud service providers (CSPs). In this context, cryptographic access control (CAC) — which consists in enforcing access control (AC) policies through cryptography — is the natural solution to regulate data sharing among authorized users (e.g., employees) while securing data and preventing unauthorized access. Nonetheless, the use of CAC entails the execution of several cryptographic computations (e.g., encryption and decryption computations), especially when considering dynamic AC policies requiring frequent distribution and revocation of users' privileges. Consequently, CAC is typically computationally demanding and may even result to be impractical in some real-world scenarios. In this thesis, we propose an extended role-based access control (RBAC) scheme to mitigate the computational overhead of CAC. Our extended RBAC scheme allows for expressing high-level RBAC policies which are automatically compiled into two sub-policies enforced in a hybrid fashion by a (computationally-light) centralized traditional RBAC enforcement mechanism and a (computationally-demanding) CAC scheme. The automatic compilation is controlled by a customizable security model defining the levels of trust assumed on, e.g., users and CSPs, an approach which allows to determine whether the execution of certain cryptographic computations is necessary or superfluous on a case-by-case basis. Moreover, we apply our extended RBAC scheme to a concrete CAC scheme and provide a proof-of-concept implementation in the Prolog language. Finally, we provide a practical application of our proof-of-concept implementation on a concrete scenario and analyze the cryptographic computational costs incurred.

1 Introduction

Context. In recent years, cloud services have become increasingly popular, with many companies moving their technological infrastructure to the cloud. According to Eurostat [5], in 2023, 45.2% of European companies were using cloud services, an increase of 4.2% compared to 2021. Indeed, major cloud service providers (CSPs) like Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform (GCP) now offer a wide range of mature and rich-feature cloud services, with data storage being one of the most commonly entrusted services to the cloud [5]. Cloud infrastructures operate on a responsibility-sharing model, where the CSP and the company collaborate to ensure the security of cloud services. Although specific responsibilities may vary depending on the chosen paradigm (e.g., IaaS, PaaS, and SaaS[1]), in general the CSP is responsible for the protection *of* the cloud, while the company is responsible for the protection *in* the cloud (i.e., the protection of the applications executed and the data stored in the cloud services). In this context, data breaches become more relevant as the company has no control over the protection of the part of the infrastructure that is the responsibility of the CSP. In 2022, the Identity Theft Centre reported an alarming 1,774 data breaches, which exposed nearly 400 million user records [9]. A data breach can have severe consequences for a company, including financial problems due to compensation and fines, as well as reputational damage [1, 11, 12]. In Europe, fines may reach up to 4% of the company's global turnover [4]. In addition, CSPs are generally considered to be honest-but-curious [7], meaning that they are assumed to behave correctly

[1] `https://www.redhat.com/en/topics/cloud-computing/iaas-vs-paas-vs-saas`

when performing the operations entrusted to them (e.g., preserve the integrity of the stored data), but may use the data for other purposes (e.g., advertising, market analysis, AI training).

For all of the above reasons, protecting data stored in the cloud requires addressing many threats (e.g., external attackers, malicious insiders, honest-but-curious CSPs) to prevent unauthorized access and ensure confidentiality and integrity. CSPs often offer a range of data protection and audit mechanisms, such as identity and access management, logs, encrypted disks, and key management. Encryption of data by the CSP can improve security by mitigating the effects of data breaches or internal threats (e.g., a disloyal employee of the CSP). However, it is important to note that this is not a definitive solution, as trust in the CSPs is being shifted from data custody to key custody. In other words, there is the need for a solution that guarantees the protection of data in an honest-but-curious (also called partially trusted) CSP.

The literature has responded to this need with a wide range of proposals, such as cryptography, steganography, and hardware-based confidential computing. One of these proposals is cryptographic access control (CAC) — an approach to access control (AC) based on full data encryption — which is diametrically opposed to full trust in CSP. CAC require CSPs to be trusted only for the integrity of resources (usually files), which can be guaranteed by an honest-but-curious CSP. CAC is commonly used to protect the confidentiality of sensitive data hosted in the cloud from both external attackers and CSP itself while enforcing AC policies. In CAC, data is encrypted, and access permissions (such as read access) to the encrypted data are represented by the secret decryption cryptographic keys, which are distributed by a trusted administrator — e.g., the information technology (IT) administrator of the company — to authorized employees only.

Problem. Intuitively, CAC has both advantages and disadvantages. On the one hand, CAC guarantees confidentiality and data integrity even when the CSP is honest-but-curious; on the other hand, CAC requires cryptographic computations that entail a — often non-negligible — computational overhead that limits the applicability of CAC in real-world scenarios. For instance, when an employee is leaving the company, it is essential to regenerate multiple keys and re-encrypt all resources the employee had access to. Otherwise, an employee who previously cached the secret decryption keys could still access data by downloading a resource and decrypting it, possibly colluding with the partially trusted CSP. While necessary, these cryptographic computations may render CAC impractical for at least the most dynamic real-world scenarios [7].

Then, we note that CAC can be used as enforcement mechanism for one of the most widely used AC models, i.e., role-based access control (RBAC) [3], which includes users, roles, and resources. In brief, a user represents a person, device, or software, while a role represents a job function. Users are assigned to one or more roles, and roles are associated with permissions on resources, allowing for specific operations to be performed. However, assuming a CAC scheme exists that can dynamically manage security levels based on resource sensitivity, user trust, or other security requirements, RBAC alone cannot manage such security levels. In fact, RBAC does not provide suitable abstractions for specifying additional information and constraints that may be useful in reducing the computational overhead of CAC. For example, if an employee who is trusted by the administrator is leaving the company, it may not be necessary to regenerate keys and re-encrypt resources. In addition, in CAC it is not possible to store public resources, i.e., resources that can be accessed by everyone. Indeed, CAC expects all resources to be encrypted — without modification, a completely different process would be necessary to store public files.

In general, all of the previous problems stem from the fact that CAC typically assumes that the CSP is not trusted to see the contents of any resource — the CSP is only considered trustworthy for the availability and the integrity of resources. Also, CAC usually assumes that any user would try to access any resource at any time, and that all resources are sensitive. In practice, however, the CSPs may be trusted to protect some resources, users may have different levels of trustworthiness, and only some resources may be sensitive. For example, if a user loses access to a resource because they have been fired, they may be perceived as less trustworthy. Conversely, if they have been promoted, they may be considered more trustworthy. To the best of our knowledge, no prior solution to those problems has been proposed in the literature.

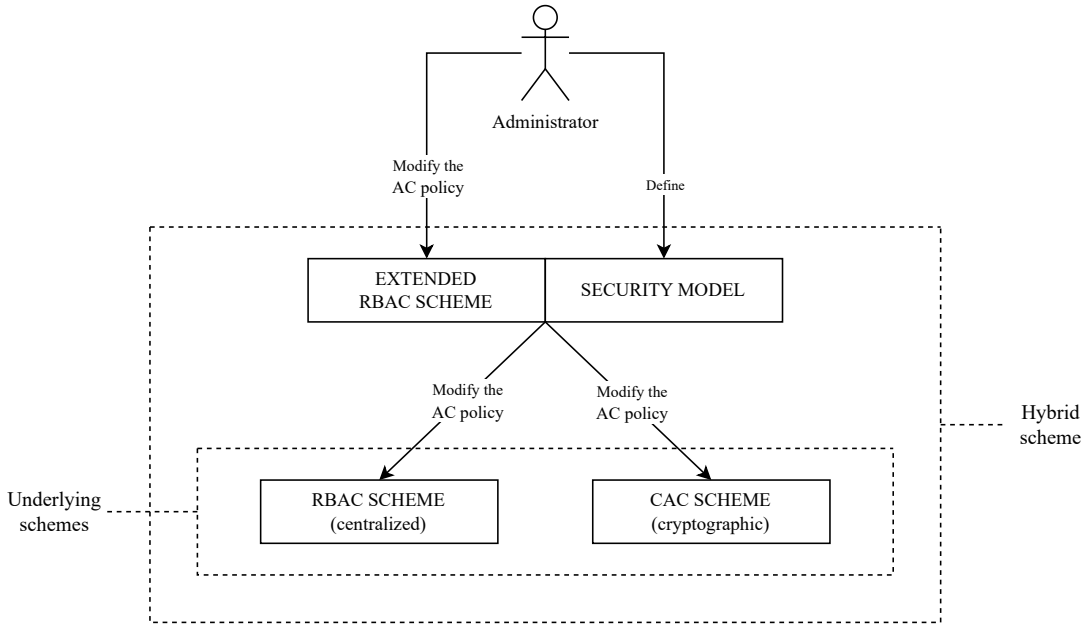


Figure 1.1: The Hybrid System

Solution. To address the issues mentioned earlier, we propose a new AC scheme that extends RBAC and allows the company administrator to specify high-level policies that — according to a specific security model (e.g., defining the levels of trust assumed on users and CSPs) — are automatically compiled and enforced by a hybrid scheme consisting of a centralized RBAC enforcement mechanism and CAC (see Figure 1.1). By defining appropriate formalisms, our extended RBAC scheme can reduce the computational overhead of cryptographic computations executed in CAC and enhance its expressiveness. This is achieved by allowing the specification of predicates (i.e., facts about users, roles, resources, and the CSP) that control the tunable aspects of CAC; for instance, our extended RBAC scheme allows to specify the characteristics of entities (e.g., whether a resource should be stored encrypted or unencrypted, whether the CSP is trusted to protect a resource), the risks of collusion between users and the CSPs, and the steps to be taken when access is revoked (e.g., whether it is necessary to regenerate keys and re-encrypt resources after revoking access privileges to a user).

Contributions. In this thesis, we make a number of contributions which we described below:

- we define an extended RBAC scheme that allows for specifying predicates defining under what conditions certain cryptographic computations of CAC should be executed;
- we define suitable formalisms allowing to automatically compile the AC policy specified in the extended RBAC scheme into two lower-level AC policies to be enforced by a centralized RBAC enforcement mechanism and CAC, respectively. Moreover, we analyze the computational costs of the cryptographic operations of the resulting CAC scheme;
- we implement our extended RBAC scheme and the formalisms in Prolog to provide a proof-of-concept of our solution.

Structure. In Chapter 2, we explain fundamental concepts of AC (Section 2.1) and CAC (Section 2.2). In Chapter 3, we provide an overview of the proposed hybrid scheme (Section 3.1), examine the extended RBAC scheme (Section 3.2), delve into the CAC scheme (Section 3.3), and explore cryptographic costs (Section 3.4). Chapter 4 presents an implementation of our model, including a list of predicates (Section 4.1), an implementation of some components of the hybrid scheme in Prolog(Section 4.3), and a concrete demonstration and cryptographic computation analysis of our example (Section 4.2).

Table 1.1: Symbols

Category	Symbol	Description
AC	Γ	the set of states
	γ	a state, $\gamma \in \Gamma$
	Q	the set of queries
	$\vdash$	the entailment function
	Ψ	the state-change rules
	ψ	a state-change rule, $\psi \in \Psi$
RBAC	U	the set of users
	u	a user, $u \in U$
	R	the set of roles
	r	a role, $r \in R$
	F	the set of resources (files)
	f	a resource (file), $f \in F$
	UR	the set of user-role assignments
	OP	the set of operations
	op	an operation, $op \in OP$
	$read$	read operation, $read \in OP$
	$write$	write operation, $write \in OP$
	RW	both read and write operations
	PR	the set of all possible permissions
	FR	is the permission-role assignement
Extended RBAC	P	the set of predicates
	PA	the set of predicate-entity assignement
CAC	$\mathtt{Gen}^{\mathtt{Pub}}$	generate asymmetric key-pair
	$\mathtt{Gen}^{\mathtt{Sig}}$	generate signing key-pair
	$\mathtt{Gen}^{\mathtt{Sym}}$	generate symmetric key
	$\mathtt{k}_a^{\mathtt{enc}}$	encryption key of entity a
	$\mathtt{k}_a^{\mathtt{dec}}$	decryption key of entity a
	$\mathtt{k}_a^{\mathtt{sig}}$	signing key of entity a
	$\mathtt{k}_a^{\mathtt{ver}}$	verification key of entity a
	$\mathtt{k}_f$	encryption key of file f
	$\mathtt{Enc}^{\mathtt{Pub}}_{\mathtt{k}_a^{\mathtt{enc}}}(...)$	encrypt with key $\mathtt{k}_a^{\mathtt{enc}}$ (asymmetric)
	$\mathtt{Dec}^{\mathtt{Pub}}_{\mathtt{k}_a^{\mathtt{enc}}}(...)$	decrypt with key $\mathtt{k}_a^{\mathtt{enc}}$ (asymmetric)
	$\mathtt{Enc}^{\mathtt{Sym}}_{\mathtt{k}_a^{\mathtt{enc}}}(...)$	encrypt with key $\mathtt{k}_a^{\mathtt{enc}}$ (symmetric)
	$\mathtt{Dec}^{\mathtt{Sym}}_{\mathtt{k}_a^{\mathtt{enc}}}(...)$	decrypt with key $\mathtt{k}_a^{\mathtt{enc}}$ (symmetric)
	$\mathtt{Sign}_a^{\mathtt{Sig}}$	signature made by a
	$\mathtt{Ver}^{\mathtt{Sig}}_{\mathtt{k}_a}(...) = 1$	verification with the key of a
Costs	$\mathtt{Gen}^{\mathtt{Sig}}$	generation of a signing key-pair
	$\mathtt{Gen}^{\mathtt{Pub}}$	generation of an asymmetric key-pair
	$\mathtt{Gen}^{\mathtt{Sym}}$	generation of a symmetric key
	$\mathtt{Enc}^{\mathtt{Sym}}$	encryption with a symmetric algorithm
	$\mathtt{Enc}^{\mathtt{Pub}}$	encryption with an asymmetric algorithm
	$\mathtt{Dec}^{\mathtt{Sym}}$	decryption with a symmetric algorithm
	$\mathtt{Dec}^{\mathtt{Pub}}$	decryption with an asymmetric algorithm
	$\mathtt{Sign}^{\mathtt{Sig}}$	signing data
	$\mathtt{Ver}^{\mathtt{Sig}}$	verification signed data

2 Background

This chapter explains the concepts of AC and CAC. Section 2.1 provides a definition of AC and explains the RBAC model in details. Section 2.2 introduces the concept of CAC and describes how CAC may be used as an enforcement mechanism for RBAC by presenting the operations of the CAC scheme in [7], which is one of the bases of our work.

2.1 Access Control

Samarati and de Capitani di Vimercati [13] describe AC as "the process of mediating every request to resources maintained by a system and determining whether the request should be granted or denied". Conceptually, AC can be divided into three levels: policy, model, and enforcement. A policy is a high-level set of rules that specifies when access should be allowed or denied, a model is a mathematical representation that defines how policies are encoded, and the enforcement is the practical implementation of a model and enforces policies (e.g. a hardware device that mediates access to a server). As described in [10], an AC model can also be called AC scheme and be seen as a state transition system $\langle \Gamma, Q, \vdash, \Psi \rangle$ where Γ is the set of states, Q is the set of queries, $\vdash: \Gamma \times Q \to \{true, false\}$ is the entailment relation that determines whether a query is true or not in a given state, and Ψ is the set of state-change rules. A state-change rule has the effect of modifying the state, i.e., $\gamma \mapsto_\psi \gamma'$ — that is, the state-change rule $\psi \in \Psi$ transforms the state $\gamma \in \Gamma$ into the state $\gamma' \in \Gamma$.

There are many AC models proposed in the literature, supporting different characteristics and complexities; attribute-based access control (ABAC) [8] and RBAC [14] are two of the most commonly used AC models [3]. Compared to ABAC, RBAC is particularly popular in companies due to its simpler implementation and role assignment process, e.g, in RBAC, roles can be intuitively mapped to job qualifications (e.g., employee, manager) within the company. Additionally, when a user is employed, they are assigned to the appropriate AC role that corresponds to their assigned tasks [3]. RBAC abstracts real entities by defining the concepts of user, role, and resource. Furthermore, RBAC allows for defining operations on resources. A permission is the association of a resource and an operation to a role. Users are assigned roles and roles are associated with permissions — where a permission describes an operation (e.g., *read*, *write*) on a resource (e.g., file). Formally, a state $\gamma \in \Gamma$ in RBAC can be described as a tuple $\langle U, R, F, UR, FR \rangle$ where:

- U is the set of users;
- R is the set of roles;
- F is the set of resources;
- $UR \subseteq U \times R$ the set of user-role assignments;
- $FR \subseteq R \times PR$ the set of role-permission assignments;
- OP is the set of all operations, where operations for RBAC are defined "on the type of system in which it is implemented" [6]. In this case, since we are dealing with files, we choose $OP = \{write, read\}$. This choice is consistent with the CAC scheme proposed in [7] and described in Section 2.2.
- $PR = OP \times F$ is the set of all possible permissions;

We report in Table 2.1 the set of state-change rules Ψ for RBAC — according to [6] — while the entailment function $\vdash$ defined for Q consists only of the query $(\textbf{\textit{canDo}}, u, \langle op, f \rangle)$ that indicates

whether user u has access to resource f with permission op. The entailment function $\vdash$ is defined as follow:

$$\vdash (\langle U, R, F, UR, FR\rangle, (\textbf{\textit{canDo}}, u, \langle op, f\rangle)) = \exists r \in R : (u, r) \in UR \wedge (r, \langle op, f\rangle) \in FR$$

2.2 Cryptographic Access Control

In cloud-based applications, CAC can be used to ensure the confidentiality and integrity of data when the CSP is honest-but-curious. With CAC, all files are encrypted and freely distributed by the CSP, with the encryption ensuring that only authorized users — which possess the secret decrypting keys — can access the files. CAC is an AC mechanism that concretely solves the key distribution problem by implementing an AC model using cryptography. Garrison et al. proposed a scheme for applying CAC to RBAC [7]. Their solution is described below. We assume that an authenticated and encrypted channel exists between each user and the administrator (e.g. using Transport Layer Security (TLS)). Each user u and role r has two asymmetric key pairs $(\mathtt{k}^{\mathtt{enc}}, \mathtt{k}^{\mathtt{dec}})$ and $(\mathtt{k}^{\mathtt{ver}}, \mathtt{k}^{\mathtt{sig}})$ for en/decryption and for creating/verifying digital signatures, respectively. $\mathtt{k}^{\mathtt{dec}}$ and $\mathtt{k}^{\mathtt{sig}}$ are the private keys and $\mathtt{k}^{\mathtt{enc}}$ and $\mathtt{k}^{\mathtt{ver}}$ are the public keys. The encryption/decryption key of the role is used to decrypt the key of the files. Each file f is encrypted with a symmetric key $\mathtt{k}_{f.name}$, obtaining an encrypted file as $\mathtt{Enc}^{\mathtt{Sym}}_{\mathtt{k}_{f.name}}(f.content)$. To assign a user u to a role r, the administrator of the policy encrypts the decryption and signing keys $(\mathtt{k}^{\mathtt{dec}}_r, \mathtt{k}^{\mathtt{sig}}_r)$ of r with the public key of u, i.e., $\mathtt{k}^{\mathtt{enc}}_u$, resulting in $\mathtt{Enc}^{\mathtt{Pub}}_{\mathtt{k}^{\mathtt{enc}}_u}(\mathtt{k}^{\mathtt{dec}}_r, \mathtt{k}^{\mathtt{sig}}_r)$. To grant permission for a role r to access a file f, the administrator of the policy encrypts the file's key $\mathtt{k}_{f.name}$ with the public key of the role resulting in $\mathtt{Enc}^{\mathtt{Pub}}_{\mathtt{k}^{\mathtt{enc}}_r}(\mathtt{k}_{f.name})$.

To decrypt a file, the procedure is:

- u decrypts $\mathtt{Dec}^{\mathtt{Pub}}_{\mathtt{k}^{\mathtt{dec}}_u}(\mathtt{Enc}^{\mathtt{Pub}}_{\mathtt{k}^{\mathtt{enc}}_u}(\mathtt{k}^{\mathtt{dec}}_r, \mathtt{k}^{\mathtt{sig}}_r))$, obtaining $(\mathtt{k}^{\mathtt{dec}}_r, \mathtt{k}^{\mathtt{sig}}_r)$;
- u decrypts $\mathtt{Dec}^{\mathtt{Pub}}_{\mathtt{k}^{\mathtt{dec}}_r}(\mathtt{Enc}^{\mathtt{Pub}}_{\mathtt{k}^{\mathtt{enc}}_r}(\mathtt{k}_{f.name}))$, obtaining $\mathtt{k}_{f.name}$;
- u decypts $\mathtt{Dec}^{\mathtt{Sym}}_{\mathtt{k}_{f.name}}(\mathtt{Enc}^{\mathtt{Sym}}_{\mathtt{k}_{f.name}}(f.content))$, obtaining $f.content$.

The CAC scheme in [7] requires the storage of additional data, such as keys for en/decryption and for creating/verifying digital signatures, AC policy status, and version numbers. These data are called metadata and are digitally signed. The metadata are organized into tuples, each of which is signed by the author. The provided tuples include: `RK`, which contains the $\mathtt{k}^{\mathtt{dec}}_r$ and $\mathtt{k}^{\mathtt{sig}}_r$ keys of a role signed with a user's key $\mathtt{k}^{\mathtt{enc}}_u$, `FK`, which contains the symmetric key $\mathtt{k}_f$ of a file f encrypted with a role's public key $\mathtt{k}^{\mathtt{enc}}_r$, and `F`, which represents the encrypted files. The CSP stores the encrypted data and metadata, while each user keeps their own key on their own secure device (e.g. laptop, smartphone, hardware security module (HSM)), assuming that the device is secure and that an attacker does not obtain the user's key. Finally, CAC is conceptually composed of these entities [2]:

Table 2.1: The Ψ Of RBAC

State-change rules	**New state**
`addUser`(u)	$\langle U \cup \{u\}, R, F, UR, FR\rangle$
`deleteUser`(u)	$\langle U \setminus \{u\}, R, F, UR, FR\rangle$
`addRole`(r)	$\langle U, R \cup \{r\}, F, UR, FR\rangle$
`deleteRole`(r)	$\langle U, R \setminus \{r\}, F, UR, FR\rangle$
`addResource`(f)	$\langle U, R, F \cup \{f\}, UR, FR\rangle$
`deleteResource`(f)	$\langle U, R, F \setminus \{f\}, UR, FR\rangle$
`assignUserToRole`(u, r)	$\langle U, R, F, UR \cup \{(u, r)\}, FR\rangle$
`revokeUserFromRole`(u, r)	$\langle U, R, F, UR \setminus \{(u, r), FR\}\rangle$
`assignPermissionToRole`$(r, \langle op, f\rangle)$	$\langle U, R, F, UR, FR \cup \{(r, \langle op, f\rangle)\}\rangle$
`revokePermissionFromRole`$(r, \langle op, f\rangle)$	$\langle U, R, F, UR, FR \setminus \{(r, \langle op, f\rangle)\}\rangle$

- **proxy**: this is software installed on the user's device that performs cryptographic computations (such as encryption, decryption, signing, verification, tuple generation, and verification) on behalf of the user. It also manages the keys;
- **metadata manager**: it stores the metadata (i.e., the tuples), the set of the metadata is the current AC state;
- **data manager**: it stores the encrypted data;
- **reference monitor**: it receives the encrypted data and metadata from the proxy, verifies that the metadata is well-formed, signed and valid for the current AC state. If all is well, it stores the metadata in the metadata manager and the encrypted data in the data manager.

3 The Proposed Hybrid Scheme

This chapter describes the design and the components of our hybrid scheme. In Section 3.1 we provide an overview of our hybrid scheme, in Section 3.2 we describe the extended RBAC scheme, and in Section 3.3 we describe the CAC scheme modified according to our extended RBAC scheme.

3.1 High-Level Structure of the Hybrid scheme

Security models typically assume that the CSP is either completely trusted or honest-but-curious. However, in practice, the actual situation is often somewhere in between, which can allow for more efficient operations and a reduced amount of cryptographic computations. To the best of our knowledge, there is no AC scheme in the literature allowing administrators to specify the security model for their scenario. To this end, we provide an extended RBAC scheme. At a high level, the extended RBAC scheme is automatically compiled into a centralized AC scheme and a CAC scheme. The centralized AC scheme can be enforced by a traditional RBAC enforcement mechanisms (e.g., those provided directly by CSPs). The CAC scheme, as proposed in [7], is not able to handle a dynamic security model (e.g., it is not always necessary to re-encrypt a file), so it is necessary to make the changes described in Section 2.2. The administrator operates on the extended RBAC model by invoking its state-change rules, which in turn invoke the state-change rules of the centralized AC and CAC, so that enforcement is performed only by these two mechanisms. To define the security model, we define the concept of predicate, introduce new queries, and modify the entailment as described below.

Predicates. Predicates are facts, characteristics, or requirements related to users, files, or the CSP. Predicates are dependent on company processes. For example, if files have different levels of sensitivity, the administrator can introduce a predicate for each level. This results in $\texttt{topSecret}(f)$ for top secret files, $\texttt{secret}(f)$ for secret files, and $\texttt{public}(f)$ for public files. Another example is a company where employees have a smart card that contains the keys $\mathtt{k}_u^{\mathtt{sig}}$ and $\mathtt{k}_u^{\mathtt{dec}}$ for CAC, the administrator might want to model the case where the card is lost and all files need to be re-encrypted. To achieve this, a $\texttt{lostKeyCard}(u)$ predicate could be introduced. Our proposal only supports unary predicates, which are predicates applied to a single entity. Although it is possible to introduce predicates on more than one entity, we chose to maintain greater simplicity by only supporting unary predicates.

Queries. There are several aspects of CAC that can be customized, and to control each one, new queries were introduced in the extended RBAC scheme. These queries serve as the interface between the extended RBAC scheme and the administrator's chosen security model. For instance, the $(\textbf{\textit{isEncryptionNeeded}}, f)$ query determines whether a file should be encrypted or not. Section 3.2 provides a detailed discussion of queries and their meanings.

Entailment Instantiation. The instantiation of the entailment function should result in a *true* or *false* answer when called with one of the new queries. The answer may be dependent on the predicate assignment. For instance, if a `confidential` predicate exists, the entailment function is instantiated so that when a file f has the `confidential`(f) predicate, the entailment function called with ($\boldsymbol{isEncryptionNeeded}, f$) returns *true*; *false* otherwise.

Figure 3.1 illustrates the operations performed in the hybrid scheme. Specifically, Figure 3.1a describes the process an administrator follows when invoking a state-change rule. The proxy receives the invocation and retrieves metadata, including predicates, from the metadata manager. As a result of the execution of the state-change rule — and according to the retreived metadata — the proxy updates the centralized AC policy and metadata in the metadata manager to reflect the changes in the policy. Finally, it encrypts or decrypts any files for which the protection enforcement mechanism has been altered. Figure 3.1b illustrates a user reading a file. The user sends a request to the proxy, which retrieves metadata from the metadata manager. The proxy then requests the file from the data manager, which in turn consults the centralized RBAC mechanism to determine if it can send the file to the user. If authorized, the proxy receives the file and delivers it to the user, decrypting it if necessary. Finally, in Figure 3.1c, a user writes a file. The plaintext file is sent to the proxy, which retrieves the metadata and determines whether to encrypt the file. If encryption is necessary, the proxy encrypts the file and sends it, along with the new metadata, to the reference monitor. Firstly, the reference monitor retrieves metadata from the metadata manager. Secondly, the reference monitor verifies that the data received from the user is well-formed, checks if the user has permission to write, and ensures that the metadata is consistent. If everything is in order, the metadata is written and the new data is sent to the data manager. The data manager then verifies with the centralized RBAC mechanism whether the user is authorized to write the file. If authorized, the file is written.

3.2 The Extended Role-Based Access Control Scheme

To support security model handling in the extended RBAC model, we introduce a set of predicates, assign predicates to entities, add new queries, and redefine the entailment function.

Predicates are defined by the administrator based on the needs of the company. To enable this, a new set of predicates P was introduced, with each predicate being represented as a string. As a result, the state transition system is redefined by adding the set P, resulting in the system being represented as $\langle \Gamma, Q, \vdash, \Psi, P \rangle$. Predicates are applied to an entity (i.e., user, role, the CSP), and they are unary. A new set, PA, was introduced to store the assignment. PA is defined as $PA \subseteq P \times (U \cup R \cup F)$. Consequently, the state was extended to also contain the set PA, resulting in $\langle U, R, F, UR, FR, PA \rangle$. New queries have been added to the set of queries Q, on top of the already existing ($\boldsymbol{canDo}, u, \langle op, f \rangle$) query. Each query adjusts a specific tunable aspect of the system, such as whether to encrypt a file or not, and whether to re-encrypt a file. Table 3.1 describes the queries, which are not customizable by the administrator. The entailment function $\vdash$ serves as the connection between company phenomenons (the predicates) and customizable features (the queries). It is necessary for the administrator to define the entailment function for new queries based on the chosen security model.

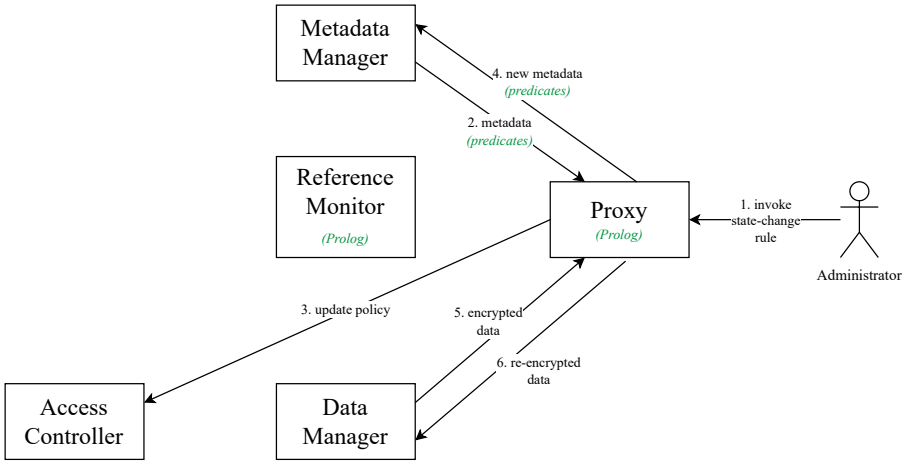
Note that the administrator cannot modify the entailment function for the query ($\boldsymbol{canDo}, u, \langle op, f \rangle$), as doing so would prevent the use of a centralized RBAC and CAC mechanism. Additionally, modifying the ($\boldsymbol{canDo}, u, \langle op, f \rangle$) query would result in a model that is no longer RBAC.

Figure 3.2 illustrates the predicates, queries, and entailment function, and distinguishes between the extended RBAC scheme and the security model.

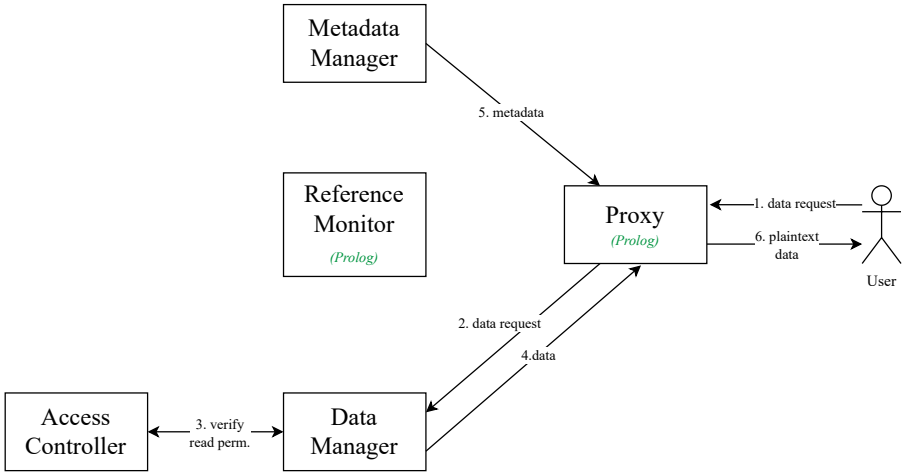
Section 4.1 contains an example of a realistic security model instantiation.

3.3 The Modified Cryptographic Access Control Scheme

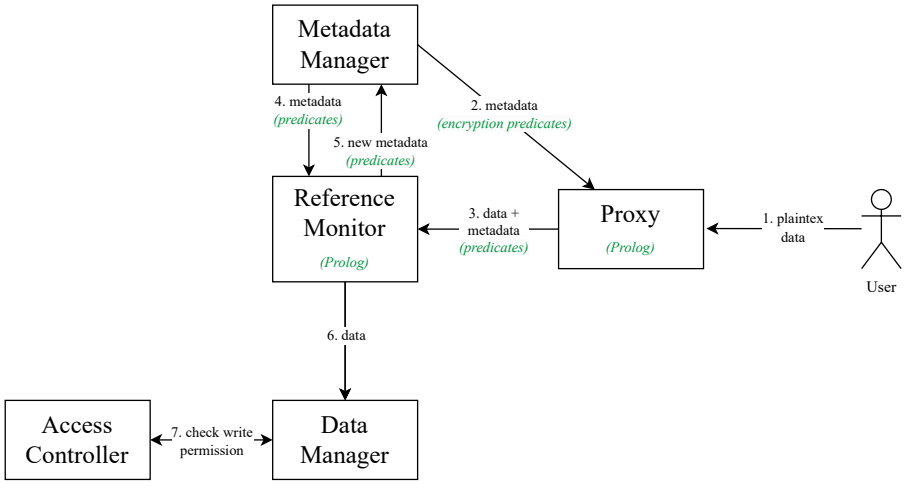
The scheme proposed in [7] assumes a specific security model. For example, it assumes that the CSP is honest-but-curious and therefore not trusted to protect any files. It also assumes that users could always collude with the CSP. Based on the previous security model, the scheme determines which operation to perform, resulting in many computationally expensive operations, such as always re-encrypting all files. To compile the security model chosen by the administrator in CAC, modifications are required.



(a) State-change rule invocation



(b) File reading



(c) File writing

Figure 3.1: Operations In Hybrid Scheme

The changes can be categorized into two groups: isolating specific instructions, such as separating the revocation part of a permission from the re-encryption of files that a user previously had access to, and implementing functionality that was not originally provided, such as the ability to perform eager re-encryption.

The purpose of the changes is to ensure the following characteristics:

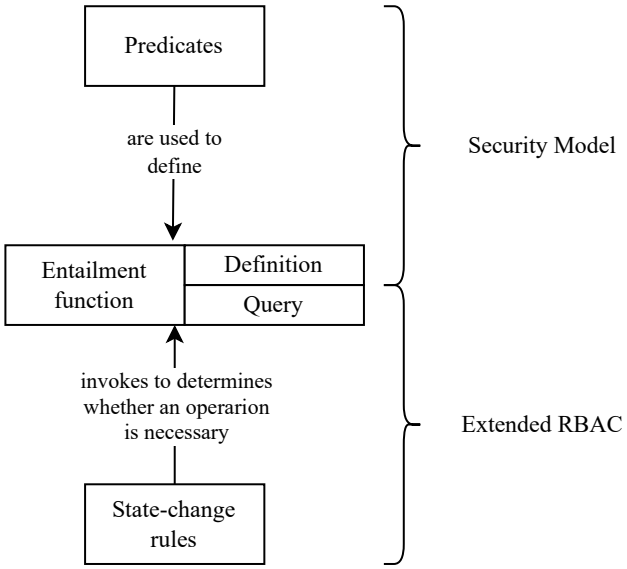


Figure 3.2: The Interaction Between Components

- **role key rotation**: the original scheme assumes that a user can save the role key and continue to use it even if access is revoked, so the keys are always changed when a revocation occurs. Not all security models provide for this, so it's necessary to make this operation optional;

- **file re-encryption**: whenever a permission is revoked or a user is revoked from a role, the file is re-encrypted. This operation may not be necessary (e.g. because a user has been promoted to another role and is considered trusted). Re-encryption should be separate from revocation and should be optional;

- **eager/lazy re-encryption**: the original scheme only implements lazy re-encryption, meaning that each time a file is re-encrypted, a new key and corresponding tuples are generated. However, the file is not actually re-encrypted until the first write, which improves performance. There is

Table 3.1: The Q of the hybrid model

Query	**Description**
$(\boldsymbol{canDo}, u, \langle op, f \rangle)$	Determines whether user u has access to file f to perform the action op
$(\boldsymbol{isEncryptionNeeded}, f)$	Determines whether it is necessary to encrypt the file f
$(\boldsymbol{isReencryptionNeededOnRP}, r, \langle op, f \rangle)$	Determines whether it is necessary to re-encrypt the file f when the permission $\langle op, f \rangle$ is revoked
$(\boldsymbol{isEagerReencNeededOnRP}, r, \langle op, f \rangle)$	Determines whether it is necessary immediately to re-encrypt the file f when the permission $\langle op, f \rangle$ is revoked
$(\boldsymbol{isRoleKeyRotationNeededOnRUR}, u, r)$	Determines whether it is necessary to rotate the key of role r when the membership of the user u to the role r is revoked
$(\boldsymbol{isReencryptionNeededOnRUR}, u, r, f)$	Determines whether it is necessary to re-encrypt the file f when the membership of the user u to the role r is revoked
$(\boldsymbol{isEagerReencNeededOnRUR}, u, r, f)$	Determines whether it is necessary to re-encrypt immediately the file f when the membership of the user u to the role r is revoked

no mention of eager re-encryption in the original scheme. However, it is intended to allow for eager re-encryption, where a file can be re-encrypted immediately, or lazy re-encryption, where it is re-encrypted at the first write, from time to time.

To achieve the features described above, the state-change rule has been modified as follows. `deleteUserK`(u) now only checks that the user is no longer assigned to a role. `deleteResourceK`(f) now simply checks that no associated permissions exist and deletes the file tuple. `deleteRoleK`(r) now simply checks that no user is assigned to role r and that no file has a permission associated with r. The state-change rules `revokeUserFromRoleK`(u, r) and `revokePermissionFromRoleK`$(r, \langle op, f\rangle)$ have been modified to exclude the part that re-encrypts files. This part has been moved to a separate state-change rule called `prepareReencryptionK`(f), which prepares the file for re-encryption. A state-change rule `reencryptResourceK`(f) has been introduced to immediately re-encrypt a pending file. To achieve lazy re-encryption, it is sufficient to call `prepareReencryptionK`(f). For eager re-encryption, it is also necessary to call `reencryptResourceK`(f) immediately afterward. The part that regenerates the role key r in the state-change rule `revokeUserFromRoleK`(u, r) was removed. This part was moved to a separate state-change rule called `rotateRoleKeyK`(r). The outcome can be found in Figure 3.4.

Figure 3.3: The state-change rules Ψ of the hybrid model

`addUser`(u)
- Update state with $\langle U \cup \{u\}, R, F, UR, FR, PA\rangle$
- `addUserC`(u)
- `addUserK`(u)

`deleteUser`(u)
- Update state with $\langle U \setminus \{u\}, R, F, UR, FR, PA\rangle$
- For each $(u, r) \in UR$:
  - `revokeUserFromRoleC`(u, r)
  - `revokeUserFromRoleK`(u, r)
- `deleteUserC`(u)
- `deleteUserK`(u)

`addRole`(r)
- Update state with $\langle U, R \cup \{r\}, F, UR, FR, PA\rangle$
- `addRoleC`(r)
- `addRoleK`(r)

`deleteRole`(r)
- Update state with $\langle U, R \setminus \{r\}, F, UR, FR, PA\rangle$
- For each $(r, \langle op, f\rangle) \in FR$:
  - `revokePermissionFromRole`$(r, \langle op, f\rangle)$
- `deleteRoleC`(r)
- `deleteRoleK`(r)

`addResource`(f)
- Update state with $\langle U, R, F \cup \{f\}, UR, FR, PA\rangle$
- `addResourceC`(f)

`deleteResource`(f)
- Update state with $\langle U, R, F \setminus \{f\}, UR, FR, PA\rangle$
- `deleteResourceC`(f)
- For each $(r, \langle op, f\rangle) \in FR$:
  - `revokePermissionFromRole`$(r, \langle op, f\rangle)$
- If ***isEncryptionNeeded***(f):
  - `deleteResourceK`(f)

`assignUserToRole`(u, r)
- Update state with $\langle U, R, F, UR \cup \{(u, r)\}, FR, PA\rangle$
- `assignUserToRoleC`(u, r)
- `assignUserToRoleK`(u, r)

`assignPermissionToRole`$(r, \langle op, f\rangle)$
- Update state with $\langle U, R, F, UR, FR \cup (r, \langle op, f\rangle), PA\rangle$
- `assignPermissionToRoleC`$(r, \langle op, f\rangle)$
- If ***isEncryptionNeeded***(f):
  - `assignPermissionToRoleK`$(r, \langle op, f\rangle)$

`revokeUserFromRole`(u, r)
- `revokeUserFromRoleC`(u, r)
- For each $(r, \langle op, f\rangle) \in FR$ with ***isEncryptionNeeded***(f)
  - `revokeUserFromRoleK`(u, r)
  - If ***isReencryptionNeededOnRUR***(u, r, f):
    - `prepareReencryptionK`(f)
  - If ***isEagerReencNeededOnRUR***(u, r, f):
    - `reencryptResourceK`(f)
- `revokeUserFromRoleK`(u, r)
- If exists $(r, \langle -, f\rangle) \in FR$ with ***isEncryptionNeeded***(f) and ***isRoleKeyRotationNeededOnRUR***(u, r):
  - `rotateRoleKeyK`(r)
- Update state with $\langle U, R, F, UR \setminus \{(u, r)\}, FR, PA\rangle$

`revokePermissionFromRole`$(r, \langle op, f\rangle)$
- `revokePermissionFromRoleC`$(r, \langle op, f\rangle)$
- If ***isEncryptionNeeded***(f):
  - `revokePermissionFromRoleK`$(r, \langle op, f\rangle)$
  - If ***isReencryptionNeededOnRP***$(r, \langle op, f\rangle)$:
    - `prepareReencryptionK`(f)
  - If ***isEagerReencNeededOnRP***$(r, \langle op, f\rangle)$:
    - `reencryptResourceK`(f)
- Update state with $\langle U, R, F, UR, FR \setminus (r, \langle op, f\rangle), PA\rangle$

`updateEncryptedFiles`$(files_1, files_2)$
- For each $f \in (file_1 \setminus file_2)$:
  - For each $(r, \langle op, f\rangle) \in PA$:
    - `revokePermissionFromRoleK`$(r, \langle op, f\rangle)$
  - `deleteResourceK`(f)
- For each $f \in (file_2 \setminus file_1)$:
  - `addResourceK`(f)
  - For each $(r, \langle op, f\rangle) \in PA$:
    - `assignPermissionToRoleK`$(r, \langle op, f\rangle)$

`assignPredicate`(p, a)
- Set $files_1 \leftarrow \{f \in F | $ ***isEncryptionNeeded***$(f)\}$
- Update state with $\langle U, R, F, UR, FR, PA \cup \{p, a\}\rangle$
- Set $files_2 \leftarrow \{f \in F | $ ***isEncryptionNeeded***$(f)\}$
- `updateEncryptedFiles`$(files_1, files_2)$

`revokePredicate`(p, a)
- Set $files_1 \leftarrow \{f \in F | $ ***isEncryptionNeeded***$(f)\}$
- Update state with $\langle U, R, F, UR, FR, PA \setminus \{p, a\}\rangle$
- Set $files_2 \leftarrow \{f \in F | $ ***isEncryptionNeeded***$(f)\}$
- `updateEncryptedFiles`$(files_1, files_2)$

Figure 3.4: CAC scheme [7] modified

$\texttt{addUserK}(u)$
- User u generate encryption key pair $(\texttt{k}^{\texttt{enc}}_u, \texttt{k}^{\texttt{dec}}_u) \leftarrow \texttt{Gen}^{\texttt{Pub}}$ and signature key pair $(\texttt{k}^{\texttt{ver}}_u, \texttt{k}^{\texttt{sig}}_u) \leftarrow \texttt{Gen}^{\texttt{Sig}}$
- User u sends $\texttt{k}^{\texttt{enc}}_u$, $\texttt{k}^{\texttt{ver}}_u$ to admin
- Admin adds $(u, \texttt{k}^{\texttt{enc}}_u, \texttt{k}^{\texttt{ver}}_u)$ to USERS

$\texttt{deleteUserK}(u)$
- Verifies that there are no tuples $\langle \texttt{RK}, u, -, -, sig\rangle$ with $\texttt{Ver}^{\texttt{Sig}}_{\texttt{k}^{\texttt{ver}}_{SU}}(\langle \texttt{RK}, u, -, -\rangle, sig) = 1$

$\texttt{addResourceK}(u, f)$
- Generate symmetric key $\texttt{k} \leftarrow \texttt{Gen}^{\texttt{Sym}}$
- Send $\langle \texttt{F}, f.name, 1, \texttt{Enc}^{\texttt{Sym}}_k(f.content), u, \texttt{Sign}^{\texttt{Sig}}_u\rangle$ and $\langle \texttt{FK}, SU, \langle f.name, RW\rangle, 1, \texttt{Enc}^{\texttt{Pub}}_{\texttt{k}^{\texttt{enc}}_{SU}}(k), u, \texttt{Sign}^{\texttt{Sig}}_u\rangle$ to R.M.
- The R.M. receives $\langle \texttt{F}, fn, 1, c, u, sig\rangle$ and $\langle \texttt{FK}, SU, \langle fn, RW\rangle, 1, c', u, sig'\rangle$ and verifies that the tuple are well-formed and the signatures are valid, i.e., $\texttt{Ver}^{\texttt{Sig}}_{\texttt{k}^{\texttt{ver}}_u}(\langle \texttt{F}, fn, 1, c, u\rangle, sig) = 1$ and $\texttt{Ver}^{\texttt{Sig}}_{\texttt{k}^{\texttt{ver}}_u}(\langle \texttt{FK}, SU, \langle fn, RW\rangle, 1, c', u\rangle, sig') = 1$
- If verifications is successful, the R.M. adds $(f.name, 1)$ to FILES and stores $\langle \texttt{F}, fn, 1, c, u, sig\rangle$ and $\langle \texttt{FK}, SU, \langle fn, RW\rangle, 1, c', u, sig'\rangle$

$\texttt{deleteResourceK}(f)$
- Verifies that there are no tuples $\langle \texttt{FK}, -, \langle f.name, -\rangle, -, -, -, sig\rangle$ with $\texttt{Ver}^{\texttt{Sig}}_{\texttt{k}^{\texttt{ver}}_{SU}}(\langle \texttt{FK}, -, \langle f.name, -\rangle, -, -, -, -\rangle, sig) = 1$
- Remove $(f.name, v_{f.name})$ from FILES
- Delete $\langle \texttt{F}, f.name, -, -, -, -\rangle$

$\texttt{addRoleK}(r)$
- Generate encryption key pair $(\texttt{k}^{\texttt{enc}}_{(r,1)}, \texttt{k}^{\texttt{dec}}_{(r,1)}) \leftarrow \texttt{Gen}^{\texttt{Pub}}$ and $(\texttt{k}^{\texttt{ver}}_{(r,1)}, \texttt{k}^{\texttt{sig}}_{(r,1)}) \leftarrow \texttt{Gen}^{\texttt{Sig}}$
- Add $(r, 1, \texttt{k}^{\texttt{enc}}_{(r,1)}, \texttt{k}^{\texttt{ver}}_{(r,1)})$ to ROLES
- Send $\langle \texttt{RK}, SU, (r, 1), \texttt{Enc}^{\texttt{Pub}}_{\texttt{k}^{\texttt{enc}}_{SU}}(\texttt{k}^{\texttt{dec}}_{(r,1)}, \texttt{k}^{\texttt{sig}}_{(r,1)}), \texttt{Sign}^{\texttt{Sig}}_{SU}\rangle$ to R.M.

$\texttt{deleteRoleK}(r)$
- Verifies that there are no tuples $\langle \texttt{FK}, (r, -), -, -, -, -, sig\rangle$ with $\texttt{Ver}^{\texttt{Sig}}_{\texttt{k}^{\texttt{ver}}_{SU}}(\langle \texttt{FK}, (r, -), -, -, -, -\rangle, sig) = 1$
- Verifies that there are no tuples $\langle \texttt{RK}, -, (r, -), -, sig\rangle$ with $\texttt{Ver}^{\texttt{Sig}}_{\texttt{k}^{\texttt{ver}}_{SU}}(\langle \texttt{RK}, -, (r, -), -\rangle, sig) = 1$
- Remove $(r, v_r, -, -)$ from ROLES

$\texttt{assignUserToRoleK}(u, r)$
- Find $\langle \texttt{RK}, SU, (r, v_r), c, sig\rangle$ with $\texttt{Ver}^{\texttt{Sig}}_{\texttt{k}^{\texttt{ver}}_{SU}}(\langle \texttt{RK}, SU, (r, v_r), c\rangle, sig) = 1$
- Decrypt keys $(\texttt{k}^{\texttt{dec}}_{(r,v_r)}, \texttt{k}^{\texttt{sig}}_{(r,v_r)}) = \texttt{Dec}^{\texttt{Pub}}_{\texttt{k}^{\texttt{dec}}_{SU}}(c)$
- Send $\langle \texttt{RK}, u, (r, v_r), \texttt{Enc}^{\texttt{Pub}}_{\texttt{k}^{\texttt{enc}}_u}(\texttt{k}^{\texttt{dec}}_{(r,v_r)}, \texttt{k}^{\texttt{sig}}_{(r,v_r)}), \texttt{k}^{\texttt{ver}}_u\rangle$ to R.M.

$\texttt{rotateRoleKeyK}(r)$
- Generate new role key $(\texttt{k}^{\texttt{enc}}_{(r,v_r+1)}, \texttt{k}^{\texttt{dec}}_{(r,v_r+1)}) \leftarrow \texttt{Gen}^{\texttt{Pub}}$ and $(\texttt{k}^{\texttt{ver}}_{(r,v_r+1)}, \texttt{k}^{\texttt{sig}}_{(r,v_r+1)}) \leftarrow \texttt{Gen}^{\texttt{Sig}}$
- For all $\langle \texttt{RK}, u, (r, v_r), c, sig\rangle$ and $\texttt{Ver}^{\texttt{Sig}}_{\texttt{k}^{\texttt{ver}}_{SU}}(\langle \texttt{RK}, u, (r, v_r), c\rangle, sig) = 1$:
  - Send $\langle \texttt{RK}, u, (r, v_r + 1), \texttt{Enc}^{\texttt{Pub}}_{\texttt{k}^{\texttt{enc}}_u}(\texttt{k}^{\texttt{dec}}_{(r,v_r+1)}, \texttt{k}^{\texttt{sig}}_{(r,v_r+1)}), \texttt{Sign}^{\texttt{Sig}}_{SU}\rangle$ to R.M.
- For every fn such that there exists $\langle \texttt{FK}, (r, v_r), \langle fn, op\rangle, v_{fn}, c, SU, sig\rangle$ with $\texttt{Ver}^{\texttt{Sig}}_{\texttt{k}^{\texttt{ver}}_{SU}}(\langle \texttt{FK}, (r, v_r), p, v_{fn}, c, SU\rangle, sig) = 1$:
  - For every $\langle \texttt{FK}, (r, v_r), \langle fn, op'\rangle, v, c', SU, sig\rangle$ with $\texttt{Ver}^{\texttt{Sig}}_{\texttt{k}^{\texttt{dec}}_{SU}}(\langle \texttt{FK}, (r, v_r), \langle fn, op'\rangle, v, c', SU\rangle, sig) = 1$:
    - Decrypt $\texttt{k} = \texttt{Dec}^{\texttt{Pub}}_{\texttt{k}^{\texttt{dec}}_{(r,v_r)}}(c')$
    - Send $\langle \texttt{FK}, (r, v_r + 1), \langle fn, op'\rangle, v, \texttt{Enc}^{\texttt{Pub}}_{\texttt{k}^{\texttt{enc}}_{(r,v_r+1)}}(k), SU, \texttt{Sign}^{\texttt{Sig}}_{SU}\rangle$ to R.M.
- Update r in ROLES, i.e., replace $(r, v_r, \texttt{k}^{\texttt{enc}}_{(r,v_r)}, \texttt{k}^{\texttt{ver}}_{(r,v_r)})$ with $(r, v_r + 1, \texttt{k}^{\texttt{dec}}_{(r,v_r+1)}, \texttt{k}^{\texttt{ver}}_{(r,v_r+1)})$

$\texttt{prepareReencryptionK}(f)$
- For every $\langle \texttt{FK}, (r, v_r), p := \langle f.name, op\rangle, v, c, SU, sig\rangle$ with $\texttt{Ver}^{\texttt{Sig}}_{\texttt{k}^{\texttt{dec}}_{SU}}(\langle \texttt{FK}, (r, v_r), p, v, c, SU\rangle, sig) = 1$:
  - Generate new symmetric key $\texttt{k}' \leftarrow \texttt{Gen}^{\texttt{Sym}}$ for p
  - For all $\langle \texttt{FK}, id, \langle f.name, op'\rangle, v_{fn}, c'', SU, sig\rangle$ with $\texttt{Ver}^{\texttt{Sig}}_{\texttt{k}^{\texttt{ver}}_{SU}}(\langle \texttt{FK}, id, \langle f.name, op'\rangle, v_{fn}, c'', SU\rangle, sig) = 1$:
    - Send $\langle \texttt{FK}, id, \langle f.name, op\rangle, v_{fn} + 1, \texttt{Enc}^{\texttt{Pub}}_{id}(k'), SU, \texttt{Sign}^{\texttt{Sig}}_{SU}\rangle$
- Increment v_{fn} in FILES, i.e., set $v_{fn} := v_{fn} + 1$

$\texttt{assignPermissionToRoleK}(r, \langle op, f\rangle)$
- For all $\langle \texttt{FK}, SU, \langle f.name, RW\rangle, v, c, id, sig\rangle$ with $\texttt{Ver}^{\texttt{Sig}}_{\texttt{k}^{\texttt{ver}}_{id}}(\langle \texttt{FK}, SU, \langle f.name, RW\rangle, v, c, id\rangle, sig) = 1$:
  - If this adds Write permission to existing Read permission, i.e., $op = RW$ and there exists $\langle \texttt{FK}, (r, v_r), \langle f.name, read\rangle, v, c', SU, sig\rangle$ with $\texttt{Ver}^{\texttt{Sig}}_{\texttt{k}^{\texttt{ver}}_{SU}}(\langle \texttt{FK}, (r, v_r), \langle fn, read\rangle, v, c', SU\rangle, sig) = 1$:
    - Send $\langle \texttt{FK}, (r, v_r), \langle f.name, RW\rangle, v, c', SU, \texttt{Sign}^{\texttt{Sig}}_{SU}\rangle$ to R.M.
    - Delete $\langle \texttt{FK}, (r, v_r), \langle f.name, read\rangle, v, c', SU, sig\rangle$
  - If the role has no existing permission for the file, i.e., there does no exist $\langle \texttt{FK}, (r, v_r), \langle f.name, op'\rangle, v, c', SU, sig\rangle$ with $\texttt{Ver}^{\texttt{Sig}}_{\texttt{k}^{\texttt{ver}}_{SU}}(\langle \texttt{FK}, (r, v_r), \langle f.name, op'\rangle, v, c, SU\rangle, sig) = 1$:
    - Decrypt key $k = \texttt{Dec}^{\texttt{Pub}}_{\texttt{k}^{\texttt{dec}}_{SU}}(c)$
    - Send $\langle \texttt{FK}, (r, v_r), \langle f.name, op\rangle, v, \texttt{Enc}^{\texttt{Pub}}_{\texttt{k}^{\texttt{enc}}_{(r,v_r)}}(k), SU, \texttt{Sign}^{\texttt{Sig}}_{SU}\rangle$ to R.M.

$\texttt{revokePermissionFromRoleK}(r, \langle op, f\rangle)$
- If $op = write$:
  - For all $\langle \texttt{FK}, (r, v_r), \langle f.name, RW\rangle, v, c, SU, sig\rangle$ with $\texttt{Ver}^{\texttt{Sig}}_{\texttt{k}^{\texttt{ver}}_{SU}}(\langle \texttt{FK}, (r, v_r), \langle f.name, RW\rangle, v, c, SU\rangle, sig) = 1$:
    - Send $\langle \texttt{FK}, (role, v_r), \langle fn, read\rangle, v, c, SU, \texttt{Sign}^{\texttt{Sig}}_{SU}\rangle$ to R.M.
    - Delete $\langle \texttt{FK}, (r, v_r), \langle fn, RW\rangle, v, c, SU, sig\rangle$
- If $op = RW$:
  - Delete all $\langle \texttt{FK}, (r, -), \langle fn, -\rangle, -, -, -, -\rangle$

$\texttt{revokeUserFromRoleK}(u, r)$
- Delete all $\langle \texttt{RK}, -, (r, v_r), -, -\rangle$
- Delete all $\langle \texttt{FK}, (r, v_r), -, -, -, -, -\rangle$

$\texttt{read}(u, fn)$
- Find $\langle \texttt{F}, fn, v, c, id, sig\rangle$ with valid ciphertext c and valid signature sig, i.e., $\texttt{Ver}^{\texttt{Sig}}_{\texttt{k}^{\texttt{ver}}_{id}}(\langle \texttt{F}, fn, v, c, id\rangle, sig) = 1$
- Find a role r such that the following hold:
  - u in role r, i.e., there exists $\langle \texttt{RK}, u, (r, v_r), c', sig\rangle$ with $\texttt{Ver}^{\texttt{Sig}}_{\texttt{k}^{\texttt{ver}}_{SU}}(\langle \texttt{RK}, u, (r, v_r), c'\rangle, sig) = 1$
  - r has read access to version v of fn, i.e., there exists $\langle \texttt{FK}, (r, v_r), \langle fn, op\rangle, v, c'', SU, sig'\rangle$ with $\texttt{Ver}^{\texttt{Sig}}_{\texttt{k}^{\texttt{ver}}_{SU}}(\langle \texttt{FK}, (r, v_r), \langle fn, op\rangle, v, c'', SU\rangle, sig') = 1$
- Decrypt role key $\texttt{k}^{\texttt{dec}}_{(r,v_r)} = \texttt{Dec}^{\texttt{Pub}}_{\texttt{k}^{\texttt{dec}}_u}(c')$
- Decrypt file key $k = \texttt{Dec}^{\texttt{Pub}}_{\texttt{k}^{\texttt{dec}}_{(r,v_r)}}(c'')$
- Decrypt file $f = \texttt{Dec}^{\texttt{Sym}}_k(c)$

$\texttt{write}(u, f)$
- Find a role r such that the folloing hold:
  - u is in role r, i.e., there exists $\langle \texttt{RK}, u, (r, v_r), c, sig\rangle$ with $\texttt{Ver}^{\texttt{Sig}}_{\texttt{k}^{\texttt{ver}}_{SU}}(\langle \texttt{RK}, u, (r, v_r), c\rangle, sig) = 1$
  - r has write access to the newest version of fn, i.e., there exists $\langle \texttt{FK}, (r, v_r), \langle fn, RW\rangle, v_{fn}, c', SU, sig'\rangle$ and $\texttt{Ver}^{\texttt{Sig}}_{\texttt{k}^{\texttt{ver}}_{SU}}(\langle \texttt{FK}, (r, v_r), \langle fn, RW\rangle, v_{fn}, c', SU\rangle, sig') = 1$
- Decrypt role key $\texttt{k}^{\texttt{dec}}_{(r,v_r)} = \texttt{Dec}^{\texttt{Pub}}_{\texttt{k}^{\texttt{dec}}_u}(c)$
- Decrypt file key $k = \texttt{Dec}^{\texttt{Pub}}_{\texttt{k}^{\texttt{dec}}_{(r,v_r)}}(c')$
- Send $\langle \texttt{F}, fn, v_{fn}, \texttt{Enc}^{\texttt{Sym}}_k(f.content), (r, v_r), \texttt{Sign}^{\texttt{Sig}}_{(r,v_r)}\rangle$ to R.M.
- The R.M. receives $\langle \texttt{F}, fn, v, c'', (r, v_r), sig''\rangle$ and verifies the following:
  - The tuple is well-formed with $v = v_{fn}$
  - The signature is valid, i.e., $\texttt{Ver}^{\texttt{Sig}}_{\texttt{k}^{\texttt{ver}}_{(r,v_r)}}(\langle \texttt{F}, fn, v, c'', (r, v_r)\rangle, sig'') = 1$
  - r has write access to the newest version of fn, i.e., there exists $\langle \texttt{FK}, (r, v_r), \langle fn, op\rangle, v_{fn}, c', SU, sig\rangle$ and $\texttt{Ver}^{\texttt{Sig}}_{\texttt{k}^{\texttt{ver}}_{SU}}(\langle \texttt{FK}, (r, v_r), \langle fn, op\rangle, v_{fn}, c', SU\rangle, sig') = 1$
- If verification is successful, the R.M. replaces $\langle \texttt{F}, fn, -, -, -, -\rangle$ with $\langle \texttt{F}, fn, v_{fn}, c'', (r, v_r), sig''\rangle$

$\texttt{reencryptResourceK}(f)$:
- $f = \texttt{read}(u, fn)$
- $\texttt{write}(u, f)$

3.4 Analysis of the Cryptographic Computational Overhead

In this chapter, we calculate the computational overhead resulting from cryptographic operations. The number of cryptographic primitives executed for each state-change rule, and for read and write operations, is calculated.

The following cryptographic primitives are required: $\texttt{Gen}^{\texttt{Pub}}$ and $\texttt{Gen}^{\texttt{Sig}}$ are the operations used to create a key pair for de/encryption and for signing and verification, respectively. $\texttt{Gen}^{\texttt{Sym}}$ is the key generation operation for the symmetric cipher. The encryption operation for symmetric ciphers is $\texttt{Enc}^{\texttt{Sym}}$, and for asymmetric ciphers is $\texttt{Enc}^{\texttt{Pub}}$. Similarly, the decryption operations for symmetric and asymmetric ciphers are $\texttt{Dec}^{\texttt{Sym}}$ and $\texttt{Dec}^{\texttt{Pub}}$, respectively. Finally, for digital signatures, signing is represented by the $\texttt{Sign}^{\texttt{Sig}}$ operation, and verification of cipher data is represented by $\texttt{Ver}^{\texttt{Sig}}$.

In comparison to the CAC scheme proposed in [7], $\texttt{addResource}(f)$ only requires cryptographic operations if the resource f is protected with CAC. $\texttt{revokeUserFromRole}(u, r)$ considers the fact that not all files require re-encryption and the computational cost of eager re-encryption. Similarly, $\texttt{revokePermissionFromRole}(r, \langle op, f \rangle)$ also considers files that do not need to be re-encrypted and eager re-encryption. If the file is not protected with CAC, $\texttt{assignPermissionToRole}(r, \langle op, f \rangle)$ does not require cryptographic operations. Additionally, $\texttt{assignPredicate}(p, a)$ and $\texttt{revokePredicate}(p, a)$ consider files that require protection with CAC or no longer require it.

The cost details can be found in Table 3.2.

4 Implementation and Demonstration

To illustrate the functionality of our system, we presented a generic security model for a company in Section 4.1. Following this, in Section 4.2, we suggest a potential implementation in Prolog for certain components of the hybrid model. In Section 4.2, we provide a practical example that involves an administrator invoking some state-change rules. Lastly, in Section 4.3, we analyze the costs of our proposal.

4.1 Considered Security Model

To demonstrate the functioning of our proposed scheme, we imagined a generic company and assumed its security model. Therefore, we then defined the set of predicates P as:

- **enc**(f): indicates that the file f needs to be encrypted;
- **enc**(r): indicates that all files to which role r has access must be encrypted;
- **enc**(u): indicates that all files to which user u has access must be encrypted;
- **eager**(f): indicates that the file f must be re-encrypted immediately;
- **eager**(r): indicates that all files to which role r has access must be re-encrypted immediately;
- **eager**(u): indicates that all files to which user u has access must be re-encrypted immediately;
- **colludingProne**(u): indicates that the user u may collude with the CSP to obtain the access to a file;
- **cspNoEnforce**(f): indicates that the CSP is not trust to protect the file f.

The entailment function resolves queries using predicates. Two helper functions, *isEncryptionNeededFile*(f) and *isEagerReencNeededFile*(f), were defined to determine if file f should be protected by CAC (and therefore encrypted) and when re-encryption of a file should occur in eager mode. In a real implementation, it is possible to replace a call to either of these functions with their body. The complete definition of the entailment function is shown in Figure 4.1. Each line of a function represents a boolean condition that must be considered in disjunction with the other lines.

4.2 Prolog Implementation

To demonstrate the functionality of our proposal, we implemented the extended RBAC scheme in Prolog. The program comprises four files: `rbac_spec.pro`, which manages the centralized AC, `cac_spec.pro`, which manages the CAC proposed in Section 3.3, `hybrid_spec.pro`, which manages the extended RBAC model, and `predicates.pro`, which defines the security model as outlined in Section 4.1. The security model of a company can be redefined by an administrator through the modification of the `predicates.pro` file. This file contains the definitions of predicates set P and entailment function $\vdash$.

The extended RBAC scheme's state-change rules are invoked by the administrator through a Prolog query. The query name coincides with the names defined in Section 3.2, and the parameters

Table 3.2: Computational costs of operations

Operation	**Cryptographic operations**
`addUser`(u)	$\texttt{Gen}^{\texttt{Pub}} + \texttt{Gen}^{\texttt{Sig}}$
`addRole`(r)	$\texttt{Gen}^{\texttt{Pub}} + \texttt{Gen}^{\texttt{Sig}} + 2 \cdot \texttt{Enc}^{\texttt{Pub}} + \texttt{Sign}^{\texttt{Sig}}$
`addResource`(f)	If ***isEncryptionNeeded***(f) then $2 \cdot (\texttt{Sign}^{\texttt{Sig}} + \texttt{Ver}^{\texttt{Sig}}) + \texttt{Gen}^{\texttt{Sym}} + \texttt{Enc}^{\texttt{Pub}} + \texttt{Enc}^{\texttt{Sym}}$
`deleteResource`(f)	No cryptographic computation needed
`assignUserToRole`(u, r)	$2 \cdot (\texttt{Enc}^{\texttt{Pub}} + \texttt{Dec}^{\texttt{Pub}}) + \texttt{Sign}^{\texttt{Sig}} + \texttt{Ver}^{\texttt{Sig}}$
`revokeUserFromRole`(u, r)	It is the sum of: • if ***isRoleKeyRotationNeededOnRUR***(u, r) then $\texttt{Dec}^{\texttt{Pub}} + \texttt{Gen}^{\texttt{Pub}} + \texttt{Gen}^{\texttt{Sig}} + (2 \cdot \lvert\{(-, r) \in UR\}\rvert)$, else no cryptographic computation needed; • $\texttt{Gen}^{\texttt{Sym}} \cdot \lvert\{(r, \langle -, f\rangle) \in FR : f \in F \wedge$ ***isReencryptionNeededOnRUR***$(u, r, f)\}\rvert$ • $(\texttt{Enc}^{\texttt{Pub}} + \texttt{Sign}^{\texttt{Sig}} + \texttt{Ver}^{\texttt{Sig}}) \cdot \lvert\{(-, \langle -, f\rangle) \in FR : (r, \langle -, f\rangle) \in FR \wedge$ ***isReencryptionNeededOnRUR***$(u, r, f)\}\rvert$ • $(\texttt{Dec}^{\texttt{Sym}} + \texttt{Enc}^{\texttt{Sym}}) \cdot \lvert\{(r, f) \in FR : f \in F \wedge$ ***isEagerReencNeededOnRUR***$(u, r, f)\}\rvert$
`assignPermissionToRole`$(r, \langle op, f\rangle)$	If not ***isEncryptionNeeded***(f) then no cryptographic computation is needed. If $\exists(r, \langle -, f\rangle) \in FR$ then $\texttt{Sign}^{\texttt{Sig}} + 2 \cdot \texttt{Ver}^{\texttt{Sig}}$, else $\texttt{Sign}^{\texttt{Sig}} + 2 \cdot \texttt{Ver}^{\texttt{Sig}} + \texttt{Enc}^{\texttt{Pub}} + \texttt{Dec}^{\texttt{Pub}}$
`revokePermissionFromRole`$(r, \langle op, f\rangle)$	If not ***isEncryptionNeeded***(f) $\vee$ not ***isReencryptionNeededOnRP***$(r, \langle op, f\rangle)$ then no cryptographic computation is needed. If $op = write$ then $2 \cdot (\texttt{Ver}^{\texttt{Sig}} + \texttt{Sign}^{\texttt{Sig}})$. If ***isEagerReencNeededOnRP***$(r, \langle op, f\rangle)$ then $\texttt{Gen}^{\texttt{Sym}} + \texttt{Dec}^{\texttt{Sym}} + \texttt{Enc}^{\texttt{Sym}} + (\texttt{Ver}^{\texttt{Sig}} + \texttt{Enc}^{\texttt{Pub}} + \texttt{Sign}^{\texttt{Sig}}) \cdot \lvert\{(r', \langle op, f\rangle) \in FR : r \neq r'\}\rvert$, else $\texttt{Gen}^{\texttt{Sym}} + (\texttt{Ver}^{\texttt{Sig}} + \texttt{Enc}^{\texttt{Pub}} + \texttt{Sign}^{\texttt{Sig}}) \cdot \lvert\{(r', \langle op, f\rangle) \in FR : r \neq r'\}\rvert$
`read`(u, fn)	$3 \cdot \texttt{Ver}^{\texttt{Sig}} + 2 \cdot \texttt{Dec}^{\texttt{Pub}} + \texttt{Dec}^{\texttt{Sym}}$
`write`(u, f)	$5 \cdot \texttt{Ver}^{\texttt{Sig}} + 2 \cdot (\texttt{Dec}^{\texttt{Pub}} + \texttt{Sign}^{\texttt{Sig}}) + \texttt{Enc}^{\texttt{Sym}}$
`assignPredicate`(p, a) and `revokePredicate`(p, a)	It is the sum of: • $(2 \cdot (\texttt{Sign}^{\texttt{Sig}} + \texttt{Ver}^{\texttt{Sig}}) + \texttt{Gen}^{\texttt{Sym}} + \texttt{Enc}^{\texttt{Pub}} + \texttt{Enc}^{\texttt{Sym}}) \cdot \lvert\{f \in F :$ ***isEncryptionNeeded***(f) is changed to *true*$\}\rvert$ • $(3 \cdot \texttt{Ver}^{\texttt{Sig}} + 2 \cdot \texttt{Dec}^{\texttt{Pub}} + \texttt{Dec}^{\texttt{Sym}}) \cdot \lvert\{f \in F :$ ***isEncryptionNeeded***(f) is changed to *false*$\}\rvert$

are passed as arguments. For instance, the query `revokePermissionFromRole` is invoked as follows: `revokePermissionFromRole`$(r, \langle op, f \rangle)$. Prolog will invoke the state-change rules of the centralized AC system, which are identifiable by the words `[TRADIT]` at the beginning and light green color in *stdout*. The logic part of CAC is implemented by Prolog, which interfaces with a second component for cryptographic operations and interaction with the monitor, data manager, and metadata manager. The Prolog program outputs tuples to *stdout* that are to be created or deleted. These tuples are preceded by the word `[CRYPT]` and are colored purple.

The Prolog program implementation is fully represented in Appendix A.

4.3 Practical Demonstration

To demonstrate our scheme's functionality, we execute a sequence of state-change rules Ψ. Our example is based on the security model proposed in Section 4.1. The process begins with an initial empty state $(\{\} = U, \{\} = R, \{\} = F, \{\} = UR, \{\} = FR, \{\} = PA) \in \Gamma$.

Between lines 1 and 20, users and roles are created, and users are assigned to roles. Tuples for users are not created at this stage because, for reasons of efficiency, these operations are performed only when actually needed, Between lines 21 and 45, resources are created, and permissions are assigned. Finally, the membership of user *alice* in the *engineers* role is revoked. The file *engineers* will be re-encrypted due to the following conditions being met:

- the file *engineers* is $\mathbf{enc}(engineers)$, and consequently is also $isEncryptionNeededFile(engineers)$;
- the CSP is $\mathbf{cspNoEnforce}(engineers)$;
- exists the user *david* that is $\mathbf{colludingProne}(david)$ and that $\boldsymbol{canDo}(david, \langle -, engineers \rangle)$;
- the user *alice* is $\mathbf{eager}(alice)$, and consequently *engineers* is $\mathbf{eager}(engineers)$.

The computational cost for this example is:

- `addUser(alice)`: $\texttt{Gen}^{\texttt{Pub}} + \texttt{Gen}^{\texttt{Sig}}$;
- `addUser(david)`: $\texttt{Gen}^{\texttt{Pub}} + \texttt{Gen}^{\texttt{Sig}}$;
- `addUser(bob)`: $\texttt{Gen}^{\texttt{Pub}} + \texttt{Gen}^{\texttt{Sig}}$;
- `addRole(staff)`: $\texttt{Gen}^{\texttt{Pub}} + \texttt{Gen}^{\texttt{Sig}} + 2 \cdot \texttt{Enc}^{\texttt{Pub}} + \texttt{Sign}^{\texttt{Sig}}$;

Figure 4.1: The instantiation of the entailment function $\vdash$

$isEncryptionNeededFile(f)$:
- $\mathbf{enc}(f)$
- $\exists r \in R : \mathbf{enc}(r) \wedge (r, \langle -, f \rangle) \in FR$
- $\exists u \in U, r \in R : \mathbf{enc}(u) \wedge (u, r) \in UR \wedge (r, \langle -, f \rangle) \in FR$

$isEagerReencNeededFile(f)$:
- $\mathbf{eager}(f)$
- $\exists r \in R : \mathbf{eager}(r) \wedge (r, \langle -, f \rangle) \in FR$
- $\exists u \in U, r \in R : \mathbf{eager}(u) \wedge (u, r) \in UR \wedge (r, \langle -, f \rangle) \in FR$

$\boldsymbol{isEncryptionNeeded}(f)$:
- $isEagerReencNeededFile(f)$

$\boldsymbol{isEagerReencNeededOnRP}(r, \langle op, f \rangle)$:
- $\boldsymbol{isReencryptionNeededOnRP}(r, \langle op, f \rangle) \wedge isEagerReencNeededFile(f)$

$\boldsymbol{isEagerReencNeededOnRUR}(u, r, f)$:
- $\boldsymbol{isReencryptionNeededOnRUR}(u, r, f) \wedge isEagerReencNeededFile(f)$

$\boldsymbol{isReencryptionNeededOnRUR}(u, r, f)$:
- $isEncryptionNeededFile(f) \wedge \mathbf{cspNoEnforce}(f) \wedge (\exists u' \in U : \mathbf{colludingProne}(u') \wedge \boldsymbol{canDo}(u', \langle -, f \rangle))$

$\boldsymbol{isRoleKeyRotationNeededOnRUR}(u, r)$:
- $\mathbf{colludingProne}(u) \wedge (u, r) \in UR$

$\boldsymbol{isReencryptionNeededOnRP}(r, \langle op, f \rangle)$:
- $isEncryptionNeededFile(f) \wedge \mathbf{cspNoEnforce}(f) \wedge (\exists u \in U : \mathbf{colludingProne}(u) \wedge \boldsymbol{canDo}(u, \langle read, f \rangle))$

- `addRole(engineers)`: $\texttt{Gen}^{\texttt{Pub}} + \texttt{Gen}^{\texttt{Sig}} + 2 \cdot \texttt{Enc}^{\texttt{Pub}} + \texttt{Sign}^{\texttt{Sig}}$;
- `assignUserToRole(alice, staff)`: $2 \cdot (\texttt{Enc}^{\texttt{Pub}} + \texttt{Dec}^{\texttt{Pub}}) + \texttt{Sign}^{\texttt{Sig}} + \texttt{Ver}^{\texttt{Sig}}$;
- `assignUserToRole(alice, engineers)`: $2 \cdot (\texttt{Enc}^{\texttt{Pub}} + \texttt{Dec}^{\texttt{Pub}}) + \texttt{Sign}^{\texttt{Sig}} + \texttt{Ver}^{\texttt{Sig}}$;
- `assignUserToRole(bob, staff)`: $2 \cdot (\texttt{Enc}^{\texttt{Pub}} + \texttt{Dec}^{\texttt{Pub}}) + \texttt{Sign}^{\texttt{Sig}} + \texttt{Ver}^{\texttt{Sig}}$;
- `assignUserToRole(david, staff)`: $2 \cdot (\texttt{Enc}^{\texttt{Pub}} + \texttt{Dec}^{\texttt{Pub}}) + \texttt{Sign}^{\texttt{Sig}} + \texttt{Ver}^{\texttt{Sig}}$;
- `assignPredicate(shuttle, enc)`: no cryptographic computation;
- `assignPredicate(temperature, enc)`: no cryptographic computation;
- `addResource(shuttle)`: $2 \cdot (\texttt{Sign}^{\texttt{Sig}} + \texttt{Ver}^{\texttt{Sig}}) + \texttt{Gen}^{\texttt{Sym}} + \texttt{Enc}^{\texttt{Pub}} + \texttt{Enc}^{\texttt{Sym}}$;
- `addResource(budget)`: no cryptographic computation;
- `addResource(temperature)`: $2 \cdot (\texttt{Sign}^{\texttt{Sig}} + \texttt{Ver}^{\texttt{Sig}}) + \texttt{Gen}^{\texttt{Sym}} + \texttt{Enc}^{\texttt{Pub}} + \texttt{Enc}^{\texttt{Sym}}$;
- `assignPermissionToRole(staff, budget)`: no cryptographic computation;
- `assignPermissionToRole(engineers, shuttle)`: $\texttt{Sign}^{\texttt{Sig}} + 2 \cdot \texttt{Ver}^{\texttt{Sig}} + \texttt{Enc}^{\texttt{Pub}} + \texttt{Dec}^{\texttt{Pub}}$;
- `assignPermissionToRole(engineers, temperature)`: $\texttt{Sign}^{\texttt{Sig}} + 2 \cdot \texttt{Ver}^{\texttt{Sig}} + \texttt{Enc}^{\texttt{Pub}} + \texttt{Dec}^{\texttt{Pub}}$;
- `assignPredicate(alice, eager)`: no cryptographic computation;
- `assignPredicate(shuttle, cspNoEnforce)`: no cryptographic computation;
- `assignUserToRole(david, engineers)`: $2 \cdot (\texttt{Enc}^{\texttt{Pub}} + \texttt{Dec}^{\texttt{Pub}}) + \texttt{Sign}^{\texttt{Sig}} + \texttt{Ver}^{\texttt{Sig}}$;
- `assignPredicate(david, collusionProne)`: no cryptographic computation;
- `revokeUserFromRole(alice, engineers)`: $2 \cdot (\texttt{Dec}^{\texttt{Pub}} + \texttt{Gen}^{\texttt{Pub}} + \texttt{Gen}^{\texttt{Sig}} + \texttt{Gen}^{\texttt{Sym}} + \texttt{Enc}^{\texttt{Pub}} + \texttt{Sign}^{\texttt{Sig}} + \texttt{Ver}^{\texttt{Sig}} + \texttt{Dec}^{\texttt{Sym}} + \texttt{Enc}^{\texttt{Sym}})$.

The computational cost of this example is: $7 \cdot \texttt{Gen}^{\texttt{Sig}} + 7 \cdot \texttt{Gen}^{\texttt{Pub}} + 4 \cdot \texttt{Gen}^{\texttt{Sig}} + 4 \cdot \texttt{Enc}^{\texttt{Sym}} + 20 \cdot \texttt{Enc}^{\texttt{Pub}} + 2 \cdot \texttt{Dec}^{\texttt{Sym}} + 14 \cdot \texttt{Dec}^{\texttt{Pub}} + 15 \cdot \texttt{Sign}^{\texttt{Sig}} + 15 \cdot \texttt{Ver}^{\texttt{Sig}}$. Using the costs in Table 4.1 and assuming that each file occupies 10MB, the total cost is 2255.453ms.

5 Conclusion

CAC is an effective (but not particularly efficient) solution to protect sensitive data stored in the cloud from external attackers, malicious insiders, and honest-but-curious CSPs. More in detail, the

Table 4.1: Computational Costs of Cryptographic Primitives

Cryptographic Primitive	**Cost**
$\texttt{Gen}^{\texttt{Sym}}$	0.033ms
$\texttt{Gen}^{\texttt{Pub}}$ and $\texttt{Gen}^{\texttt{Sig}}$	112.673ms
$\texttt{Enc}^{\texttt{Sym}}$	11.860ms/MB
$\texttt{Enc}^{\texttt{Pub}}$	0.417ms
$\texttt{Dec}^{\texttt{Sym}}$	16.820ms/MB
$\texttt{Dec}^{\texttt{Pub}}$	11.986ms
$\texttt{Sign}^{\texttt{Sig}}$	1.656ms
$\texttt{Ver}^{\texttt{Sig}}$	0.269ms

computational overhead affecting CAC is — to a significant extent — due to the default security model usually considered in the design of CAC schemes, i.e., the CSP is always partially trusted with respect to any resource, all users are always untrusted, and all resources are always sensitive. However, we note that such a security model may not suit every scenario, and instead the capability of expressing an alternative security model may considerably reduce the computational overhead of CAC, hence making CAC a viable solution to enforce AC policies over cloud-hosted data. Therefore, in this thesis we proposed a hybrid AC scheme comprising an extended RBAC model whose policies are automatically compiled and enforced by a centralized RBAC enforcement mechanism and CAC. The extended RBAC model allows administrators to define security models through predicates specified for users, roles, resources, and CSPs. Besides, our extended RBAC model includes new queries allowing to fine-tune the execution of cryptographic operations in the underlying CAC scheme — which we modified accordingly. Moreover, administrators can define predicates and redefine the entailment function in such a way to best represent their specific scenario. Also, we analyzed the computational costs of state-change rules in the extended RBAC scheme with respect to the cryptographic computations involved. Finally, we provide a proof-of-concept implementation in Prolog and a consequent demonstration of our contributions in a reasonable security model concretely defining the predicates and the entailment function used in our extended RBAC scheme, calculating the number and type of cryptographic computations involved and the corresponding execution time.

Figure 4.2: The example code

```
1. addUser(alice)
2. [TRADIT] Create user alice
3. addUser(david)
4. [TRADIT] Create user david
5. addUser(bob)
6. [TRADIT] Create user bob
7. addRole(staff)
8. [TRADIT] Create role staff
9. addRole(engineers)
10. [TRADIT] Create role engineers
11. assignUserToRole(alice, staff)
12. [TRADIT] Assign alice to staff
13. assignUserToRole(alice, engineers)
14. [TRADIT] Assign alice to engineers
15. assignUserToRole(bob, staff)
16. [TRADIT] Assign bob to staff
17. assignUserToRole(david, staff)
18. [TRADIT] Assign david to staff
19. assignPredicate(shuttle, enc)
20. assignPredicate(temperature, enc)
21. addResource(shuttle)
22. [TRADIT] Create file shuttle
23. [CRYPTO] +Resource(shuttle, 1)
24. addResource(budget)
25. [TRADIT] Create file budget
26. addResource(temperature)
27. [TRADIT] Create file temperature
28. [CRYPTO] +Resource(temperature, 1)
29. assignPermissionToRole(staff, budget)
30. [TRADIT] Assign budget access to staff
29. assignPermissionToRole(engineers,
    shuttle)
30. [TRADIT] Assign shuttle access to
    engineers
31. [CRYPTO] +Role(engineers, 1)
32. [CRYPTO] +PermissionRole(engineers, 1,
    shuttle, 1)
33. [CRYPTO] +User(alice)
34. [CRYPTO] +UserRole(alice, engineers, 1)
35. assignPermissionToRole(engineers,
    temperature)
36. [TRADIT] Assign temperature access to
    engineers
37. [CRYPTO] +PermissionRole(engineers, 1,
    temperature, 1)
38. assignPredicate(alice, eager)
39. assignPredicate(shuttle, cspNoEnforce)
40. assignUserToRole(david, engineers)
41. [TRADIT] Assign david to engineers
42. [CRYPTO] +User(david)
43. [CRYPTO] +UserRole(david, engineers, 1)
44. assignPredicate(david, collusionProne)
45. revokeUserFromRole(alice, engineers)
46. [TRADIT] Revoke alice from engineers
47. [CRYPTO] -UserRole(alice, engineers, 1)
48. [CRYPTO] +Resource(shuttle, 2)
49. [CRYPTO] +PermissionRole(engineers, 1,
    shuttle, 2)
50. [CRYPTO] -PermissionRole(engineers, 1,
    shuttle, 1)
51. [CRYPTO] -Resource(shuttle, 1)
```

5.1 Future Work

Our work can be extended to enhance the expressiveness of the proposed RBAC model by supporting negative permissions, that is, allowing the overriding of the normal behavior of RBAC by permitting or prohibiting a specific user from accessing a particular resource. Additionally, we believe it is possible to reduce the number of asymmetric operations in the CAC scheme we consider in favor of symmetric operations. Moreover, to enhance efficiency, cryptographic operations may be performed only when strictly necessary. For example, the asymmetric keys of a role may be created only if a resource protected with CAC is associated with that role. Another possible future direction is to further investigate the computational overhead of our hybrid system and compare it with that of the vanilla CAC scheme — as proposed in [7] — for different sequences of state-change-rules. Additionally, it would be interesting to identify real-world scenarios over which to apply our hybrid system and evaluate the (eventual) performance improvement. Finally, we are considering replacing Prolog with more efficient languages or technologies. For example, Datalog — which focuses on querying data in relational databases — is based on Prolog but offers better performance in queries with dynamic data, which is precisely the situation of our proposed hybrid scheme.

Bibliography

[1] Barrett. Security news this week: The pentagon left data exposed in the cloud. https://www.wired.com/author/brian-barrett/.

[2] Stefano Berlato, Roberto Carbone, Adam J. Lee, and Silvio Ranise. Formal modelling and automated trade-off analysis of enforcement architectures for cryptographic access control in the cloud. *ACM Trans. Priv. Secur.*, 25(1), nov 2021.

[3] Fangbo Cai, Nafei Zhu, Jingsha He, Pengyu Mu, Wenxin Li, and Yi Yu. Survey of access control models and technologies for cloud computing. *Cluster Computing*, 22:6111–6122, 2019.

[4] European Parliament and Council of European Union. Regulation (eu) no 2016/679, 2016. https://eur-lex.europa.eu/legal-content/EN/TXT/HTML/?uri=CELEX:32016R0679#d1e6226-1-1.

[5] Eurostat. Cloud computing - statistics on the use by enterprises. https://ec.europa.eu/eurostat/statistics-explained/index.php?title=Cloud_computing_-_statistics_on_the_use_by_enterprises.

[6] American National Standard for Information Technology. Role based access control. Standard, American National Standards Institute, Inc., February 2004.

[7] William C. Garrison, Adam Shull, Steven Myers, and Adam J. Lee. On the practicality of cryptographically enforcing dynamic access control policies in the cloud. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 819–838, May 2016.

[8] Jeremy Horwitz and Ben Lynn. Toward hierarchical identity-based encryption. In *Advances in Cryptology - EUROCRYPT 2002, International Conference on the Theory and Applications of Cryptographic Techniques*, pages 466–481, 04 2002.

[9] Identity Tefth Resource Center. Itrc - 2022 data breach report, 2022.

[10] Ninghui Li and Mahesh V. Tripunitara. Security analysis in role-based access control. *ACM Trans. Inf. Syst. Secur.*, 9(4):391–420, nov 2006.

[11] Newman. Blame human error for wwe and verizon's massive data exposures. https://www.wired.com/story/amazon-s3-data-exposure/.

[12] Newman. The scarily common screw-up that exposed 198 million voter records. https://www.wired.com/story/voter-records-exposed-database/.

[13] Pierangela Samarati and Sabrina Capitani de Vimercati. Access control: Policies, models, and mechanisms. In Riccardo Focardi and Roberto Gorrieri, editors, *Foundations of Security Analysis and Design*, pages 137–196, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[14] Ravi Sandhu. Access control: principle and practice. *Advances in Computers*, 46:237 – 286, 10 1998.

Appendix A Prolog Code

A.1 hybrid_spec.pro

```
% dynamic predicates

:- dynamic user/1.
:- dynamic role/1.
:- dynamic file/1.
:- dynamic isMember/2.
:- dynamic allow/2.


% import

:- ensure_loaded('cac_spec.pro').
:- ensure_loaded('rbac_spec.pro').
:- ensure_loaded('predicates.pro').


% utils

encryptFile :-
    foreach(
        (
            isEncryptionNeeded(FILE),
            \+ encryptedFileK(FILE)
        ),
        (
            addResourceK(FILE),
            foreach(
                allow(ROLE, FILE),
                assignPermissionToRole(ROLE, FILE)
            )
        )
    ).

decryptFile :-
    foreach(
        (
            \+ isEncryptionNeeded(FILE),
            encryptedFileK(FILE)
        ),
        (
            foreach(
```

```
            allow(ROLE, FILE),
            revokePermissionFromRole(ROLE, FILE)
        ),
        deleteResourceK(FILE)
    )
).

updateEncryptedFile :-
    encryptFile, decryptFile.


% operations

addUser(USER) :-
    assert(user(USER)),
    addUserC(USER).

deleteUser(USER) :-
    user(USER),

    % delete memberships
    foreach(
        isMember(USER, ROLE),
        revokeUserFromRole(USER, ROLE)
    ),

    retract(user(USER)),
    deleteUserC(USER),
    deleteUserK(USER).

addRole(ROLE) :-
    assert(role(ROLE)),
    addRoleC(ROLE).

deleteRole(ROLE) :-
    role(ROLE),

    foreach(
        allow(ROLE, FILE),
        revokePermissionFromRole(ROLE, FILE)
    ),

    foreach(
        isMember(USER, ROLE),
        revokePermissionFromRole(USER, ROLE)
    ),

    retract(role(ROLE)),
    deleteRoleC(ROLE),
    deleteRoleK(ROLE).

addResource(FILE) :-
    assert(file(FILE)),
```

```
    addResourceC(FILE),
    (
        isEncryptionNeeded(FILE) -> addResourceK(FILE);
        true
    ).

    deleteResource(FILE) :-
        file(FILE),

        foreach(
            allow(ROLE, FILE),
            retract(allow(ROLE, FILE))
        ),

        retract(file(FILE)),
        deleteResourceC(FILE),
        deleteResourceK(FILE).

deleteResource(FILE) :-
    file(FILE),

    foreach(
        allow(ROLE, FILE),
        revokePermissionFromRole(ROLE, FILE)
    ),

    retract(file(FILE)),
    deleteResourceC(FILE),
    deleteResourceK(FILE).


assignUserToRole(USER, ROLE) :-
    user(USER), role(ROLE), % checks
    assert(isMember(USER, ROLE)),
    assignUserToRoleC(USER, ROLE),
    (
        file(FILE), allow(ROLE, FILE),
        isEncryptionNeeded(FILE)
        ->
        addUserK(USER), addRoleK(ROLE),
        assignUserToRoleK(USER, ROLE)
        ;
        true
    ).

assignPermissionToRole(ROLE, FILE) :-
    role(ROLE), file(FILE), % checks
    assert(allow(ROLE, FILE)),
    assignPermissionToRoleC(ROLE, FILE),
    (
        isEncryptionNeeded(FILE)
        ->
        addRoleK(ROLE),
```

```
        assignPermissionToRoleK(ROLE, FILE),
        foreach(
            isMember(USER, ROLE),
            (
                addUserK(USER),
                assignUserToRoleK(USER, ROLE)
            )
        )
        ;
        true
    ).

revokePermissionFromRole(ROLE, FILE) :-
    role(ROLE), file(FILE),
    revokePermissionFromRoleC(ROLE, FILE),
    revokePermissionFromRoleR(ROLE, FILE),
    (
        isReencryptionNeededOnRP(ROLE, FILE)
        ->
        prepareReenctyptionResourceR(FILE),
        (
            isEagerReencNeededOnRP(ROLE, FILE)
            ->
            reencryptResourceR(FILE)
            ;
            true
        )
        ;
        true
    ),

    % must be done at last!
    retract(allow(ROLE, FILE)).

revokeUserFromRole(USER, ROLE) :-
    user(USER), role(ROLE),
    revokeUserFromRoleC(USER, ROLE),
    revokeUserFromRoleK(USER, ROLE),

    % role tuple
    (
        isRoleKeyRotationNeededOnRUR(USER, ROLE)
        ->
        rotateRoleKeyK(ROLE)
        ;
        true
    ),

    % files
    (
        isReencryptionNeededOnRUR(USER, ROLE, FILE)
        ->
        prepareReenctyptionResourceR(FILE),
```

```
        (
            isEagerReencNeededOnRUR(USER, ROLE, FILE)
            ->
            reencryptResourceR(FILE)
            ;
            true
        )
        ;
        true
    ),

    retract(isMember(USER, ROLE)).


% queries

canAccessPure(USER, FILE) :-
    user(USER), file(FILE), % checks
    role(ROLE),
    isMember(USER, ROLE),
    allow(ROLE, FILE).
```

A.2 cac_spec.pro

```
% dynamic predicates

:- dynamic userPublicKey/1. % user
:- dynamic rolePublicKey/2. % role, key_version
:- dynamic encFile/2. % file, key_version
:- dynamic roleKey/3. % user, role, role_key_version
:- dynamic fileKey/4. % role, role_key_version, file, file_key_version


% utils

cacTuple(OP, TYPE, TEXT, CONTENT) :-
    atom_concat("[CRYPTO]␣", OP, P1),
    atom_concat(P1, TYPE, P2),
    atom_concat(P2, "(", P3),
    atom_concat(P3, TEXT, P4),
    atom_concat(P4, ")~n", P5),
    ansi_format([fg(magenta)], P5, CONTENT).


% operations

% user management

addUserK(USER) :- userPublicKey(USER).
addUserK(USER) :-
    \+ userPublicKey(USER),
    assert(userPublicKey(USER)),
```

```
    cacTuple("+", "User", "~a", [USER]).

deleteUserK(USER) :- \+ userPublicKey(USER).
deleteUserK(USER) :- userPublicKey(USER), roleKey(USER, _, _).
deleteUserK(USER) :-
    userPublicKey(USER), \+ roleKey(USER, _, _), % checks
    retract(userPublicKey(USER)),
    cacTuple("-", "User", "~a", [USER]).

% role management

cac_createRoleTuple(ROLE, ROLE_VERSION) :-
    assert(rolePublicKey(ROLE, ROLE_VERSION)),
    cacTuple("+", "Role", "~a,␣~a", [ROLE, ROLE_VERSION]).

cac_deleteRoleTuple(ROLE, ROLE_VERSION) :-
    retract(rolePublicKey(ROLE, ROLE_VERSION)),
    cacTuple("-", "Role", "~a,␣~a", [ROLE, ROLE_VERSION]).

addRoleK(ROLE) :- rolePublicKey(ROLE, _).
addRoleK(ROLE) :-
    \+ rolePublicKey(ROLE, _),
    cac_createRoleTuple(ROLE, 1).

deleteRoleK(ROLE) :- \+ rolePublicKey(ROLE, _).
deleteRoleK(ROLE) :- rolePublicKey(ROLE, _), fileKey(ROLE, _, _, _).
deleteRoleK(ROLE) :- rolePublicKey(ROLE, _), roleKey(_, ROLE, _).
deleteRoleK(ROLE) :-
    rolePublicKey(ROLE, ROLE_VERSION),
    \+ fileKey(ROLE, _, _, _), \+ roleKey(_, ROLE, _),
    cac_deleteRoleTuple(ROLE, ROLE_VERSION).


rotateRoleKeyK(ROLE) :- \+ rolePublicKey(ROLE, _).
rotateRoleKeyK(ROLE) :-
    rolePublicKey(ROLE, ROLE_VERSION),

    % new role key
    sum_list([ROLE_VERSION, 1], NEW_ROLE_VERSION),

    % new role
    cac_createRoleTuple(ROLE, NEW_ROLE_VERSION),

    % copy members
    foreach(
        roleKey(USER, ROLE, ROLE_VERSION),
        (
            cac_createUserRoleTuple(USER, ROLE, NEW_ROLE_VERSION),
            cac_deleteUserRoleTuple(USER, ROLE, ROLE_VERSION)
        )
    ),

    % copy permissions
```

```
    foreach(
        fileKey(ROLE, ROLE_VERSION, FILE, FILE_VERSION),
        (
            cac_createPermissionRole(ROLE, NEW_ROLE_VERSION, FILE, FILE_VERSION),
            cac_deletePermissionRole(ROLE, ROLE_VERSION, FILE, FILE_VERSION)
        )
    ),

    % delete old role
    cac_deleteRoleTuple(ROLE, ROLE_VERSION).

% resource management

cac_createResourceTuple(FILE, FILE_VERSION) :-
    assert(encFile(FILE, FILE_VERSION)),
    cacTuple("+", "Resource", "~a,␣~a", [FILE, FILE_VERSION]).

cac_deleteResourceTuple(FILE, FILE_VERSION) :-
    retract(encFile(FILE, FILE_VERSION)),
    cacTuple("-", "Resource", "~a,␣~a", [FILE, FILE_VERSION]).

addResourceK(FILE) :- encFile(FILE, _).
addResourceK(FILE) :-
    \+ encFile(FILE, _),
    cac_createResourceTuple(FILE, 1).

deleteResourceK(FILE) :- \+ encFile(FILE, _).
deleteResourceK(FILE) :- encFile(FILE, _), fileKey(_, _, FILE, _).
deleteResourceK(FILE) :-
    encFile(FILE, _), \+ fileKey(_, _, FILE, _),

    foreach(
        encFile(FILE, FILE_VERSION),
        cac_deleteResourceTuple(FILE, FILE_VERSION)
    ).

prepareReenctyptionResourceR(FILE) :- \+ encFile(FILE, _).
prepareReenctyptionResourceR(FILE) :- % the file hasn't already reencrypted
    encFile(FILE, OLD_VERSION),
    encFile(FILE, CURRENT_VERSION),
    OLD_VERSION < CURRENT_VERSION, % find versions

    sum_list([CURRENT_VERSION, 1], NEW_VERSION), % new version key

    % create new version
    cac_createResourceTuple(FILE, NEW_VERSION),

    % copy permission (and delete current ones)
    foreach(
        fileKey(ROLE, ROLE_VERSION, FILE, OLD_VERSION),
        (
            cac_createPermissionRole(ROLE, ROLE_VERSION, FILE, NEW_VERSION),
            cac_deletePermissionRole(ROLE, ROLE_VERSION, FILE, CURRENT_VERSION)
```

```
        )
    ),

    % delete current version
    cac_deleteResourceTuple(FILE, CURRENT_VERSION).
prepareReenctyptionResourceR(FILE) :- % the file has only one version
    encFile(FILE, CURRENT_VERSION),
    sum_list([CURRENT_VERSION, 1], NEW_VERSION), % new version key

    % create new version
    cac_createResourceTuple(FILE, NEW_VERSION),

    % copy permission
    foreach(
        fileKey(ROLE, ROLE_VERSION, FILE, CURRENT_VERSION),
        cac_createPermissionRole(ROLE, ROLE_VERSION, FILE, NEW_VERSION)
    ).

reencryptResourceR(FILE) :- \+ encFile(FILE, _).
reencryptResourceR(FILE) :-
    % checks
    encFile(FILE, OLD_VERSION),
    encFile(FILE, NEW_VERSION),
    OLD_VERSION < NEW_VERSION,

    % delete old permissions
    foreach(
        fileKey(ROLE, ROLE_VERSION, FILE, OLD_VERSION),
        cac_deletePermissionRole(ROLE, ROLE_VERSION, FILE, OLD_VERSION)
    ),

    % delete old resource
    cac_deleteResourceTuple(FILE, OLD_VERSION).

% user-role management

cac_createUserRoleTuple(USER, ROLE, ROLE_VERSION) :-
    assert(roleKey(USER, ROLE, ROLE_VERSION)),
    cacTuple("+", "UserRole", "~a,␣~a,␣~a", [USER, ROLE, ROLE_VERSION]).

cac_deleteUserRoleTuple(USER, ROLE, ROLE_VERSION) :-
    retract(roleKey(USER, ROLE, ROLE_VERSION)),
    cacTuple("-", "UserRole", "~a,␣~a,␣~a", [USER, ROLE, ROLE_VERSION]).

assignUserToRoleK(USER, ROLE) :- roleKey(USER, ROLE, _).
assignUserToRoleK(USER, ROLE) :-
    userPublicKey(USER), rolePublicKey(ROLE, ROLE_VERSION), % checks
    \+ roleKey(USER, ROLE, ROLE_VERSION), % already member
    cac_createUserRoleTuple(USER, ROLE, ROLE_VERSION).

revokeUserFromRoleK(USER, ROLE) :- \+ roleKey(USER, ROLE, _).
revokeUserFromRoleK(USER, ROLE) :-
    userPublicKey(USER), rolePublicKey(ROLE, ROLE_VERSION), % checks
```

```
    roleKey(USER, ROLE, ROLE_VERSION), % is member
    cac_deleteUserRoleTuple(USER, ROLE, ROLE_VERSION).

% permission-role management

cac_createPermissionRole(ROLE, ROLE_VERSION, FILE, FILE_VERSION) :-
    assert(fileKey(ROLE, ROLE_VERSION, FILE, FILE_VERSION)),
    cacTuple("+", "PermissionRole", "~a,␣~a,␣~a,␣~a", [ROLE, ROLE_VERSION, FILE,
        ↪ FILE_VERSION]).

cac_deletePermissionRole(ROLE, ROLE_VERSION, FILE, FILE_VERSION) :-
    retract(fileKey(ROLE, ROLE_VERSION, FILE, FILE_VERSION)),
    cacTuple("-", "PermissionRole", "~a,␣~a,␣~a,␣~a", [ROLE, ROLE_VERSION, FILE,
        ↪ FILE_VERSION]).

assignPermissionToRoleK(ROLE, FILE) :- fileKey(ROLE, _, FILE, _).
assignPermissionToRoleK(ROLE, FILE) :-
    rolePublicKey(ROLE, ROLE_VERSION), encFile(FILE, _), % checks

    % assign permission to each file version
    foreach(
        encFile(FILE, FILE_VERSION),
        cac_createPermissionRole(ROLE, ROLE_VERSION, FILE, FILE_VERSION)
    ).

revokePermissionFromRoleR(ROLE, FILE) :- \+ fileKey(ROLE, _, FILE, _).
revokePermissionFromRoleR(ROLE, FILE) :-
    rolePublicKey(ROLE, _), encFile(FILE, _), % checks
    fileKey(ROLE, _, FILE, _), % permission is assigned

    foreach(
        fileKey(ROLE, ROLE_VERSION, FILE, FILE_VERSION),
        cac_deletePermissionRole(ROLE, ROLE_VERSION, FILE, FILE_VERSION)
    ).

% queries

encryptedFileK(FILE) :-
    encFile(FILE, _).
```

A.3 rbac_spec.pro

```
% dynamic predicates

:- dynamic userC/1. % user name
:- dynamic roleC/1. % role name
:- dynamic isMemberC/2. % user, role
:- dynamic fileC/1. % file name
:- dynamic allowC/2. % role, file


% utils
```

```
rbacPrint(TEXT, CONTENT) :-
    atom_concat("[TRADIT]␣", TEXT, O1),
    atom_concat(O1, "~n", O2),
    ansi_format([fg(cyan)], O2, CONTENT).


% operations

addUserC(USER) :- userC(USER).
addUserC(USER) :-
    \+ userC(USER),
    assert(userC(USER)),
    rbacPrint("Create␣user␣~a", [USER]).


addRoleC(ROLE) :- roleC(ROLE).
addRoleC(ROLE) :-
    \+ roleC(ROLE),
    assert(roleC(ROLE)),
    rbacPrint("Create␣role␣~a", [ROLE]).

addResourceC(FILE) :- fileC(FILE).
addResourceC(FILE) :-
    \+ fileC(FILE),
    assert(fileC(FILE)),
    rbacPrint("Create␣file␣~a", [FILE]).

assignUserToRoleC(USER, ROLE) :- isMemberC(USER, ROLE).
assignUserToRoleC(USER, ROLE) :-
    userC(USER), roleC(ROLE), % checks
    \+ isMemberC(USER, ROLE), % already member
    assert(isMemberC(USER, ROLE)),
    rbacPrint("Assign␣~a␣to␣~a", [USER, ROLE]).

assignPermissionToRoleC(ROLE, FILE) :- allowC(ROLE, FILE).
assignPermissionToRoleC(ROLE, FILE) :-
    roleC(ROLE), fileC(FILE), % checks
    \+ allowC(ROLE, FILE), % already has permission
    assert(allowC(ROLE, FILE)),
    rbacPrint("Assign␣~a␣access␣to␣~a", [FILE, ROLE]).

revokePermissionFromRoleC(ROLE, FILE) :- \+ allowC(ROLE, FILE).
revokePermissionFromRoleC(ROLE, FILE) :-
    roleC(ROLE), fileC(FILE), % checks
    allowC(ROLE, FILE), % permission is assigned
    retract(allowC(ROLE, FILE)),
    rbacPrint("Revoke␣~a␣access␣by␣~a", [FILE, ROLE]).

revokeUserFromRoleC(USER, ROLE) :- \+ isMemberC(USER, ROLE).
revokeUserFromRoleC(USER, ROLE) :-
    userC(USER), roleC(ROLE),
    isMemberC(USER, ROLE),
```

```
    retract(isMemberC(USER, ROLE)),
    rbacPrint("Revoke␣~a␣from␣~a", [USER, ROLE]).

deleteResourceC(FILE) :- \+ fileC(FILE).
deleteResourceC(FILE) :-
    fileC(FILE),

    % delete associated permissions
    foreach(
        allowC(ROLE, FILE),
        revokePermissionFromRoleC(ROLE, FILE)
    ),

    % delete allow
    retract(fileC(FILE)),
    rbacPrint("Delete␣resource␣~a", [FILE]).


deleteRoleC(ROLE) :- \+ roleC(ROLE).
deleteRoleC(ROLE) :-
    roleC(ROLE),

    % delete associated permissions
    foreach(
        allowC(ROLE, FILE),
        revokePermissionFromRoleC(ROLE, FILE)
    ),

    % delete members
    foreach(
        isMemberC(USER, ROLE),
        revokeUserFromRoleC(USER, ROLE)
    ),

    % delete role
    retract(roleC(ROLE)),
    rbacPrint("Revoke␣role␣~a", [ROLE]).

deleteUserC(USER) :- \+ userC(USER).
deleteUserC(USER) :-
    userC(USER),

    % delete memberships
    foreach(
        isMemberC(USER, ROLE),
        revokeUserFromRoleC(USER, ROLE)
    ),

    retract(userC(user)),
    rbacPrint("Delete␣user␣~a", [USER]).
```

A.4 predicates.pro

```
% imports

:- ensure_loaded('hybrid_spec.pro').


% predicates definition

:- dynamic enc/1. % file | role | file
:- dynamic eager/1. % file | role | file
:- dynamic cspNoEnforce/1. % file
:- dynamic collusionProne/1. % user


% combinations rules

encF(FILE) :- file(FILE), enc(FILE).
encF(FILE) :- file(FILE), role(ROLE), allow(ROLE, FILE), enc(ROLE).
encF(FILE) :- file(FILE), user(USER), enc(USER), canAccessPure(USER, FILE).

eagerF(FILE) :- file(FILE), eager(FILE).
eagerF(FILE) :- file(FILE), role(ROLE), allow(ROLE, FILE), eager(ROLE).
eagerF(FILE) :- file(FILE), user(USER), eager(USER), canAccessPure(USER, FILE).

isEncryptionNeeded(FILE) :- encF(FILE).

isReencryptionNeededOnRP(ROLE, FILE) :-
    role(ROLE), file(FILE), % checks
    enc(FILE), cspNoEnforce(FILE), % file encrypted & csp not trusted to protect
        ↪ this file
    user(USER), collusionProne(USER), canAccessPure(USER, FILE). % exists a user
        ↪ that can access and is not trusted

isEagerReencNeededOnRP(ROLE, FILE) :-
    role(ROLE), file(FILE), % checks
    isReencryptionNeededOnRP(ROLE, FILE), % need reencryption
    eagerF(FILE). % eager file (also indirectly)

isRoleKeyRotationNeededOnRUR(USER, ROLE) :-
    user(USER), role(ROLE),
    collusionProne(USER), isMember(USER, ROLE).

isReencryptionNeededOnRUR(USER, ROLE, FILE) :-
    user(USER), role(ROLE), file(FILE), % checks
    enc(FILE), cspNoEnforce(FILE), % file encrypted & csp not trusted to protect
        ↪ this file
    user(OTHER_USER), collusionProne(OTHER_USER), canAccessPure(USER, FILE). %
        ↪ exists a user that can access and is not trusted

isEagerReencNeededOnRUR(USER, ROLE, FILE) :-
    role(ROLE), file(FILE), % checks
    isReencryptionNeededOnRUR(USER, ROLE, FILE), % need reencryption
```

```
eagerF(FILE). % eager file (also indirectly)
```