© ACM, (2007). This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in Proceedings of the Proceedings of the 2nd ACM Symposium on Information, Computer and Communications Security, Singapore http://doi.acm.org/10.1145/1229285.1229308

Trust Management Services in Relational Databases

Sabrina De Capitani di Vimercati
DTI - University of Milan
26013 Crema - Italy
decapita@dti.unimi.it

Sushil Jajodia
George Mason University
Fairfax, VA 22030-4444
jajodia@gmu.edu

Stefano Paraboschi
DIGI - University of Bergamo
24044 Dalmine - Italy
parabosc@unibg.it

Pierangela Samarati
DTI - University of Milan
26013 Crema - Italy
samarati@dti.unimi.it

ABSTRACT

Trust management represents today a promising approach for supporting access control in open environments. While several approaches have been proposed for trust management and significant steps have been made in this direction, a major obstacle that still exists in the realization of the benefits of this paradigm is represented by the lack of adequate support in the DBMS.

In this paper, we present a design that can be used to implement trust management within current relational DBMSs. We propose a trust model with a SQL syntax and illustrate the main issues arising in the implementation of the model in a relational DBMS. Specific attention is paid to the efficient verification of a delegation path for certificates. This effort permits a relatively inexpensive realization of the services of an advanced trust management model within current relational DBMSs.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Relational databases*; H.2.7 [**Database Management**]: Database Administration—*Security, integrity, and protection*

General Terms

Security

Keywords

Trust, relational DBMS, credentials, access control

1. INTRODUCTION

Governments, large companies, and many other organizations are required to offer access to information contained within their information systems to a multitude of users. Users can be internal or external, and typically access the data from their clients connected to a network. The size and dynamics of the user community in this scenario set requirements that cannot be easily solved by traditional authorization and access control solutions [14]. It is often impractical to assume the creation and management of an account for each and every user on each system: it is complex both on the provider's side (each account has to be managed, privileges have to be explicitly assigned, and individual credentials have to be securely kept) and on the client side (every one experiences problems in managing the accounts and passwords she has) [17]. The case for governments and public services is particularly significant: there is a strong interest in allowing citizens to access the information that each public organization keeps on them, while guaranteeing simplicity to the user in managing accesses.

A partial solution to the user-side management of accounts is represented by Single-Sign-On (SSO); the considerable interest on this technology is a clear signal of the size of the problem. At the same time SSO is clearly insufficient, because it deals only with the sharing of authentication within a single organization.

As an alternative, trust management systems allow possibly unknown parties to establish trust based on (certified) information that each party can present to the counterpart at the time of interaction. Servers supporting trust management can then regulate access based on attributes (identities or more general properties) that clients requesting access present [2, 3, 5, 7, 17]. This is often the base on which flexible authorizations can be defined, using certified attributes as parameters in the specification of the resource or subject of an authorization.

Some of the components on which trust management is based, like the presence of a public-key infrastructure supporting the distribution and verification of certificates, already are effective components of many important services, as demonstrated by IPsec and SSL/TLS. Trust management aims at extending their use from authentication on network connections to flexible access control for structured resources.

While several approaches have been proposed for trust management and significant steps have been made in this direction [1], a major obstacle that still exists in the realization of the benefits of this paradigm is represented by the lack of adequate support in the DBMS. Indeed, while current DBMSs offer some support for certificates, they exploit them only for the specific task of user authentication. The major commercial DBMSs (specifically the last releases of

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ASIACCS'07, March 20-22, 2007, Singapore.
Copyright 2007 ACM 1-59593-574-6/07/0003 ...$5.00

Oracle Server and Microsoft SQL Server 2005) emphasize in their documentation the possibility to manage certificates and to allow users to establish database connections using SSL/TLS or other PKI-based solutions. However, the information appearing in certificates can at most be used to assign a specific user id or role to the session activated with the connection; no support is offered within the system to use certified attributes to specify flexible authorizations. PostgreSQL presents a similar profile: it is possible to combine it with OpenSSL in order to introduce a robust and flexible authentication service, but it is not possible to integrate this mechanisms with DBMS authorizations.

The availability of a trust management service within the DBMS would considerably increase the impact and applicability of this access control paradigm. As a matter of fact, DBMSs are not only the backbone of old-style business applications, but are responsible for the management of most of the information that is accessed using a Web browser or a Web service invocation. Lack of support of trust in the DBMS would require the DBMS to rely on the overlying application layer for enforcing trust management functions; this is clearly in contrast with the DBMS long term evolution, which has continuously extended the DBMS with richer functions for data access and management. Only by including trust management functions in the DBMS itself the system can give the database administrator (DBA) guarantees of full control on the access control policies. The development of a trust management component within a database management system would offer considerable advantages in terms of organization of access privileges (the authorization policy is defined together with the data) and of robustness, since there would be a strong guarantee that all accesses to the data satisfy the protection requirements, independently from the characteristics of the application environment. In general, this design would satisfy a classical access control principle that imposes to "keep the access control mechanism close to the resource".

Another demonstration of the need for an integration between trust management services and DBMSs derives from the analysis of the architecture often used in applications, where access control requirements in a scenario with many users are managed by defining a set of ad-hoc mechanisms, realized using the functionality of the procedural extension of SQL. Such an approach satisfies the requirements only partially and exhibits clear shortcomings in terms of performance, usability, and integrity. The solution presented in this paper instead designs a novel service integrated within the database architecture. The advantages are greater performance, greater usability, and increased robustness. In particular, our solution has been developed with the following requirements in mind.

- *Seamless integration with the DBMS*. DBMSs have an integrated module for controlling access to resources. It is important for trust management solutions not to subvert such a control, but complement it. Also, it is important to be able to express and represent trust information as part of the database schema, with an adequate representation within the catalog of the DBMS.
- *Abstractness and usability*. One of the main characteristics of a DBMS is the abstractness of its structure and the declarativeness of the languages accessing it; it is important that the trust management solution maintains such characteristics.
- *Expressiveness*. The trust management solution should be expressive to make it possible to specify in a flexible way different protection requirements that may need to be imposed on different data.
- *Scalability*. The trust management solution should ensure scalability with respect to the potentially high number of users, resources, and policies that may need to be managed in the context of large distributed open systems.

The contribution of the paper is twofold. First, a trust management model for DBMSs is proposed. The model identifies and adapts trust management concepts for their handling within relational databases. It is accompanied by a SQL syntax, which allows a seamless integration with existing database services and demonstrates the high-level abstraction that a database designer can use to apply these concepts. Second, the paper illustrates the basic techniques on which a mechanism efficiently enforcing the model within a modern relational engine can be built.

The remainder of the paper is organized as follows. Section 2 introduces the base elements of a trust management model for DBMSs and defines the framework within which the model should operate. Section 3 proposes SQL statements for representing the trust model. Section 4 presents an algorithm for retrieving a valid delegation chain in support of a certificate presented by the client. Section 5 discusses integration with current relational database engines. Section 6 discusses how to implement our proposal in PostgreSQL. Section 7 describes related work. Finally, Section 8 presents our concluding remarks.

2. BASE ELEMENTS OF THE MODEL

The first step for introducing a model and related language for defining and managing trust within the DBMS is the identification of the *concepts* that should be captured for providing trust management. It is also necessary to define the *framework* within which the model should operate.

2.1 Base concepts

By analyzing the needs of a trust management system, and considering the results of previous work in the area, we identify the following concepts that need to be captured.

Identity. It corresponds to a *public key*. The trust management service is based on the services of asymmetric cryptography. A basic assumption is that in offering access to information resources the database does not consider if a client really corresponds to a specified physical-world entity. Instead, the assumption is that every client interacting with the database presents an identity and, as long as the client demonstrates knowledge of the private key corresponding to the identity, the client *owns* the identity.

Authority. It represents an identity (i.e., a public key) responsible for producing and signing certificates. The need for capturing this concept comes from the fact that a party accepts certificates signed by identities that it trusts (or chains of certificates eventually leading to them) [8].

Certificate. It involves two identities: the *issuer* producing the certificate, and the *subject* receiving it. The integrity of the certificate is guaranteed by the presence of a cryptographic signature created by the issuer. A certificate then includes: the *issuer*'s public key, the *subject*'s public key, a *validity period*, and a *signature*.

We distinguish two types of certificates: *attribute certificates* and *delegation certificates*.[1]

- An *attribute certificate* binds attribute information to the certificate subject. It contains a list of pairs $\langle attribute_name, attribute_value \rangle$.
- A *delegation certificate* binds the trustworthiness of pairs of authorities together. More precisely, an authority may issue a delegation certificate asserting that it trusts another authority (or authorities having certain attributes) for issuing certain attribute certificates. Delegation has been an important topic in research on trust management. The model presented in this paper considers a form of delegation where authorities can either give unrestricted delegation to other authorities, or delegate other authorities only on specified attributes (e.g., a health agency can issue a certificate delegating physicians to certify a restricted set of properties over patients). A delegation certificate contains a list (possibly empty) of *attribute_name* terms, representing the attributes on which the subject has been delegated.

Policy. It defines the rules regulating access to resources, based on the identities owned by the client and on the information provided by the attribute and delegation certificates. A major contribution of our solution is the capability to express powerful rules that complement and nicely integrate with the native access control solution of the DBMS.

2.2 Framework assumptions

The goal of this work is to present an approach for allowing the DBMS to understand and reason about trust and regulate access to its data accordingly. We are not concerned with the low-level services (certificate formats, cryptographic protocols, and so on) required to create, exchange and verify certificates, or to delegate authority; the model is built assuming the presence and correct behaviour of traditional solutions developed for that and already available. Specifically, issues like certificate revocation, network retrieval of certificates, credential negotiation, and robust cryptography, are all assumed to be managed by an underlying implementation of the certificate management services [12, 13]. Reuse of existing implementations is particularly significant in this environment, where the large number, variety and distribution of the players, combined with the need for a consensus on the standards used, gives a strong "first-mover advantage" to existing solutions.

Our proposal represents a convenient strategy to realize the benefits of a richer trust management model without the need to update the underlying infrastructure. This is particularly significant considering that currently X.509 is the format typically used for certificates and it presents significant limitations in its design (it allows a restricted form of delegation and it focuses on the certification of real-world identities, an intrinsically hard problem that the solution is not able to solve completely). The integration with a richer model like the one proposed in this work can significantly increase the flexibility in the use of X.509 certificates.

3. SQL MODEL FOR TRUST MANAGEMENT

Our solution builds on an analysis of previous proposals, which however are not directly applicable to the DBMS scenario. Although existing approaches are expressive, they are unmanageable in practice, because most of them are based on expressive logics that cannot be put at work in real DBMSs. Therefore, since all relational DBMSs support SQL, our trust management model is based on a SQL syntax. In this way, we make trust management work in DBMSs, while enjoying a high expressiveness and flexibility thanks to the coupling with SQL and existing DBMS services.

We consider the different concepts identified in the previous section and propose a possible SQL syntax for defining them. The concept of identity corresponds to a public key and we do not need to explicitly represent it in the model. Also, delegation certificates are not explicitly represented in the model, which is focused on the specification of policies based on certified attributes. They are instead considered in the definition of certified attributes, which can be asserted by a trusted authority or by an authority delegated by it.

Each trust concept is represented by a SQL statement resulting in the construction of a corresponding schema object (see Section 5.1). Note that the introduction of SQL statements for the representation of the model is a critical success factor for a trust management solution in relational DBMSs, otherwise DBAs would be required to express trust using either external or low-level SQL constructs. The introduction of specific SQL constructs for the management of a novel security service is both compatible with the typical DBMS approach, where new constructs are always used to represent novel functions (e.g., SQL:2003/Foundation has more than 200 constructs), as well as the canonical security design approach, which imposes to keep the policy and its management clear and separated from the mechanism and implementation details.

[1] Even if some certificate formats combine the two aspects in a single certificate, it is possible to consider separately the two types of certificates.

3.1 Authority

The concept of authority defines the identities that are trusted to issue certificates. Its definition binds a name to the public key of the authority. Considering the features of current X.509 certificates, which use a predefined schema to describe the *Distinguished Name* of authorities, we also envision the introduction of a predefined set of attributes in the authority description. The syntax of the SQL statement is as follows.

```
create authority AuthorityName
 [imported by FileName]
 [public_key = AttrValue
 {,AttrName = AttrValue}]
```

This statement permits to define authority *AuthorityName* either by importing its description from an existing certificate stored in a file (clause `imported by`) or by explicitly introducing its attributes. The attributes follow the schema specified by X.509 [9] and must include the authority public key.

EXAMPLE 1. *The following* `create authority` *statement defines authority* DOH *(Department of Health) specifying its X.509 attributes, namely: common name (*`CN`*), organization (*`O`*), and country (*`C`*).*

```
create authority DOH
  public_key = '14:c9:ec....:4f:91:51',
  CN = 'Department of Health',
  O = 'Government',
  C  = 'IT'
```

A critical aspect for scalability is represented by the ability of defining an authority based on its certified attributes, instead of its identity. To this purpose, we propose the concept of *authority class* and the following SQL statement.

```
create authorityclass AuthorityClassName
 authoritative
 AuthorityClassOrName [with [no] delegation]
 {,AuthorityClassOrName [with [no] delegation]}
 [except AuthorityName {, AuthorityName}]
 ( AttrName AttrDomain [check (Condition)]
   {,AttrName AttrDomain [check (Condition)]}
   [, check (Condition)])
```

This statement allows the definition of authority class *AuthorityClassName*. The syntax is rich and reuses many features that SQL offers for the definition of tables. The description of the meaning and role of each term of the syntax appears in the next subsection, because its features are identical to those used for trust tables. The main difference in the management of authority classes compared with trust tables (see next subsection) is that trust tables represent properties obtained by certificates where the subject is the identity interacting with the database, whereas authority classes are defined based on attribute certificates where the subject is an authority. The syntax is recursive, and an authority class can be defined starting from another authority class.

EXAMPLE 2. *The following* `create authorityclass` *statement defines the* HealthGovAgency *class as any agency holding a certificate issued from the* Department of Health *(DOH) proving that the agency is specialized in healthcare and has paid the registration tax.*

```
create authorityclass HealthGovAgency
 authoritative DOH with delegation
 (regtax varchar(10) check (regtax='paid'),
  specialty varchar(15)
  check (specialty='healthcare') )
```

3.2 Certified attributes (trust tables)

Traditional approaches to trust management (e.g., [4, 10, 13, 19]) usually do not assume a (pre)declaration of the attributes that will be used in the policy, but simply use attributes in the policy. However, since DBMS engines need a structured organization of the data, the consideration of a DBMS context requires the explicit identification, in terms of names and types, of all the attributes that will be used in the trust model. In our solution, the concept of trust table responds to this need, as it represents the means by which it is possible to consider the information provided through certificates in rules and queries. The concept of trust table captures several aspects:

- the *certified attributes* that characterize the identity making a request to the database. The idea is that a client presents a set of certificates and the information extracted from them is stored in a relational table that associates this information with the session that manages the dialog with the client;
- the declaration of the *authorities trusted for asserting* those attributes;
- the declaration of whether possible *delegated authorities* are accepted (as well as a possible list of excluded authorities);
- the specification of possible *conditions on the value of attributes* that can be accepted; it allows filtering of certificates based on the values of the attributes appearing in them.

The proposed syntax for the definition of a trust table is as follows.

```
create trusttable TrustTableName
 [authoritative
 AuthorityClassOrName [with [no] delegation]
 {,AuthorityClassOrName [with [no] delegation]}]
 [except AuthorityName {, AuthorityName}]
 ( AttrName AttrDomain [check (Condition)]
   {,AttrName AttrDomain [check (Condition)]}
   [, check (Condition)])
```

The interpretation of the options is as follows. The *TrustTableName* represents a name associated with the set of attributes extracted from certificates signed by given authorities. The `authoritative` clause describes the authorities that are trusted as signers of certificates producing the

specified set of attributes. If the `authoritative` clause is missing, we assume that the DBMS uses a certificate verification service, independent from the proposed SQL trust model, which identifies the trusted certificates in autonomy. If `with` ([`no`]) `delegation` is specified, the module responsible for verifying the integrity of the certificates is (not) permitted to consider certificate chains. The `except` clause allows the specification of exceptions. It can be used by the DBA to exclude specific authorities that she does not want to consider for the specific trust table (even if they have received a delegation for the specified set of attributes). The reason can be that the authority is not trusted by the DBA or that a more specific trust table is used to manage certificates issued from that authority.

The `check` clause is a powerful mechanism that SQL offers for the description of integrity constraints. The trust table uses this mechanism to introduce constraints on the values of the certificate attributes.

EXAMPLE 3. *The following trust table* Physicians *specifies properties defining the attributes characterizing medical doctors. The* `check` *clause imposes the non nullity of the license_number.*

```
create trusttable Physicians
 authoritative HealthGovAgency with delegation
 except HealthSchoolAuth
 (code char(9),
  name varchar(25),
  license_number int
    check (license_number is not null),
  specialty varchar(20)
 )
```

3.3 Policy

A trust management policy regulates access to resources based on the attributes stated by verified certificates. Supporting a trust management policy requires then to provide the DBMSs with means to exploit certified attributes to regulate access. In this section, we show how certified attributes are used by the DBMS to regulate role activation and user identifier enabling. This provides a dynamic component for managing subjects, whose access is then regulated by classical authorizations (for roles and/or users) within the DBMS itself. We also illustrate how trust management can be used to enrich access control with context-dependent restrictions.

3.3.1 Trust policy

The trust policy represents the mechanism by which data access privileges are assigned to the clients, based on the information presents in the trust table. The trust policy allows the system to associate with a client a given role, subject to the satisfaction of a condition that can refer to the trust table attributes. The condition is expressed in the SQL syntax for query predicates, and uses the SQL dot notation to refer to trust table attributes (preceding them with the name of the trust table). The following SQL statement defines a trust policy.

```
create trustpolicy [PolicyName]
  [for Role [autoactivate] | Userid ]
 where Condition
```

Here, *Condition* is any predicate that can appear in the `where` clause of a SQL query and can refer to the trust table using its name and specifying the attributes contained in it. The *Role* is a set of privileges and it has to be a previously defined SQL role (the concept of role has been introduced in the SQL standard by SQL:1999 [6]). A role can be dynamically activated by all users authorized for it. The semantics of the statement is therefore that all users presenting certificates satisfying the condition are authorized to activate the specified role or are enabled the user id. The role activation is automatic if the `autoactivate` option is specified. If the `for` clause is omitted, the user satisfying the condition is assigned the privileges of the predefined identifier `PUBLIC`, which everybody is allowed to activate.

Since trust management systems are typically used to enforce attribute-based access control (which departs from the classical mechanism based on user identifiers) this statement would be typically used to establish role activation. The reason for considering trust policy statements referring to user identifiers is to support authentication certificates, that is, certificates stating a correspondence between a trust management identity and a user identifier internal to the database.[2]

EXAMPLE 4. *The following policy activates role* Cardiologist *for each user presenting a certificate from a health government agency (see Example 3) proving that the user is a doctor specialized in cardiology.*

```
create trustpolicy
  for Cardiologist autoactivate
  where Physicians.specialty = 'Cardiology'
```

3.3.2 Support for context-based restrictions

SQL provides some support for *content-based* access control, via the use of views, but it does not provide support for *context-based* access control, where access to data (or to views over them) may depend on properties of the user (or its session) such as time, the machine from which the user connected, and so on. Our trust management solution can be exploited to provide such a functionality. Also, coupled with the view mechanism it can provide a means to specify accesses where each user has a particular view over the data, depending on its certified properties. This technique is simple, yet effective, and powerful. The specification of the certificate attributes follows an approach similar to the one used for the definition of trust policy conditions, thus referencing certificate attributes using a dot notation. A small difference is that the trust tables are assumed to be directly available in the definition of the trust policy condition, whereas they have to be explicitly cited in the `from` clause of the query defining the view.

[2]When the client satisfies the conditions of many trust policies, she would receive a grant to activate multiple roles, and if the trust policies specify the `autoactivate` option, they will be all activated at the same time. The concurrent activation of multiple privileges does not create a critical situation, thanks to the absence of negative authorizations in SQL that permits an immediate combination of different authorization profiles based on set union.

```
SATISFIES(cert, TT)
If cert.issuer ∈ TT.Roots then return valid(cert.id)  /* the issuer is a root auth. for TT */
Roots := {auth ∈ TT.Roots | TT.Roots.del_flag=true}  /* Determine authorities of domain TT which can delegate */
If Roots = ∅ then return(false)  /* No delegation allowed */

/* Phase 1: graph construction*/
For dc ∈ Deleg_Certs do /* Construct labeled graph G=(V,E) of delegation certificates */
   V := V ∪ {dc.subject, dc.issuer}
   E := E ∪ {⟨dc.subject, dc.issuer⟩}
   λ(⟨dc.subject, dc.issuer⟩).Attributes := dc.Attributes
   λ(⟨dc.subject, dc.issuer⟩).cost := dc.cost
   λ(⟨dc.subject, dc.issuer⟩).id := dc.id

/* Phase 2: find supporting chains */
Tocheck := cert.Attributes ∩ TT.Attributes /*Initialize set of attributes for which chains have to be found */
For a ∈ Tocheck do /* Initialize Cost and Pred of each node n */
   For n ∈ V do
      Cost_n[a] := ∞ /* lowest cost of path for a ending in node n */
      Pred_n[a] := null /* predecessor of n in such a path */
MAKENULL(Queue) /* Create priority queue of edges with information of attributes and cost of path ending with them */
For e ∈ {⟨n1, n2⟩ ∈ E | n1=cert.issuer} do  /* Add to Queue all edges outgoing from cert.issuer */
   p_attrs := λ(e).Attributes
   p_cost := λ(e).cost
   INSERT([n1,n2,p_attrs,p_cost], Queue)
MAKENULL(Verify_Queue) /* Create priority queue describing the root nodes of the solution chains */
While Tocheck ≠ ∅ ∧ Queue ≠ ∅  /* Extract from Queue the element with minimum path cost */
   [from,to,p_attrs,p_cost] := EXTRACT_MIN_p_cost(Queue)
   A := ∅ /* Keeps track of attributes verified along the chain */
   For a ∈ p_attrs ∩ Tocheck  /* For each attribute still to be verified that belongs to the extracted edge ⟨from,to⟩,
      If Cost_to[a] > p_cost then  if the cost of the path expressed by the element is smaller than the one of current path,
         Cost_to[a] := p_cost        update solution to include the extracted edge */
         Pred_to[a] := from
         A := A ∪ a
      If A ≠ ∅ then
         If to ∈ Roots then
            Tocheck := Tocheck − A
            INSERT([to,A,p_cost], Verify_Queue)
         else For e ∈ {⟨n1, n2⟩ ∈ E | n1=to} do
               p_attrs := λ(e).Attributes ∩ A ∩ Tocheck
               If p_attrs ≠ ∅ then
                  p_cost := p_cost + λ(e).cost
                  INSERT([to,n2,p_attrs,p_cost], Queue)
If Tocheck ≠ ∅ then return(false) /* No chain covering all attributes in Tocheck is found */

/* Phase 3: verify chains */
Tocheck := cert.Attributes ∩ TT.Attributes /* Initialize attributes to check for verification */
While Tocheck ≠ ∅ ∧ Verify_Queue ≠ ∅
   A := Tocheck /* Initialize attributes covered by a verified path */
   [to,p_attrs,p_cost] := EXTRACT_MAX_p_cost(Verify_Queue)
   If p_attrs∩ Tocheck ≠ ∅ then
      Let a be any attribute in p_attrs ∩ Tocheck
      Repeat  /* go back in the chain for a from to to cert.issuer */
         from := Pred_to[a]
         If valid(λ(⟨from,to⟩).id) then
            A := A ∩ λ(⟨from,to⟩).Attributes
            to := from
         else /* certificate id is not valid */
               to := cert.issuer /* set condition for termination */
               A := ∅ /* no attribute verified along the chain */
      Until to = cert.issuer
      Tocheck := Tocheck − A /* remove verified attributes from Tocheck */
If Tocheck ≠ ∅ then return(false) /* Not all attributes verified */
return(true)
```

Figure 1: Chain verification algorithm

EXAMPLE 5. *The following view grants each physician access to the data of her patients (having the physician recorded as their primary doctor).*

```
create view PatientView as
  select Patients.*
  from Patients, Physicians
  where Physicians.code = Patients.doctor_code
```

4. DELEGATION CHAIN VERIFICATION

One of the most critical components of every trust management proposal is the design of the algorithm responsible for the identification of the delegation chains. Many models have been proposed for the management of this important step, both in centralized and distributed contexts, considering several alternative models for the representation of delegation (e.g., [12]). Unfortunately, all the models that offer the representation of a flexible delegation mechanism use algorithms for the verification of delegation chains that are extremely difficult to apply in the database context, due to their computational cost. It is indeed hard to convince a database implementor to integrate within the relational engine a logic model checker, for the verification of certificates presented by a client. We instead show that our model permits the application of an algorithm that is able to identify, with an acceptable computational effort, if a set of certificates produces a delegation path for a set of attributes appearing in a certificate. Also, with an approach similar to the one used by DBMS query optimizers, the algorithm can apply a cost model that estimates the computational effort required for the verification of delegation chains. In this way, the algorithm is also able to identify, for each attribute in the certificate, the path in the delegation graph that requires the minimal cost for its verification. The algorithm is therefore a crucial component of the proposed approach since it shows that the complexity of our delegation mechanism remains manageable by a DBMS.

We assume that the system has knowledge of all the delegation certificates *Deleg_Certs* needed for the verification. Each delegation can be either unrestricted or applicable only to a subset of attributes. We also assume that each delegation certificate is associated with a cost representing an estimate of the computational effort required for the certificate verification. The reason for capturing cost information is that cryptographic functions are computationally expensive and it is therefore important to minimize their use. The cost information can be used to model the lower cost of using certificates cached as valid in prior verification as well as the different higher costs of retrieving certificates from remote directories. Finally, we assume that the cryptographic check over certificates is carried out by invoking an external function, called **valid**.

4.1 Algorithm

We consider a client that presents a certificate *cert*. For each trust table TT with a structure that is compatible with *cert*, the algorithm illustrated in Figure 1 determines whether *cert* satisfies TT either directly or via a delegation chain, returning *true* or *false* accordingly.

The algorithm starts by checking if the issuer of the certificate (*cert.issuer*) belongs to the set of root authorities of TT. If this is the case, the algorithm terminates right away returning the outcome of function **valid** over the certificate. Otherwise (the issuer is not a root authority) the algorithm proceeds to determine whether the certificate is supported by a chain of delegation certificates in *Deleg_Certs*. If delegation is not allowed (the set of root authorities for which delegation is allowed is empty), the algorithm terminates returning *false*; otherwise, it proceeds to retrieve a possible delegation chain.

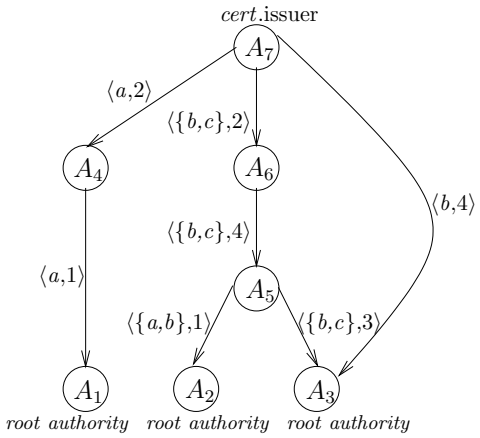

Figure 2: An example of delegation graph

The process of determining a delegation chain can be seen as composed of three parts: first the algorithm *defines a graph* representing all delegation certificates in *Deleg_Certs*, then it *finds support chains* for the attributes involved, and finally it *checks the validity of the certificates in the chains*.

Phase 1: graph construction. The *delegation graph* G representing all the delegation certificates *Deleg_Certs* has a node for every issuer and every subject of certificates, and an edge for each certificate *dc*, going from the subject of *dc* to its issuer. The edge is labeled (via a function λ) with a triple reporting the (*Attributes*) in the certificate, the cost for checking its validity, and its identifier. As an example, suppose that certificate *cert* contains three attributes, namely, a, b, c, and that the issuer of this certificate is authority A_7. Figure 2 illustrates an example of delegation graph with three root authorities (A_1, A_2, and A_3) and seven delegate certificates (one for each edge in the graph) involving authorities A_4, A_5, A_6, and A_7. Each edge is associated with a label that specifies the delegated attributes and the cost, respectively.[3]

Phase 2: find supporting chains. Finding a support chain for an attribute a means finding a path in the graph starting from the issuer of *cert* and ending in one of the root authorities for TT (set *Roots*) such that the set of attributes of all the edges in the path includes a. The cost of a support chain is defined as the sum of the costs of the edges belonging to the chain; the shortest chain for an attribute is the chain with minimum cost that exists for it.

[3]For the sake of simplicity, we omit the identifiers associated with the delegation certificates corresponding to the edges of the graph.

The process for finding supporting chains is performed via a Dijkstra-like process, with a **while** cycle that iterates until either a chain has been retrieved for all attributes (*Tocheck* is empty) or there are no more edges to examine (*Queue* is empty). When a path (chain) ending in a root authority is found, *Verify_Queue* is updated accordingly. *Verify_Queue* keeps track of the root authorities reached by a support chain for some attributes. At the end of the **while** cycle, if *Tocheck* is not empty, then no chain has been found for some attributes and the algorithm terminates returning *false*. Otherwise, the algorithm proceeds verifying the chains retrieved. The algorithm makes use of classical Dijkstra-like structures to maintain information on the paths being found.

Phase 3: verify chains. The *chain verification* process starts by initializing variable *Tocheck* to the set of attributes to be verified. Then, it processes chains to be verified (*Verify_Queue*) in decreasing order of cost. For each chain, the certificates corresponding to the edges are checked via a call to function **valid** and if the chain is correctly verified, the attributes certified by it removed from those to be checked. The process (controlled by the **while** cycle) continues until there are no more attributes to be verified (*Tocheck* is empty) or there are no more chains to process (*Verify_Queue* is empty). In this latter case, not all attributes have been verified and the algorithm terminates returning *false*; otherwise it returns *true*.

As an example, for the delegation graph in Figure 2, our algorithm determines two verification chains:$\langle A_7, A_4, A_1\rangle$ for attribute a and $\langle A_7, A_6, A_5, A_3\rangle$ for attributes $\{b, c\}$.

Note that the reason why the elements in *Verify_Queue* are processed in decreasing order of cost is to minimize the number of chains to be verified. For instance, with respect to the delegation graph in Figure 2, our algorithm verifies both attribute b and attribute c with path $\langle A_7, A_6, A_5, A_3\rangle$ with cost 9, instead of considering two paths: $\langle A_7, A_3\rangle$ for attribute b with cost 4 and $\langle A_7, A_6, A_5, A_3\rangle$ for attribute c with cost 9.

In summary, the algorithm is able to efficiently identify (with a computational effort that grows almost linearly with the number of edges in the graph multiplied by the number of attribute labels) the presence of a delegation chain supporting the certificate in the graph. We observe that the algorithm has a tuple oriented structure: at each iteration, in the search for the minimum cost path, a single edge is considered. This is the reason while we have chosen not to use SQL instructions.

5. INTEGRATION WITHIN A DBMS

There are few principles that have to be followed in the integration within a current DBMS of the trust management services we described. First, the implementation in real systems of these services can be successful only if it is focused on a few components, otherwise, it could introduce many side effects, in terms of functionality or performance, which would create problems in current database applications. Second, the implementation has to require a modest coding effort; apart from the increase in costs that can make this extension too expensive in the eye of the DBMS producer, it would be considerably more difficult to have a guarantee on the robustness in terms of security. Third, there is a need for a good integration with current SQL constructs, in order to minimize the effort required to the database designer in the modelling of application requirements for access control. Our proposal has been designed taking into account all these principles ensuring seamless integration with existing DBMSs.

One key aspect deserving mention in the implementation of our solution concerns role activation. The SQL standard, since SQL:1999, offers supports for roles; however the SQL model with roles binds roles to user identities and therefore must be adapted to support role activation (i.e., granting of privileges) on the basis of certified attributes. We solve this problem by using sessions (instead of user ids) as target of grant statements triggered by the trust policy to enable role activation (see Section 3.3.1).

5.1 Translation of the SQL Constructs

The SQL statements that we presented for the definition of authorities, trust tables, and trust policies facilitate the integration of these aspects within relational databases. We describe in more details how each SQL trust statement can be translated into traditional SQL structures. The table in the appendix summarizes the discussion. The goal is not to suggest a strategy for DBAs to represent trust directly in the DBMS (a higher level representation has to be used to model an access policy), but to demonstrate the compatibility of our proposal with the internal architecture of current DBMSs.

The description of authorities within the schema requires to introduce a table in the database catalog, which we call `Authority`, which has to present two non-null attributes, *name* and *public_key*, storing the name and the public key of each authority, respectively. The specification of a `create authority` statement therefore corresponds to the insertion of a tuple in the `Authority` table, where the authority name and the public key are those indicated in the statement.

Statements `create authorityclass` and `create trusttable` produce a more extensive impact on the catalog. Each authority class produces a table corresponding to the authority class description. Analogously, each trust table produces a table to contain the attribute values obtained from client certificates. Depending on the feature set of the DBMS, these tables can be managed as *global temporary tables*.[4] Global temporary tables are described in the SQL standard [6] and represent tables that are part of the database schema, but that differ from base tables because their content cannot be shared between different sessions; a session can then use a global temporary table to store information that is then needed within the same session and that must not be shared with other sessions. The advantage of global temporary tables is typically greater performance, due to the fact that locks, and in general concurrency control mechanisms, are not used

[4]For database servers that do not support global temporary tables, it is necessary to simulate their services using in the schema of the trust table an additional attribute, the session identifier *session_id*. The goal of this attribute is to associate each certificate with the session on which it has been presented.

to access the table; an additional benefit is that a rigid separation of the information pertaining to distinct sessions is automatically supported, with automatic removal of the information at the closure of the session. Both the `create authorityclass` statement and the `create trusttable` statement correspond to a `create global temporary table` statement, where the name of the global temporary table is the name of the authority class and the name of the trust table, respectively, and the schema is the list of attributes and constraints defined by the `create authorityclass` and `create trusttable` statements.

The management of the optional clauses on trusted authorities, delegation, and exceptions forces the introduction of additional tables, which have an important role in the evaluation, database-side, of the validity of the certificates presented by the client. More precisely, tables `AuthorityClass` and `AuthorityTT` are needed to store the list of authorities trusted (clause `authoritative`) for asserting that a given party has given attributes and for producing certificates that can contain the list of attributes defined in the trust table, respectively. These tables have three attributes, *name*, *authority*, and *delegation_flag*, storing the name of the authority class or trust table, the name of the authority, and whether chains of delegated authorities with *authority* as a starting point are acceptable (*delegation_flag* is set to *true*) or not (*delegation_flag* is set to *false*). Analogously, tables `NotAuthorityClass` and `NotAuthorityTT` are needed to store the list of authorities that are not trusted (clause `except`) in the specific authority class and trust table, respectively. These tables have two attributes, *name* and *authority*, storing the name of the authority class or trust table and the name of the authority, respectively.

Each `create authorityclass` statement and each `create trusttable` statement are therefore translated into one or more `insert` statements that have to be executed on these tables: one `insert` statement on tables `AuthorityClass` and `AuthorityTT` for each authority specified in the `authoritative` clause and one `insert` statement on tables `NotAuthorityClass` and `NotAuthorityTT` for each authority specified in the `except` clause, as reported in the appendix.

The `create trustpolicy` statement is represented as a trigger whose condition is the condition specified in the policy. The trigger event reacts to the insertion of a tuple in the trust table referred in the condition. The trigger action grants the session a privilege to activate the role (and activates it automatically if the `autoactivate` clause is specified).

6. PROTOTYPE IMPLEMENTATION

To demonstrate the realizability within current DBMSs of the proposed trust management model, a related project is currently producing an implementation in the PostgreSQL system. PostgreSQL is a well-known open-source DBMS, already chosen as a testbed for the implementation of novel database technologies by many other research initiatives.

The first design issue was the choice of the component responsible for the cryptographic functions used for the verification of certificates. The OpenSSL system has been selected, which is one of the most used implementations of the SSL protocol. The PostgreSQL distribution already offers the possibility to integrate the OpenSSL system, but it only uses it for the realization of a dialog between a client and the PostgreSQL server using an SSL connection. Since PostgreSQL does not support global temporary tables[5], trust tables have been managed by regular tables that explicitly represent the session identifier together with the attributes obtained by certificates. The model presented in the paper restricts the use of trust table attributes to two situations: the definition of trust policies and the definition of views. The management of `create trustpolicy` statements is based on the use of triggers. The management of the `create view` statement follows the same approach used in PostgreSQL for the management of views, which uses *rules*. PostgreSQL rules are *rewriting* rules that capture references to views and replace each occurrence of the view in a SQL statement with the corresponding query. PostgreSQL rules can be immediately adapted to the representation of views using certificate attributes. Starting from a view definition, a rule is produced that replaces the view occurrence with the query that defined the view, extended with the predicate that restricts the evaluation of certificate attributes only to the tuples of the trust table that refer to the session in which the SQL command is executed.

EXAMPLE 6. *Consider the following view, which uses an attribute of trust table* Physicians.

```
create view PatientView as
  select Patients.*
  from Patients, Physicians
  where Physicians.code = Patients.doctor_code
```

The system produces the following PostgreSQL rule.

```
create rule PatietViewsSelect as
 on select to PatientView
 do
   select Patients.*
   from Patients, Physicians
   where Physicians.code = Patients.doctor_code
     and Physicians.SessionId = session_identifier()
```

7. RELATED WORK

Trust management has received considerable interest in the research community. Much of this research, however, focuses on the formalization and analysis of the expressive power of authorization systems, without addressing the practical details and strategies for integrating and implementing trust management in the DBMS. In contrast, our approach can be easily incorporated into a DBMS, increasing the expressiveness of the access control model in terms of protection requirements that can be supported.

The term *trust management* was first introduced in [3] by Blaze, Feigenbaum, and Lacy, where the authors presented

[5]Note that temporary tables in PostgreSQL are only *local*, that is, they have to be created with a `create table` statement within a session, and are dropped at the end of the session.

a trust management system, called PolicyMaker, where authorizations are associated with keys rather than with users' identities. An application sends to the execution environment a request for actions, a policy, and a set of credentials and the execution environment returns an answer to the question of whether the credentials prove that the request complies with the policy. The KeyNote system [2], the successor of PolicyMaker, refines the idea of PolicyMaker into a more practical system. REFEREE (Rule-controlled Environment For Evaluation of Rules, and Everything Else) [5] is a trust management system for Web applications. Like PolicyMaker, it supports full programmability of assertions (i.e., policies and credentials). While these approaches provide an interesting framework for reasoning about trust between unknown parties, the flexibility of the delegation mechanism is difficult to integrate with a DBMS, and suggests an access control model which strictly merges authentication and policy evaluation, in a way that is difficult to integrate with database access control.

Other approaches use digital certificates to establish properties of their holder, delegation and revocation of credentials, and evaluation of credential chains [8, 12, 15]. In [12] the authors present an algorithm for discovering credential chains expressed using a role-based trust management language, called RT_0. Wang et. al [15] propose a framework that models an attribute-based access control system using logic programming with set constraints of a computable set theory. The Simple Public Key Infrastructure (SPKI) 2.0 [8] is a digital-certificate schema where SPKI certificates can be seen as tuples that can bind names to keys, names to privileges, and privileges to keys. All these proposals permit a powerful representation of privileges based on the information presented in certificates, but none of them is able to offer guarantees on the computational effort required for certificate verification.

X.509 [9] focused on the definition of a binding between keys and names, and X.509 v.3 certificates extended this binding to general attributes. X.509 is currently the most successful solution, but its delegation model and certificate structure are quite rigid. The model proposed in this paper permits to exploit the existing X.509 infrastructure for the realization of flexible policies.

Other complementary approaches (e.g., [4, 10, 11, 13, 16, 17, 18, 19]) propose solutions for specifying and enforcing access control policies based on certified attributes. These proposals focus in particular on the assumption that parties may be unknown a-priori and therefore propose approaches and strategies for parties to communicate to each other their policies as well as releasing their certificates, possibly undertaking a multi-step negotiation process. In this paper, we have assumed the client to present all certificates needed for an access at the request time. This does not rule out compatibility with the different proposals supporting trust negotiation. Our assumption is essentially that the negotiation is completed before the trust management service starts the verification process.

8. CONCLUSIONS

Even if trust management mechanisms have been proposed a few years ago, their adoption has until now been limited. This is mostly due to the obstacles arising in the implementation of a working infrastructure for the management and exchange of certificates, as testified by the time and effort spent for the realization of the current infrastructure based on X.509 certificates. Part of the responsibility can also be assigned to the absence of a clear strategy for the integration of these services with database servers, which today manage most of the information for which it is important to define a rich and flexible access control model. Many trust management proposals present mechanisms that are quite powerful, but that are difficult to integrate with current DBMSs. Indeed, most previous proposals had as main aim the increase in expressive power, in order to represent evermore complex and sophisticated scenarios.

Our approach has set as the primary requirement its compatibility with consolidated DBMS practices. The strict integration with the full set of current DBMS services provides to our model a considerable expressive power. Exploiting the integration of the policy with the active components (triggers, procedures, constraints, roles, transactions) and rich storage services offered by SQL, we were able to adequately represent all the scenarios that we analyzed. The solution presented in this paper is designed to be immediately implemented by DBMS producers and used by DBAs. We believe that our solution represents a good trade off between functionality and applicability. Indeed, while not enjoying the complete functionality of trust management approaches, it captures their essential features and functions thus enabling the use of trust management concepts in practice.

9. ACKNOWLEDGMENTS

This work was supported in part by the European Union within the PRIME Project in the FP6/IST Programme under contract IST-2002-507591.

10. REFERENCES

[1] C. Ardagna, E. Damiani, S. De Capitani di Vimercati, S. Foresti, and P. Samarati. Trust management. In *Security, Privacy and Trust in Modern Data Management*. Springer, 2006.

[2] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. Keromytis. *The KeyNote Trust Management System (Version 2)*, internet rfc 2704 edition, 1999.

[3] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *Proc. of the 17th Symposium on Security and Privacy*, Oakland, California, USA, May 1996.

[4] P. Bonatti and P. Samarati. A unified framework for regulating access and information release on the web. *Journal of Computer Security*, 10(3):241–272, 2002.

[5] Y. Chu, J. Feigenbaum, B. LaMacchia, P. Resnick, and M. Strauss. REFEREE: Trust management for web applications. *The World Wide Web Journal*, 2(3):127–139, 1997.

[6] Database language SQL – part 2: Foundation (SQL/foundation). ISO International Standard, ISO/IEC 9075:1999, 1999.

[7] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, and P. Samarati. Access control policies and languages in open environments. In *Security in Decentralized Data Management*. Springer, 2006.

[8] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. SPKI certificate theory. RFC2693, September 1999.

[9] R. Housley, W. Ford, W. Polk, and D. Solo. *Internet X.509 Public Key Infrastructure Certificate and CRL Profile*, rfc 2459 edition, January 1999. http://www.ietf.org/rfc/rfc2459.txt.

[10] K. Irwin and T. Yu. Preventing attribute information leakage in automated trust negotiation. In *Proc. of the 12th ACM CCS*, Alexandria, VA, USA, Nov. 2005.

[11] N. Li, J. Mitchell, and W. Winsborough. Beyond proof-of-compliance: Security analysis in trust management. *Journal of the ACM*, 52(3):474–514, May 2005.

[12] N. Li, W. Winsborough, and J. Mitchell. Distributed credential chain discovery in trust management. *Journal of Computer Security*, 11(1):35–86, February 2003.

[13] J. Ni, N. Li, and W. Winsborough. Automated trust negotiation using cryptographic credentials. In *Proc. of the 12th ACM CCS*, Alexandria, VA, USA, Nov. 2005.

[14] P. Samarati and S. De Capitani di Vimercati. Access control: Policies, models, and mechanisms. In R. Focardi and R. Gorrieri, editors, *Foundations of Security Analysis and Design*, LNCS 2171. Springer-Verlag, 2001.

[15] L. Wang, D. Wijesekera, and S. Jajodia. A logic-based framework for attribute based access control. In *Proc. of the 2004 ACM Workshop on Formal Methods in Security Engineering*, Washington DC, USA, October 2004.

[16] J. Warner, V. Atluri, and R. Mukkamala. An attribute graph based approach to map local access control policies to credential based access control policies. In *Proc. of the International Conference on Information Systems Security (ICISS 2005)*, Kolkata, India, December 2005.

[17] M. Winslett, N. Ching, V. Jones, and I. Slepchin. Using digital credentials on the World-Wide Web. *Journal of Computer Security*, 1997.

[18] T. Yu and M. Winslett. A unified scheme for resource protection in automated trust negotiation. In *Proc. of the IEEE Symposium on Security and Privacy*, Oakland, CA, USA, May 2003.

[19] T. Yu, M. Winslett, and K. Seamons. Supporting structured credentials and sensitive policies through interoperable strategies for automated trust negotiation. *ACM Transactions on Information and System Security*, 6(1):1–42, February 2003.

APPENDIX
A. TRANSLATION OF SQL TRUST STATEMENTS

SQL trust statement	SQL statement
create authority *AuthorityName* public_key = *PublicKeyValue*	insert into Authority values(*AuthorityName*,*PublicKeyValue*)

create authorityclass *AuthorityClassName*
[authoritative
AuthorityClassOrName [with [no] delegation]
{,*AuthorityClassOrName* [with [no] delegation]}]
[except *AuthorityName*{, *AuthorityName*}]
(*AttrName AttrDomain* [check (*Condition*)]
{,*AttrName AttrDomain* [check (*Condition*)]}
[, check (*Condition*)])
	create global temporary table *AuthorityClassName*
	(*AttrName AttrDomain* [check (*Condition*)]
	{,*AttrName AttrDomain* [check (*Condition*)]}
	[, check (*Condition*)])
	
	For *AuthorityClassOrName* in authoritative and with delegation **do**
	insert into AuthorityClass
	values(*AuthorityClassName*,*AuthorityClassOrName*,*true*)
	
	For *AuthorityClassOrName* in authoritative and with no delegation **do**
	insert into AuthorityClass
	values(*AuthorityClassName*,*AuthorityClassOrName*,*false*)
	
	For *AuthorityName* in except **do**
	insert into NotAuthorityClass
	values(*AuthorityClassName*,*AuthorityName*)

create trusttable *TrustTableName*
[authoritative
AuthorityClassOrName [with [no] delegation]
{,*AuthorityClassOrName* [with [no] delegation]}]
[except *AuthorityName*{, *AuthorityName*}]
(*AttrName AttrDomain* [check (*Condition*)]
{,*AttrName AttrDomain* [check (*Condition*)]}
[, check (*Condition*)])
	create global temporary table *TrustTableName*
	(*AttrName AttrDomain* [check (*Condition*)]
	{,*AttrName AttrDomain* [check (*Condition*)]}
	[, check (*Condition*)])
	
	For *AuthorityClassOrName* in authoritative and with delegation **do**
	insert into AuthorityTT
	values(*TrustTableName*,*AuthorityClassOrName*,*true*)
	
	For *AuthorityClassOrName* in authoritative and with no delegation **do**
	insert into AuthorityTT
	values(*TrustTableName*,*AuthorityClassOrName*,*false*)
	
	For *AuthorityName* in except **do**
	insert into NotAuthorityTT
	values(*TrustTableName*,*AuthorityName*)

create trustpolicy *PolicyName* for *Role* autoactivate as *Condition* /* *Condition* on trust table *TrustTableName* */	create trigger *PolicyName* after insert on *TrustTableName* for each row when *Condition* grant *Role* to *session id* set role *Role*