Integrating Trust Management and Access Control in Data-Intensive Web Applications

SABRINA DE CAPITANI DI VIMERCATI and SARA FORESTI,
DTI, Università degli Studi di Milano
SUSHIL JAJODIA, CSIS, George Mason University
STEFANO PARABOSCHI and GIUSEPPE PSAILA, DIIMM, Università degli Studi di Bergamo
PIERANGELA SAMARATI, DTI, Università degli Studi di Milano

The widespread diffusion of Web-based services provided by public and private organizations emphasizes the need for a flexible solution for protecting the information accessible through Web applications. A promising approach is represented by credential-based access control and trust management. However, although much research has been done and several proposals exist, a clear obstacle to the realization of their benefits in data-intensive Web applications is represented by the lack of adequate support in the DBMSs. As a matter of fact, DBMSs are often responsible for the management of most of the information that is accessed using a Web browser or a Web service invocation.

In this article, we aim at eliminating this gap, and present an approach integrating trust management with the access control of the DBMS. We propose a trust model with a SQL syntax and illustrate an algorithm for the efficient verification of a delegation path for certificates. Our solution nicely complements current trust management proposals allowing the efficient realization of the services of an advanced trust management model within current relational DBMSs. An important benefit of our approach lies in its potential for a robust end-to-end design of security for personal data in Web scenario, where vulnerabilities of Web applications cannot be used to violate the protection of the data residing on the database server. We also illustrate the implementation of our approach within an open-source DBMS discussing design choices and performance impact.

Categories and Subject Descriptors: H.2.4 [**Database Management**]: Systems—*Relational databases*; H.2.7 [**Database Management**]: Database Administration—*Security, integrity, and protection*; H.3.5 [**Information Storage and Retrieval**]: Online Information Services—*Web-based services*; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection

General Terms: Management, Security

Additional Key Words and Phrases: Trust management, relational databases, access control

---

This paper extends the previous work by the authors which appeared under the title "Trust management services in relational databases," in *Proceedings of the 2nd ACM Symposium on Information, Computer and Communications Security* (ASIACCS'07).
This work was partially supported by the EC within the 7FP under grant agreement 257129 (PoSecCo); by the Italian Ministry of Research project PEPPER (2008SY2PH4) and by the Universita degli Studi di Milano project PREVIOUS. The work of Sushil Jajodia was partially supported by the National Science Foundation under grants CT-20013A and CCF-1037987; by the Air Force Office of Scientific Research under grant FA9550-09-1-0421; and by the Army Research Office under DURIP grant W911NF-11-1-0340.
Authors' addresses: S. De Capitani di Vimercati, S. Foresti, and P. Samarati, DTI, Univ. degli Studi di Milano, 26013 Crema, Italy; S. Jajodia, CSIS, George Mason Univ., Fairfax, VA 22030-4444, USA. S. Paraboschi and G. Psaila, DIIMM, Univ. degli Studi di Bergamo, 24044 Dalmine, Italy.

DOI 10.1145/2180861.2180863 http://doi.acm.org/10.1145/2180861.2180863

ACM Reference Format:
De Capitani di Vimercati, S., Foresti, S., Jajodia, S., Paraboschi, S., Psaila, G., and Samarati, P. 2012. Integrating trust management and access control in data-intensive Web applications. ACM Trans. Web 6, 2, Article 6 (May 2012), 43 pages.
DOI = 10.1145/2180861.2180863 http://doi.acm.org/10.1145/2180861.2180863

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 can access the data from their clients connected to a network. The size and dynamics of the user community in this scenario impose requirements that cannot be easily solved by traditional authorization and access control solutions. In particular, 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 user's side (every one experiences problems in managing their accounts and passwords) [Winslett et al. 1997]. The case for governments and public services is particularly significant: there is a strong interest in allowing citizens to access in a simple way the information on them maintained by public organizations, while guaranteeing protection of this information from unauthorized accesses.

Several proposals have recently tried to provide an answer to the problem above, at different levels. Single Sign-On solutions represent an immediate approach for the management of the client-side issues, but fall short in satisfying the requirements of wide-scale open systems, as they deal only with the sharing of authentication within a single organization. More sophisticated solutions have considered credential/attribute-based access control and "trust management," a term with many facets that we characterize here as a label identifying access control systems where the access policy refers to information that is provided by user certificates (e.g., Bonatti and Samarati [2002]; Irwin and Yu [2005]; Li et al. [2005a, 2005b]; Wang et al. [2004]; Warner et al. [2005]; Winslett et al. [1997]; Yu and Winslett [2003]). Recent evolutions of these approaches have considered limiting the disclosure of the personal information provided by the user to the server, thanks to a negotiation phase or to the services of an identity provider [Lee et al. 2008; Lee and Winslett 2008b; Ryutov et al. 2005; Winsborough and Li 2006; Winslett et al. 2002; Yu et al. 2000, 2001, 2003; Yu and Winslett 2003]. Most of these proposals have generated considerable interest in the research community, but until now have found limited application in Web environments. One of the obstacles to the realization of the benefits of trust management is represented by the lack of adequate support in the DBMS.

The availability of a trust management service within the DBMS would considerably increase the impact and applicability of credential/attribute-based access control. 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. On the other hand, research on database access control in the academic and industrial communities has recently shown a significant interest in the definition of *fine-granularity access control* (FGAC) (e.g., Agrawal et al. [2005]; Kabra et al. [2006]; Murthy and Sedlar [2007]). Typically, fine-granularity access control systems define authorizations that depend on information extracted from the user profile. Some proposals assume that this data is stored within the DBMS, while other proposals assume that it is provided, in an unspecified way, at session startup. The approach presented in this article supports flexible authorizations, and goes a step further, thanks to the consideration of credentials which permit a clear

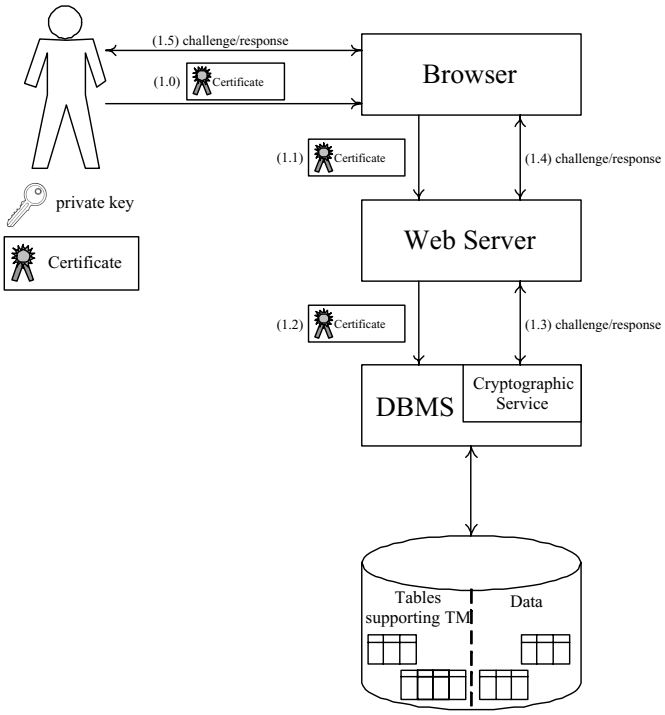


Fig. 1. Architecture of the system.

management of user attributes, compliant with the architecture of future Internet applications.

1.1. Scenario

The architecture we consider for the use of the trust management services assumes a typical three-layer structure, with a browser working as a front-end to a Web server accessing data stored on a DBMS. Current three-layer architectures assume that user privileges on accessing the data are described in the Web application, which has continuous privileged access to the DBMS, with full control on all the data accessible by the user. If the client authenticates using certificates or the authentication process exploits trust management functions, the functionalities supporting certificates and/or trust management are realized within the Web server.

Figure 1 illustrates a high-level architecture of systems resulting from the integration of a DBMS with the trust management functionality. This architecture is different from the typical three-layer architecture characterizing traditional systems described above. In this architecture, the interaction among the components follows these steps. The client makes available a set of certificates to the browser (step 1.0). The browser forwards these certificates to the Web server (step 1.1) which in turn forwards them to the DBMS (step 1.2). The DBMS verifies, for each certificate in the set, that the owner of the certificate corresponds to the access requestor, executing a challenge/response verification (steps 1.3, 1.4, and 1.5). The challenge/response phase can rely on an implementation of the Needham–Schroeder–Lowe protocol [Lowe 1996; Needham and Schroeder 1978], which offers mutual authentication guarantees to each party. The cryptographic module within the DBMS sends a random value as a challenge; the client encrypts the challenge with her private key; and the cryptographic module

verifies that encrypting the response with the public key in the certificate produces the challenge back. All the exchanges between the client and the DBMS are routed through the Web server and the browser. After the challenge/response verification, the cryptographic service will verify that the signing authority of each certificate is trusted. This check is performed by analyzing the list of trusted certification authorities, which is a part of the system configuration. Finally, the set of certificates presented by the client and verified by the DBMS is used to establish the client's access privileges to the tables in the database.

The integration between trust management services and the DBMS allows the realization of an *end-to-end* design in the management of access control, providing a mechanism that exhibits a computational complexity compatible with a performance-conscious environment as DBMSs and Web systems. Such a design enjoys the following benefits.

—It is more robust from a security point of view because it is not vulnerable to errors in the application. For instance, SQL injection is today a common and critical vulnerability in many Web applications. Systems using a trust management solution within the DBMSs would be immediately protected against this attack because the portion of data accessible to each user would only depend on the credentials presented by the user, and manipulation of the Web application could not be used to exploit it. In general, this design would satisfy a classical access control principle that imposes to "keep the access control mechanism close to the resource" [Saltzer and Schroeder 1975].
—It makes it possible to define once, for all applications and access paths, the set of privileges of every user. For environments where the Web system consists of several Web applications, the advantage of having a strict integration between the access policy and the data facilitates their management, providing an immediate guarantee that the policy will be satisfied by every access.
—It facilitates the integration of all the data collections that are used by different applications. Today, most systems that support distinct applications distribute the information in different databases to introduce a protection layer against possible misuses of data. The availability of an integrated trust management service would facilitate the creation of a single database, with all the advantages in terms of reduced complexity of the system, better consistency checks, and easier support for all the applications that have to access data belonging to separate domains.

We therefore propose a novel service integrated within the database architecture, which has been developed with the following requirements in mind.

—*Seamless integration with the DBMS*. A trust management solution should not subvert the control of the module of the DBMS that monitors access to resources, but should complement it. Also, a trust management solution should 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*. A trust management solution should maintain both the abstractness of the structure of the DBMS and the declarativeness of the languages accessing it.
—*Expressiveness*. A 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*. A 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 proposed solution is based on the assumption that a client presents all certificates needed for an access at request time. This does not rule out compatibility with the different proposals supporting trust negotiation (e.g., Lee and Winslett [2006]; Lee et al. [2008]; Winslett et al. [2002]; Yu et al. [2000]; Yu and Winslett [2003]; Yu et al. [2001, 2003]). Our assumption is essentially that the identification of the certificates that the client should provide is completed before our trust management service starts the verification process.

1.2. Contribution and Structure of the Paper

The contribution of this article is threefold. First, we provide an expressive and flexible model for defining and managing trust within DBMSs. The proposed model is based on the definition of new SQL statements for integrating trust management in relational DBMSs. Second, we propose an algorithm that determines (if any) the delegation chains for (a possible subset of) the attributes listed in a certificate. The algorithm computes the delegation chains by taking into consideration the fact that the computational effort required for chain verification should be minimized. Third, we empirically verify the soundness of the proposed technique by integrating our model with a well-known open-source DBMS, PostgreSQL, and evaluate the impact on system performance.

The remainder of the article 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, which allow a seamless integration with existing DBMSs. Section 4 presents an algorithm for retrieving a valid delegation chain in support of a set of certificates presented by a client. Section 5 discusses integration with current relational database engines, and specifically the implementation of our proposal in PostgreSQL. Section 6 highlights the practical impact of our solution on current and future ICT infrastructures for credential and identity management. Section 7 describes related work. Finally, Section 8 presents our concluding remarks. The article also includes an Appendix that reports an analysis of the complexity of the problem of computing minimum cost delegation chains (Appendix A.1); a detailed description of the functions called by the algorithm introduced in Section 4 (Appendix A.2); and the proofs of the theorems stated in the article (Appendix A.3).

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. 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 leading to them).

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 (including identity) to the certificate subject. It contains a list of pairs ⟨*attribute_name*, *attribute_value*⟩.
—A *delegation certificate* issued by an authority asserts 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 article 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 of 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 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. The crucial feature is a strict and smooth integration between the trust management services and traditional database access control. The potential applications of this approach are particularly significant for the Web scenario, but they can find application in most environments where relational DBMSs are used (see Section 6.2).

We are not concerned with the low-level services (such as certificate formats, cryptographic protocols) required to create, exchange, and verify certificates, or to delegate authority; the model is built assuming the presence and correct behavior 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 service [Lee et al. 2008; Li et al. 2005a, 2003]. 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, give a strong "first-mover advantage".

Our proposal represents a convenient strategy to realize the benefits of a richer trust management model without the need to update the underlying infrastructure (see Section 6.1). This is particularly significant considering that currently X.509 [Housley et al. 2002] 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 X.509 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.

[1]Even if some certificate formats do not consider these two types of certificates as different, it is possible to consider them separately.

3. SQL MODEL FOR TRUST MANAGEMENT

Our solution builds on an analysis of previous proposals, most of which are not directly applicable to the DBMS scenario. Although existing approaches are expressive, they are unmanageable in practice. A major limitation is that most of them are based on expressive logics that cannot be put to work in real DBMSs. Therefore, since all relational DBMSs support SQL, our trust management model is based on 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 with existing DBMS services.

We consider the 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. Delegation certificates are implicitly represented in our model through the definition of certified attributes, for which we can specify the authorities trusted for asserting such attributes and whether delegated authorities can also be accepted (Section 3.2).

Each trust concept is represented by a SQL statement, resulting in the construction of a corresponding schema object (see Section 5). 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 compatible with both the typical DBMS approach, where new constructs are always used to represent novel functions (e.g., SQL:2003/Foundation has more than 200 constructs), and the canonical security design approach, which keeps the policy and its management clear and separated from the mechanism and from implementation details.

3.1. Authority

The concept of authority defines the identities that can 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 consider the introduction of a predefined set of attributes in the authority description. The syntax of the SQL statement is then as follows.[2]

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

This statement permits the definition of an authority with the name *AuthorityName* whose public key is stored in attribute `public_key`, which must always be specified. The attributes describing the authority may possibly be imported from an existing certificate stored in file *FileName* (clause `imported by`). In this case, the attributes follow the schema specified by X.509 [Housley et al. 2002], and must include the authority's public key.

Example 3.1. The following `create authority` statement defines authority *Government*, representing the governmental department of health, specifying its X.509 attributes, namely: common name (`CN`), organization (`O`), and country (`C`).

```
create authority Government
 (public_key = '14:c9:ec....:4f:91:51',
```

[2]Note that, since there is also the need to manage the evolution of the policy, the `drop` and `alter` statements are required for the removal and update of authorities as well as for all the other components of the policy schema, which will be described next. We omit the description of these statements.

```
CN = 'Department of Health',
O = 'Governmental',
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 an authority class with the name *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 clauses `authoritative`, `with [no] delegation`, `except`, and `check` appears in Section 3.2, since they are also used for defining trust tables. The main difference in the management of authority classes, compared with trust tables, is that trust tables represent attributes 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 definition of an authority class is recursive, and an authority class can then be defined starting from another authority class.

Example 3.2. The following `create authorityclass` statement defines the *ClassHospital* class as any private or public institution holding a certificate issued by the *National Healthcare System* (*NationalHealthcare*), proving that the institution has the authorization to operate as a hospital in a given city.

```
create authorityclass ClassHospital
 authoritative NationalHealthcare with delegation
 (authorization varchar(30) check (authorization is not null),
  city varchar(20))
```

3.2. Certified Attributes (Trust Tables)

Traditional approaches to trust management (e.g., Bonatti and Samarati [2002]; Irwin and Yu [2005]; Li et al. [2005a]; Yu et al. [2003]) do not usually assume a (pre)declaration of the attributes that will be used in the policy, but directly 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 since queries can refer to the information provided via certificates. The concept of trust table captures the following 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, thus filtering certificates on the basis of the values of the attributes appearing in them.

The proposed syntax for the definition of a trust table 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 follows. The *TrustTableName* represents a name associated with the set of attributes extracted from certificates signed by the given authorities.

The `authoritative` clause describes the authorities (or classes of authorities) that are trusted as signers of certificates, including the required set of attributes. If the `authoritative` clause associated with a trust table contains a set of authority names only, certificates provided by them will be analyzed to extract certified attributes. If the `authoritative` clause contains the name of authority classes, the certificates considered for the corresponding trust table can also be those issued by authorities that belong to the specified authority classes. An authority *AuthorityName* belongs to an authority class if it has a certificate that satisfies the restrictions specified in the definition of the authority class. Since, in turn, this certificate can be issued either by an authority (different from *AuthorityName*) explicitly listed in the `authoritative` clause of the `create authorityclass` statement or by an authority (different from *AuthorityName*) belonging to an authority class listed in the same clause, the verification process is recursive. If the `authoritative` clause is missing, no specific authority is associated with the trust table, and every certificate is analyzed with respect to the configuration of the underlying engine that verifies certificates' validity.

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 given trust table (even if they have received a delegation for it). The reason for exceptions can be that the named 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 statement uses the `check` clause to introduce constraints on the values of the certified attributes. The conditions that can be specified in the `check` clause are the same conditions that can appear in the `where` clause of a SQL query and can reference any attribute in the trust table.

Example 3.3. The following trust table, *Physician*, specifies the attributes characterizing physicians working for a specific hospital. The authorities trusted to issue certificates that include the attributes mentioned in the trust table are the authority class *ClassHospital* of hospitals; authority *Government* representing the department of health; authority *Board* representing the research board; and authority class *ClassResearchInstitute* of research institutes. We assume that authority *LocalHospital* cannot be considered for this trust table. The `check` clause imposes the non-nullity of

attribute *number*, representing the registration number of the physician to the public register.

```
create trusttable Physician
 authoritative ClassHospital with no delegation,
               Government with delegation,
               Board with delegation,
               ClassResearchInstitute with delegation
 except LocalHospital
 (number char(10) check (number is not null),
  project varchar(20),
  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 providing the DBMS with means to exploit certified attributes to regulate access. In this section we show how certified attributes can be used by the DBMS to enable roles and user identifiers. 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-based restrictions.

3.3.1. Trust Policy. The trust policy represents the mechanism by which data access privileges are assigned to clients, based on the information in the trust tables. The trust policy allows the system to associate a given role with a client, 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
```

This statement allows the definition of a trust policy *PolicyName*, where *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 previously defined SQL role, that is, a set of privileges that can be dynamically activated by users granted the privilege to activate the role [Database language SQL 1999]. The semantics of the statement is that all users presenting certificates satisfying the condition are authorized to activate the specified role, or are associated with the specified user identifier *UserId*. If the `autoactivate` option is indicated, the role activation is automatic.[3] If the `for` clause is omitted, the user satisfying the condition is assigned the privileges of the predefined identifier `PUBLIC`, which every user within the system 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 typically be used to establish role activation. The reason

[3]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 all be activated at the same time. The concurrent activation of multiple roles does not create a critical situation, thanks to the absence of negative authorizations in SQL, which permits an immediate combination of different authorization profiles based on the set union.

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.

Example 3.4. The following policy activates the role *Cardiologist* for each user presenting a certificate (see Example 3.3) proving that she is a physician whose specialty is cardiology.

```
create trustpolicy RoleCardiologist
 for Cardiologist autoactivate
 where Physician.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 it) may depend on properties of the user (or her session) such as the time or the machine from which the user connected, and so on. Our trust management solution can be used 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 her certified attributes. This technique is simple, yet effective and powerful. The specification of context-based restrictions relies on the traditional SQL syntax for the definition of views. When it is necessary to define restrictions on the value of certified attributes, trust tables may appear in the `from` clause of the SQL statement that defines a view. The specification of the certified attributes in the definition of a view follows an approach similar to the one used for the definition of trust policy conditions, thus referencing certified attributes using the SQL dot notation (i.e., *TrustTableName.AttName*). A small difference is that trust tables are assumed to be directly available in the definition of the trust policy condition, whereas they have to be explicitly listed in the `from` clause of the query defining the view.

Example 3.5. The following view grants each physician access to the health examinations data of her patients. In fact, each physician will be able to see all the patients who have the physician's `number` in the `doctor_code` attribute.

```
create view PatientView as
 select Examinations.*, Patients.*
 from Examinations join Patients on Examinations.patient_id=Patients.id
      join Physician on Physician.number=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., Li et al. [2003]). Unfortunately, all the models that offer a flexible delegation mechanism use algorithms for the verification of delegation chains that are difficult to apply in the database context. The main reason for this difficulty is that database implementors are reluctant to integrate a logic model checker within the relational engine. We instead show that our model permits the application of an algorithm that is able to identify if a given certificate (set thereof) produces a delegation path for (a possible subset of) the attributes it certifies, without the need of integrating a logic model checker. Also, with an approach similar to the one used by DBMS query optimizers, our algorithm can apply a cost model that estimates the computational effort required for the verification of delegation chains. The experimental results show

that the complexity of our delegation mechanism remains manageable by a DBMS (see Section 5.3).

We assume that the system has knowledge of all the delegation certificates *Deleg_Certs*, all the authority certificates *Authority_Certs* that certify some attributes of the authorities, and of the policy adopted by the system, composed of a set $\mathcal{TT}$ of trust tables, a set *Auth* of authorities, and a set $\mathcal{AC}$ of authority classes. This data is needed for the verification of a given set *Cert* of certificates presented by a client during a session. We also assume that each 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 low cost of using certificates cached as valid in prior verification, as well as the high 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. Our goal is then to compute a minimum cost delegation chain for all the certificates presented by a client. Since the problem of determining a minimum cost delegation chain is NP-hard (see Theorem A.1 in Appendix A), we propose the heuristic algorithm described in the remainder of this section.

4.1. Algorithm

Figure 2 illustrates our heuristic algorithm for the computation of the minimum cost delegation chains for the certificates presented by a client. Appendix A.2 reports a detailed description of the functions called by the algorithm along with their pseudocode. The algorithm receives as input the set *Cert* of certificates presented by a client during a session, the set $\mathcal{TT}$ of trust tables, authorities *Auth*, authority classes $\mathcal{AC}$, delegation certificates *Deleg_Certs*, and authority certificates *Authority_Certs*. For each certificate *cert* in *Cert*, the algorithm first calls function CHECKCORRECTNESS that performs all the noncryptographic controls (e.g., expiration time) on *cert*. If function CHECKCORRECTNESS returns true for each trust table TT in $\mathcal{TT}$ such that *cert* is *compatible* with TT (i.e., *cert* contains all the attributes required by TT and *cert* satisfies all the `check` conditions specified in the definition of TT), the algorithm calls function **Satisfy** with parameters *cert* and TT. Function **Satisfy** returns a set *ver_list* of certificates forming the delegation chains (if any) supporting all the common attributes in *cert* and TT. If all the certificates in *ver_list* are valid, the algorithm inserts a tuple in trust table TT whose attributes values are extracted from the corresponding attributes in *cert*; the algorithm terminates returning an error message, otherwise. We now describe how function **Satisfy** works.

Function **Satisfy** is the core component of our algorithm. The function takes as input a certificate *cert* and an entity E, which may either be a trust table or an authority class compatible with *cert*. It returns a set of certificates that compose the delegation chains supporting all the attributes in $cert.attributes \cap Attributes(TT)$, and rooted at an authority (or authority class) that is trusted with respect to TT. If such delegation chains do not exist, the function returns an empty set of certificates.

Function **Satisfy** first checks whether *cert.issuer* is a valid authority with respect to entity E. Three cases may occur: (1) *cert.issuer* appears in the `except` clause of E (i.e., *cert.issuer* is in $Except(E)$), and the function terminates, returning an empty set of certificates; (2) *cert.issuer* appears in the `authoritative` clause of E (i.e., *cert.issuer* is in $Authoritative(E)$), and the function terminates, returning *cert* as the unique certificate composing the delegation chain; or (3) *cert.issuer* is a member of an authority class in the `authoritative` clause of E, and the delegation chains proving this membership are stored in variable $cert.issuer.ac\_ver\_list(E)$. To verify whether *cert.issuer* is a member of an authority class in the `authoritative` clause of E, function **Satisfy** calls function

```
/* cert.id: identifier of certificate cert */
/* cert.issuer, cert.subject: issuer and subject of certificate cert */
/* cert.attributes: attributes in certificate cert */
/* cert.cost: cost for the verification of certificate cert */
/* E: trust table or authority class */
/* Except(E): authority names in the except clause of E */
/* Authoritative(E): authority names and classes in the authoritative clause of E */
/* Attributes(E): attributes in E */
/* auth.ac_visited(E): flag set to true if the algorithm has already checked whether */
/*                     auth belongs to a class in Authoritative(E) */
/* auth.ac_ver_list(E): set of delegation chains proving that auth belongs to a class in Authoritative(E) */

Input: Cert: set of certificates presented by a client during a session
       TT, Auth, AC: set of trust tables, authorities, and authority classes
       Deleg_Certs: set of delegation certificates
       Authority_Certs: set of certificates assigning attributes to authorities
Output: tuples in TT

MAIN
for each cert∈Cert do
  if CheckCorrectness(cert) then /* makes noncryptographic controls */
    for each TT∈{TT' | TT'∈TT∧ Compatible(cert, TT')} do
      /* certificate cert can contribute to populate TT */
      ver_list := Satisfy(cert, TT)
      /* check the validity of certificates */
      if ver_list≠∅ then
        for each cert'∈ver_list do
          if ¬Valid(cert'.id) then exit /* cryptographic controls fail */
        Insert into TT a tuple with values of the attributes extracted from cert

Input: cert: certificate
       E: trust table or authority class
Output: set ver_list of certificates proving that cert matches with E

SATISFY(cert, E): ver_list
/* Preliminary steps that verify whether cert.issuer is a valid authority w.r.t. E */
if cert.issuer∈Except(E) then return(∅)
if cert.issuer∈Authoritative(E) then return(cert)
for each authority name auth in Authoritative(E) do
  auth.ac_visited(E) := FALSE
/* check if cert.issuer belongs to an authority class in the authoritative clause of E */
cert.issuer.ac_ver_list(E) := CheckClasses(cert.issuer, E)
if ∀auth∈Authoritative(E) the delegation flag is set to false then
  return(cert.issuer.ac_ver_list(E)) /* no delegation */
/* Phase 1: find supporting chains */
Candidates := FindChain(cert, E)
if Candidates=∅ then return(∅) /* no chain covering all attributes exists */
/* Phase 2: build the set of certificates to be validated */
ver_list := BuildVerificationList(cert, E, Candidates)
return(ver_list) /* return the list of certificates for validation */
```

Fig. 2. Certificate verification algorithm.

CheckClasses (Figure 12 in Appendix A). For each authority class in *Authoritative*(*E*), **CheckClasses** recursively calls function **Satisfy** and returns the delegation chains (if any) with minimum cost, proving that *cert.issuer* is a member of the authority class. Such delegation chains are stored in variable *cert.issuer.ac_ver_list*(*E*). Furthermore, **CheckClasses** inserts a virtual delegation certificate representing the computed delegation chains. Intuitively, this virtual delegation certificate represents the fact that *cert.issuer* is trusted to produce certificates for attributes in *Attributes*(*E*), since it is a member of an authority class listed in *Authoritative*(*E*).

After the analysis of *cert.issuer*, function **Satisfy** checks the delegation flag of all the authorities and classes in the `authoritative` clause of E. If all such authorities and authority classes have the delegation flag set to false, function **Satisfy** terminates by returning the set *cert.issuer.ac_ver_list*(E) of certificates. In fact, if delegation chains cannot be considered, *cert* is a valid certificate with respect to E only if it has been directly issued by an authority that belongs to an authority class in the `authoritative` clause of E. If at least an authority or a class in the `authoritative` clause of E has the delegation flag set to true, function **Satisfy** searches a set of delegation chains that reaches an authority (or a class) in the `authoritative` clause of TT and that supports all the common attributes between *cert* and TT. To this purpose, the set *Deleg_Certs* of delegation certificates (also including virtual delegation certificates) is seen as a *delegation graph*, where there is a node for each issuer and subject of the delegation certificates, and there is an edge for each delegation certificate going from the issuer of the certificate to its subject. Each edge is labeled with a pair ⟨*attributes*,*cost*⟩, where *attributes* is the set of attributes asserted by the corresponding delegation certificate and *cost* is the cost for verifying the certificate. The process of finding delegation chains consists in (i) finding supporting chains for the attributes considered (function **FindChain**); and (ii) removing redundant supporting chains (function **BuildVerificationList**). We assume that the delegation graph is acyclic and that the subgraphs of *Deleg_Certs* necessary for verifying different certificates do not have common edges (i.e., common certificates). Function **FindChain** (Figure 13 in Appendix A) adopts a Dijkstra-like approach to determine, for each attribute that appears both in *cert* and in TT, the minimum cost path reaching *cert* from an authority (which belongs to an authority class) in the `authoritative` clause of TT with the delegation flag set to true. We note that function **FindChain** invokes function **CheckClasses** to verify whether the authorities along the computed paths belong to a trusted authority class. Function **BuildVerificationList** (Figure 14 in Appendix A) analyzes the paths computed by function **FindChain** and removes possible redundancies.

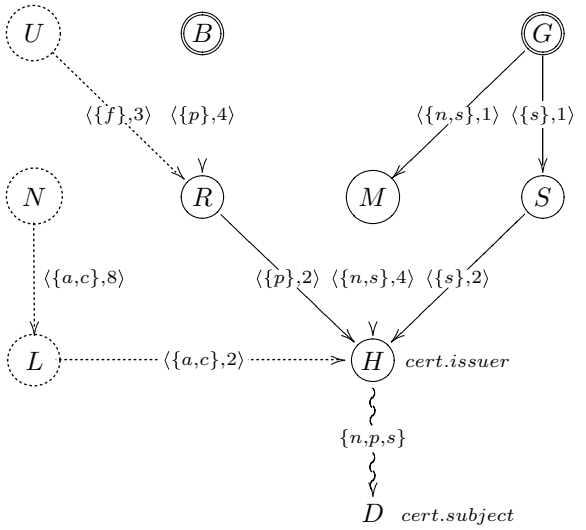
The nonredundant delegation chains obtained by function **Satisfy** are finally returned.

Example 4.1. Consider a certificate *cert* issued by authority *Hospital (H)*, with subject *Doctor (D)*, and certifying attributes *number (n)*, *project (p)*, and *specialty (s)*. Figure 3(a) illustrates the set of authority certificates *Authority_Certs* and the set of delegation certificates *Deleg_Certs* available in the system and involved in the processing of *cert*. It is easy to see that *cert* is compatible with the trust table *Physician* (see Example 3.3), and since the issuer of *cert* is not an authority listed directly in the `except` or `authoritative` clause of *Physician*, we need to check the existence of delegation chains supporting attributes {*n, p, s*}. Figure 3(b) illustrates the delegation graph corresponding to the set of certificates in Figure 3(a). Here, the authorities directly listed in the `authoritative` clause of *Physician* are represented through a double circle. Dotted edges and nodes represent the delegation certificates and authorities needed for verifying whether an authority belongs to an authority class. The curly edge represents certificate *cert*.

Function **Satisfy** first calls procedure **CheckClasses** to verify whether H is a member of the *ClassHospital* authority class directly listed in the `authoritative` clause of *Physician*. Since the dotted path in Figure 3(b) starting from N and ending in H is a delegation chain supporting such a membership (see also the definition of *ClassHospital* in Example 3.2), function **CheckClasses** adds a virtual delegation certificate, where the issuer is the virtual authority $\mathcal{C}$, the subject is H, the attributes are those mentioned in the *Physician* trust table, and the cost is the sum of the costs associated with the dotted edges (see the dashed edge in Figure 4). Function **Satisfy** then calls

Issuer	Subject	Attributes	Cost
EuropeanUnion (U)	ResearchInst (R)	founding (f)	3
NationalHealthcare (N)	LocalHealthcare (L)	authorization, city (a, c)	8
LocalHealthcare (L)	Hospital (H)	authorization, city (a, c)	2
Board (B)	ResearchInst (R)	project (p)	4
ResearchInst (R)	Hospital (H)	project (p)	2
Government (G)	MedicalBoard (M)	number, specialty (n, s)	1
Government (G)	School (S)	specialty (s)	1
MedicalBoard (M)	Hospital (H)	number, specialty (n, s)	4
School (S)	Hospital (H)	specialty (s)	2
Hospital (H)	Doctor (D)	number, project, specialty (n, p, s)	3

(a)



(b)

Fig. 3. An example of certificates (a) and corresponding delegation graph (b).

function **FindChain** that finds the paths with minimum cost that support attributes: $n, G \rightarrow M \rightarrow H; p, \mathcal{C} \rightarrow R \rightarrow H$; and $s, G \rightarrow S \rightarrow H$. We note that function **FindChain** while searching for the path supporting attribute p, adds another virtual delegation certificate where the issuer is again the virtual authority $\mathcal{C}$; the subject is R; the attributes are those mentioned in the *Physician* trust table; and the cost is the sum of the costs associated with the dotted edge from U to R (see Figure 4). Function **Satisfy** finally calls function **BuildVerificationList**, which removes the redundant delegation chain $G \rightarrow S \rightarrow H$ supporting attribute s. In fact, path $G \rightarrow M \rightarrow H$ supports both attributes n and s. The certificates that need to be verified are therefore the ones along paths $\mathcal{C} \rightarrow R \rightarrow H$ and $G \rightarrow M \rightarrow H$, and the path represented by virtual certificate $\mathcal{C} \rightarrow R$ (i.e., $U \rightarrow R$). Example A.2 in Appendix A describes in more detail the execution, step-by-step, of functions **FindChain** and **BuildVerificationList**.

4.2. Correctness and Complexity

We now state the correctness of our approach and discuss its complexity.

THEOREM 4.2 (TERMINATION AND CORRECTNESS). *Given a finite set of delegation certificates Deleg_Certs, a finite set of authority certificates Authority_Certs, and a certificate*

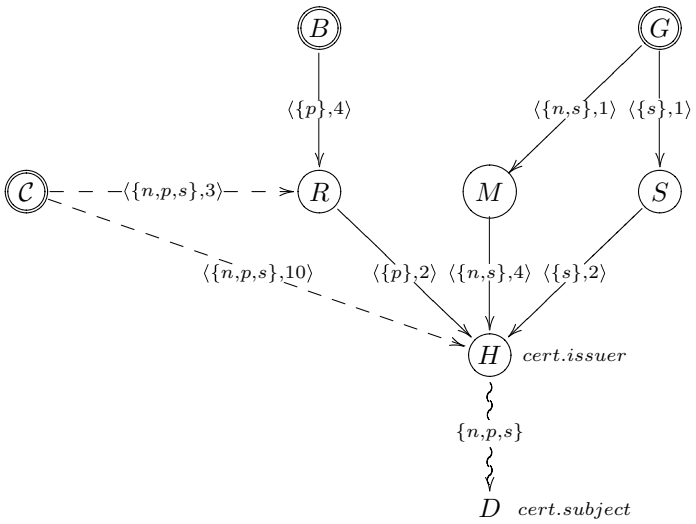


Fig. 4. Delegation graph for Example 4.1 extended with virtual certificates.

cert compatible with an entity E, function **Satisfy** *terminates and determines a correct set of delegation chains for cert if such delegation chains exist.*

PROOF. The proof is presented in Appendix A.3. □

We adopt the following notational conventions with respect to a given set *Deleg_Certs* of delegation certificates; a set *Authority_Certs* of authority certificates; a set $\mathcal{AC}$ of authority classes; a set *Auth* of authorities; a certificate *cert*; and an entity (i.e., a trust table or an authority class) E compatible with *cert*: N_d denotes the number of delegation certificates (i.e., $N_d = |Deleg\_Certs|$); N_{auth} denotes the number of authority certificates (i.e., $N_{auth} = |Authority\_Certs|$); $N_{\mathcal{AC}}$ denotes the number of authority classes (i.e., $N_{\mathcal{AC}} = |\mathcal{AC}|$); N_E denotes the number of attributes listed in the definition of E (i.e., $N_E = |Attributes(E)|$).

THEOREM 4.3 (COMPLEXITY). *Function* **Satisfy** *finds delegation chains for a certificate cert compatible with an entity E in* $O(N_d \cdot N_E \cdot N_{\mathcal{AC}} \cdot N_{auth} \cdot \log(N_d \cdot N_E))$.

PROOF. The proof is presented in Appendix A.3. □

In terms of quality of the solution returned by the algorithm, we observe that the heuristic algorithm computes a solution that offers guarantees in terms of distance from the optimum. If we consider separately the evaluation of every trust table or authority class, the solution returned exhibits the minimum possible cost for the attribute with the highest cost, due to the fact that the algorithm applies a search strategy that is similar to the Dijkstra algorithm for the identification of the path with minimal cost. Indeed, if there are no restrictions on attribute delegations, the solution identified for a trust table will be optimal; otherwise, the cost of the identified solution cannot be greater than $n - 1$ times the cost of the optimal, where n is the number of attributes. In general, we expect that in most scenarios the delegation graph will have a simple structure and the proposed algorithm will be satisfactory in terms of the quality of the solution, at the same time being able to manage complex situations.

5. INTEGRATION WITHIN A DBMS

There are few principles that have to be followed in the integration within a current DBMS of our trust management service. First, the implementation in real systems of this service 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, so as to minimize the effort required to the database designer in the modeling of application requirements for access control. Our proposal has been designed taking into account all these principles, building our extensions on existing functionalities and catalogs, and ensuring seamless integration with existing DBMSs.

One key aspect in the implementation of our solution deserving mention concerns role activation. In fact, our approach is not directly enforceable within current DBMSs for two main reasons. First, our trust policies require support for role activation (i.e., granting privileges) on the basis of certified attributes (see Section 3.3.1), while the SQL standard binds roles to user identities only. Second, our model requires that a client satisfying the conditions of different trust policies to activate multiple roles within a session, while SQL supports the activation of one role per session for each client. Section 5.2 shows how to overcome these limitations of the SQL standard with minimal changes to the DBMS code.

Since trust tables and trust policies extend the schema of the database, the implementation of the trust management service requires extending the database catalog. In particular, it is necessary to introduce catalog tables (and their schemas) for authorities as well as for trust tables and trust policies.

A trust table behaves as a *global temporary table*. Global temporary tables are described in the SQL standard [Database language SQL 1999] and represent tables that are part of the database schema, but that differ from base tables because their content cannot be shared among different sessions. A session can then use a global temporary table to store information that is needed within the same session and that must be protected from access by 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 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.

However, since several DBMSs do not support global temporary tables, it may be necessary to simulate their services defining an additional attribute, that is, the session identifier, in the schema of classical relational tables that implement trust tables. The goal of this attribute is to associate each tuple with the session on which it has been presented. We discuss this issue in Section 5.2.

The effectiveness of our approach and the impact on performance have been evaluated by implementing a prototype. We now present the technical details characterizing the implementation, and then present the set of experiments we conducted to evaluate the impact of our trust management service on performance.

5.1. Design Choices

PostgreSQL is a well-known open-source DBMS, the current incarnation of Postgres, a system initially developed at Berkeley under the supervision of Michael Stonebraker

[1987]. PostgreSQL has been chosen for our prototype since it is commonly considered the most advanced and interesting of the current open-source database servers.

The availability of the component responsible for the cryptographic functions used for the verification of certificates is fundamental. One of the most used implementations of the SSL protocol is the OpenSSL system, which is already included in recent distributions of PostgreSQL. In fact, PostgreSQL offers the possibility to realize a dialog between clients and the PostgreSQL server using an SSL connection.

The development of the prototype has been performed by taking into account the following major requirements which have to be satisfied by every implementation of our trust management service.

—*Minimal impact on the DBMS engine.* The prototype has been developed to verify that the approach can be realized with a small impact on a current DBMS engine.
—*Behavior of trust tables equivalent to global temporary tables, as defined in the recent SQL standard.* We have previously introduced the notion of trust table as a global temporary table, where the logged user can find tuples describing all the certificates provided at login time. In the SQL standard definition, global temporary tables are visible across sessions, but a session can see only tuples created by itself. Since PostgreSQL does not support global temporary tables (temporary tables in PostgreSQL correspond to *local* temporary tables, 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 mechanism to create an equivalent concept within the prototype must be developed.
—*Isolation of sessions.* The implementation of the trust management service within the prototype has to offer a strong guarantee on *isolation of sessions*, as far as grant of privileges is concerned. This is a crucial feature of the proposed model which would potentially require a significant rewriting of all the code dealing with the assignment of privileges. Due to the requirement on the minimization of the impact on the database server, we excluded the full revision of the architecture of the access control component. Instead, we devised to introduce for each session a novel generic user identifier as a reference for the assignment of privileges. This user identifier initially has no role or privilege, and it acquires them when trust policies are fired, based on properties extracted from certificates. The critical aspect in this design is to decide whether user identifiers have to be reused from an existing pool or have to be created from scratch at the start of each session. The advantage of having a pool of predefined user identifiers is that the cost of creating user identifiers is only paid at system initialization time. The pool must have a size adequate to support the maximum expected number of concurrent sessions. The major disadvantage of this solution is the need to carefully remove every privilege from the user identifier at session closure, with an explicit revoke statement, otherwise privileges would be transferred from one session to the next one using the same user identifier. We chose the second option and create, at the start of each session, a new user identifier. The motivation is that this solution offers a robust and automatic guarantee of isolation among sessions. A performance evaluation showed that the cost for creating user identifiers is small, and is compensated by the automatic revoke of all the privileges that follow the deletion of the user identifier, without the need to pay the cost of execution of explicit SQL statements in the alternative approach.

5.2. Prototype Features

The prototype has been built starting from the PostgreSQL 8.3.0 release. The implementation was mostly focused on the revision of the login and logout processes and on the extension of the catalog to manage the information describing trust tables and

PHYSICIAN_TTABLE

session_id	number	project	specialty
001	025	allergies	dermatologist
001	025	stress diseases	dermatologist
002	048	pediatric diseases	cardiologist

(a)

PHYSICIAN

number	project	specialty
048	pediatric diseases	cardiologist

(b)

Fig. 5. An example of trust table (a) and corresponding view for session 002 (b).

trust policies. The additional relations in the catalog and their configuration at the time a new session is created permit reusing the existing services of the DBMS for the enforcement of the access control policy. The implementation required to adapt C modules and the Lex/Yacc sources in the PostregSQL distribution, modifying and adding in total around 6,000 lines of code.

Login Manager. The login manager was modified to extract information from certificates and to insert it into trust tables. The properties that are defined in trust tables appear in the standard properties or *extensions* of X.509 certificates sent by clients. An X.509 v3 [Housley et al. 2002] extension is an application-specific property reported in X.509 certificates, which is assigned a unique numeric code defined as a sequence of 4 numbers, each of them with a value up to 255. Furthermore, X.509 v3 allows the association of a name with the numerical identifiers. For each extension, the certificate describes both the numerical identifier and the associated name (if any).

The extraction of named extensions occurs by checking them against the available trust table schemas. The extracted property values are then available for user identification and role assignment. This function is realized by the *policy evaluation mechanism*, which fires policies activated by tuples inserted into trust tables.

Management of Trust Tables. To make trust tables really part of the SQL environment, their definition must be part of the database schema. For this reason, we extended the standard database catalog with a number of tables; the names of the tables follow the style of PostgreSQL, which uses a "pg_" prefix (e.g., the catalog table describing the defined trust tables is called *pg_trust_table*).

The realization of trust tables in the prototype had to overcome the limitation of PostgreSQL, which does not support global temporary tables as defined in the SQL standard. In our prototype, we build, for each trust table named *trust_table_name*, a new regular table with the name *trust_table_name*, followed by the suffix *"_ttable"*. The schema of the new table presents all the attributes declared in the trust table definition, extended with attribute *session_id*, containing the session identifier. For instance, from the definition of trust table *Physician*, the system will create a table *Physician_ttable*(*session_id, number, project, specialty*). Figure 5(a) illustrates an example of trust table, *Physician_ttable*, with three tuples referred to two different sessions, namely session 001 and session 002.

Tuples are inserted into this table when a connection request succeeds and a new session is created. For each certificate matching the corresponding trust table, a tuple is inserted into the new table, with the session identifier and the trust table attribute values extracted from the certificate. Note that this table does not use the session identifier as a primary key, since several tuples with the same session identifier might be inserted into the trust table. This is the case when multiple certificates are sent by the client and several of them match the same trust table: in this case, for each matched certificate, a tuple with the same session identifier is inserted into the trust table.

The management of trust tables is then completed by the definition of views over the supporting tables storing certificates, named as the trust tables. For instance, for *Physician* we can introduce a view with the following SQL statement.

```
create view Physician as
 (select number, project, specialty
  from Physician_ttable
  where session_id = pg_backend_pid())
```

In this way, the user can only see the tuples concerning the current session, since the PostgreSQL function `pg_backend_pid()` returns the identifier of the current session issuing the query. Access is provided only to properties extracted from the session certificates. Of course, the user is allowed by standard authorizations to access the view but not the underlying table. As an example, Figure 5(b) illustrates the *Physician* view resulting from selecting tuples with `session_id`=002 from the trust table in Figure 5(a).

Management of Trust Policies. Trust policies assign roles to sessions by considering the information appearing in the trust tables. Trust policies are managed by creating, for each policy, a stored procedure (a *PL/pgSQL function*, in the PostgreSQL terminology), which evaluates the policy condition, and, if it is successful, grants the roles specified in the policy to the user generated for the session, activating them if the `autoactivate` option is specified in the trust policy definition.

The support for trust policies within the catalog is realized by two tables, *pg_trust_policy*(*tpolicy_id*, *tpolicy_name*, *condition*, *role*, *function_id*) and *pg_tpolicy_and_ttable*(*tpolicy_id*, *ttable_id*). For each trust policy, a row appears in table *pg_trust_policy*. Each trust policy is internally identified by a unique numerical identifier (attribute *tpolicy_id*). Attribute *tpolicy_name* is the policy name (this is unique as well, thus it is a secondary key of the table); attribute *condition* describes the trust policy condition (defined in the `create trustpolicy` statement); attribute *role* denotes the role to assign to users if the trust policy is fired; and attribute *function_id* denotes the internal identifier of the *PL/pgSQL* function that evaluates the trust policy condition. Table *pg_tpolicy_and_ttable* denotes the trust tables involved in trust policy conditions; a distinct table is used, because trust policy conditions can refer to more than one trust table.

The *PL/pgSQL* function applies the trust policy condition to the join of all the trust tables that appear in it. At trust policy definition time, the *PL/pgSQL* function code is translated into a C function, which is compiled and whose executable version is stored into the database. Furthermore, the execution plan for the query corresponding to the trust policy is also prepared and stored into the database, obtaining high performance when the function is executed.

Login Process. The login process performs the following steps in the activation of a session where certificates are presented by the client.

(1) *Certified login.* This is the login based on SSL certificates. When a login request comes from a client, the login manager checks the presented certificates and identifies those that contain properties of the client (i.e., attribute certificates); the client may also present delegation certificates and certificates declaring properties of certification authorities.
   (a) *Trust table identification.* For each attribute certificate, the properties are extracted from the certificates and checked, accessing the catalog, against the schema of identified trust tables. The current implementation supports properties with the use of X.509 v3 extensions. Future systems will use more flexible certificate formats, like the one provided by SAML [Lockhart et al. 2007]. The

login process identifies the set of trust tables that are compatible with the certificate content.

(b) *Certification authority verification.* For each pair of an attribute certificate and compatible trust table, the certification authority that issued the certificate is searched in table *pg_trust_ca* (i.e., the catalog table that describes registered certification authorities identified by an internal numerical identifier *ca_id*, with the unique name *ca_name* given in the `create authority` statement, and a unique public key *ca_publickey* which identifies certificates issued by the authority): if not present, the process verifies the presence of a correct delegation chain, with an approach that follows the algorithm in Section 4. Another option for the implementation of the authority verification process could be to rely on the integration of a logic-based reasoner. This alternative would offer a different and potentially more compact description of the verification algorithm, but it would also require the strict combination of such a tool with the relational engine, greatly increasing the implementation effort.

(c) *Property extraction from certificates.* If the above check is successful, a new tuple with the values of the attributes involved in the verified certificate and with the current session identifier is inserted into the matched trust table.

(2) *Temporary user creation.* To guarantee isolation among different sessions, after the completion of a successful authentication, the login manager creates a new user identifier with no connection rights (only the current session can exploit this user id). The name of this user, called *temporary session user*, is obtained by adding a fixed textual prefix to a string that contains the session identifier. At this point, the new session is activated. Since a temporary session user is created for each session, the same client can open distinct sessions with the database server, presenting different certificates; privileges will be assigned to each session depending only on the certificates that have been presented.

(3) *Granting access to trust table views.* For each trust table, the temporary session user is granted the privilege of accessing the view with the same name (with the restriction on the session id hidden in the view definition). In this way, the user can query the content of trust tables.

(4) *Trust policy activation.* For each trust table, the login manager analyzes the catalog and selects the trust policies that refer to trust tables in which at least a certificate has been inserted. For each selected trust policy, the corresponding *PL/pgSQL* function is executed. The previously compiled function evaluates the trust policy condition against the current state of the trust tables. If the query result is not empty, the role associated with the policy is granted to the user and possibly activated.

Logout Process. When the user logs out, it is necessary to revoke roles granted in the login phase. The process performs the following steps.

(1) *Revoke of access to trust table views.* For each trust table, the authorization to query the corresponding view is revoked.

(2) *Deletion of session tuples.* For each trust table, tuples presenting the session identifier are deleted.

(3) *Deletion of temporary session user.* All roles granted to the temporary session user are revoked, the temporary session user is dropped, and the session is closed.

Comparison Between Triggers and Stored Procedures. An important issue in the realization of the model proposed in this article is the construction of the mechanism guaranteeing that, as soon as a certificate is presented within a session that satisfies the condition of a trust policy, the privileges specified by the trust policy must be

assigned to the session. There are two alternative approaches that can be adopted. The first approach uses triggers, which have to monitor insertion events on trust tables and react to such insertions, evaluating the trust policy condition on the corresponding session. The approach adopted in the prototype instead introduces a set of stored procedures into the database, which are responsible for the evaluation of the trust policy condition and the corresponding activation of privileges for the session within which the procedure has been activated.

In terms of software design, the use of triggers provides the benefit that it assigns the responsibility for the activation of trust policy privileges to a separate component, thus reducing the effort needed for revising the login manager (a separate policy-definition-time module will be responsible for the creation of a correct set of triggers). The design advantage of procedures lies in their scalability; complex trust policies (i.e., policies that refer in their condition to a number of trust tables) require the definition of a single stored procedure that evaluates the policy condition and verifies whether the collection of information in the trust tables assigned to the session satisfies the predicate; the rule is then invoked by the login manager when it has finished inserting information into the trust tables. The management of a complex policy by triggers instead requires the definition of a collection of triggers, at least one for every trust table referred to in the trust policy, and particular attention has to be paid to produce a set of triggers which correctly considers every situation that may produce an assignment of privileges to users (the triggers have to be written considering traditional approaches to the automatic generation of sets of triggers [Ceri et al. 1994]).

Overall, the above design considerations give a preference to the solution using stored procedures. In the construction of the prototype, both approaches have been initially implemented to get some concrete support for the choice of one alternative over the other. The result of this effort confirms the observations above. In terms of performance, experiments showed that for simple policies (a single trust table appearing in the trust policy condition) triggers were able to offer better results. The explanation of this difference is the way in which triggers are managed by the database, which optimizes the execution of triggers in several ways. For instance, the keyword `new` used in trigger conditions to refer to newly inserted tuples in the trust tables, is managed by the DBMS with a direct access to its memory representation within the working memory of the relational engine; stored procedures instead require access to the content of tables, using traditional SQL mechanisms. Experiments also confirm that, as the number of trust tables in the trust policy condition grows, the time required for the management of sessions correspondingly increases; when trust policies are defined over three trust tables, the execution times with stored procedures become less than the execution times for triggers. In light of these results and the above considerations, we opted to continue the development of the prototype using the approach based on stored procedures, which guarantees a better software design and a more robust behavior in terms of performance, immune from the presence of complex conditions in the access control policy.

5.3. Experiments

To concretely evaluate the impact of our approach on a traditional DBMS, we measured the performance of the login and logout processes, which are the only crucial phases requiring a performance evaluation, since our solution adapts the traditional access control model of SQL to the presence of certificates. We therefore run a set of experiments that open a connection to the database, execute a query in memory, and close the connection. Each run of our experiments opens and closes 1000 sessions. The results illustrated in the following are computed as the average of the values obtained in 10 runs.

The experiments were performed on a PC with two Intel Xeon Quad 2.0GHz L3-4MB, 12GB RAM, four 1-Tbyte disks, and a Linux Ubuntu 9.04 operating system. The database used for the experiments contains 100 tables, with 200,000 tuples each. We initially designed a generic application supporting the access requirements of a small group of users and manually created the trust tables and trust policies. Configurations with a large number of trust tables and trust policies were then produced by writing synthetic definitions. The conditions in the trust policy include a conjunction of predicates, each of which compares one attribute of the trust policy with a constant value. The synthetic data appears adequate to test the performance of our approach because the structure, the number and size of attributes, and the number of tuples composing both trust tables and trust policies have negligible impact on system performance (as we experimentally verified before proceeding with our evaluation). The number of trust tables and of trust policies in the system are instead two important parameters that have an impact on the time required for the completion of the login and logout processes.

The experiments investigated several aspects of system behavior, focusing on the performance of the system when the number of trust tables, trust policies, enabled roles, and concurrent sessions increase. To validate the efficiency of the proposed model, we considered configurations representing systems with a size that exceeds the size expected for systems relying on a trust management service integrated within DBMSs. In particular, we considered configurations where the number of trust tables, trust policies, and concurrent sessions vary in the ranges 0 to 20, 1 to 100, and 1 to 600, respectively. Note that the number of trust tables is lower than the number of trust policies, since in real-world applications the number of certificate formats (modeled through trust tables) is usually limited with respect to the rules regulating access to the data (expressed through trust policies). Also, 100 trust policies appear adequate, even for modeling configurations with sophisticated access control regulations. In the following, we discuss the experimental results obtained with the configurations described above to test the scalability of the system. All the experiments evaluate the time necessary to open a connection to the database, execute a query in memory, and close the connection. A breakdown of the components shows that the time required to execute the query in memory and the time required for the logout remain stable in the different configurations, with an average of less than 3ms and limited variance. The trends shown in the experiments are due only to the management of the login. We also run a few experiments, which we do not discuss in this article, where we compared the performance for the execution of queries in two configurations, one where the user had only a single TM-free privilege active and another with multiple privileges assigned by the trust management component; no difference in performance was observed. The configurations with zero trust tables and zero trust policies show execution times identical to those observed for the TM-free system.

Figures 6 and 7 show how the time increases with the number of trust tables and trust policies in the system, respectively. In particular, Figure 6 shows how the time varies when the number of trust tables varies from 0 to 20. The configuration with no trust tables models a situation where certificates are used for authentication, even if not stored in a trust table (i.e., they are not subsequently used for access control enforcement). Figure 7 compares the times in a system characterized by 5, 10, 15, and 20 trust tables, when the number of trust policies varies from 0 to 100. As the two graphs in Figures 6 and 7 show, the time grows linearly, both with the number of trust tables (Figure 6) and with the number of trust policies (Figure 7). From these results, we also observed that for a fixed number of trust policies (or trust tables, resp.), the execution time increases less than 1ms for every trust table (or trust policy, resp.) added to the system.

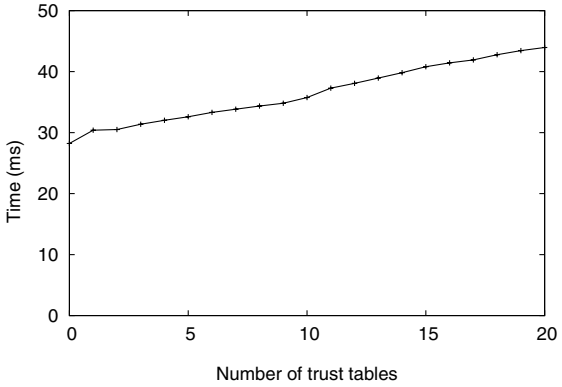


Fig. 6. Time necessary to open a connection to the database, execute a query in memory, and close the connection, varying the number of trust tables.

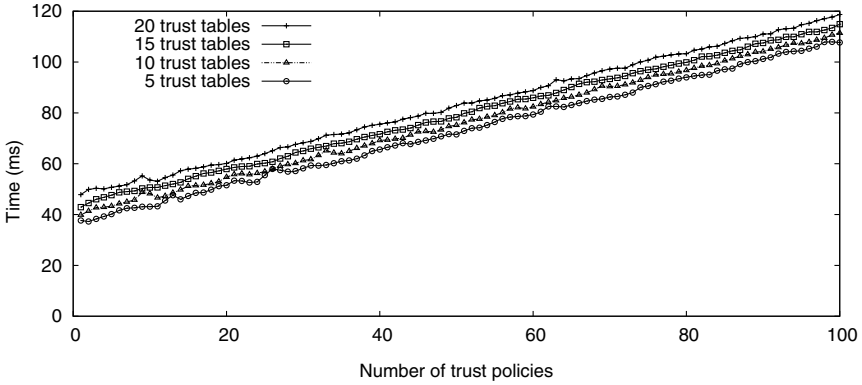


Fig. 7. Time necessary to open a connection to the database, execute a query in memory, and close the connection, varying the number of trust tables and trust policies.

Figure 8 illustrates how the time increases with the number of enabled roles. The graph illustrates the times obtained with a system characterized by 20 trust tables, a number of trust policies that varies from 0 to 100, and a number of enacted roles that varies according to three different scenarios, depending on how many different roles are enabled by the trust policies in the system. In the first scenario, each trust policy enables a single PostgreSQL role. In the second scenario, the first 10 trust policies enable 10 different roles; each of the following trust policy enables one of the first 10 roles. In the third scenario, each trust policy enables a different role. As the graph shows, the number of enacted roles has a significant impact on system performance. When the number of enacted roles grows with the number of trust policies, the execution times double. This is due to the time necessary for the execution of the PostgreSQL statement responsible for role-enabling, which is characterized by relatively heavy processing. However, the scalability of the system is not compromised, since the linearity in the time is not affected by role activation, as clearly shown in the graph. Also, we expect that, in real systems, even configurations characterized by a large number of trust tables and trust policies will typically require the activation of a limited number of roles at each session.

Figure 9 compares the times in a system without trust policies, with a number of trust tables varying from 0 to 20, and with a number of concurrently active sessions that

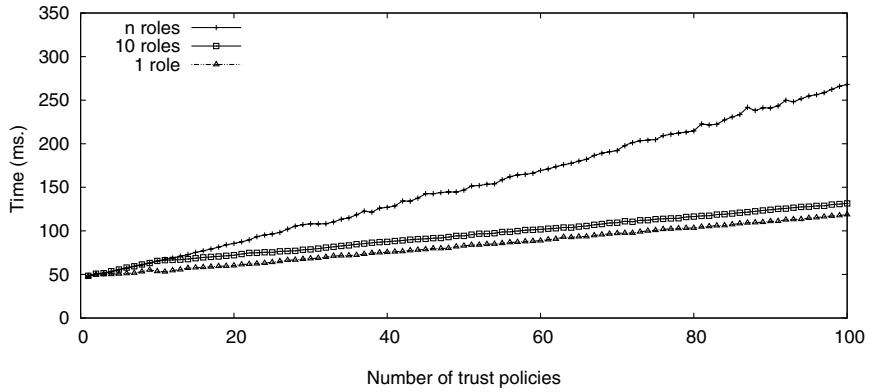


Fig. 8. Time necessary to open a connection to the database, execute a query in memory, and close the connection in a system with 20 trust tables, varying the number of trust policies and enacted roles.

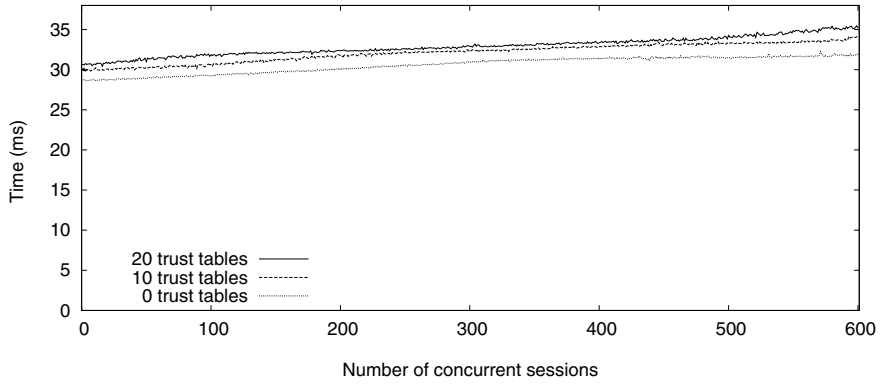


Fig. 9. Time necessary to open a connection to the database, execute a query in memory, and close the connection in a system with 20 trust tables, varying the number of concurrent sessions.

varies from 0 to 600. As expected, the time grows with the number of concurrent active sessions, since the data structures necessary to support the activation, deactivation, and management of sessions (i.e., operating systems structures, internal PostgreSQL structures, and the temporary tables for credentials management) increase. The time however grows linearly with the number of concurrent sessions, thus confirming the scalability of the system, even with respect to concurrent accesses to the system.

Finally, we tested the effectiveness and efficiency of the chain verification algorithm described in Section 4. In particular, we considered two scenarios that differ from each other in how the client provides all the credentials needed for the verification process. In the first case, the client provides the credentials at session startup. The experiments performed in this case prove that the execution time of the chain verification algorithm is negligible compared to the time necessary to populate the relational structures used to keep track of the credentials provided at login time, even if the number of certificate authorities in the system grows. In the second case, the credentials are stored in a table and the algorithm explores the delegation graph by retrieving one-by-one at each step the credentials necessary for the verification process. Figure 10 reports the execution time of the chain verification algorithm, varying the number of certificate authorities from 1 to 100, with the certificates forming a linear chain (the worst case in terms of performance). As the graph shows, the execution time grows linearly with the number

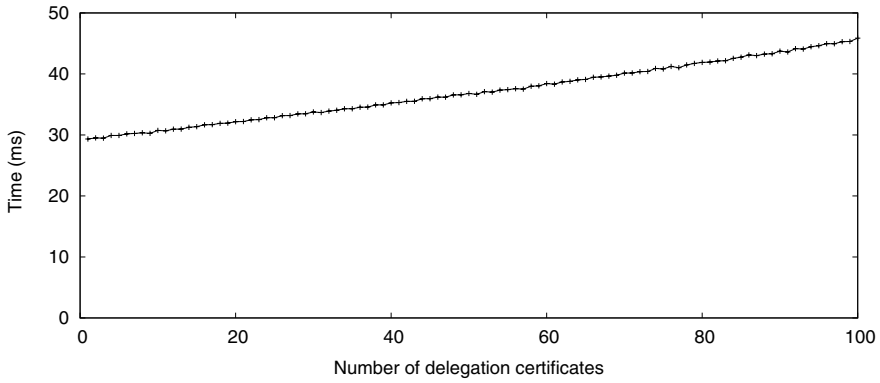


Fig. 10. Execution time of the chain verification algorithm, varying the number of certificate authorities.

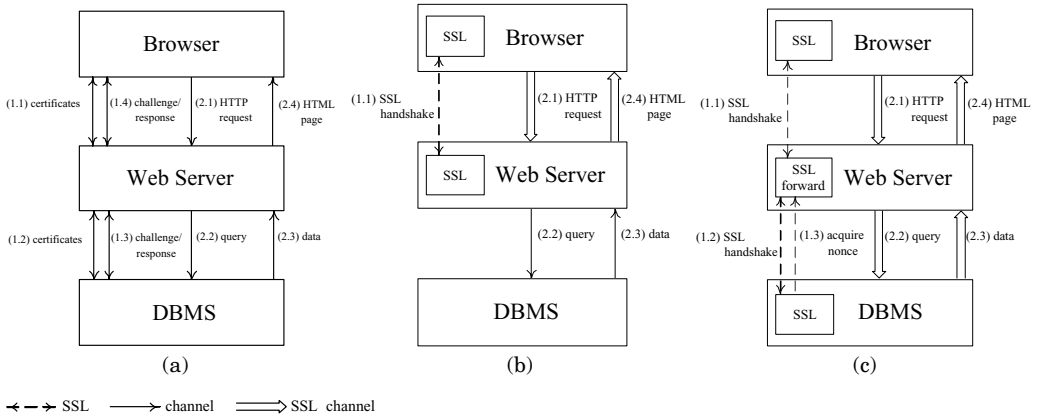


Fig. 11. Interactions between browser, Web server, and DBMS in our design (a); in current SSL-based architectures (b); and in the implementation of our approach over SSL (c).

of certificates retrieved, thus confirming the scalability of the system even in scenarios characterized by complex delegations.

In conclusion, the experiments have shown that the system is able to manage, with a limited overhead, configurations characterized by a significant number of trust tables, trust policies, concurrent sessions, and elements of the certificate chains. The impact on performance from the increase of these parameters was always modest. This confirms that our approach is scalable and does not introduce limitations on the performance of the system.

6. DEPLOYMENT IN CURRENT CREDENTIAL MANAGEMENT INFRASTRUCTURES

The architecture used as a reference for the design presented in this article is the one described in Figure 1. Figure 11(a) illustrates in more detail the interactions between browser, Web server, and DBMS in such a context, also assuming mutual authentication between the browser and the Web server. The Web server represents the platform where both the HTTP server and the Web application are executed, typically separated from the DBMS environment. The architecture in Figure 11(a) is the credential management architecture that is expected to characterize future information systems, with the availability of rich identity management architectures. They will support the flexible transfer among the parties of certified assertions. Relational servers enriched with

trust management components will allow Web application designers to easily build robust and flexible access control over the data the Web applications are interested to access. Future credential management architectures will rely on flexible certificate formats like SAML, or technologies like anonymous credentials, and the design presented in the article is ready to support these novel features. SAML [Lockhart et al. 2007] is an OASIS initiative for the definition of a *Security Assertion Markup Language*, an XML format for the representation of secure assertions. Given an assertion format, classical approaches for the mapping to relational data can be used to define the format of the corresponding trust tables. Anonymous credentials allow the user to expose only a subset of the information contained in the credential (Idemix and U-Prove [Brands 2000; Camenisch and Lysyanskaya 2001] represent the most significant proposals). Anonymous credentials can be well integrated with the design we proposed: it is sufficient to allow attributes in the trust tables to assume null values (for information that may be missing in the certificate).

When considering the current scenario, some restrictions emerge that limit the realization of the architecture in Figure 11(a). Current client-side Web tools, where SSL/TLS is commonly used to support the verification of certificates, require in fact a small adaptation for the integration with the proposed database trust management services. We discuss how to implement our approach with current client-side Web tools in the next section.

6.1. Adaptation to the Current Technological Landscape

Proprietary systems and Web applications targeting a restricted user community may be able to apply today ad-hoc solutions and anticipate the evolution of technology. Instead, Web systems that need to reach a large external user community have to consider as a hard constraint the kind of support available in the client environment. Web browsers today permit access to Web applications in a secure way using HTTPS, that is, HTTP over an SSL [Freier et al. 1996] (or TLS[4] [Dierks and Rescorla 2008]) communication channel. Support is offered for mutual authentication based on the use of client-side and server-side certificates.

Let us consider how the mutual authentication is realized for a Web application using certificates and SSL. Figure 11(b) illustrates the use of SSL in a 3-layer architecture. The user connects with the browser specifying the url of the resource she wants to access, for instance, `https://example.com/resource`. The browser opens an SSL channel to the Web server at the `example.com` address. If the server is configured for mutual authentication (e.g., in Apache it is sufficient to insert the option `SSLVerifyClient require` into the configuration file), the SSL server will ask for a user certificate. The browser asks the user to select it within the set of user certificates (the choice can be made automatic). The SSL handshake protocol is then executed, in which each party, after the verification of the correctness of the exchanged certificates, sends to the other party a selected nonce encrypted with the public key appearing in the certificate (step 1.1). The handshake protocol terminates with the computation of a shared master secret that derives from the values of the nonces chosen by each party, and is the basis for the symmetric key used for encrypting the communication channel. The browser then sends its HTTP request to the Web server on the SSL channel (step 2.1), which forwards the query to the DBMS (step 2.2). The DBMS returns the data in the query result to the Web server (step 2.3), which forwards the HTML page to the browser on the SSL channel (step 2.4). We note that the handshake has been designed to be robust against man-in-the-middle attacks: even knowledge of the private key of one party does

[4]It is common to use the term SSL/TLS to refer to the almost identical SSL and TLS protocols; to simplify the presentation, we only refer to SSL, but what is said can be fully applied to TLS.

not allow another party to compute the master secret, since the nonces required for the computation of the master secret leave each party encrypted only with the public key of the other party. This choice improves security, but it also introduces a rigidity in SSL, which is acknowldged as one of its weaknesses for use in advanced applications.

The security of the trust management system presented in the article depends on the ability of the DBMS to be able to evaluate that the entity having access to the private key corresponding to the public key in the certificate is actually online at the start of the session. This requirement creates the need for the DBMS to be able to realize the SSL handshake directly with the client. There are two obstacles to manage. The first is that the browser only interacts with the HTTPS server. The DBMS is hidden from the client behind the Web server and is not remotely accessible (the configuration of networks according to best practice makes DBMSs inaccessible from external networks). The second and harder obstacle is that database queries are produced by the Web application, not by the client. The client accesses Web pages that present data extracted from the database by SQL queries produced by the Web application.

The solution we designed and implemented is illustrated in Figure 11(c) and relies on forwarding the SSL handshake.[5] We implemented the forwarding service in the SSL management component of the application server (we modified the Java implementation of SSL used by Tomcat[6]). The browser interacts with the Web server (step 1.1), and, for the setup of the SSL connection, the Web server forwards the packets to the DBMS server, monitoring their content (step 1.2). When the handshake ends, we let the Web application have access to the nonce stored within the SSL implementation on the DBMS (step 1.3). To this purpose, we introduced a simple and protected interface to access the server-side nonce in the C implementation offered by OpenSSL. From the knowledge of the nonce, the Web server can derive the master secret and obtain the encryption key, which will be used to both communicate with the client and with the DBMS server. On the DBMS side, at the end of the SSL handshake, the certificate presented by the client is used to establish the client's access privileges to the tables in the database. The certificate is matched with the *trust tables*, and the information within the certificate creates a new tuple in every trust table for which there is a match. *Trust policies* make use of trust tables for the definition of the access privileges of the connected client. After the handshake protocol terminates, the browser can send its HTTP request to the Web server (step 2.1), which forwards it to the DBMS (step 2.2). The DBMS returns the data in the query result to the Web server (step 2.3), which forwards the HTML page to the browser (step 2.4). We note that all these communications exploit the encrypted channel established by the SSL handshake.

6.2. Advantages for the Personal Medical Portfolio Application

To illustrate the advantages of our approach, we consider as a reference application the access to personal medical portfolios, an application currently offered by the public health management services in Lombardy (the Italian administrative region where most of the authors of this article reside). This application for access to personal medical portfolios already requires the use of a smart-card issued by the regional administration and distributed to each of the more than 9 million Lombardy residents. Each smart card contains a certificate associated with the citizen identifier; the smart card can execute computations using the private key when the correct PIN is input by the user. PCs with smart card readers are available in government offices for public use. USB

[5]There are solutions for SSL proxying, also known as SSL tunneling, but they solve the problem of providing SSL protection where the existing application does not natively support SSL. They cannot be used in our scenario.

[6]http://tomcat.apache.org/.

smart card readers are also sold at a subsidized price to facilitate the use of smart cards and to let citizens access their records from their homes. The sensitive data contained in the tables are always associated with the citizen identifier.

A crucial requirement of this application is to offer strong security guarantees in a scenario where the Web application is large and presents many components, and the collection of sensitive data is used by many of its modules. Also, there is the involvement of external auditors that want a guarantee of protection without having to analyze a large volume of code. The Italian Privacy Directive enriches the European Directive and forces organizations managing private medical data to pass examinations by certified auditors, who have to verify that protection measures adequately protect the data.

These requirements are difficult to support using current SSL-based architectures. If a single account on the database is used by the Web application to access the data (as is commonly the case), then vulnerabilities within the Web application can be exploited to get access to the complete collection of protected data. If a DBMS account for each citizen is used to segregate the sensitive data, a significant overhead is imposed in the management of the DBMS, while not increasing assurance that the client who owns the sensitive data is really behind the Web application requesting such data. Vulnerability analysis is a very long and expensive process that can only limit the risks but cannot guarantee that applications are completely free from vulnerabilities. If there is more than one module, the analysis has to be repeated for each module, and vulnerabilities arising from the interaction among the separate modules have to be checked. Finally, the security policy used by the Web application, even if defined in an explicit way, will typically be supported by functions within the application that offer weaker guarantees than what is expected from an internal DBMS component.

Compared to this, both the architectures in Figure 11(a) and in Figure 11(c) permit the realization of an *end-to-end* design, with strong isolation of the protected data from vulnerabilities in the Web server and in the Web application. Also, considering that there are multiple applications and modules accessing the same collection of sensitive data, a centralized definition of the access policy offers more robust guarantees that there will be no way to bypass the protections due to their interaction or to the incomplete representation of security requirements within any of the applications or modules. Auditors will also be able to certify compliance in a more effective and efficient way. An additional benefit of the approach for the application is that a single repository can become the collection point for all medical information, facilitating the construction of applications operating over it. Without robust security guarantees, the integration of a large amount of sensitive data would create a significant risk.

7. RELATED WORK

The two research areas that have a strong relationship with the approach presented in this article are the work on fine granularity access control for databases and the work on trust management.

7.1. Fine Granularity Access Control for Relational Databases

The need of extending and enriching the SQL access control model for supporting fine-grained and expressive access control authorizations has been well recognized. This topic has been considered both in the research and in the industry communities, as testified by several research proposals and by recent releases of major commercial database engines.

The motivation for this interest lies in the perceived need for a more flexible and high-level approach, satisfying the requirements of many important applications, with a specific impact on the Web scenario, often considered the one that most requires

improved support. The overall goal is to obtain access control services characterized by improved flexibility, performance, and robustness.

We synthetically present the most important recent proposals. We will then detail the similarities and the differences with the trust management solution discussed in this article.

Agrawal et al. [2005] proposes a fine-grained access control model on a relational database to support privacy policies. Fine granularity is realized by the use of "restrictions," identifying columns, rows, and cells where access limitations have to be applied. Additionally, as typical for the privacy context, the concept of "purpose" is defined to qualify access to personal data. However, the definition of the restrictions is not fully integrated with the SQL syntax. The restrictions are realized with the use of query rewriting, and the paper presents the mapping between a P3P privacy policy, typically used in the Web to express privacy preferences, and a set of internal tables supporting application of the restrictions.

Kabra et al. [2006] present a fine granularity access control model, focusing on redundancy and information leakage and showing that when query rewriting is used to support restrictions, information leakage can occur. They present a several techniques that can be applied to avoid the leakage, taking into account the impact that a restriction can have on query processing.

In a related line of work, Olson et al. [2008] propose a solution to improve the expressiveness of database access control rules, which allows the formulation of privileges as queries on the data themselves. To this purpose, the authors propose a formal framework (based on Transaction Datalog [Bonner 1997]) to represent reflective policies, that is, policies which may depend on the data included in the database they regulate.

Like our proposal, Chaudhuri et al. [2007] present an extension of the SQL access control model within the Microsoft SQL Server environment. The `grant` statement is enriched with a `where` clause that can be used to restrict (horizontally) the tuples accessible to a user (e.g., a function like `userid()` in the `where` clause can be used to specify that a user is granted access to her own tuples). Although this proposal represents an important step towards the improvement of the flexibility and expressivity of the SQL access control model, it does not support credential-based access control. Our approach goes one step further by illustrating how a trust management service can be integrated with the traditional SQL access control model with a minimal impact on relational DBMSs.

Recently, the major DBMS companies have recognized the importance of incorporating support for certificate-based authentication into DBMSs. For instance, the Oracle Server (since release 10) and the Microsoft SQL Server 2005 emphasize in their documentation the possibility of using certificates and allowing users to establish database connections using SSL/TLS or other PKI-based solutions. However, the information in certificates can at most be used to assign a specific user or role identifier to the session activated with the connection; no support is offered within the system to use certified attributes to specify flexible authorizations. Analogously, PostgreSQL can be combined with OpenSSL to introduce a robust and flexible authentication service; but this mechanism is not integrated with DBMS authorizations.

A few commercial systems have taken some steps toward increasing the flexibility of the access control model. The Oracle Virtual Private Database (VPD) is the approach that received the greatest attention. Steps in this direction have also been taken by the Sybase Server and by the Microsoft SQL Server 2005 Analysis Services, which supports the definition of access privileges on the results of aggregate queries.

Murthy and Sedlar [2007] present a fine-granularity access control solution for Oracle Server VPD, using as a specific application the definition of privileges for access to XML data. The authors illustrate their implementation of the VPD solution in

Oracle, discussing several detailed technical aspects that characterize this approach and improve its efficiency. In terms of design, the implementation adopts various low-level technical solutions that depend on specific aspects of the Oracle platform. For instance, it uses string rewriting to retrieve the user parameters that drive the access control restrictions; the description of how this service is integrated with the optimizer is difficult to extend to other systems. On the other hand, some design aspects have larger applications. For instance, it separates the user parameters from the session, using "lightweight sessions". In this way, the system is able to keep a pool of sessions continuously open, reusing them on subsequent independent user requests. The paper also considers the problem of protecting the data in the application server cache. The approach proposed consists in transferring access restrictions together with the data.

All the motivations justifying the above investigations also apply to our proposal. In general, the solution presented here takes into account the additional requirement of a stricter integration with user authentication based on certificates, which most relational engines already support, but which are kept separate from the specification of the access control policy.

An interesting comparison can be made between our model and the other approaches at a deeper technical level. First, all the proposals above do not discuss the integration with robust authentication services like those offered by credentials. Second, the approach we present uses tables for the storage of user credentials, whereas all the other approaches store the profiles of the users in contextual information available within the session, typically in a flat record. The use of a relational table to store the credentials may cause a performance overhead with respect to storing the profiles of the users in contextual information, due to the need to insert all the data characterizing the user (derived from credentials in our system) into a relational structure. However, the experimental results shown in Section 5.3 confirm that, even for complex policies and sessions presenting a variety of credentials, the execution times required for the processing of the SQL statements responsible for the insertion of the data into the tables produce an acceptable increase in the time required to open a session. The use of relational tables to store credentials also offers the following significant benefits.

—The model is able to manage configurations where a user presents several credentials, whereas in other FGAC solutions, a single collection of scalar parameters is used to characterize the client. The use of a single access profile may satisfy scenarios where the certificates are used as a surrogate for classical user identification, but it is cumbersome to use when the scenario may allow a client to present separate credentials.
—When the information derived from credentials is stored into tables, the definition of authorizations fully benefits from the power and declarativeness of SQL, with the potential to represent complex access restrictions in a compact and clear form, presenting a natural integration with the sophisticated SQL processing architecture of modern relational engines.
—The increase on the time required to open a session directly depends on the number of certificates with different schemas and the number of trust policies. When there is a limited number of trust tables or trust policies, the execution time in our prototype shows a limited impact (see Section 5.3). On the other hand, when the number of trust tables and the number of trust policies increase, the execution time of our prototype also increases. However, in this case the solutions that store user profiles in contextual information would need to be adapted to be able to store all the certificates within a single record in the context session. This modification is however difficult to realize.

7.2. Trust Management

Trust management has received considerable interest in the research community. Much of this research focuses on two main aspects: (i) the development of trust management systems that allow possibly unknown parties to establish trust based on certified information; and (ii) the management of credentials. Early trust management systems (e.g., PolicyMaker [Blaze et al. 1996]; Keynote [Blaze et al. 1999]; and REFEREE [Chu et al. 1997]) associate authorizations with keys rather than with user identities. These systems therefore use credentials to describe specific delegation of trust among keys and to bind public keys to authorizations. Alternative solutions exploit digital certificates to establish properties of their holders (e.g., Bonatti and Samarati [2002]; Irwin and Yu [2005]; Li et al. [2005a, 2005b]; Wang et al. [2004]; Warner et al. [2005]; Winslett et al. [1997]; Yu and Winslett [2003]). In these approaches, the access decision of whether or not a party may execute an access depends on properties that the party may have, and can be proven by presenting one or more certificates. Since parties may be unknown a priori, they undertake a trust negotiation process, gradually releasing to each other credentials and policies (e.g., Lee et al. [2008]; Lee and Winslett [2008b]; Ryutov et al. [2005]; Winsborough and Li [2006]; Winslett et al. [2002]; Yu et al. [2000, 2001, 2003]; Yu and Winslett [2003]). With respect to the credential management, different proposals have addressed the problem of establishing a Public Key Infrastructure, which is at the basis of credential management and of discovering credential chains. Li et al. [2003] focus on the presentation of algorithms for discovering credential chains expressed using a role-based trust management language, called RT_0. The proposed algorithms have the goal of considering only those credentials that are relevant for the discovery of a credential chain, thus minimizing the computational time. The Simple Public Key Infrastructure (SPKI) 2.0 [Ellison 1999; Ellison et al. 1999] is a digital-certificate schema based on public key encryption that allows the integration of certificates in access control models. SPKI is usually adopted in combination with the Simple Distributed Security Infrastructure (SDSI) [Rivest and Lampson 1996], which is a public-key infrastructure used for defining groups and certificates for their membership. SDSI has been designed for distributed systems, exploiting local name spaces instead of hierarchical structures. Deciding whether or not a given principal (or set of principals) is authorized to access a resource may require determining a delegation chain proving that the principal is authorized to access the resource [Clarke et al. 2001]. Li and Mitchell [2006] present a first-order logic semantics for SPKI/SDSI that has been used to analyze the design of SPKI/SDSI. Other complementary works have proposed solutions for verifying the consistency affecting distributed trust-based systems (e.g., Lee and Winslett [2008a]; Lee et al. [2007]) and for verifying security properties (e.g., Reith et al. [2007]).

While providing advancement and interesting solutions for dealing with certificates and managing trust, all the proposals above do not address the problem of enforcement within the underlying data management system. Their solutions are therefore complementary to our approach, which aims at bringing the access control management down at the DBMS level to guarantee flexibility and protection to data accessed via Web services and applications, maintaining a limited computational effort.

8. CONCLUSIONS

Even though trust management mechanisms were proposed a few years ago, their adoption has been limited until now. This is mostly due to the obstacles 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 their main aim, the increase in expressive power, 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 our model with 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 article is designed to be immediately implemented by DBMS producers and used by DBAs. We believe that our solution presents a good balance between functionality and applicability. Indeed, while not enjoying all the functionality of some logic-based trust management approaches, it captures the features and functions that are needed to enable the use of trust management concepts in practice.

APPENDIX

A. ALGORITHM

A.1. NP-Hardness

The problem of computing minimum cost delegation chains supporting all the attributes in certificate *cert* is NP-hard, even for configurations with no authority classes. In fact, the minimum set cover problem reduces to it in polynomial time, as formally stated by the following theorem.

THEOREM A.1 (NP-HARDNESS). *The problem of computing minimum-cost delegation chains supporting all the attributes in a certificate is NP-hard.*

PROOF. The proof is a reduction from the NP-hard problem of the minimum set cover, which can be formulated as follows: given a collection S of subsets of a finite set U, determine a subset $S' \subseteq S$ such that every element in U belongs to at least one set in S' and the cardinality of S' is minimized.

Given a finite set U and a collection S of subsets of U, the problem of computing a minimum set cover for U can be translated into an equivalent instance of the problem of computing a minimum cost supporting chain, as follows. Each element in the finite set U translates to an attribute certified by *cert* and that belongs to TT. Any subset s in S translates to a delegation certificate in *Deleg_Certs*, issued by an authority in the `authoritative` clause of TT, supporting the attributes in s, and with cost equal to 1. The minimum-cost supporting chain for *cert* corresponds to a subset S' of S that completely covers U and with minimum cardinality. □

A.2. Functions Invoked by Function Satisfy

We discuss in detail the functions invoked by function **Satisfy**, briefly described in Section 4, and provide an example of their execution.

CheckClasses. Function **CheckClasses** (see Figure 12) receives as input the authority *auth* to be checked, a trust table or authority class E, and verifies whether *auth* is a member of an authority class in the `authoritative` clause of E (i.e., if *auth* has an authority certificate that directly or indirectly proves that it belongs to a class trusted for E). If this is the case, **CheckClasses** returns the delegation chains proving such a membership; an empty set, otherwise. Variable *min_ac_ver_list* contains the delegation chains with minimum cost, proving that *auth* is a member of a class trusted for E; variable *min* corresponds to the cost of *min_ac_ver_list*; and variable *ver_chain*[*auth*, *AC*] contains the delegation chains proving that *auth* is a member of class *AC*. Initially, variables *min_ac_ver_list* and *min* are set to NULL and ∞, respectively.

```
/* min_ac_ver_list: minimum cost chain proving that auth belongs to a class in E */
/* ver_chain[auth, AC]: minimum cost chain proving that auth belongs to AC */

Input: auth: authority
       E: trust table or authority class
Output: min_ac_ver_list: set of certificates proving that auth is an authority trusted for E

CHECKCLASSES(auth, E): min_ac_ver_list
let AuthoritativeClass be the set of authority classes in Authoritative(E) with the delegation flag set to true
min_ac_ver_list := NULL
min := ∞
for each AC∈AuthoritativeClass do
    if ver_chain[auth, AC]=NULL then /* auth has never been checked for AC */
       min_chain := NULL
       for each authority_cert∈{authority_cert' | authority_cert'∈Authority_Certs ∧
                                    authority_cert'.subject=auth ∧
                                    COMPATIBLE(authority_cert', AC)} do
          ac_ver_list := Satisfy(authority_cert, AC)
          if ac_ver_list≠∅ then
             if (COST(ac_ver_list)<COST(min_chain)) ∨ (min_chain=NULL) then
                min_chain := ac_ver_list
       if min_chain≠NULL then
          /* minimum cost paths for verifying that auth is a member of AC */
          ver_chain[auth, AC] := min_chain
       else
          ver_chain[auth, AC] := ∅ /* auth does not belong to AC */
    if ver_chain[auth, AC]≠∅ then /* auth belongs to AC */
       if COST(ver_chain[auth, AC])<min then
          min_ac_ver_list := ver_chain[auth, AC]
          min := COST(ver_chain[auth, AC])
if min_ac_ver_list≠∅ then
    c.cost := COST(min_ac_ver_list)
    c.issuer := C /* virtual authority */
    c.subject := auth
    c.attributes := Attributes(E)
    Deleg_Certs := Deleg_Certs ∪ {c} /* fictitious edge added to the delegation graph */
auth.ac_visited(E) := TRUE
return (min_ac_ver_list)
```

Fig. 12. Function checking if *auth* belongs to a class in *Authoritative*(*E*).

Let *AuthoritativeClass* be the set of authority classes in *Authoritative*(*E*) with the delegation option. For each authority class *AC* in *AuthoritativeClass*, if *ver_chain*[*auth*, *AC*] is NULL, **CheckClasses** checks whether *auth* is a member of *AC*. Variable *min_chain* is set to NULL and is used to store the chains with minimum cost, supporting the membership of *auth* to *AC*. For each authority certificate *authority_cert* issued for *auth* and compatible with *AC*, **CheckClasses** recursively calls function **Satisfy**. At the end of the recursive call, variable *ac_ver_list* contains a set (possibly empty) of certificates forming the delegation chains for *auth*. If *ac_ver_list* is not empty, *auth* is a member of class *AC*, and therefore if the cost of *ac_ver_list* is less than the cost of the current best solution (*min_chain*), *ac_ver_list* becomes the new best solution and *min_chain* is set to *ac_ver_list*. When all authority certificates compatible with *AC* have been verified, if *min_chain* is different from NULL, variable *ver_chain*[*auth*, *AC*] is set to *min_chain*; otherwise, *auth* is not a member of *AC*, and therefore *ver_chain*[*auth*, *AC*] is set to ∅. If *auth* is a member of *AC* (i.e., *ver_chain*[*auth*, *AC*] is not NULL), **CheckClasses** compares the cost of the delegation chain in *ver_chain*[*auth*, *AC*] with the cost (variable *min*) of the delegation chain that is currently the chain with minimum cost and that proves that *auth* is a member of an authority class in the `authoritative` clause of *E*. If the cost of the delegation chain in *ver_chain*[*auth*, *AC*] is less than *min*, *ver_chain*[*auth*, *AC*]

```
/* Queue: priority queue of edges to be examined */
/* [from, to, p_attrs, p_cost] in Queue: last edge of a path from from to cert.issuer */
/*                                      supporting p_attrs with cost p_cost */
/* predecessor[auth, a]: predecessor in the shortest path from auth to cert.issuer, supporting a */
/* cost[auth, a]: cost of the shortest path from auth to cert.issuer, supporting a */

Input: cert: certificate
       E: trust table or authority class
Output: Candidates: priority queue of the first edges in delegation chains
        [from, A, p_cost] in Candidates: delegation path from a valid authority from to cert.issuer
                                         supporting A with cost p_cost

FINDCHAIN(cert, E): Candidates
ToCheck := cert.attributes∩Attributes(E) /* attributes to be checked for supporting chains */
MAKENULL(Queue)
MAKENULL(Candidates)
for each del_cert∈{c | c∈Deleg_Certs∧ c.subject=cert.issuer} do
    if del_cert.issuer∉Except(E) then
      /* add to Queue all edges outgoing from cert.issuer */
      INSERT([del_cert.issuer, del_cert.subject, del_cert.attributes∩ToCheck, del_cert.cost], Queue)
 /* Dijkstra-like visit of the dynamically built delegation graph */
while ToCheck≠∅ ∧ Queue≠∅ do
    [from, to, p_attrs, p_cost] := EXTRACTMIN(Queue)
    A:= ∅
    for each a∈(p_attrs∩ToCheck) do /* for each attribute still to be verified that belongs */
        if cost[from, a]=NULL then     /* to the extracted edge keep the path with lower cost */
          cost[from, a] := p_cost
          predecessor[from, a] := to
          A := A∪{a}
    if A≠∅ then
      if (from∈Authoritative(E) and has delegation flag set to true) ∨ (from=C) then
          ToCheck := ToCheck − A
          INSERT([from, A, p_cost], Candidates)
      else /* check if from belongs to an authority class in Authoritative(E) */
          if from.ac_visited(E)=FALSE then
            from.ac_visited(E) := CheckClasses(from,E)
          for each del_cert∈{c | c∈Deleg_Certs ∧ del_cert.subject=from} do
              p_attrs := del_cert.attributes∩A
              if p_attrs≠∅ then
                p_cost := p_cost + del_cert.cost
                if del_cert.issuer∉Except(E) then
                  /* add to Queue edges outgoing from from */
                  INSERT([del_cert.issuer,del_cert.subject,p_attrs,p_cost], Queue)
if ToCheck≠∅ then return(∅) /* no chain covering all attributes in ToCheck is found */
else return(Candidates)
```

Fig. 13. Function determining supporting chains.

becomes the new best solution. Therefore, *min_ac_ver_list* is set to *ver_chain*[*auth*, *AC*] and *min* is set to the cost of *ver_chain*[*auth*, *AC*].

When all classes in *AuthoritativeClass* have been processed, if *min_ac_ver_list* is not NULL, **CheckClasses** adds a virtual delegation certificate c to *Deleg_Certs*, where the issuer is a virtual authority $\mathcal{C}$, the subject is *auth*, the cost of the certificate is COST(*min_ac_ver_list*), and the set of certified attributes is *Attributes*(*E*). Finally, flag *auth.ac_visited*(*E*) is set to true, meaning that it has already been analyzed as to whether *auth* is a member of an authority class listed in *Authoritative*(*E*). The function then returns the verification chain *min_ac_ver_list*.

FindChain. Function **FindChain** (see Figure 13) receives as input a certificate *cert*, and a trust table or an authority class *E*. It returns a (possibly empty)

```
/* predecessor[auth, a]: predecessor in the shortest path from auth to cert.issuer, supporting a */

Input: cert: certificate
       E: trust table or authority class
       Candidates: priority queue of the first edges in delegation chains
Output: ver_list: set of certificates proving that cert matches with E

BUILDVERIFICATIONLIST(cert, E, Candidates): ver_list
ToCheck := cert.attributes∩Attributes(E) /* initialize attributes to check for verification */
ver_list := cert /* set of certificates that need validation */
while ToCheck≠∅ ∧ Candidates≠∅ do
    A := ToCheck /* initialize attributes covered by a verified path */
    /* extract the maximum cost verification path */
    [auth, p_attrs, p_cost] := EXTRACTMAX(Candidates)
    if (p_attrs∩ToCheck)≠∅ then
       Let a be any attribute in p_attrs∩ToCheck
       if auth=C then
          ver_list := predecessor[auth, a].ac_ver_list(E)
       repeat
          next := predecessor[auth, a] /* starting from the root and going down to cert */
          del_cert := EXTRACT(Deleg_Certs|issuer=auth ∧ subject=next ∧ a∈attributes)
          ver_list := ver_list∪del_cert /* add the edge to the path */
          A := A∩del_cert.attributes
          auth := del_cert.subject
       until auth=cert.issuer /* stop when reaching cert */
       ToCheck := ToCheck − A
if ToCheck≠∅ then return(∅)
else return(ver_list) /* return the list of certificates for validation */
```

Fig. 14. Function building the set of certificates to be validated.

set of supporting chains for the attributes in variable *ToCheck*, initially set to *cert.attributes*∩*Attributes*(*E*). Finding a supporting chain for an attribute a means finding a path in the delegation graph ending in *cert.issuer* and starting at one of the authorities in *Authoritative*(*E*) or at an authority that belongs to one of the authority classes listed in *Authoritative*(*E*), such that the labels of all edges in the path include attribute a. The process for finding supporting chains is performed via a Dijkstra-like visit of the delegation graph, possibly extended with the virtual certificates. The **while** loop 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) ends in $\mathcal{C}$ or in an authority in the `authoritative` clause of *E*, an element of the form [*from*, *A*, *p_cost*] is added to the *Candidates* queue, where *from* represents the authority from which there is a path reaching *cert.issuer*; *A* is the set of supported attributes; and *p_cost* is the cost of the path. Otherwise, if the authority *from* reached by the path is not a valid authority for *E* and flag *from.ac_visited*(*E*) is false, function **FindChain** calls function **CheckClasses** on authority *from* to check if it belongs to an authority class in the `authoritative` clause of *E*.

At the end of the **while** loop, if *ToCheck* is not empty (i.e., no chain has been found for some attributes), **FindChain** returns an empty set (and consequently function **Satisfy** terminates, returning an empty verification list); it returns priority queue *Candidates*, representing the delegation chains for the attributes of interest, otherwise.

BuildVerificationList. Function **BuildVerificationList** (see Figure 14) receives as input a certificate *cert*, a trust table or an authority class *E*, and a priority queue *Candidates* of delegation chains, and returns the list of certificates that need to be validated, by removing redundant chains from *Candidates*. Variable *ToCheck* is initialized to the set of attributes to be verified (i.e., *cert.attributes*∩*Attributes*(*E*)). To minimize

	ToCheck	Queue	Assignments	Candidates
	$\{n,p,s\}$	$\emptyset$		
cert	$\{n,p,s\}$	$[R,H,\{p\},2]*$ $[S,H,\{s\},2]*$ $[M,H,\{n,s\},4]*$ $[\mathcal{C},H,\{n,p,s\},10]*$		
$[R,H,\{p\},2]$	$\{n,p,s\}$	$[S,H,\{s\},2]$ $[M,H,\{n,s\},4]$ $[\mathcal{C},R,\{p\},5]*$ $[B,R,\{p\},6]*$ $[\mathcal{C},H,\{n,p,s\},10]$	*predecessor*$[R,p]$:=H *cost*$[R,p]$:=2	
$[S,H,\{s\},2]$	$\{n,p,s\}$	$[G,S,\{s\},3]*$ $[M,H,\{n,s\},4]$ $[\mathcal{C},R,\{p\},5]$ $[B,R,\{p\},6]$ $[\mathcal{C},H,\{n,p,s\},10]$	*predecessor*$[S,s]$:=H *cost*$[S,s]$:=2	
$[G,S,\{s\},3]$	$\{n,p\}$	$[M,H,\{n,s\},4]$ $[\mathcal{C},R,\{p\},5]$ $[B,R,\{p\},6]$ $[\mathcal{C},H,\{n,p,s\},10]$	*predecessor*$[G,s]$:=S *cost*$[G,s]$:=3	$[G,\{s\},3]*$
$[M,H,\{n,s\},4]$	$\{n,p\}$	$[\mathcal{C},R,\{p\},5]$ $[G,M,\{n\},5]*$ $[B,R,\{p\},6]$ $[\mathcal{C},H,\{n,p,s\},10]$	*predecessor*$[M,n]$:=H *cost*$[M,n]$:=4 *predecessor*$[M,s]$:=H *cost*$[M,s]$:=4	$[G,\{s\},3]$
$[\mathcal{C},R,\{p\},5]$	$\{n\}$	$[G,M,\{n\},5]$ $[B,R,\{p\},6]$ $[\mathcal{C},H,\{n,p,s\},10]$	*predecessor*$[\mathcal{C},p]$:=R *cost*$[\mathcal{C},p]$:=5	$[\mathcal{C},\{p\},5]*$ $[G,\{s\},3]$
$[G,M,\{n\},5]$	$\{\}$	$[B,R,\{p\},6]$ $[\mathcal{C},H,\{n,p,s\},10]$	*predecessor*$[G,n]$:=M *cost*$[G,n]$:=5	$[G,\{n\},5]*$ $[\mathcal{C},\{p\},5]$ $[G,\{s\},3]$

Fig. 15. Execution of function **FindChain** for Example 4.1.

the number of chains to be evaluated, function **BuildVerificationList** processes the chains in *Candidates* in decreasing order of cost. Indeed, since the starting authorities of the chains in *Candidates* are inserted in increasing order of cost, the chain with the maximum cost is necessary to support at least one attribute a that is not covered by the other chains whose starting point is already in *Candidates*. Therefore, if the most expensive chain extracted from *Candidates* also supports an additional attribute a', the chain associated with a' whose starting point is in *Candidates* will not need to be considered. For each chain, the corresponding certificates are added to the set *ver_list* of certificates that need validation. For edges starting at the virtual authority $\mathcal{C}$ and ending in an authority *auth*, the certificates listed in $auth.ac\_ver\_list(\boldsymbol{E})$ are added to *ver_list*. The attributes certified by a chain of certificates are then removed from *ToCheck*. The process (controlled by the **while** loop) continues until there are no more attributes to be verified (*ToCheck* is empty) or there are no more chains to be processed (*Candidates* is empty). The function then returns the set *ver_list* of certificates, which is in turn returned by function **Satisfy** as the result of the evaluation of certificate *cert*.

Example A.2. With reference to Example 4.1, we illustrate the execution step-by-step of functions **FindChain** and **BuildVerificationList**. Figure 15 illustrates how the content of *ToCheck*, *Queue*, *predecessor*[], *cost*[], and *Candidates* is modified by function **FindChain**. Initially, the set of attributes to be checked is $\{n, p, s\}$ and *Queue* is empty, as indicated in the first line of the table in Figure 15. The effect of the first **for** loop in function **FindChain** is the insertion into *Queue* of all edges entering H, as indicated in the second line of the table. The effect of the **while** loop in function **FindChain** is represented by the other lines of the table that show the elements with minimum cost in the order in which they are extracted from *Queue*, and show

	ToCheck	Candidates	Verification List
	$\{n,p,s\}$	$[G,\{n\},5]$ $[\mathcal{C},\{p\},5]$ $[G,\{s\},3]$	$\emptyset$
$[G,\{n\},5]$	$\{p\}$	$[\mathcal{C},\{p\},5]$ $[G,\{s\},3]$	$[G,M,\{n,s\},1]*$ $[M,H,\{n,s\},4]*$
$[\mathcal{C},\{p\},5]$	$\{\}$	$[G,\{s\},3]$	$[G,M,\{n,s\},1]$ $[M,H,\{n,s\},4]$ $[U,R,\{f\},3]*$ $[R,H,\{p\},2]*$

Fig. 16. Execution of function **BuildVerificationList** for Example 4.1.

how the content of *ToCheck*, *Queue*, *predecessor*[], *cost*[], and *Candidates* changes as each element is processed. Note that an $*$ near an element in *Queue* and *Candidates* indicates that the element has been inserted during the processing of the current extracted element. Furthermore, during the processing of element $[R,H,\{p\},2]$, function **FindChain** calls procedure **CheckClasses** which adds another virtual delegation certificate where the issuer is again the virtual authority $\mathcal{C}$, the subject is R, the attributes are those mentioned in the *Physician* trust table, and the cost is the sum of the costs associated with the dotted edge from U to R. Figure 16 lists the elements with maximum cost in the order in which they are extracted from *Candidates* by function **BuildVerificationList**, and shows how the content of *ToCheck*, *Candidates*, and *ver_list* changes as each element is processed. It is important to note here that the path represented by $[G,\{s\}, 3]$, which is an element inserted into *Candidates* by function **FindChain**, is not visited, since attribute s is already supported by another path that is needed for supporting attribute n.

A.3. Proof of Correctness and Complexity Theorems

We introduce three lemmas used in the proof of Theorem 4.6 on the termination and correctness of our delegation chain verification algorithm.

LEMMA A.3. *Given a finite set of delegation certificates, Deleg_Certs, a finite set of authority certificates Authority_Certs and a certificate cert compatible with an entity E, each element* [*from, to, p_attrs, p_cost*] *in the Queue used by function* **FindChain** *represents a path starting from from, ending to cert.issuer, and supporting attributes p_attrs* $\subseteq$ {*cert.attributes*$\cap$*Attributes(E)*} *with cost p_cost.*

PROOF. We prove this property by induction.

Base Case. This property is satisfied before entering the **while** loop, since *Queue* contains all edges (delegation certificates) entering *cert.issuer* and supporting attributes in *del_cert.attributes*$\cap$*ToCheck*, where *ToCheck* is initialized to *cert.attributes* $\cap$*Attributes*(*E*).

Recursive Case. Suppose that this property is valid for all elements in *Queue*. We now prove that the new elements added to *Queue* always satisfy the above property. Let [*from*, *to*, *p_attrs*, *p_cost*] be the extracted element from *Queue*. By induction, we know that it represents a path from *from* to *cert.issuer* with cost *p_cost*, and supports a set of attributes *p_attrs* $\subseteq$ {*cert.attributes*$\cap$*Attributes*(*E*)}. If there is at least one attribute in {*p_attrs*$\cap$*ToCheck*} for which the extracted path is the current supporting path with minimum cost (i.e., *cost*[*from*, *a*]=NULL), and *from* is neither an authority directly listed in the authoritative clause of *E* nor the virtual authority $\mathcal{C}$, the attribute is added to *A*, where *A* $\subseteq$*p_attrs*, and the edges entering *from* (i.e., the delegation certificates *del_cert* with subject *from*) are added to *Queue*.

For each edge, function **FindChain** then adds to *Queue* an element of the form [*del_cert.issuer*, *from*, *A*∩*del_cert.attributes*, *p_cost*+*del_cert.cost*] which therefore represents a path from *del_cert.issuer* to *cert.issuer*, supporting a set of attributes included in *cert.attributes*∩*Attributes*(*E*). □

LEMMA A.4. *Given a finite set of delegation certificates Deleg_Certs, a finite set of authority certificates Authority_Certs, and a certificate cert compatible with an entity E, function* **FindChain** *terminates and determines correct delegation chains.*

PROOF. We prove the termination of function **FindChain** by induction.

Base Case. Function **FindChain** includes a **for each** loop followed by a **while** loop. The **for each** loop iterates on the delegation certificates in *Deleg_Certs* with subject *cert.issuer*. Since *Deleg_Certs* is finite, the **for each** loop terminates. The **while** loop terminates when either *ToCheck* becomes empty or *Queue* becomes empty. Initially, *ToCheck* contains all attributes in *cert.attributes*∩*Attributes*(*E*) and *Queue* contains all edges (delegation certificates) entering *cert.issuer*. At each iteration, the number of attributes in *ToCheck* may only decrease, one element is extracted from *Queue*, and new elements may be added to *Queue*. In particular, since the attributes in *ToCheck* are removed when **FindChain** has found a supporting path for them, two cases may occur. In the first case, **FindChain** finds a supporting path for all attributes in *ToCheck*, and therefore it becomes empty. In the second case, there is at least one attribute such that the supporting path does not exist. The **while** loop therefore terminates only if *Queue* becomes empty, and we then need to prove that the number of elements inserted into *Queue* is finite.

At each iteration of the **while** loop, an element [*from*, *to*, *p_attrs*, *p_cost*] with minimum cost is extracted from *Queue*. If for all attributes *p_attrs*, function **FindChain** has already found a delegation chain (i.e., *p_attrs*∩*ToCheck* is empty), no element is added to *Queue*. Otherwise, *p_attrs*∩*ToCheck* is not empty. Now let *a* be an attribute in {*p_attrs*∩*ToCheck*}. If the extracted element represents an edge in the first path supporting *a* and passing through *from* (Lemma A.3), attribute *a* is added to *A* and the predecessor of *from* becomes authority *to* (i.e., *predecessor*[*from*, *a*] := *to*), thus correctly building the delegation path. Otherwise, if function **FindChain** has already analyzed another path supporting *a* and passing through *from*, *p_cost* must be greater than or equal to the cost of the other path, since, at each iteration of the **while** loop, the path with minimum cost is extracted. The edges entering *from* are possibly added to *Queue* only if *A* is not empty (in the worst case this may happen as many times as the number of attributes in *cert*) and *from* is neither an authority directly listed in the authoritative clause of *E* nor the virtual authority $\mathcal{C}$. If *A* is not empty and *from* is an authority directly listed in the authoritative clause of *E* or corresponds to $\mathcal{C}$, the set *ToCheck* is modified by removing the set of attributes *A* and no element is added to *Queue*. Since however a finite number of elements are inserted into *Queue* and at each iteration of the **while** loop an element is extracted from *Queue*, the **while** loop terminates. If after the termination of the **while** loop *ToCheck* is not empty, function **FindChain** returns an empty set.

Recursive Case. We first note that function **FindChain** terminates if **CheckClasses** terminates. Function **CheckClasses** is composed of two nested **for each** loops, which iterate on a finite number of authority classes and authority certificates, and therefore the operations inside these two loops are executed a finite number of times. In particular, we are interested in the number of calls to function **Satisfy**. Given an authority *auth*, and an authority class *AC*, the membership of *auth* to *AC* is checked only if *ver_chain*[*auth*, *AC*] is NULL and there exists at least one authority certificate issued for *auth* and compatible with *AC*. In particular, the number of times that function **Satisfy**

is called for checking whether *auth* is a member of *AC* is equal to the number of authority certificates issued for *auth* and compatible with *AC*. Since the delegation graph is acyclic, the recursive call to **Satisfy** cannot reach *auth* again. Furthermore, since the number of authority classes and delegation certificates is finite and since each call to function **Satisfy** visits a different portion of such a graph, the recursive call to **Satisfy** terminates (provided function **Satisfy** terminates, as proved by Theorem 4.6). When the inner **for each** loop also terminates, *ver_chain*[*auth*, *AC*] becomes different from NULL and therefore the membership of *auth* to *AC* will not be checked any more. □

LEMMA A.5. *Given a finite set of delegation certificates Deleg_Certs, a finite set of authority certificates Authority_Certs, and a certificate cert compatible with an entity E, function* **BuildVerificationList** *terminates.*

PROOF. Function **BuildVerificationList** is composed of a **while** loop and an innermost **repeat-until** loop. The **while** loop terminates when either *ToCheck* becomes empty or *Candidates* becomes empty. Initially, *ToCheck* contains all attributes in *cert.attributes*∩*Attributes*(*E*) and *Candidates* contains a finite set of elements of the form [*auth*, *p_attrs*, *p_cost*] that represents a path staring from *auth* and ending in *cert.issuer*, with cost *p_cost*, and supporting attributes *p_attrs* (Lemma A.4). Since at each iteration of the **while** loop, an element [*auth*, *p_attrs*, *p_cost*] with maximum cost is extracted from *Candidates* and no element is added, *Candidates* will become empty. The innermost **repeat-until** loop is executed whenever the extracted element represents a path supporting at least one attribute *a* for which the corresponding delegation chain has not been visited yet, that is, $a \in p\_attrs \cap ToCheck$. The **repeat-until** loop follows the corresponding path by exploiting the *predecessor* global variable, and the corresponding delegation certificates are added to *ver_list*. Since function **FindChain** correctly builds the delegation chains through variable *predecessor* (Lemma A.4) and the graph is acyclic, at each iteration the **repeat-until** loop visits an edge of the delegation chain until *cert.issuer* is reached. Consequently, the **repeat-until** loop terminates, and if *ToCheck* is empty, the function returns the set *ver_list* of certificates supporting attributes in *cert.attributes*∩*Attributes*(*E*). □

THEOREM 4.6 (TERMINATION AND CORRECTNESS). *Given a finite set of delegation certificates Deleg_Certs, a finite set of authority certificates Authority_Certs, and a certificate cert compatible with an entity E, function* **Satisfy** *terminates and determines a correct set of delegation chains for cert, if such delegation chains exist.*

PROOF. We first prove that the function terminates and then that if it returns a nonempty set of certificates, they correspond to delegation chains that support all attributes in *cert.attributes*∩*Attributes*(*E*).

The preliminary operations of function **Satisfy** check whether or not *cert.issuer* appears in *Except*(*E*) and *Authoritative*(*E*). Since such sets are finite, the preliminary operations terminate. Analogously, since both function **FindChain** and function **BuildVerificationList** terminate (Lemma A.4 and Lemma A.5); both Phase 1 and Phase 2 of function **Satisfy** also terminate.

We now prove by contradiction that if there are delegation chains supporting all attributes in *cert.attributes*∩*Attributes*(*E*), function **Satisfy** returns them. Suppose that all attributes in *cert.attributes*∩*Attributes*(*E*) are supported by a unique delegation chain and let *s* be the set of delegation certificates forming the delegation chains for such attributes. Suppose also that function **Satisfy** returns an empty set. In this case, solution *s* cannot coincide with *cert*, since **Satisfy** checks if *cert* represents a solution to the problem. Then set *s* has to include more than one certificate. Since function **Satisfy** returns the empty set, function **FindChain** returned an empty *Candidates* list. Therefore, the **while** loop in function **FindChain** terminates because *Queue* becomes

empty, while *ToCheck* includes at least one attribute a. By assumption, we know that there is a supporting path p for attribute a also, going from a valid authority for E to *cert.issuer*. By Lemma A.3, the edge in p entering *cert.issuer* and represented by element [*from*, *cert.issuer*, *p_attrs*, *p_cost*] is inserted into *Queue* before the **while** loop. Since, at the end of the **while** loop, *Queue* is empty, within an iteration of the **while** loop this edge is extracted from *Queue*, and therefore any edge entering *from*, including the edge in path p, is inserted into *Queue*. By recursively applying this observation, we can conclude that all edges in p are evaluated and added to *Queue* until the function reaches an authority directly listed in the authoritative clause of E or the virtual authority $\mathcal{C}$. Attribute a is then removed from *ToCheck*, thus contradicting our assumption that a remains in *ToCheck*.

Function **Satisfy** can also return an empty set if function **BuildVerificationList** returns an empty *ver_list*; that is, if the function cannot reconstruct a delegation chain for all the attributes in *cert.attributes*∩*Attributes*(E). Suppose that a is the attribute for which it cannot reconstruct the delegation chain. Since a belongs to *cert.attributes*∩*Attributes*(E), function **FindChain** has removed a from *ToCheck* and added an element to *Candidates* containing a. Therefore, such an element is extracted from *Candidates* by function **BuildVerificationList** because, by assumption, there is only a unique path supporting a, and this path is reconstructed, since at each iteration of the **repeat-until** loop, variable *next* is assigned to the *predecessor* associated with *auth* for a. For Lemma A.3, *cert.issuer* is reached by the **repeat-until** loop and *ToCheck* is updated removing a. □

THEOREM 4.7 (COMPLEXITY). *Function* **Satisfy** *finds delegation chains for a certificate cert compatible with an entity E in* $O(N_d \cdot N_E \cdot N_{\mathcal{AC}} \cdot N_{auth} \cdot \log(N_d \cdot N_E))$.

PROOF. The complexity of function **Satisfy** is obtained by evaluating the complexity of the preliminary checks and of the two phases composing it. We first evaluate the complexity of **Satisfy** in the base case, thus assuming that function **CheckClasses** is never called.

Preliminary Checks. The preliminary checks in **Satisfy** require (in total) time proportional to |*Authoritative*(E)| + |*Except*(E)|.

Phase 1. The cost of this phase is the cost of function **FindChain**. The **for each** loop of the function requires time proportional to N_d, and the cost of the **while** loop of the function requires time proportional to the number of iterations of such a loop. In the worst case, the **while** loop terminates when *Queue* is empty. As already noted, any certificate (edge) in *Deleg_Certs* is inserted into *Queue* at most as many times as the number of attributes N_E. This implies that the maximum number of elements in *Queue* is $N_d \cdot N_E$. All operations performed within the **while** loop have a constant cost, but the INSERT operation on priority queue *Queue*, whose cost is $O(\log(N_d \cdot N_E))$. We can then conclude that the cost of this first phase is proportional to $N_d \cdot N_E \cdot \log(N_d \cdot N_E)$.

Phase 2. The cost of this phase is the cost of function **BuildVerificationList** that visits (a subset of) the paths found in the previous phase. The cost of this second phase is then proportional to N_d.

Overall, the time complexity is proportional to |*Authoritative*(E)| + |*Except*(E)| + $N_d \cdot N_E \cdot \log(N_d \cdot N_E) + N_d$. If we assume that all operations performed by function **Satisfy** have a constant cost and c_{max} is the maximum cost, the time complexity is in $O(c_{max} \cdot N_d \cdot N_E \cdot \log(N_d \cdot N_E)) = O(N_d \cdot N_E \cdot \log(N_d \cdot N_E))$.

Consider now the recursive execution of function **Satisfy**. As already noted, in the worst case, the function is recursively called $N_{\mathcal{AC}} \cdot N_{auth}$ times. Since the time complexity

of each call is $O(N_d \cdot N_E \cdot \log(N_d \cdot N_E))$ and in the worst case the whole delegation graph is visited, the overall time complexity is then $O(N_d \cdot N_E \cdot N_{AC} \cdot N_{auth} \cdot \log(N_d \cdot N_E))$. □

REFERENCES

AGRAWAL, R., BIRD, P., GRANDISON, T., KIERNAN, J., LOGAN, S., AND RJAIBI, W. 2005. Extending relational database systems to automatically enforce privacy policies. In *Proceedings of the 21st International Conference on Data Engineering*.

BLAZE, M., FEIGENBAUM, J., IOANNIDIS, J., AND KEROMYTIS, A. 1999. The KeyNote trust management system (version 2). Internet RFC 2704. http://www.crypto.com/papers/rfc2704.txt.

BLAZE, M., FEIGENBAUM, J., AND LACY, J. 1996. Decentralized trust management. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE, Los Alamitos, CA.

BONATTI, P. AND SAMARATI, P. 2002. A unified framework for regulating access and information release on the Web. *J. Comput, Secur. 10,* 3, 241–272.

BONNER, A. 1997. Transaction datalog: A compositional language for transaction programming. In *Proceedings of the 6th International Workshop on Database Programming Languages*.

BRANDS, S. 2000. *Rethinking Public Key Infrastructure and Digital Certificate*s. MIT Press, Cambridge, MA.

CAMENISCH, J. AND LYSYANSKAYA, A. 2001. An efficient system for non-transferable anonymous credentials with optional anonymity revocation. In *Proceedings of the 20th Annual International Conference on the Theory and Applications of Cryptographic Techniques*.

CERI, S., FRATERNALI, P., PARABOSCHI, S., AND TANCA, L. 1994. Automatic generation of production rules for integrity maintenance. *ACM Trans. Datab. Syst. 19,* 3, 367–422.

CHAUDHURI, S., DUTTA, T., AND SUDARSHAN, S. 2007. Fine-grained authorization through predicated grants. In *Proceeding of the 23rd IEEE International Conference on Data Engineering*. IEEE, Los Alamitos, CA.

CHU, Y., FEIGENBAUM, J., LAMACCHIA, B., RESNICK, P., AND STRAUSS, M. 1997. REFEREE: Trust management for Web applications. *World Wide Web J.l 2,* 3, 127–139.

CLARKE, D., ELIEN, J., ELLISON, C., FREDETTE, M., MORCOS, A., AND RIVEST, R. 2001. Certificate chain discovery in SPKI/SDSI. *J.Comput. Secur. 9,* 4, 285–322.

DE CAPITANI DI VIMERCATI, S., JAJODIA, S., PARABOSCHI, S., AND SAMARATI, P. 2007. Trust management services in relational databases. In *Proceedings of the 2nd ACM Symposium on Information, Computer and Communications Security*. ACM, New York.

DIERKS, T. AND RESCORLA, E. 2008. The transport layer security (TLS) protocol (version 1.2). Internet RFC 5246. http://tools.ietf.org/rfc/rfc5246.txt.

ELLISON, C. 1999. SPKI requirements. Internet RFC 2692. http://www.ietf.org/rfc/rfc2692.txt.

ELLISON, C., FRANTZ, B., LAMPSON, B., RIVEST, R., THOMAS, B., AND LONEN, T. 1999. SPKI certificate theory. Internet RFC 2693. http://www.ietf.org/rfc/rfc2693.txt.

FREIER, A. O., KARLTON, P., AND KOCHER, P. C. 1996. The SSL protocol (version 3.0). Netscape's final SSL3.0 draft. http://www.mozilla.org/projects/security/pki/nss/ssl/draft302.txt.

HOUSLEY, R., POLK, W., FORD, W., AND SOLO, D. 2002. Internet X.509 public key infrastructure certificate and CRL profile. Internet RFC 3280. http://www.ietf.org/rfc/rfc3280.txt.

IRWIN, K. AND YU, T. 2005. Preventing attribute information leakage in automated trust negotiation. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*. ACM, New York.

ISO. 1996. Database language SQL – part 2: Foundation (SQL/foundation) 1999. ISO International Standard, ISO/IEC9075.

KABRA, G., RAMAMURTHY, R., AND S. SUDARSHAN, S. 2006. Redundancy and information leakage in fine-grained access control. In *Proceeding of the 2006 ACM SIGMOD International Conference on Management of Data*. ACM, New York.

LEE, A., MINAMI, K., AND WINSLETT, M. 2007. Lightweight consistency enforcement schemes for distributed proofs with hidden subtrees. In *Proceedings of the 12th ACM Symposium on Access Control Models and Technologies*. ACM, New York.

LEE, A. AND WINSLETT, M. 2006. Safety and consistency in policy-based authorization systems. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*. ACM, New York.

LEE, A. AND WINSLETT, M. 2008a. Enforcing safety and consistency constraints in policy-based authorization systems. *ACM Trans. Inf. Syst. Secur. 12,* 2, 1–33.

LEE, A. AND WINSLETT, M. 2008b. Towards an efficient and language-agnostic compliance checker for trust negotiation systems. In *Proceedings of the 3rd ACM Symposium on Information, Computer and Communications Security*. ACM, New York.

LEE, A., WINSLETT, M., BASNEY, J., AND WELCH, V. 2008. The trust authorization service. *ACM Trans. Inf. Syst. Secur. 11,* 1, 1–33.

LI, J., LI, N., AND WINSBOROUGH, W. 2005a. Automated trust negotiation using cryptographic credentials. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*. ACM, New York.

LI, N., MITCHELL, J., AND WINSBOROUGH, W. 2005b. Beyond proof-of compliance: Security analysis in trust management. *J. ACM 52,* 3, 474–514.

LI, N. AND MITCHELL, J. 2006. Understanding SPKI/SDSI using first-order logic. *Int. J, Inf. Secur. 5,* 1, 48–64.

LI, N., WINSBOROUGH, W., AND MITCHELL, J. 2003. Distributed credential chain discovery in trust management. *J. Comput. Secur. 11,* 1, 35–86.

LOCKHART, H., WISNIEWSKI, T., CANTOR, S., MISHRA, P., AND LIEN, J. 2007. Security assertion markup language (SAML) V2.0 tech. overview. OASIS working draft. http://www.oasisopen.org/committees/download.php/22553/sstc-saml-tech-overview-2200-draft-13.pdf.

LOWE, G. 1996. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Proceedings of the 2nd International Workshop on Tools and Algorithms for Construction and Analysis of Systems*.

MURTHY, R. AND SEDLAR, E. 2007. Flexible and efficient access control in Oracle. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM, New York.

NEEDHAM, R. M. AND SCHROEDER, M. D. 1978. Using encryption for authentication in large networks of computers. *Comm. ACM 21,* 12, 993–999.

OLSON, E., GUNTER, C., AND MADHUSUDAN, P. 2008. A formal framework for reflective database access control policies. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*. ACM, New York.

REITH, M., NIU, J., AND WINSBOROUGH, W. 2007. Apply model checking to security analysis in trust management. In *Proceedings of the 23rd International Workshop on Data Engineering*.

RIVEST, R. AND LAMPSON, B. 1996. SDSI - A simple distributed security infrastructure. http://people.csail.mit.edu/rivest/sdsi10.html.

RYUTOV, T., ZHOU, L., NEUMAN, C., LEITHEAD, T., AND SEAMONS, K. 2005. Adaptive trust negotiation and access control. In *Proceedings of the 10th ACM Symposium on Access Control Models and Technologies*. ACM, New York.

SALTZER, J. AND SCHROEDER, M. 1975. The protection of information in computer systems. *Proc. IEEE 63,* 9, 1278–1308.

STONEBRAKER, M. 1987. The design of the POSTGRES storage system. In *Proceedings of the 13th International Conference on Very Large Data Bases*.

WANG, L., WIJESEKERA, D., AND JAJODIA, S. 2004. A logic-based framework for attribute based access control. In *Proceedings of the ACM Workshop on Formal Methods in Security Engineering*. ACM, New York.

WARNER, J., ATLURI, V., AND MUKKAMALA, R. 2005. An attribute graph-based approach to map local access control policies to credential based access control policies. In *Proceedings of the 1st International Conference on Information Systems Security*.

WINSBOROUGH, W. AND LI, N. 2006. Safety in automated trust negotiation. *ACM Trans. Inf. Syst. Secur. 9,* 3, 352–390.

WINSLETT, M., CHING, N., JONES, V., AND SLEPCHIN, I. 1997. Using digital credentials on the World Wide Web. *J. Comput. Secur. 5,* 3, 255–267.

WINSLETT, M., YU, T., SEAMONS, K., HESS, A., JACOBSON, J., JARVIS, R., SMITH, B., AND YU, L. 2002. Negotiating trust on the Web. *IEEE Internet Comput. 6,* 6, 30–37.

YU, T., MA, X., AND WINSLETT, M. 2000. PRUNES: An efficient and complete strategy for automated trust negotiation over the internet. In *Proceedings of the 7th ACM Conference on Computer and Communications Security*. ACM, New York.

YU, T. AND WINSLETT, M. 2003. A unified scheme for resource protection in automated trust negotiation. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE, Los Alamitos, CA.

YU, T., WINSLETT, M., AND SEAMONS, K. 2001. Interoperable strategies in automated trust negotiation. In *Proceedings of the 8th ACM Conference on Computer and Communications Security*. ACM, New York.

YU, T., WINSLETT, M., AND SEAMONS, K. 2003. Supporting structured credentials and sensitive policies through interoperable strategies for automated trust negotiation. *ACM Trans. Inf. Syst. Secur. 6,* 1, 1–42.

Received January 2009; revised August 2011; accepted October 2011