Trust Management in P2P systems using Standard TuLiP

Marcin Czenko, Jeroen Doumen, and Sandro Etalle

Abstract In this paper we introduce Standard TuLiP - a new logic based Trust Management system. In Standard TuLiP, security decisions are based on security credentials, which can be issued by different entities and stored at different locations. Standard TuLiP directly supports the distributed credential storage by providing a sound and complete Lookup and Inference AlgoRithm (LIAR). In this paper we focus on (a) the language of Standard TuLiP and (b) on the practical considerations which arise when deploying the system. These include credential encoding, system architecture, system components and their functionality, and also the usability issues.

1 Introduction

In the context of the I-Share project [7] we are developing a security infrastructure for secure content sharing in P2P networks, and for secure internet TV. The underlying idea is that users of a P2P system organise themselves in so-called virtual communities [10] sharing tastes, interests, or business objectives. These virtual communities need a highly decentralised yet fine-grained policy enforcement system to protect data from undesired disclosure. Traditional approaches based on access control are not adequate as they expect that the system entities can be statically

Marcin Czenko
Department of Computer Science
University of Twente, The Netherlands , e-mail: marcin.czenko@utwente.nl

Jeroen Doumen
Department of Computer Science
University of Twente, The Netherlands, e-mail: jeroen.doumen@utwente.nl

Sandro Etalle
Eindhoven University of Technology and
University of Twente, The Netherlands, e-mail: s.etalle@tue.nl

enumerated and assigned the appropriate privileges in advance. In virtual communities the users often do not know each other and need to have reason to *trust* other peers before taking an action. It was shown [8] that trust is a significant factor in the success of such systems.

The obvious choice was to have a policy specification and enforcement system based on *Trust Management* [4, 5, 12, 15], in which security decisions are based on security *credentials*. In Trust Management a credential represents a permission or a capability assigned by the credential issuer to the credential subject. Credentials can be simple *facts* assigning a specific permission or a *role* to a specific user, or they can express more sophisticated *rules* describing roles of groups of users without enumerating their members. An important feature of Trust Management is that credentials can be issued by different authorities and stored at different locations. The user can access a resource if it can be proven that she has a certain role. This is done by evaluating a chain of credentials.

In the setting of our project, the trust management system has to meet the following (rather common) requirements: First, the policy language should be simple (possibly based on a well-known language), extensible (e.g. by interfacing it with external components, like constraint solvers), it should allow calculations and should be able to express complex policies. Secondly, the underlying architecture should be completely decentralised; in particular, the credential storage should be not only decentralised, but one should be able to determine whether a credential should be stored by the issuer or by some other entity (e.g. the so-called subject of the credential), like in the RT [13] trust management system. Thirdly, the system should enjoy a sound and complete decision algorithm, i.e. an algorithm which - in spite of the decentralised storage of the credentials - will always be able to make the appropriate decision and deliver the correct chain of credentials supporting it, if one exists (more about this later).

Present TM systems do not satisfy all three conditions; RT [13, 12] comes very close by satisfying the second and third requirements, but at the cost of a syntax which is too inflexible for our purposes (see our [6] for a discussion on this). Other systems either do not support decentralised storage or do not enjoy a sound and complete decision algorithm (see the Related Work section for the details).

To meet all requirements, we have developed a new trust management system: Standard TuLiP. Standard TuLiP is based on the theoretical basis laid in Core TuLiP [6], i.e. on the same concept of credential storage system and on a similar decision algorithm (which in turn is inspired by the architecture of RT [13]). But while Core TuLiP is more or less a theoretical exercise based on a very restricted syntax, Standard TuLiP is a full fledged Trust Management system with not only a more flexible syntax, but with the support of a whole distributed infrastructure, with APIs for the specification, the validation and the storage of the credentials, APIs for interrogating the decision procedure and a number of changes w.r.t. Core TuLiP which make it amenable for a practical deployment (to mention one, the choice of including the mode in the credential specification, which allows to reduce dramatically the workload of the lookup algorithm).

In this paper we present the Standard TuLiP system. In particular, we concentrate on the practical issues related to its deployment and use. We start with Sect. 2 where we introduce the XML syntax of Standard TuLiP credentials and policies and how they are represented in the logic programming form. In Sect. 3 we show how we can specify the credential storage by using modes. We introduce the notion of *traceable* credentials in order to guarantee that all required credentials will be found later when needed in a proof. Section 4 deals with the architecture of Standard TuLiP. We introduce basic components, show their functionality and also say how they communicate with each other. In particular we show how credentials are stored and how we find them. Then, in Sect. 5 we show the system from the user perspective: we answer questions like how to write credentials, send queries, and we also discuss the problem of credential and user identifier revocation. We finish the paper with Related Work in Sect. 6 and Conclusions and Future Work in Sect. 7.

2 Policies

Standard TuLiP is a credential-based, role-based Trust Management system. Informally, a credential is a signed statement determining which role can be assigned to an entity. A role can then be further associated with permissions, capabilities, or actions to be performed. For example, the University of Twente may issue a credential saying that *Alice* is a student of it, which directly or indirectly may give Alice a certain set of permissions (like buying a book in an online store at a discount price). Here, the University of Twente is called the *issuer* of the credential, Alice is called its *subject*, and student is the *role name*. A credential is always signed by its issuer, as it is the issuer who has the authority of associating certain rights with the subject. A credential can also contain additional information about the subject. For instance, a student usually has a student number, she belongs to a certain department, etc. This information is stored in the *properties* section of a credential. Standard TuLiP uses XML [21] as a language for credential representation. The use of XML is convenient for several reasons. Firstly, XML is a widely accepted medium for electronic data exchange and is widely supported by many commercial and free tools. Secondly, the use of XML namespaces [22] can help in avoiding name conflicts and facilitates the definition of common vocabulary.

We distinguish two types of credentials: the basic credential, and the conditional credential. The first is just a direct role assignment (e.g. “Alice is a student”), while the latter can express role assignments under some constraints.

Basic credentials. Figure 1 shows the XML encoding of a basic Standard TuLiP credential, which consists of a single *credential* XML element. The *credential* XML element, in turn, contains a single *permission* XML element, which consists of a *role name*, *mode*, *issuer*, *subject*, and optionally *properties* XML elements. The meaning of the *mode* XML element will be explained later in this paper. The *issuer* element consists of a single *entityID* element which contains a public identifier of the credential issuer. Similarly, the *subject* element contains the *entityID* element containing

```
<?xml version="1.0" encoding="UTF−8"?>
<credential xmlns="urn:ewi:namespaces:tulip"
notBefore="2007−02−12T20:00:00" notAfter="2008−02−12T20:00:00">
    <permission>
        <rolename>student</rolename>
        <mode>oi</mode>
        <issuer><entityID>ut−pub−key</entityID></issuer>
        <subject><entityID>alice−pub−key</entityID></subject>
        <properties>
            <studentid>0176453</studentid>
            <department>ewi</department>
            <study>cs</study>
        </properties>
    </permission>
    <Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
        <SignedInfo>
            ...
            <Reference URI="">
                ...
                <DigestValue>5WlwStu5ouu94nb5rwQ6BhFOPWc=</DigestValue>
            </Reference>
        <SignedInfo>
        <SignatureValue>signature−value</SignatureValue>
    </Signature>
</credential>
```

Fig. 1: A basic Standard TuLiP XML credential.

the public identifier of the subject. The optional *properties* XML element can include arbitrary XML content describing additional properties of the issuer and/or the subject. Every credential includes the time period in which it is considered valid - this is done by using the *notBefore* and *notAfter* attributes of the "credential" XML element. Each Standard TuLiP credential is signed by the issuer's private key. Standard TuLiP uses public (RSA) keys as public identifiers. By doing this, every credential can be immediately validated without the need for an external PKI infrastructure. The signature is contained in the *Signature* XML element. We use the enveloped XML signature format [20] (more precisely, a digest value is computed over the top-level element, which is then included in the DigestValue element of the SignedInfo element of the signature, and then the signature is made of the SignedInfo element and included in the SignatureValue element). Notice also the use of the *urn:ewi:namespaces:tulip* namespace in the top level credential XML element. This is required for every valid Standard TuLiP credential. By using namespaces, Standard TuLiP credentials can be easily distinguished from other credential formats and this even allows for different credential formats to be mixed in a single XML document.

Conditional credentials. Sometimes we need more sophisticated types of statement: consider an online store which gives a discount to students of the University

of Twente. Instead of giving each student a basic credential granting the discount, it is much more efficient to associate the discount role to everyone who has a student role at the University of Twente, using a variable as subject.

Conditional Standard TuLiP credentials contain an additional *provided* XML element. This element includes one or more *condition* elements which specify additional conditions that must be satisfied before the specified role name can be associated with the credential subject. A *condition* XML element is similar to the *permission* XML element in that it also contains the role name, mode, issuer, subject and optional properties XML elements. In the condition XML element, the issuer, subject, and properties XML elements can contain variables referring to the elements from the preceding conditions and/or to the subject and properties elements from the permission XML element. Notice that the credential issuer cannot contain a variable as the issuer of the credential must always be known (otherwise one would not be able to verify the credential signature). The *provided* part of the credential can also contain a constraint which in turn can refer to built-in functions (to manipulate values taken by the variables). The presence of variables allows us to interface easily with external functions (e.g. arithmetic solvers, constraint solvers, programs written in other languages). The only requirement is that these calls should respect the input-output flow dictated by the mode of the credential; modes are discussed later and it is beyond the scope of this paper to explain in detail how the interfacing with external functions takes place.

A Standard TuLiP security *policy* is defined by a set of credentials.

Queries. When *eStore* wants to check if *alice* is a student of the University of Twente it sends a *query* to the University of Twente. In Standard TuLiP, queries are also encoded as XML documents. The structure of an XML representation of a query is very similar to that of the provided part of a Standard TuLiP XML credential. The top-level element is the *query* XML element. It consists of a one or more condition XML elements each of which contains the role name, mode, issuer, subject, and optionally the properties XML elements. If a query contains only one condition element we call it a *basic query*. Besides the query conditions, every query reports the public identifier of the entity making the query. Each Standard TuLiP query also contains a unique *ID* and *IssueInstant* attributes inside the top-level query XML element. The ID attribute allows the system to check whether the received response corresponds to the earlier issued query. The IssueInstant attribute carries the time and date of the request which allows the responding entity to filter out erroneous requests (like the ones with the time in the future), or to check whether the time matches the validity of the credentials used in answering the query.

A query is always about a specific set of permissions. However, they can be of different types. For instance “Is *alice* a student of the University of Twente?” and “Give me all the students of the University of Twente” are queries of different type. In Standard TuLiP the policy writer can restrict the type of queries one can ask by using modes. We discuss this in Sect. 3.

Semantics. In order to give Standard TuLiP credentials formal meaning they are translated to the equivalent logic programming form. In this representation every

credential is represented by a definite clause containing one or more the so called *credential atoms* and/or *built-in constraints*.

Definition 1 (credential atoms,credentials,queries). A *credential atom* is a predicate of the form:

$$rolename(issuer, subject, properties).$$

A *credential* is a definite clause of the form $P \leftarrow C_1, \ldots, C_n$, where P is a credential atom, and $C_1, \ldots, C_n$ are credential atoms or built-in constraints. The credential atom P in the head of the clause corresponds to the permission XML element and every credential atom or built-in constraint C_i in the body of the clause corresponds to a condition in the provided part of the corresponding XML credential encoding. A query is represented by a sequence of credential atoms and/or built-in constraints $C_1, \ldots, C_n$, where each C_i corresponds to a query condition.

The var XML elements are straightforwardly mapped to logical variables. For space reasons, we do not show the actual mapping from the content of the properties XML element to the corresponding logic programming term.

A *policy* is a logic program containing one or more credentials.

Example 1. The policy modelling the scenario presented above is represented by the following logic program:

$$student(ut\text{-}pub\text{-}key, alice\text{-}pub\text{-}key, properties). \quad (1)$$

$$discount(eStore\text{-}pub\text{-}key, X, Y) \leftarrow student(ut\text{-}pub\text{-}key, X, Y). \quad (2)$$

Here credential (1) is a direct translation of credential shown in Fig. 1 and *properties* is the Prolog term representing the content of the corresponding "properties" XML element.

Given a role name p, the set of all credentials having p as a role name of the credential atom occurring in the head is called the *definition* of p. Every credential from the definition of p is called a defining credential of p. For any given credential atom A, we denote the issuer of A by $issuer(A)$ and the subject of A by $subject(A)$.

We will often refer to the logic programming representation of the credentials as it makes the notation easier. For sake of clarity, we will sometimes write the credential atoms without the last argument.

Expressiveness. Standard TuLiP is expressive enough to model complex policies like thresholds and separation of duty. Below we show examples of a threshold and separation of duty policy. In order to emphasise relative simplicity of Standard TuLiP, below we first show the policy encoded in RT^T (in fact we use a dialect of RT called RT_1^T as we also use arguments) and then its equivalent in Standard TuLiP.

Example 2. Threshold Policy: A says that an entity is a member of role $A.r$ and has the properties A_1 and A_2 if one member of $B.s$ and one member of $C.t$ say the same (in RT "?" denotes a variable).

$$
\begin{aligned}
RT_1^T: \;& A.r(?A_1, ?A_2) \leftarrow A.r_1.r(?A_1, ?A_2) \\
& A.r_1 \leftarrow B.s \odot C.t \\
\text{Standard TuLiP}: \;& r(a, X, \text{prop} : [p_1 : A_1, p_2 : A_2]) :- s(b, Y), t(c, Z), \\
& \qquad r(Y, X, \text{prop} : [p_1 : A_1, p_2 : A_2]), r(Z, X, \text{prop} : [p_1 : A_1, p_2 : A_2]).
\end{aligned}
$$

Example 3. Separation of Duty Policy: A says that an entity is a member of role $A.r$ and has the properties A_1 and A_2 if two different entities - one being a member of $B.s$ and the second being a member of $C.t$ - say the same.

$$
\begin{aligned}
RT_1^T: \;& A.r(?A_1, ?A_2) \leftarrow A.r_1.r(?A_1, ?A_2) \\
& A.r_1 \leftarrow B.s \otimes C.t \\
\text{Standard TuLiP}: \;& r(a, X, \text{prop} : [p_1 : A_1, p_2 : A_2]) :- s(b, Y), t(c, Z), Y \neq Z. \\
& \qquad r(Y, X, \text{prop} : [p_1 : A_1, p_2 : A_2]), r(Z, X, \text{prop} : [p_1 : A_1, p_2 : A_2]).
\end{aligned}
$$

RT^T is a member of the RT Trust Management Framework [12]. In order to handle policies like those presented in Example 2 and Example 3, RT^T introduces the so called *manifold roles*, which allow not only entities but also sets of entities to be members of a role. With Standard TuLiP, we have the same syntax and the same semantics for all sorts of supported policies. Notice also, that Standard TuLiP can be easily extended to a general purpose logic programming language by relaxing the restriction on the number of arguments in the credential atoms and their corresponding modes values (though at the cost of limited flexibility of the distributed storage in some cases).

3 Storage and Modes

The content of this section is not new in the sense that the results we report here are a natural extension of the material we present in [6] for Core TuLiP. Nevertheless, we include them for sake of completeness. Standard TuLiP is a distributed system in which credentials are stored by various peers (not necessarily by those issuing the credential). The following standard example shows that the location where the credentials are stored can affect the efficiency and the correctness of the whole TM system.

Example 4. We extend the example presented in Sect. 2. Now, *eStore* gives a discount to any student from any university accredited by *accBoard*. This is modelled as follows (logic notation):

$$
\begin{aligned}
& \mathit{discount}(\mathit{eStore\text{-}pub\text{-}key}, X) \leftarrow \mathit{accredited}(\mathit{accBoard\text{-}pub\text{-}key}, Y), \mathit{student}(Y, X). && (1) \\
& \mathit{accredited}(\mathit{accBoard\text{-}pub\text{-}key}, \mathit{ut\text{-}pub\text{-}key}). && (2) \\
& \mathit{student}(\mathit{ut\text{-}pub\text{-}key}, \mathit{alice\text{-}pub\text{-}key}). && (3)
\end{aligned}
$$

Now, suppose that credential (1) is stored by *eStore*, credential (2) by the *accBoard*, and credential (3) by *alice*: if one wants to know whether *alice* can have a discount at *eStore*, then one needs to evaluate the following query: $\leftarrow \mathit{accredited}(\mathit{accBoard}$

-pub-key$,Y),$*student*$(Y,$*alice*$)$. The most efficient way of answering this query is first to fetch credential (3) from *alice*. From credential (3), we immediately know that *alice* is a student of *ut*. Now, it is sufficient to check if *ut* is an accredited university. This can be done by either fetching credential (2) from *accBoard*, or from *ut*. Notice however, that if we store both credential (2) and (3) at the *ut* then we would not be able to find them. In this case contacting *alice* would not help as *alice* does not store any related credentials. Similarly, querying *accBoard* will not bring us any closer, as *accBoard* lets universities store the accreditation credentials.

Standard TuLiP uses the notion of *mode* to handle distributed storage. In logic programming, the mode of a predicate indicates which predicate arguments are *input* and which are *output* arguments. An input argument must be ground (completely instantiated) when atom is evaluated. In Standard TuLiP modes are assigned directly to role names and there are only three possible mode values: *ii*, *io*, or *oi*, where *i* stands for "input" and *o* for "output". Here, the first character points to the issuer and the second to the subject. For instance, if role name *r* has mode *oi*, it means that - in order to be able to find a credential of the form $r(issuer, subject, properties)$ - "subject" must be known. The most common mode is *io*, but *oi* is also useful to be able to store a credential in a place different from the issuer. The mode value *oo* is not allowed because this would allow queries in which both the issuer and subject are unknown, and we would not know where to start looking.

To be precise, in Standard TuLiP modes determine three things: (1) the storage location of the credentials defining a given role name, (2) the types of queries in which the role name can be used, and (3) they guarantee the soundness and completeness of the method of credential discovery. We now illustrate these three aspects.

Storage. The mode of a role name *p* indicates where the credentials defining the corresponding credential atom should be stored. Standard TuLiP introduces the notion of a *depositary* to be the entity which should store the given credential.

Definition 2 (depositary). Let p be a role name and let $c = P \leftarrow C_1, \ldots, C_n$ be a credential defining p. Then:

- if $mode(p) \in \{ii, io\}$ then the depositary of c is the credential issuer $(= issuer(P))$.
- if $mode(p) = oi$ then the depositary of c is the credential subject $(= subject(P))$.

There are exceptions: with mode *oi* it is also possible to store the credential at an entity other than the credential issuer or the subject. For details we refer the reader to [6].

Example 5. Referring to Example 4, assume that we have the following mode assignments: $mode(discount) = ii$, $mode(accredited) = io$, and $mode(student) = oi$. According to Definition 3, credential (1) should be stored by *eStore*, credential (2) by *accBoard*, and credential (3) by *alice*.

Queries. Recall that a Standard TuLiP query is a sequence of one or more credential atoms, each of which can be seen as a basic query itself. In Standard TuLiP we identify three types of (basic) queries:

Type 1: "Given two entities a and b, check if b has role name p as said by a." This query can be answered for any valid mode assignment for p.
Type 2: "Given entity a and role name p, find all entities b such that b has role name p as defined by a." This query can be answered only if $mode(p) = io$.
Type 3: "Given entity b and role name p find all entities a such that a says that b has role name p." This query can be answered if $mode(p) = oi$.

One can also consider a more general form of the query of Type 3: "Given entity b find all role names b has." This query is not supported in Standard TuLiP. The reason for this is purely of syntactic nature as we explain in [6].

The classification above is of purely syntactical nature, and one still has to guarantee that given a supported query, it can be answered. Basically, a query can be answered - either positively or negatively - if all the related credentials can be found. We guarantee this by the soundness and completeness of the credential discovery method.

Soundness and Completeness. Storing the credentials in the right place does not yet guarantee their discoverability. To ensure this we require the credentials to be *traceable*:

Definition 3 (traceable). We say that a credential is *traceable* if it is *well-moded* and the depositary of the credential is as given by Definition 2.

We use the standard definition of well-modedness as given in [1]. Standard TuLiP comes with a terminating *sound* and *complete* Lookup and Inference AlgoRithm (LIAR). Assuming that all credentials are traceable and given a well-moded query, the soundness result guarantees that LIAR produces only true answers. The completeness results on the other hand guarantees that if there exists an answer to the query then LIAR will be able to construct the proof of it. In this paper we do not give the detailed description of LIAR. For a formal description, we refer the reader to [6].

4 System Architecture

In this section we describe the architecture of Standard TuLiP. First, we present the system components and their role in the system. Then we show how the system components interoperate and we give a concrete example demonstrating this. Finally, we present the requirements Standard TuLiP has on the underlying infrastructure. In particular, we discuss how public identifiers can be mapped to physical infrastructure nodes.

System Components. In Standard TuLiP we identify the following components: (a) the LIAR engine, (b) the credential server (c) the User Client application, and (d) the mode register (see Fig. 2).

By default, every system user should run an instance of the LIAR engine, but other approaches are also possible. For instance, there can be a preselected set of

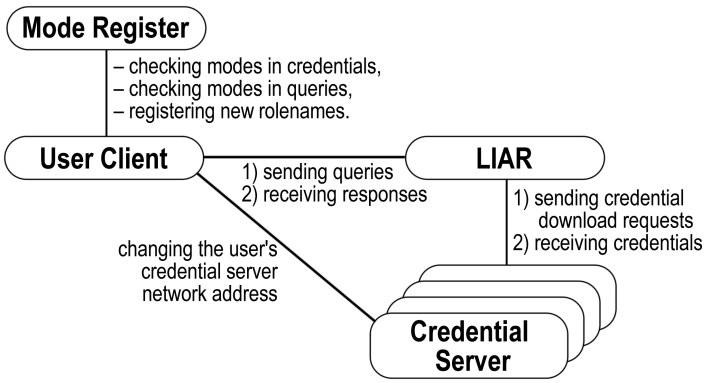

Fig. 2: Components of Standard TuLiP.

nodes having LIAR functionality, or there can even be only one instance of LIAR serving the whole community. The LIAR algorithm is implemented using YAP Prolog [14] with the external interface written in Python. This makes deployment of LIAR easier but allows us to preserve the original logic programming formalism in the "reasoning" part of the system. LIAR operates as an HTTP server when answering the queries and as a client when fetching credentials from *credential servers*. We give a more detailed functional description of LIAR later in this section.

Every user who wants to store her own credentials must run an instance of the credential server. The credential server responds to requests coming from LIAR engines and returns credentials satisfying the request. The credential server is implemented as a simple HTTP server (written in Python) and is internally connected to a *credential store*, which stores all user's credentials.

The *User Client* is a GUI application (written in Flash and Python) and provides user-friendly interface to other Standard TuLiP system components. In particular, the User Client is used for: generating the user private-public key pair, setting up and maintaining the location of the user's credential server, importing user credentials, and querying the Standard TuLiP system. Optionally, additional applications in the form of plugins can be provided. For instance one could provide a plugin having a graphical credential editor functionality. Notice that the User Client application itself does not allow the user to perform any action on a remote resource. Its main purpose is to let the user query the system.

Another important component of Standard TuLiP is the *mode register*, which is a centralised service where all the allowed role names in use and their corresponding modes are stored. The mode register is implemented as an HTTP server with a user-friendly web-interface. The role names and the associated modes are provided as Security Assertion Markup Language (SAML) *assertions* [16]. The mode register responds to SAML *attribute queries*. The answer is returned in the form of an SAML *response* [16] containing one or more assertions, each of which corresponds to the

credential atom and its associated mode(s). The mode register uses version 2.0 of the SAML standard [18].

LIAR. The basic functionality of LIAR is to wait for queries and respond to them. Recall that Standard TuLiP queries are themselves XML documents.

When LIAR receives a query from the User Client it first checks the signature on it and then is starts the evaluation process. Every time additional credentials are needed, LIAR fetches them from the location indicated by the mode information obtained by combining the information on the query and the mode register. Actually, by embedding the mode information in the credentials and the queries, the mode register does not have to be contacted in order to determine the storage location for the credentials defining a given role name. The credentials are fetched from the corresponding credential server by sending a so called *credential request*. Credential requests are XML documents specifying which credentials should be fetched. LIAR validates the received credentials by checking the signatures and validity intervals.

After evaluating the query, LIAR sends to the User Client the so called Standard TuLiP *response* (XML) document containing all answers, i.e. all instances of the query conditions satisfying the query. The top-level element of the Standard TuLiP response is the *response* XML element. Besides the unique ID and IssueInstant XML attributes it also contains *InResponseTo* XML attribute containing the value of the ID XML attribute from the corresponding query.

The following example demonstrates the system behaviour in the response to a concrete query.

Example 6. Assume we have the following set of credentials (logic notation):

$$\mathit{discount}(\mathit{ii}, \mathit{eStore\text{-}pub\text{-}key}, X) \leftarrow \mathit{accredited}(\mathit{io}, \mathit{accBoard\text{-}pub\text{-}key}, Y), \mathit{student}(\mathit{oi}, Y, X). \quad (1)$$

$$\mathit{accredited}(\mathit{io}, \mathit{accBoard\text{-}pub\text{-}key}, \mathit{ut\text{-}pub\text{-}key}). \quad (2)$$

$$\mathit{student}(\mathit{oi}, \mathit{ut\text{-}pub\text{-}key}, \mathit{alice\text{-}pub\text{-}key}). \quad (3)$$

This is the same set of credentials as in Example 4 but now including the mode argument indicating the mode of the corresponding role name. Figure 3 presents the steps performed by LIAR during evaluation of the query $\leftarrow \mathit{discount}(\mathit{ii}, \mathit{eStore\text{-}pub\text{-}key}, \mathit{alice\text{-}pub\text{-}key})$. In Fig. 3 the rounded rectangles represent the (credential servers of) entities, the arrows represent the messages being sent, and numbers above the arrows represent their order. Below, the flow of the algorithm is presented for the given query.

We assume that the instance of the LIAR algorithm is run by a user Jeroen with the public identifier *jeroen-pub-key*. In message $\boxed{1}$ LIAR receives the query in which the query issuer (Jeroen) asks whether the user with public id *alice-pub-key* has a discount at the internet store identified by *eStore-pub-key*. The query is signed by the query issuer. Before evaluating the query, LIAR checks the signature on the query, then it checks the mode of the atom $\mathit{discount}(\mathit{ii}, \mathit{eStore\text{-}pub\text{-}key}, \mathit{alice\text{-}pub\text{-}key})$. As the mode associated with role name *discount* is *ii*, LIAR knows that it should try to fetch credentials matching this query from *eStore*. This is done in messages $\boxed{2}$ and $\boxed{3}$. After receiving the matching credentials, LIAR validates each of them, which means that it checks the signatures and the validity intervals, and then every successfully validated credential rule is instantiated by unifying its head with

the query atom. In our case only one credential is fetched (credential (1)) and the resulting instance is:

$$discount(ii,eStore\text{-}pub\text{-}key,alice\text{-}pub\text{-}key) \leftarrow accredited(io,accBoard\text{-}pub\text{-}key,Y),student(oi,Y,alice\text{-}pub\text{-}key).$$

We see that in order to prove the initial query, now LIAR has to evaluate the following one:

$$\leftarrow accredited(io,accBoard\text{-}pub\text{-}key,Y),student(oi,Y,alice\text{-}pub\text{-}key).$$

In evaluating this (sub) query, LIAR checks the mode associated with role name *accredited* and notices that the associated mode value is *io*, meaning that the related credentials (if any) should be stored by *accBoard*. The *accBoard* is queried in message 4, resulting in the fact that credential (2) is fetched with message 5, it is validated, and then its instance - $accredited(io,accBoard\text{-}pub\text{-}key,ut\text{-}pub\text{-}key)$ - is used. As the body of credential (2) is empty, the query reduces to $\leftarrow student(oi,ut\text{-}pub\text{-}key,alice\text{-}pub\text{-}key)$. The mode of role name *student* is *oi* which means that the defining credentials should be stored by their subject: *alice* in this case. Alice is contacted by LIAR with message 6, and asked for all *oi* credentials she stores. In the response, in message 7, credential (3) is returned and then validated. This credential unifies with $student(oi,ut\text{-}pub\text{-}key,alice\text{-}pub\text{-}key)$ and at this point the original query has been evaluated successfully. The information about successful evaluation, containing only one condition corresponding to the $discount(ii,eStore\text{-}pub\text{-}key,alice\text{-}pub\text{-}key))$ credential atom, is sent to the User Client in message 8.

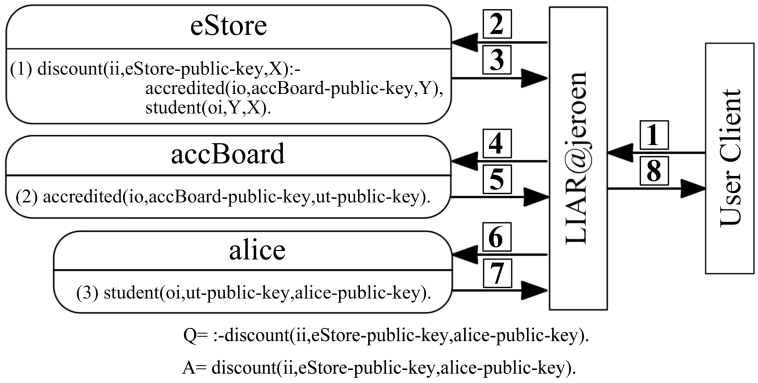

Fig. 3: Credential Discovery with LIAR

Public Identifiers. Recall that Standard TuLiP uses public keys as (public) identifiers of the users. In Example 6 we have silently assumed that there exists a mapping between public identifiers and concrete network addresses. Indeed, Standard TuLiP requires an underlying service to map public identifiers to concrete network addresses.

Distributed Hash Tables (DHT) [19] represent a class of overlay P2P systems with key-based routing functionality. They provide a look up service similar to a hash table. In DHT systems, a network location of a resource is determined by a global unique name of that resource.

Standard TuLiP can be built on top of a DHT system or an another overlay P2P network providing lookup service based on global unique identifiers. We assume that every underlying infrastructure node stores one or more user records containing at the least the current network address of the user, and the network address of the user's credential server. The user should be able to securely change the network address of her credential server. The user's current network address should be synchronised with the actual network address of the User Client application acting in the name of the user.

5 Using Standard TuLiP

In using the Standard TuLiP Trust Management system we can distinguish the following actions that may be performed by the user: (1) issuing credentials, (2) sending the queries and receiving the responses, and (3) revoking credentials and user public identifiers. Below we briefly summarise issues raised by these actions.

Writing Credentials. When issuing credentials one must be sure that any new credential is traceable. The User Client application helps in writing credentials by checking that the credentials are traceable. Before accepting the credential, it checks if *for every* mode value of the credential head there exists a permutation of the credential atoms occurring in the credential body and the corresponding mode values such that the credential is traceable. If this is not the case, the credential is refused. If for a mode value of the head there exists more than one valid mode assignment for the credential atoms in the body, the user will be allowed to choose a preferred one.

The User Client application determines the modes of the credential atoms by querying the mode register. The selected modes are then embedded into the actual credentials so that the mode register does not have to be referred to during query evaluation later. Recall that the assigned modes determine the actual credential storage location. The User Client application automatically uploads the new credentials to the suitable credential server (as given by the user record associated with the given public id).

When a user introduces a credential with a new role name, it has to be registered with the mode register. The mode register can be accessed through the TuLiP homepage, or by using a dedicated application. Each user can request the registration of additional role names and the corresponding modes by requesting it through the TuLiP web-site.

Writing Queries. Every Standard TuLiP query must be well-moded. Therefore, before sending a query, the User Client application checks for well-modedness. If

some credential atoms in the query have more than one mode value, it is possible that there will be more than one variant of mode assignment that makes the query well-moded. In such a case, the User Client lets the user to select the preferred mode assignment (e.g. the one that is likely to yield the correct answer most efficiently). The User Client application sends the query to the LIAR engine associated with the user issuing the query and presents the received response.

Revoking Credentials and Public Identifiers. Recall that in Standard TuLiP, the user's public identifier is the user's public key. The use of public keys as user identifiers is convenient as everyone can create her public identifier by simply generating a new key pair. Additionally one can sign the credentials and queries using the corresponding private key. This makes the validation possible *without* the need for any external public key infrastructure. When using public keys as user identifiers, however, one has to deal with the problems of key revocation. Notice that when the user's private key becomes compromised, it is not sufficient to revoke all the credentials issued by this user, but the user's public identifier should not be used anymore. Currently, Standard TuLiP does not support any revocation mechanisms other than the validity period specified in each credential. In the future, we plan to extend LIAR, so that it checks if the selected credential is not revoked. Instead of revoking all the user's credentials, it is also possible to revoke the user id. This can be done by issuing an *id revocation certificate* which would state that the given public identifier cannot be trusted any longer to sign the credentials. A comprehensive revocation framework for Standard TuLiP is our future work.

6 Related Work

The first trust management systems, PolicyMaker [4], KeyNote [3], and SDSI/SPKI [5], as well as e.g. [11, 9] focus on the language design without fully supporting credential distribution. In most systems, public keys are used as the identifiers of the users. This is in contrast to the traditional authorisation mechanisms based on identity-based public-key systems like X.509. The RT family of Trust Management Languages [13, 12] is the first in which the problem of credential discovery is given an extensive treatment. In particular, in [13], a type system is introduced in order to restrict the number of possible credential storage options. In [23] Winsborough and Li identify the features a "good" language for credentials should have, one of those being the support for distributed storage. As we show in [6], our system is at least as flexible as RT and all storage possibilities given by RT can be replicated here.

PeerTrust [15] is a Trust Negotiation language where the problem of the distributed storage is also taken into account. PeerTrust is based on first order Horn clauses of the form $lit_0 \leftarrow lit_1, \ldots, lit_n$, where each lit_i is a positive literal. PeerTrust supports distributed storage by allowing each literal in a rule to have an additional *Issuer* argument: lit_i @ *Issuer*. *Issuer* is the peer responsible for evaluating lit_i. The *Issuer* arguments do not, however, say where a particular credential should be stored but only who is responsible for evaluating it. It means that PeerTrust makes a silent

assumption that the credentials are stored in such a way that *Issuer* can find the proof, but it gives no clue of how this should be done. PeerTrust considers only the first two requirements mentioned in the introduction.

From the more practical approaches (but with very strong theoretical foundation as well), Bertino et al. developed Trust-$\mathscr{X}$ - a trust negotiation system [2]. Trust-$\mathscr{X}$ uses the $\mathscr{X}$-TNL trust negotiation language for expressing credentials and disclosure policies. Trust-$\mathscr{X}$ certificates are either credentials or declarations. Credentials state personal characteristics of the owner and are certified by a Credential Authority (CA). Declarations also carry personal information about its owner but are not certified. Trust-$\mathscr{X}$ does not deal with the problem of distributed credential storage and discovery. It means that the second and third requirement is not supported.

The *eXtensible Access Control Markup Language* (XACML) [17] supports distributed policies and also provides a profile for the role based access control (RBAC). However, in XACML, it is the responsibility of the *Policy Decision Point* (PDP) – an entity handling access requests – to know where to look for the missing attribute values in the request. The way missing information is retrieved is application dependent and is not directly visible in the supporting language. Thus, XACML does not support the second and the third requirement presented in Section 1.

7 Conclusions and Future Work

In this paper we presented the architecture of Standard TuLiP - a logic based Trust Management system. Standard TuLiP follows the Trust Management approach in which security decisions are based on security credentials which are issued by different entities and stored at different places. Standard TuLiP basic constituents are the Standard TuLiP Trust Management language, the mode system for the credential storage, and a terminating sound and complete Lookup and Inference AlgoRithm (LIAR) which guarantees that all required credentials can be found when needed.

Standard TuLiP is decentralised. Every user can formulate his/her own security policy and store credentials in the most convenient and efficient way for himself. Standard TuLiP does not require a centralised repository for credential storage, nor does it rely on any external PKI infrastructure. Standard TuLiP credentials are signed directly by their issuers so that no preselected Certification Authority (CA) is needed.

With this we show that it is possible to design and implement a Trust Management system that is theoretically sound yet easy and efficient to deploy and use.

Future Work. Standard TuLiP can be extended in several directions. Firstly, we plan to extend expressiveness of the Standard TuLiP Trust Management language, so that it can be used to express non-monotonic policies. Although Standard TuLiP can already be used as a Trust Negotiation language, we also plan to add a direct support to Trust Negotiation at the language level.

References

1. K. R. Apt and I. Luitjes. Verification of logic programs with delay declarations. In *AMAST*, volume 936 of *LNCS*, pages 66–90. Springer, 1995.
2. E. Bertino, E. Ferrari, and A. C. Squicciarini. Trust-$\mathcal{X}$: A Peer-to-Peer Framework for Trust Establishment. *IEEE Trans. Knowl. Data Eng.*, 16(7):827–842, 2004.
3. M. Blaze, J. Feigenbaum, J. Ioannidis, and A. Keromytis. The KeyNote Trust-Management System, Version 2. IETF RFC 2704, September 1999.
4. M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized Trust Management. In *Proc. 17th IEEE Symposium on Security and Privacy*, pages 164–173. IEEE Computer Society Press, May 1996.
5. D. Clarke, J.E. Elien, C. Ellison, M. Fredette, A. Morcos, and R. L. Rivest. Certificate Chain Discovery in SPKI/SDSI. *Journal of Computer Security*, 9(4):285–322, 2001.
6. M. R. Czenko and S. Etalle. Core TuLiP - Logic Programming for Trust Management. In *Proc. 23rd International Conference on Logic Programming, ICLP 2007, Porto, Portugal*, volume 4670 of *LNCS*, pages 380–394, Berlin, 2007. Springer Verlag.
7. Freeband Communication. *I-Share: Sharing resources in virtual communities for storage, communications, and processing of multimedia data.*
   URL: http://www.freeband.nl/project.cfm?language=en&id=520.
8. S. L. Jarvenpaa, N. Tractinsky, and M. Vitale. Consumer Trust in an Internet Store. *Inf. Tech. and Management*, 1(1-2):45–71, 2000.
9. T. Jim. SD3: A Trust Management System with Certified Evaluation. In *Proc. IEEE Symposium on Security and Privacy*, pages 106–115. IEEE Computer Society Press, 2001.
10. F. Lee, D. Vogel, and M. Limayem. Adoption of informatics to support virtual communities. In *HICSS '02: Proc. 35th Annual Hawaii International Conference on System Sciences (HICSS'02)-Volume 8*, page 214.2. IEEE Computer Society Press, 2002.
11. N. Li, B. Grosof, and J. Feigenbaum. Delegation Logic: A Logic-based Approach to Distributed Authorization. *ACM Transactions on Information and System Security (TISSEC)*, 6(1):128–171, 2003.
12. N. Li, J. Mitchell, and W. Winsborough. Design of a Role-based Trust-management Framework. In *Proc. IEEE Symposium on Security and Privacy*, pages 114–130. IEEE Computer Society Press, 2002.
13. N. Li, W. Winsborough, and J. Mitchell. Distributed Credential Chain Discovery in Trust Management. *Journal of Computer Security*, 11(1):35–86, 2003.
14. LIACC/Universidade do Porto and COPPE Sistemas/UFRJ. *YAP Prolog*, April 2006.
15. W. Nejdl, D. Olmedilla, and M. Winslett. PeerTrust: Automated Trust Negotiation for Peers on the Semantic Web. In *Secure Data Management*, pages 118–132, 2004.
16. OASIS. *Assertions and Protocols for the OASIS: Security Assertion Markup Language (SAML) V2.0*, March 2005.
17. OASIS. *eXtensible Access Control Markup Language (XACML) Version 2.0*
    URL: http://www.oasis.org, Feb 2005.
18. OASIS. *SAML V2.0 Executive Overview*, April 2005.
19. S. Ratnasamy, P. Francis, M. Handley, R. M. Karp, and S. Shenker. A scalable content-addressable network. In *SIGCOMM*, pages 161–172, 2001.
20. W3C. *XML-Signature Syntax and Processing*, Feb 2002.
21. W3C. *Extensible Markup Language (XML) 1.1 (Second Edition)*, Sep 2006.
22. W3C. *Namespaces in XML 1.0 (Second Edition)*, Aug 2006.
23. W. H. Winsborough and N. Li. Towards Practical Automated Trust Negotiation. In *POLICY*, pages 92–103. IEEE Computer Society Press, 2002.