Core TuLiP

Marcin Czenko* and Sandro Etalle*
Department of Computer Science
University of Twente
{marcin.czenko, sandro.etalle}@utwente.nl

Abstract

We propose CoreTuLiP - the core of a trust management language based on Logic Programming. CoreTuLiP is based on a subset of moded logic programming, but enjoys the features of TM languages such as RT; in particular clauses are issued by different authorities and stored in a distributed manner. We present a lookup and inference algorithm which we prove to be correct and complete w.r.t. the declarative semantics. CoreTuLiP enjoys uniform syntax and the well-established semantics and is expressive enough to model scenarios which are hard to deal with in RT.

1 Introduction

Trust management (TM) [5, 16, 7, 6, 10, 11, 13, 12, 17] is an approach to access control in decentralised distributed systems where access control decisions are based on policy statements issued by multiple principals, and stored in a distributed manner. Policy statements are often digitally signed to ensure their authenticity and integrity; such statements are sometimes called *credentials* or *certificates*, and – in many cases – can be represented as datalog clauses. Indeed, like in logic programming, in TM credentials often have to be combined together to provide a authorisation *proof* (e.g. a proof that a given user has indeed access to a given resource).

One of the peculiar features of TM (w.r.t. classical decentralised access control, but also w.r.t. logic programming) is that credentials may or may not be stored by the authority who *issues* them, therefore one of the prominent problems of TM is that of guaranteeing that – under reasonable circumstances – if there exists a proof of a certain (authorisation) statement, then it is also possible to *find* the credentials needed to construct the proof itself.

To date, one of the most successful TM systems is the RT family, defined by Li, Winsborough and Mitchell [12, 13]. This family of languages enjoys a well-defined LP-based declarative semantics, syntax similar to that of SDSI [6], and offers the possibility of storing credentials either by the *issuer* (the authority issuing them) and/or by the *subject* (the entity the credential "refers to"). The location where the credential

*This work was carried out within the Freeband I-Share project.

is stored is determined by the so-called *type* of the credential. Li et. al show that if all credentials are *well typed* then there exists a terminating credential chain discovery algorithm which determines whether a given statement is valid in the present state.

Although the RT family is successful in achieving its goals, we believe that it presents drawbacks which are worth investigating and improving. In particular, RT syntax is inflexible to the extent that to accommodate natural things such as separation of duty etc., one has to resort to a number of rather artificial extensions (RT_1 until RT^D, and RT^T), which are difficult to grasp and use. Secondly, it cannot be linked naturally with external languages. Finally, while it enjoys a declarative reading, this reading does not reflect anything of the crucial type information.

One could speculate that to solve the above problems one should simply translate RT into Logic Programming, and then use the latter to specify and prove authorisation statements. This is however inaccurate, as this translation would lose the essential elements that make RT a *trust management* language, in particular the information concerning where credentials should be stored and how they can be found when needed, which is essential in TM.

In this paper we present *CoreTuLiP*, which is the stripped-down version of the TuLiP (Trust management system based on Logic Programming) system we are developing at the University of Twente in the context of the I-Share project [9]. CoreTuLiP, is basically a subset of (function-free) moded logic programming, with the essential additional feature that the clauses are not stored at a central authority, but are distributed across the different principals involved in the system. The mode information determines *where* a clause will be stored and a form of *well-modedness* is used to guarantee that, as the computation progresses, enough information is available to *find* the clauses needed to build a proof of the query being fired. Since credentials are distributed, CoreTuLiP is not amenable to SLD resolution; like RT, CoreTuLiP requires a mix of top-down and bottom up reasoning. Here, we present a terminating algorithm which is able to answer well-moded queries, together with a a soundness and completeness result (w.r.t. the standard LP semantics under the assumptions that all clauses are aggregated in one place). Finally, we show that RT_0, the core language of the RT family is basically equivalent to a subset of CoreTuLiP; this equivalence is demonstrated also by taking into account the type of RT credentials, and thus the location where credentials are stored. Doing so, we prove that it is possible to define a true trust management language which is as expressive as RT_0 also in terms of credential distribution without giving up the established LP formalism.

CoreTuLiP, being based on LP, has a much more flexible underlying syntax than RT, and can easily accommodate extensions without changes to the syntax. Because of this, we believe and we argue that it will form a good basis for a language which will be much more expressive than RT while enjoying RT's properties as a TM language. To support our arguments, we show that it is possible to express thresholds and separation of duties (which require special addition to RT_0) without leaving the syntax of CoreTuLiP.

This paper is structured as follows: in Section 2 we introduce the basics of moded Logic Programming. In Section 3 we introduce CoreTuLiP. In Section 4 we describe LIAR, the Lookup and Inference AlgoRithm, which checks whether a fact is entailed by a set of CoreTuLiP credentials. We also show that LIAR is sound and complete

w.r.t. the standard LP semantics. In Section 5 we compare RT_0 with CoreTuLiP. Finally, we conclude the paper in Section 6 and propose future research in Section 6. Some proofs are reported in the Appendix, *which is included solely for the reader's convenience*. Should the paper be accepted we are going to remove the appendix and make it available as Technical Report.

2 Preliminaries on Logic Programs

The reader is assumed to be familiar with the terminology and the basic results of the semantics of logic programs [1, 2, 14]. Here we refer to *function-free* (Datalog-like) logic programs and we adopt the notation of [2] in the fact that we use boldface characters to denote sequences of objects; therefore $\mathbf{t}$ denotes a sequence of terms while $\mathbf{B}$ is a sequence of atoms, i.e. a query (following [2], queries are simply conjunctions of atoms, possibly empty). We denote atoms by $A, B, H, \ldots$, queries by $\mathbf{A}, \mathbf{B}, \mathbf{C}, \ldots$, clauses by $c, d, \ldots$, and programs by P. The empty query is denoted by $\Box$.

For any syntactic object (e.g., atom, clause, query) o, we denote by $Var(o)$ the set of variables occurring in o. Given a *substitution* $\sigma = \{x_1/t_1, ..., x_n/t_n\}$ we say that $\{x_1, \ldots, x_n\}$ is its *domain* (denoted by $Dom(\sigma)$) and that $Var(\{t_1, ..., t_n\})$ is its *range* (denoted by $Ran(\sigma)$). Further, we denote by $Var(\sigma) = Dom(\sigma) \cup Ran(\sigma)$. If, $t_1, ..., t_n$ is a permutation of $x_1, ..., x_n$ then we say that σ is a *renaming*. The *composition* of substitutions is denoted by juxtaposition ($\theta\sigma(X) = \sigma(\theta(X))$). We say that an syntactic object (e.g., an atom) o is an *instance* of o' iff for some σ, $o = o'\sigma$, further o is called a *variant* of o', written $o \approx o'$ iff o and o' are instances of each other. A substitution θ is a *unifier* of objects o and o' iff $o\theta = o'\theta$. We denote by *mgu*(o, o') any *most general unifier* (*mgu*, in short) of o and o'.

Computations are sequences of derivation steps. The non-empty query $q : \mathbf{A}, B, \mathbf{C}$ and a clause $c : H \leftarrow \mathbf{B}$ (renamed apart w.r.t. q) yield the resolvent $(\mathbf{A}, \mathbf{B}, \mathbf{C})\theta$, provided that $\theta = mgu(B, H)$. A *derivation step* is denoted by $\mathbf{A}, B, \mathbf{C} \stackrel{\theta}{\Longrightarrow}_{P,c} (\mathbf{A}, \mathbf{B}, \mathbf{C})\theta$. c is called its *input clause*, and B is called the *selected atom* of q. A derivation is obtained by iterating derivation steps. A maximal sequence $\delta := \mathbf{B}_0 \stackrel{\theta_1}{\Longrightarrow}_{P,c_1} \mathbf{B}_1 \stackrel{\theta_2}{\Longrightarrow}_{P,c_2} \cdots \mathbf{B}_n \stackrel{\theta_{n+1}}{\Longrightarrow}_{P,c_{n+1}} \mathbf{B}_{n+1} \cdots$ of derivation steps is called an *SLD derivation of* $P \cup \{\mathbf{B}_0\}$ provided that for every step the standardisation apart condition holds, i.e., the input clause employed at each step is variable disjoint from the initial query $\mathbf{B}_0$ and from the substitutions and the input clauses used at earlier steps. If the program P is clear from the context and the clauses $c_1, \ldots, c_{n+1}, \ldots$ are irrelevant, then we drop the reference to them. If δ is maximal and ends with the empty query ($\mathbf{B}_n = \Box$) then the restriction of θ to the variables of $\mathbf{B}$ is called its *computed answer substitution* (*c.a.s.*, for short). The length of a (partial) derivation δ, denoted by $len(\delta)$, is the number of derivation steps in δ.

Moded Programs Informally speaking, a *mode* indicates how the arguments of a relation should be used, i.e. which are the input and which are the output positions of each atom, and allow one to derive properties such as absence of run-time errors for

Prolog built-ins, absence of floundering for programs with negation [4]. Most compilers encourage the user to specify a mode declaration.

Definition 2.1 (Mode) *Consider an n-ary predicate symbol p. By a* mode *for p we mean a function m_p from $\{1, \ldots, n\}$ to $\{In, Out\}$.*

If $m_p(i) = In$ (resp. *Out*), we say that i is an $input$ (resp. *output*) *position of* p (with respect to m_p). We assume that each predicate symbol has a *unique mode* associated to it; multiple modes may be obtained by simply renaming the predicates. We use the notation $(X_1, \ldots, X_n)$ to indicate the mode m in which $m(i) = X_i$. For instance, (In, Out) indicates the mode in which the first (resp. second) position is an input (resp. output) position. To benefit from the advantage of modes, programs are required to be *well-moded* [4], which means that they have to respect some correctness conditions relating the input arguments to the output arguments. We denote by $In(A)$ (resp. $Out(A)$) the sequence of terms filling in the input (resp. output) positions of A, and by $VarIn(A)$ (resp. $VarOut(A)$) the set of variables occupying the input (resp. output) positions of A.

Definition 2.2 (Well-Moded) *A clause $H \leftarrow B_1, \ldots, B_n$ is* well-moded *if for all $i \in [1, n]$*

$$VarIn(B_i) \subseteq \bigcup_{j=1}^{i-1} VarOut(B_j) \cup VarIn(H), \text{ and}$$
$$VarOut(H) \subseteq \bigcup_{j=1}^{n} VarOut(B_j) \cup VarIn(H).$$

*A query **A** is* well-moded *iff the clause $H \leftarrow$ **A** is well-moded, where H is any (dummy) atom of zero arity. A program is* well-moded *if all of its clauses are well-moded.*

Note that the first atom of a well-moded query is ground in its input positions and a variant of a well-moded clause is well-moded. The following Lemma, due to [3], shows the "persistence" of the notion of well-modedness.

Lemma 2.3 *An SLD-resolvent of a well-moded query and a well-moded clause that is variable-disjoint with it, is well-moded.* □

As a consequence of Lemma 2.3 we have the following well-known property [3].

Corollary 2.4 *Let P be a well-moded program and $\mathbf{A}$ be a well-moded query. Then for every computed answer σ of $\mathbf{A}$ in P, $\mathbf{A}\sigma$ is ground.* □

A straightforward consequence of this Corollary is the following one:

Corollary 2.5 *Let $H \leftarrow B_1, \ldots, B_n$ be a clause in a well-moded program P. If A is a well-moded atom such that $\gamma_0 = mgu(A, H)$ and for every $i \in [1, j], j \in [1, n-1]$ there exists a successful derivation $B_i\gamma_0, \ldots, \gamma_{i-1} \xrightarrow{\gamma_i}_P \square$ then $B_{j+1}\gamma_0, \ldots, \gamma_j$ is a well-moded atom.* □

3 Core TuLiP

In CoreTuLiP there are two types of atoms: (user-defined) *credentials* and built-in *constraints*. To distinguish between the two we assume the presence of a set of *certificate predicates* (for credentials) and of a disjoint with it set of *constraint predicates*. In CoreTuLiP,

- *certificate* predicates have exactly two arguments;
- in a certificate atom we call the term filling the first argument the *issuer*, and the one filling the second argument the *recipient*.

(the relations between these notions and the notions of issuer and subject used in RT are discussed in Section 5). In the full version we are going to have certificates with more arguments and user defined predicates as well. These additions are, however, immaterial for this paper.

For example, the certificate *student*$(ut, alice)$ is *issued* by *ut* (University of Twente), and has *alice* as the recipient. With this certificate, *ut* states that *alice* belongs to the *student* set. In a practical setting, this certificate is signed by *ut*, and *ut* and *alice* are placeholders for the implementation dependent identifiers (like public keys or URIs) referring to the University of Twente and Alice respectively. In TM, certificates are always *issued* by some authority (for the sake of simplicity here we identify authorities with the set of ground terms). It is therefore natural to expect the issuer of the head of a rule to be a ground term.

Definition 3.1 *Let* $cl : H \leftarrow B_1, \ldots, B_n$ *be a clause. We say that* cl *is* well-formed *if it is well-moded and* issuer(H) *is a ground term.*

Decentralised Storage A peculiar feature of trust management systems is that certificates are stored in a distributed way. For instance, the credential *student*$(ut, alice)$ which is issued by *ut* could be stored by either *ut* or *alice*. Storing it by *alice* has the advantage that *alice* does not have to fetch the credential at *ut* every time she needs it, which in a highly distributed system may be costly. We call the *depositary* of a credential the authority where the credential is stored. In CoreTuLiP, it is the mode of the credential's head which determines its depositary. Furthermore, in CoreTuLiP we allow only one mode per relation symbol, so credentials will be stored at one place only; by allowing multiple modes we lift this limitation in the extended system.

Certificate symbols have three *legal modes*: (In, In), (In, Out), and (Out, In). The reason why the mode (Out, Out) is considered illegal is that it would allow queries with completely uninstantiated arguments like *student*(X, Y), in which neither the issuer nor the recipient is specified. Unlike in LP, such queries cannot be answered in a TM system because the system does not know where to look for relevant credentials, which could be issued and stored by any authority. By requiring that at least one of the arguments be ground, and that that the credentials be *traceable* (see below) we will be able to find the credentials we need to construct the proofs we need. For constraint predicates, on the other hand, the only legal mode is the one all-input $(In, \ldots, In)$.

The mode of a predicate determines where the credentials defining it are stored. For instance, if *mode*(*student*) is either (In, In) or (In, Out), then the above credential

will be stored at *ut*, otherwise (if the mode is (Out, In)), *alice* will store it. Storing the credential at some other place would make it unfindable. The definition below generalises this concept.

Definition 3.2 (Traceable, Depositary) *We say that a clause* $cl : H \leftarrow B_1, \ldots, B_n$ *is* traceable *if it is well-formed and one of the following conditions hold:*

1. mode$(H) \in \{(In, In), (In, Out)\}$ *– in this case* issuer(H) *is the* depositary *of the rule,*
2. mode$(H) = (Out, In)$*, and* recipient$(H)(= In(H))$ *contains a ground term - in this case* recipient(H) *is the* depositary *of the rule,*
3. mode$(H) = (Out, In)$ *and* recipient(H) *contains a variable. In this case we require that there exists a prefix* $B_1, \ldots, B_k$ *of the body such that*
   - mode$(B_1) = \ldots =$ mode$(B_k) = (Out, In)$,
   - $In(H) = In(B_1)$,
   - $In(B_{i+1}) = Out(B_i)$*, and is a variable, for* $i \in [1, k-1]$,
   - $Out(B_k)$ *contains a ground term,*

   In this case, we say that issuer$(B_k)(= Out(B_k))$ *is the* depositary *of the rule.*

The third case is a bit complex, but it has the advantage of permitting the storage of a credential at a third party (neither the issuer, not the recipient).

We can now introduce the concept of a state.

Definition 3.3 *A* state $\mathcal{P}$ *is a finite collection of pairs* (a, P_a) *where* P_a *is a collection of traceable credentials and* a *is the depositary of these credentials.*

The declarative semantics of a state is simply given in terms of logic programming as follows (where for simplicity we assume that all constraints are user-defined)

Definition 3.4 *Let* $\mathcal{P}$ *be the state* $\{(a_1, P_1), \ldots, (a_n, P_n)\}$*, and* A *be an atom*

- *We denote by* $P(\mathcal{P})$ *the set of clauses* $P_1 \cup \cdots \cup P_n$*. We call* $P(\mathcal{P})$ *the LP-counterpart of state* $\mathcal{P}$*.*
- *We say that* A *is true in state* $\mathcal{P}$ *iff* $P(\mathcal{P}) \cup \mathcal{C} \models A$*, where* $\mathcal{C}$ *is a first order theory determining the meaning of credential predicates.*

The following example clarifies the use of modes and their influence on the credential storage.

Example 3.5 To access a project document at the University of Twente (UT) one must be either a project member and a Ph.D. student at the UT or at one of the partner universities, or be approved by two different assistant professors at the UT. John and Jeroen are assistant professors at the UT. John says that a project member from one of the partner universities can access the document if she is approved by at least one project member who is also an associate professor at that university. Jeroen approves anyone who is also approved by a project leader at the UT. Sandro is a project leader at the

(c_1) *access_document*(ut, X) :–
project_member(ut, X),
prof(ut, A_1),
prof(ut, A_2),
$A_1 \neq A_2$,
approve_access(A_1, X),
approve_access(A_2, X).

(c_2) *access_document*(ut, X) :–
phd_student(P, X),
project_partner(ut, P),
project_member(P, X).

(c_3) *approve_access*$(john, X)$:–
approve_access(A, X),
associate_prof(P, A),
project_partner(ut, P),
project_member(P, A),
project_member(P, X).

(c_4) *approve_access*$(jeroen, X)$:–
approve_access(L, X),
project_leader(ut, L).

(c_5) *project_leader*$(ut, sandro)$.

(c_6) *phd_student*$(ut, marcin)$.

(c_7) *approve_access*$(sandro, rico)$.
approve_access$(jeffrey, rico)$.

(c_8) *associate_prof*$(tud, jeffrey)$.

(c_9) *project_member*$(ut, john)$.
project_member$(ut, charles)$.
prof$(ut, john)$.
prof$(ut, jeroen)$.
project_partner(ut, ut).

(c_{10}) *project_partner*(ut, tud).
project_member$(tud, jeffrey)$.
project_member$(tud, rico)$.

Figure 1: Credentials of Example 3.5.

UT. Figure 1 shows the credentials modelling the scenario. The modes are as follows: *access_document*:(In, In), *project_member*:(In, In), *prof*:(In, Out), and the remaining predicates are moded (Out, In). Therefore, credentials $c_1 - c_4$ and credentials c_9 are stored at *ut.* Credential c_5 is stored by *sandro*, c_6 by *marcin*, the two credentials c_7 by *rico*, and c_8 by *jeffrey*. *tud* stores all the credentials c_{10}.

4 The Lookup and Inference AlgoRithm (LIAR)

The goal of an authorisation system is to check whether a fact is true in a given state. Since the state $\mathcal{P}$ can be very large and distributed across different agents, it is essential to have an algorithm which takes care of computing whether a given query is true in $P(\mathcal{P})$ without having to collect the entire $P(\mathcal{P})$. An extra difficulty comes from the fact that clauses might easily be mutually recursive, and that cases 2 and 3 of Definition 3.2 make it impossible to follow a straightforward top-down reasoning.

In this section we present a suitable algorithm. Before we proceed we need the following definitions.

Definition 4.1 (Connected) *We say that two atoms A and B that have mode (Out, In) are* connected *if* recipient(A) *is ground and* recipient(A) = recipient(B).

Notation: Let A be an atom and S be a set of atoms. In the sequel we adopt the following notational conventions:

(i) We write $A \stackrel{\sim}{\in} S$ iff $\exists A' \in S$, such that $A' \approx A$.

(ii) We write $A \stackrel{\sim}{\notin} S$ iff $\nexists A' \in S$ such that $A' \approx A$.

(iii) We write $A \overset{\theta}{\hookrightarrow} S$ iff $\exists A' \overset{\sim}{\in} S$ standardised apart w.r.t. A such that $\gamma = mgu(A, A')$ and $A\theta \approx A\gamma$.

Definition 4.2 *Let A be an atomic well-moded query. We define the Lookup and Inference AlgoRithm (LIAR) which given a state $\mathcal{P}$ and a query A as an input returns the (possibly empty) sets of atoms* FACTSTACK *and* GOALSTACK. *The algorithm is reported in Figure 4.*

In the description of LIAR we assume that *dummy* is a reserved predicate symbol, with mode (Out, In). Statements in boxes are optional and included only for optimisation purposes. The algorithm extends naturally to queries containing more than one atom.

We now prove that LIAR algorithm is sound and complete w.r.t. the standard LP semantics, i.e. the centralised algorithm based on the SLD resolution. We need the following lemma.

Lemma 4.3 *Let $\mathcal{P}$ be a state and* FACTSTACK *be the result of the algorithm execution for some well-moded query. Let A be an atom in* FACTSTACK. *Then A is ground.*

Proof. See the Appendix.

The soundness result is rather straightforward.

Theorem 4.4 (soundness) *Let $\mathcal{P}$ be a state and* FACTSTACK *be the result of executing LIAR on $\mathcal{P}$ and a well-moded query. Then $\forall A \in$ FACTSTACK, $P(\mathcal{P}) \models A$.*

Proof. It is easy to see that, by construction, if an atom A is added to FACTSTACK, then CLSTACK $\models A$. Since $\forall c \in$ CLSTACK c is an instance of a clause $c' \in P(\mathcal{P})$, it follows that $P(\mathcal{P}) \models A$. □

The following completeness result guarantees among other things that – after executing LIAR on a state $\mathcal{P}$ and some well-moded query – for any goal $A \in$ GOALSTACK it holds that if there exists a successful SLD derivation of A in $P(\mathcal{P})$ with c.a.s. θ then $A \overset{\theta}{\hookrightarrow}$ FACTSTACK.

Theorem 4.5 (completeness) *Let $\mathcal{P}$ be a state and then* FACTSTACK, GOALSTACK *be the result of executing LIAR on $\mathcal{P}$ and a given well-moded goal. Then $\forall C \in$ GOALSTACK, if $\exists$ a successful SLD derivation $\delta : C \xrightarrow{\theta}_{P(\mathcal{P})} \square$ then $C \overset{\theta}{\hookrightarrow}$ FACTSTACK.*

Proof. See the Appendix.

5 CoreTuLiP vs. RT_0

In this section we are going to compare CoreTuLiP with the well-established RT_0 trust management language. We are going to show that – in most respects – CoreTuLiP is at least as expressive as RT_0. To this end, we first present a slightly simplified (yet expressively equivalent) version of RT_0 as given in [13]: A *principal* is a uniquely identified individual or process. A principal can define a *role*, which is indicated by a

INPUT : A. /* A is the initial atomic query */

Init:

CLSTACK : $\{A \leftarrow A\}$;
FACTSTACK : $\emptyset$;
GOALSTACK : $\emptyset$;
SATISFIED = **FALSE** ;
VISITED $= \emptyset$;

REPEAT

Phase 1 (Top-down resolution):
CHOOSE:
c : $H \leftarrow \mathbf{B}, C, \mathbf{D} \in$ CLSTACK and
$\mathbf{B}' \subseteq$ FACTSTACK, such that the following conditions hold:
(i) $\mathbf{B}$ and $\mathbf{B}'$ unify with mgu θ,
(ii) $C\theta$ is well-moded,
(iii) $C\theta \stackrel{\sim}{\notin}$ GOALSTACK,
(iv) **IF** $mode(C) = (Out, In)$ **THEN** *recipient*$(C\theta) \notin$ VISITED **ENDIF**
ADD $C\theta$ to GOALSTACK;
IF *mode*$(C) \in \{(In, Out), (In, In)\}$ **THEN**
FETCH at *issuer*$(C\theta)$ all clauses $\{c_1, \ldots, c_n\}$ whose head unifies with $C\theta$ with mgus $\{\gamma_1, \ldots, \gamma_n\}$ respectively ;
FOR EACH $c_i\gamma_i \in \{c_1\gamma_1, \ldots, c_n\gamma_n\}$**DO**
IF $c_i\gamma_i \stackrel{\sim}{\notin}$ CLSTACK **THEN** ADD $c_i\gamma_i$ to CLSTACK **ENDIF**
END FOR EACH
ELSEIF *mode*$(C) = (Out, In)$ **THEN**
FETCH all clauses $\{c_1, \ldots, c_n\}$ stored at *recipient*$(C\theta)$ whose head has mode (Out, In) ;
ADD *recipient*$(C\theta)$ to VISITED;
FOR EACH $c_i \in \{c_1, \ldots, c_n\}$ **DO**
IF $c_i \stackrel{\sim}{\notin}$ CLSTACK **THEN** ADD c_i to CLSTACK **ENDIF**
END FOR EACH
ENDIF
Phase 2 (Bottom-up model-building):
REPEAT
CHOOSE: $H \leftarrow \mathbf{B} \in$ CLSTACK and $\mathbf{B}' \subseteq$ FACTSTACK, such that $\mathbf{B}$ and $\mathbf{B}'$ unify with mgu θ ;
IF $H\theta \notin$ FACTSTACK **THEN** ADD $H\theta$ to FACTSTACK **ENDIF**;
IF *mode*$(H) = (Out, In)$ *AND issuer*$(H\theta) \notin$ VISITED **THEN**
ADD to CLSTACK the clause:
$dummy(X,$ *issuer*$(H\theta)) \leftarrow dummy(X,$ *issuer*$(H\theta))$
ENDIF
UNTIL nothing can be added to FACTSTACK;
IF A is ground and $A \in$ FACTSTACK **THEN** SATISFIED = **TRUE ENDIF**
UNTIL SATISFIED **OR** nothing can be added to FACTSTACK and CLSTACK;
OUTPUT = FACTSTACK;

Figure 2: The Lookup and Inference AlgoRithm (LIAR)

principal's name followed by a *role name*, separated by a dot. For instance $a.r$, and $alice.pictures$ are roles. We use names starting with a lowercase letter (sometimes with subscripts) to indicate role names. (Differently from [13], for the sake of uniformity, principals are denoted by names starting with a lowercase, typically, a, b, d.) A role denotes a set of principals – the members of the role. To indicate which principals populate a role, RT_0 allows a principal to issue four kind of statements:

- *Simple Member*: $a.r \leftarrow d$. "a asserts that d is a member of $a.r$."
- *Simple Inclusion*: $a.r \leftarrow b.r_1$. "a asserts that $a.r$ includes (all members of) $b.r_1$."
- *Linking Inclusion*: $a.r \leftarrow a.r_1.r_2$. "a asserts that $a.r$ includes $b.r_2$ for every b that is a member of $a.r_1$."
- *Intersection Inclusion*: $a.r \leftarrow b_1.r_1 \cap b_2.r_2$. "$a$ asserts that $a.r$ includes every principal who is a member of both $b_1.r_1$ and $b_2.r_2$."

An *RT policy* (indicated by $\mathcal{S}$) is a set of RT statements. Its semantics is defined by translating it into a *semantic program*, *SP*($\mathcal{S}$), which is a Prolog program with only one ternary predicate m. Intuitively, $m(a, r, d)$ indicates that d is a member of the role $a.r$. Given an RT statement c, the *semantic program* of c, *SP*(c), is defined as follows:

$$\begin{aligned}
\textit{SP}(a.r \leftarrow d)) &= m(a, r, d).\\
\textit{SP}((a.r \leftarrow b.r_1)) &= m(a, r, X) :\!- m(b, r_1, X).\\
\textit{SP}((a.r \leftarrow a.r_1.r_2)) &= m(a, r, X) :\!- m(a, r_1, Y), m(Y, r_2, X).\\
\textit{SP}((a.r \leftarrow b_1.r_1 \cap b_2.r_2)) &= m(a, r, X) :\!- m(b_1, r_1, X), m(b_2, r_2, X).
\end{aligned}$$

SP extends to the set of statements in the obvious way: $\textit{SP}(\mathcal{S}) = \{\textit{SP}(c) \mid c \in \mathcal{S}\}$. Finally, given an RT policy $\mathcal{S}$, the semantics of a role $a.r$ is defined in terms of atoms entailed by the semantic program: $[\![a.r]\!]_{SP(\mathcal{S})} = \{d \mid \textit{SP}(\mathcal{S}) \models m(a, r, d)\}$.

The type system of RT$_0$ To ensure traceability, RT_0 comes with a type system [13]. In the original presentation, each role name has two types: an issuer-side type and a subject-side type. Here – also for the sake of simplicity – we assume that each role has just one of the following three type values: *issuer-traces-all (ITA)*, *issuer-traces-def (ITD)*, and *subject-traces-all (STA)*. To extend the results we present here to the full version (i.e., including all possible combinations of RT types) we need to extend Core TuLiP in a straightforward by allowing predicates with multiple modes.

Concerning storage, if a role name r is issuer-traces-all or issuer-traces-def, then principal a has to store all the credentials defining $a.r$. When a role name r is subject-traces-all then for any credential of the form $a.r \leftarrow e$, every subject of this credential must store this credential. The successful discovery requires that each credential in the policy $\mathcal{S}$ be *well-typed*. For the sake of simplicity, we use the following definition of well typed credentials (equivalent to [13]).

Definition 5.1 *Let c be an RT_0 credential. We say that c is* well typed *iff the combination of type value assignments appears as a valid entry in Table 1.*

		$a.r \leftarrow b.r_1$		
	r_1	ITA	ITD	STA
r	ITA	OK		
	ITD	OK	OK	OK
	STA			OK

		$a.r \leftarrow a.r_1.r_2$								
	r_1	ITA			ITD			STA		
	r_2	ITA	ITD	STA	ITA	ITD	STA	ITA	ITD	STA
r	ITA	OK								
	ITD	OK	OK	OK			OK			OK
	STA									OK

		$a.r \leftarrow b_1.r_1 \cap b_2.r_2$								
	r_1	ITA			ITD			STA		
	r_2	ITA	ITD	STA	ITA	ITD	STA	ITA	ITD	STA
r	ITA	OK	OK	OK	OK			OK		
	ITD	OK	OK	OK	OK	OK	OK	OK	OK	OK
	STA			OK			OK	OK	OK	OK

Table 1: Well Typed RT_0 credentials.

For example, take the credential $c : a.r \leftarrow a.r_1.r_2$ and assume first that $type(r)$ and $type(r_1)$ is ITD, and $type(r_2)$ is STA. Then, after checking with Table 1, we see that c is well typed w.r.t. this type value assignment. On the other hand, if $type(r) = type(r_1) = type(r_2) =$ ITD, then c is not well typed as there is no valid entry for this type value assignment in Table 1. Note that simple member credentials (of the form $a.r \leftarrow a$) are always well-typed.

Three *Sorts* of Goals If the set of credentials $\mathcal{S}$ is *well-typed* then there exists a terminating algorithm supporting three *sorts* of goals.

1. "given $a.r$, list all principals in $[\![a.r]\!]_{SP(\mathcal{S})}$"; this goal can be answered provided that r is issuer-traces-all.
2. "given $a.r$ and b, check if b is a member of $[\![a.r]\!]_{SP(\mathcal{S})}$"; this goal can be answered in all cases.
3. "given b, check list all roles $a.r$ such that b is a member of $[\![a.r]\!]_{SP(\mathcal{S})}$"; this goal can be answered only partially: given b the system is able to find all *subject traceable* roles $a.r$ such that b is a member of $[\![a.r]\!]_{SP(\mathcal{S})}$.

5.1 Translating RT_0 into CoreTuLiP

We now demonstrate that CoreTuLiP is – in most cases – more expressive than RT_0 by showing that an arbitrary RT_0 policy can be translated in a straightforward way into an equivalent CoreTuLiP state. First, we define a mapping T from RT_0 to CoreTuLiP.

Definition 5.2 *Let c be an RT$_0$ credential. Then $T(c)$ is defined as follows:*

$$
\begin{array}{rcl}
T(a.r \leftarrow d) & = & r(a,d). \\
T(a.r \leftarrow b.r_1) & = & r(a,X) :- r_1(b,X). \\
T(a.r \leftarrow a.r_1.r_2) & = & \left\{ \begin{array}{l} r(a,X) :- r_2(Y,X), r_1(a,Y). \textit{ if } type(r_1) \neq \textit{ITA}, \\ r(a,X) :- r_1(a,Y), r_2(Y,X). \textit{ otherwise.} \end{array} \right. \\
T(a.r \leftarrow b_1.r_1 \cap b_2.r_2) & = & \left\{ \begin{array}{l} r(a,X) :- r_2(b_2,X), r_1(b_1,X). \textit{ if } type(r_1) \neq \textit{ITA}, \\ r(a,X) :- r_1(b_1,X), r_2(b_2,X). \textit{ otherwise.} \end{array} \right.
\end{array}
$$

Concerning the mode of the predicates, we have: if the RT$_0$ type of r is ITA then the mode of r in $T(c)$ is (In, Out), if the RT$_0$ type of r is ITD then the mode of r in $T(c)$ is (In, In), and if the RT$_0$ type of r is STA then the mode of r in $T(c)$ is (Out, In). □

The following theorem shows that from the view point of the declarative semantics $\mathcal{S}$ and $T(\mathcal{S})$ are equivalent (recall that m is the fixed predicate symbol used in *SP*($\mathcal{S}$)).

Theorem 5.3 *Let $\mathcal{S}$ be an RT$_0$ policy. Then* SP($\mathcal{S}$) $\models m(a,r,d)$ *iff* $T(\mathcal{S}) \models r(a,d)$.

Proof. See the Appendix.

The above statement proves that each RT policy can be translated into a declaratively equivalent CoreTuLiP state. Now, to prove the full equivalence we still have to prove two things, namely that (a) if an RT credential is stored at principal a then its corresponding CoreTuLiP statement is stored at a as well and that (b) the CoreTuLiP system can answer the same goals the RT system can. We start with the following:

Proposition 5.4 *Let c be an RT$_0$ credential.*

(a) if c is stored at a then $T(c)$ is also stored at a.

(b) if c is a well typed then $T(c)$ is traceable.

Proof. See the Appendix.

At last, we have to show how RT goals can be transformed into (legal, i.e., well-moded) CoreTuLiP queries. Since RT does not have a formal notation to express goals we have to be a bit verbose.

Remark 5.5 (Translating RT goals) *Let $\mathcal{S}$ be a well-typed RT$_0$ policy and $T(\mathcal{S})$ its CoreTuLiP equivalent. Let us consider the different sorts of goals supported by RT.*

Sort 1: *the general goal of this sort is "given $a.r$, list all principals in $[\![a.r]\!]_{SP(\mathcal{S})}$". This is translated into the query $r(a,X)$. Notice this goal can be answered in RT only if the role r has type ITA. But in this case the mode of r in $T(\mathcal{S})$ is (I,O), and the query $r(a,X)$ is well-moded w.r.t. it. Therefore, we can conclude that goals of sort 1 can be safely expressed in CoreTuLiP.*

Sort 2: *the general goal of this sort is "given $a.r$ and b, check if b is a member of $[\![a.r]\!]_{SP(\mathcal{S})}$". This is translated into the query $r(a,b)$, which being ground is always well-moded. Therefore, we can conclude that goals of sort 2 can be safely expressed in CoreTuLiP.*

Sort 3: *the general goal of this sort is "given b, list all $a.r$ such that b is a member of $[\![a.r]\!]_{SP(\mathcal{S})}$". Such goals have no corresponding CoreTuLiP translation. The technical reason behind this limitation is purely of syntactic nature: the translation would be higher-order query ($X(Y, b)$, where X and Y are variables). There are two reasons why we believe that in practice this limitation of CoreTuLiP w.r.t. RT is hardly relevant in practice: first, RT allows to express the query, but it is not able to give a complete answer anyhow: it can only find all such $a.r$ which are also* subject traceable. *Secondly (also because RT is not able to provide a full answer), this kind of goals is not used in practice on their own, but only as subgoals of Sort 2 goals.* □

The syntactic inability of CoreTuLiP to express goals of sort 3 is actually a conscious design choice we made to keep the syntax manageable (to express this sort goals we would need a mode "polymorphic" mode system in which the actual mode of an atom does not only depend on its predicate symbol but also on some of its arguments). Indeed, our LIAR algorithm would be able to answer such queries as well.

Summarising, Theorem 5.3, Proposition 5.4, and Remark 5.5 allow us to say that CoreTuLiP is at least as expressive as RT_0, with the small exception of sort 3 goals. In proving this, we have made the restrictive assumption that RT_0 role name has just one of the following three types: ITA (issuer-traces-all), ITD (issuer-traces-def), or STA (subject-traces-all). The extension to the full version (i.e., including all possible combinations of RT types) can be done in a straightforward way by extending Core TuLiP so that it allows predicates with multiple modes.

5.2 A Flexible Syntax

As we said already, CoreTuLiP is simply the core language of the TM system we are developing. The full language will allow credentials with more than two arguments and user defined predicates. Nevertheless, CoreTuLiP is already expressive enough to express complex policies (like thresholds or separation of duty) that in RT require the adoption of special operators (which are present in more expressive members of the RT family RT_1, RT_2, RT^T, or RT^D).

We now want to give a flavour of the syntactic difficulties one encounters with RT when expressing less than trivial statements. Consider the following statement taken from [12] "a says that an entity is a member of $a.r$ if one member of $a.r_1$ and two *different* members of $a.r_2$ all say so". This policy cannot be expressed in RT_0, and to express this in RT one needs to use the so-called *manifold roles*, which extend the notion of roles by allowing role members to be *collections* of entities (rather than just principals). This is done in RT^T by defining the operators $\odot$ and $\otimes$. A *type-5* credential of the form $a.r \leftarrow b_1.r_1 \odot b_2.r_2$ says that $\{s_1 \cup s_2\}$ is a member of $a.r$ if s_1 is a member of $b_1.r_1$ and s_2 is a member of $b_2.r_2$. A *type-6* credential $a.r \leftarrow b_1.r_1 \otimes b_2.r_2$ has a similar meaning, but it additionally requires that $s_1 \cap s_2 = \emptyset$. With these two additional types of credential one can express the above statement as follows:

$$
\begin{aligned}
a.r &\leftarrow a.r_4.r \\
a.r_4 &\leftarrow a.r_1 \odot a.r_3 \\
a.r_3 &\leftarrow a.r_2 \otimes a.r_2
\end{aligned}
$$

In CoreTuLiP, on the other hand, this policy can be expressed quite naturally with the following oneliner:

$$r(a, X) :- r_1(a, Y),\ r(Y, X),\ r_2(a, Z_1),\ r_2(a, Z_2),\ Z_1 \neq Z_2,\ r(Z_1, X),\ r(Z_2, X).$$

Notably, to express this we don't have to use manifold-like structures which are, in our opinion, rather hard to grasp.

6 Conclusions

In this paper we introduce CoreTuLiP, a true Trust Management Language which enjoys the advantages of LP syntax and of its declarative semantics. CoreTuLiP form the basis for the TuLiP TM language we are developing at the UT (which will include user-defined predicates and will enjoy of most features of moded logic programs, including interface with other languages, debugging facilities etc). The main purpose of CoreTuLiP to provide a theoretical basis for the further developments.

CoreTuLiP enjoys the advantages of trust management languages: for instance statements may be issued by multiple authorities and be stored by authorities different than the issuing one. To deal with the problem of finding the credentials when needed for a proof we define the notion of traceable credentials and present a Lookup and Inference AlgoRithm (LIAR), which we show to be correct and complete w.r.t. the standard declarative semantics.

We also compare CoreTuLiP with RT_0 and show that each RT_0 credentials and goals translates in a straightforward way into Core TuLiP (with the small exception of sort 3 goals).

The theoretical relevance of this paper is that we show that it is possible to define a true TM language without leaving the well-established LP formalism. The practical relevance lies in the much greater flexibility, extendibility and accessibility that LP language enjoy with respect to – for instance – RT. As we have discussed, to accommodate various needs, the language RT_0 has developed a relatively large number of extensions, which are in our opinion often hard to grasp. We thought that this was the price we had to pay to have a true TM language, but CoreTuLiP shows that this can be done otherwise.

Future work CoreTuLiP can be extended in several directions. First we plan to investigate extending CoreTuLiP to support non-stratified negation. This is connected to our previous work on $RT_{\ominus}$[15], where we extend RT_0 with *negation-in-context*. Secondly, we are going to add support for integrity constraints for TM systems [8]. We also plan to provide an implementation for CoreTuLiP, possibly supporting the XACML standard.

Acknowledgements We would like to thank Pieter Hartel and Jeroen Doumen from the University of Twente for their feedback and valuable comments to this paper.

References

[1] K. R. Apt. Introduction to Logic Programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, pages 495–574. Elsevier, Amsterdam and The MIT Press, Cambridge, 1990.

[2] K. R. Apt. *From Logic Programming to Prolog*. Prentice Hall, 1997.

[3] K. R. Apt and I. Luitjes. Verification of logic programs with delay declarations. In *AMAST*, volume 936 of *LNCS*, pages 66–90. Springer, 1995.

[4] K. R. Apt and E. Marchiori. Reasoning about Prolog programs: from Modes through Types to Assertions. *Formal Aspects of Computing*, 6(6A):743–765, 1994.

[5] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *Proc. of the 17th IEEE Symposium on Security and Privacy*, pages 164–173. IEEE Computer Society Press, May 1996.

[6] 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.

[7] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. SPKI certificate theory. IETF RFC 2693, September 1999.

[8] S. Etalle and W. H. Winsborough. Integrity constraints in trust management – extended abstract. In G-J. Ahn, editor, *Proc. 10th ACM Symp. on Access Control Models and Technologies (SACMAT)*, pages 1–10. ACM Press, 2005. Extended version available at CoRR: http://arxiv.org/abs/cs.CR/0503061.

[9] 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.

[10] T. Jim. SD3: A trust management system with certified evaluation. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, pages 106–115. IEEE Computer Society Press, May 2001.

[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. 2002 IEEE Symposium on Security and Privacy*, pages 114–130. IEEE Computer Society Press, May 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] J. W. Lloyd. *Foundations of Logic Programming*. Springer, 2 edition, 1993.

[15] M.Czenko, H. Tran, J. Doumen, S. Etalle, P. Hartel, and J. den Hartog. Nonmonotonic trust management for P2P applications. In *Proc. of the 1st International Workshop on Security and Trust Management*, pages 101–116. Elsevier, 2005.

[16] R. Rivest and B. Lampson. SDSI — a simple distributed security infrastructure, October 1996. Available at http://theory.lcs.mit.edu/∼rivest/sdsi11.html.

[17] S. Weeks. Understanding trust management systems. In *Proceedings of 2001 IEEE Symposium on Security and Privacy*, pages 94–105. IEEE Computer Society Press, 2001.

Appendix

In this appendix we include the proofs of various theorems from the paper. The appendix is provided solely for the reader's convenience. Should the paper be accepted we are going to remove the appendix and make it available as Technical Report.

Lemma 4.3 *Let $\mathcal{P}$ be a state and* FACTSTACK *be the result of the algorithm execution for some well-moded query. Let A be an atom in* FACTSTACK. *Then A is ground.*

Proof. The proof proceeds by induction on the size of FACTSTACK. The basic case is trivial as FACTSTACK is empty.
Now, assume that FACTSTACK contains only ground atoms. We are proving that each time a new atom is added to FACTSTACK, it is ground. Notice that an atom is added to FACTSTACK as the result of the bottom-up evaluation of the facts in FACTSTACK and a clause selected from CLSTACK. We have then two cases: (1) the clause selected from CLSTACK in Phase 2 of the algorithm has empty body, (2) the clause selected from CLSTACK in Phase 2 of the algorithm has non-empty body.
Case 1: The clause selected from CLSTACK has empty body.
In such a case, a fact can be added to FACTSTACK only if it is already in CLSTACK. Let $H.$ be a clause selected from CLSTACK. Recall that $\forall C \in$ GOALSTACK, C is well-moded.

1. *mode*$(H) \in \{(In, In), (In, Out)\}$.

   If $H. \in$ CLSTACK then there must be some C in GOALSTACK, such that $\exists \theta = mgu(H, C)$ and $\exists H'. \in \mathcal{P}$ such that H' is stored at *issuer*(C), $H = H'\theta$, and $\theta = mgu(H', C)$. But, by Definition 3.4 (State) all clauses in a state $\mathcal{P}$ are traceable, so that $\forall G \in$ GOALSTACK and $\forall A. \in \mathcal{P}$ such that *mode*$(G) \in \{(In, In), (In, Out)\}$, $A.$ is stored at *issuer*(G), and $\exists \gamma = mgu(A, G)$, $A\gamma$ is ground and will be added to CLSTACK during Phase 1 of the algorithm. Then, as a special case of the observation above, H must be ground.

2. *mode*$(H) = (Out, In)$.

   If $H. \in$ CLSTACK then there must be some C in GOALSTACK such that *mode*$(C) = (Out, In)$, $\exists c \in \mathcal{P}$ such that c is stored at *recipient*(C), and $H. = c$. But, by Definition 3.2 (Traceable, Depositary) and by the fact that every traceable clause is well-formed, $\forall G \in$ GOALSTACK such that *mode*$(G) = (Out, In)$ and $\forall D. \in \mathcal{P}$ such that D is connected to C, $D.$ is ground. Then, as a special case, $H.$ must also be ground.

Case 2: The clause selected from CLSTACK has non-empty body.
When the clause selected from CLSTACK has non-empty body, a fact can be added to FACTSTACK only be the means of the bottom-up evaluation in Phase 2 of the algorithm. Let $c : H \leftarrow \mathbf{B}$ be a clause selected from CLSTACK.

1. *mode*$(H) \in \{(In, In), (In, Out)\}$.

   In such a case each input position in the head of c is ground because before c was added to CLSTACK its head was unified with a well-moded atom from

GOALSTACK. By well-modedness of clauses, each variable V in the output position of the head of c, such that $V \notin VarIn(H)$, must occur in $\mathbf{B}$. Now, assume that $\exists \mathbf{B}' \subseteq$ FACTSTACK such that $\mathbf{B}$ and $\mathbf{B}'$ unify with mgu θ and that $H\theta$ is not ground. Then, it must be that $\exists B \in \mathbf{B}'$ such that B is not ground. But, by the inductive hypothesis, each $B \in \mathbf{B}'$ is ground. This is a contradiction so $H\theta$ must be ground.

2. *mode*$(H) = (Out, In)$.

   If *mode*$(H) = (Out, In)$, then by Definition 3.1 (Well-Formed) $Out(H)$ is ground and by Definition 3.2 (Traceable,Depositary) either $In(H)$ is ground, or $In(H)$ is a variable and $In(H) = In(B_1)$ where B_1 is the first atom in $\mathbf{B}$. Now, assume that $\exists \mathbf{B}' \subseteq$ FACTSTACK such that $\mathbf{B}$ and $\mathbf{B}'$ unify with mgu θ and that $H\theta$ is not ground. Then, it must be that $\exists B \in \mathbf{B}'$ such that B is not ground. But, by the inductive hypothesis, each $B \in \mathbf{B}'$ is ground. This is a contradiction so $H\theta$ must be ground. □

Theorem 4.5 (completeness) *Let $\mathcal{P}$ be a state and then* FACTSTACK, GOALSTACK *be the result of executing LIAR on $\mathcal{P}$ and a given well-moded goal. Then $\forall C \in$* GOALSTACK, *if $\exists$ a successful SLD derivation $\delta : C \xrightarrow{\theta}_{P(\mathcal{P})} \square$ then $C \overset{\theta}{\hookrightarrow}$* FACTSTACK.

Proof. We prove a more general proposition:

Proposition *Let $\mathcal{P}$ be a state and then* FACTSTACK, GOALSTACK *be the result of executing LIAR on $\mathcal{P}$ and a given well-moded goal. Then $\forall C \in$* GOALSTACK*:*

1. *if* mode$(C) = (In, Out)$ *or* mode$(C) = (In, In)$ *and $\exists$ successful SLD derivation $\delta : C \xrightarrow{\theta}_{P(\mathcal{P})} \square$ then $C \overset{\theta}{\hookrightarrow}$* FACTSTACK,
2. *if* mode$(C) = (Out, In)$ *and $\exists$ successful SLD derivation $D \xrightarrow{\theta}_{P(\mathcal{P})} \square$, where D is an atom connected to C then $D \overset{\theta}{\hookrightarrow}$* FACTSTACK.

Proof. The proof proceeds by induction on the length of the derivation.
Base case: $length = 1$.

1. Assume that $\delta : C \xrightarrow{\theta}_{P(\mathcal{P})} \square$ has length 1. In such a case there exists a clause $c : C'. \in P$ such that $mgu(C, C') = \theta$. Note that *mode*$(C) \in \{(In, Out), (In, In)\}$. This means that clause c is stored at *issuer*(C) (the mode is assigned to the predicate symbol and this is the same for C and C').

   Since $C \in$ GOALSTACK then:

   (a) first, at some step in Phase 1 of the algorithm, clause c was fetched at *issuer*(C') and $C'\theta$ was added to CLSTACK;
   (b) then, at some step in Phase 2 of the algorithm, $C'\theta$ was added to FACTSTACK.

   Since $mgu(C, C') = \theta$ the thesis follows.

2. Assume that $\delta : D \xrightarrow{\theta}_{P(\mathcal{P})} \Box$ has length 1. Then there exists a clause $d : D'. \in P$ such that $mgu(D, D') = \theta$. Note that since D is a well-moded goal, *recipient*(D) = *recipient*(D'). Since D is connected to C, *mode*$(D) = (Out, In)$ and d is stored at *recipient*(C) = *recipient*(D). Consequently, since $C \in$ GOALSTACK:

   (a) at some step in Phase 1 of the algorithm clause d was fetched at *recipient*(D') and added to CLSTACK;

   (b) then, at some point in Phase 2, D' was added to FACTSTACK.

   Since $mgu(D, D') = \theta$ the thesis follows.

Inductive case:

1. Assume that there exists an SLD derivation $\delta : C \xrightarrow{\theta}_{P(\mathcal{P})} \Box$, such that $length(\delta) = m > 1$. Then, by well-known result from the theory of Logic Programming (switching lemma) there exists a clause $c : H \leftarrow B_1, \ldots, B_n$ and substitutions $\gamma_0, \gamma_1, \ldots, \gamma_n$ such that:

   - $\gamma_0 = mgu(C, H)$,
   - $\forall i \in [1, n]$ there exists a successful derivation $\delta_i : B_i\gamma_0 \cdots \gamma_{i-1} \xrightarrow{\gamma_i}_{P(\mathcal{P})} \Box$ such that $length(\delta_i) < m$ with *c.a.s.* γ_i,
   - $C\theta$ is a variant of $H\gamma_0\gamma_1 \cdots \gamma_n$.

   Since $mode(C) \in \{(In, Out), (In, In)\}$ then clause c is stored at *issuer*(C), and, since $C \in$ GOALSTACK at some step in Phase 1 of the algorithm, c is fetched at *issuer*(C) and $c\gamma_0$ is added to CLSTACK. We need to prove the following claim:

   Claim 1 *For each* $i \in [1, n]$, $(B_1, \ldots, B_i)\gamma_0 \overset{\gamma_1 \cdots \gamma_i}{\hookrightarrow}$ FACTSTACK.

   Proof of Claim 1. The proof is by induction on i:

   - *Basic case*: $i = 1$.
     Notice that $B_1\gamma_0$ is well-moded. Since $B_1\gamma_0 \xrightarrow{\gamma_1}_{P(\mathcal{P})} \Box$ is a derivation of $length < m$, by inductive hypothesis on the length of the derivation, $B_1\gamma_0 \overset{\gamma_1}{\hookrightarrow}$ FACTSTACK.
   - *Inductive case*:
     Assume $(B_1, \ldots, B_{i-1})\gamma_0 \overset{\gamma_1 \cdots \gamma_{i-1}}{\hookrightarrow}$ FACTSTACK. Notice that, by Corollary 2.5, $B_i\gamma_0\gamma_1 \cdots \gamma_{i-1}$ is well-moded. Since $B_i\gamma_0\gamma_1 \cdots \gamma_{i-1} \xrightarrow{\gamma_i}_{P(\mathcal{P})} \Box$ is a derivation of $length < m$ then, by inductive hypothesis on the length of the derivation, $B_i\gamma_0\gamma_1 \cdots \gamma_{i-1} \overset{\gamma_i}{\hookrightarrow}$ FACTSTACK.
     Notice that, by Corollary 2.4, $B_i\gamma_0, \ldots, \gamma_i$ is ground. By composing the substitutions, $(B_1, \ldots, B_i)\gamma_0 \overset{\gamma_1 \cdots \gamma_i}{\hookrightarrow}$ FACTSTACK, so that the claim follows. $\Box$

Now, from Claim 1 and the fact that $c \in P$, it follows that at some stage of Phase 1 of the algorithm, $H\gamma_0\gamma_1 \cdots \gamma_n$ was added to FACTSTACK. Since $\gamma_0 = mgu(C, H)$, it follows that $C \overset{\theta}{\hookrightarrow}$ FACTSTACK, where $\theta = \gamma_0\gamma_1 \cdots \gamma_n$.

2. Assume that there exists a successful SLD derivation $\delta : D \xrightarrow{\theta}_{P(\mathcal{P})} \Box$, such that $length(\delta) = m > 1$. Then, by well-known result from the theory of Logic Programming (switching lemma) there exists a clause $c : H \leftarrow B_1, \ldots, B_n$ and substitutions $\gamma_0, \gamma_1, \ldots, \gamma_n$ such that:

   - $\gamma_0 = mgu(D, H)$,
   - $\forall i \in [1, n]$ there exists a successful derivation $\delta_i : B_i\gamma_0 \cdots \gamma_{i-1} \xrightarrow{\gamma_i}_{P(\mathcal{P})} \Box$ such that $length(\delta_i) < m$ with *c.a.s.* γ_i,
   - $D\theta$ is a variant of $H\gamma_0\gamma_1 \cdots \gamma_n$.

   Since $mode(D) = (Out, In)$ then either:

   2a $In(H)$ contains a ground term a and c is stored at a, *or*

   2b $In(H)$ is a variable. In such a case, there exists a prefix $B_1, \ldots, B_k$ of $B_1, \ldots, B_n$ satisfying the conditions of Definition 3.2, and c is stored at *issuer*$(B_k) = Out(B_k)$.

   Case 2a.

   Since $C \in$ GOALSTACK and *recipient*(C) = *recipient*(H) = *recipient*(D) then at some stage of Phase 1 of the algorithm c is fetched at *recipient*(C) and added to CLSTACK. We need to prove the following claim:

   Claim 2 *For each* $i \in [1, n]$, $(B_1, \ldots, B_i)\gamma_0 \overset{\gamma_1 \cdots \gamma_i}{\hookrightarrow}$ FACTSTACK.

   Proof of Claim 2. The proof is by induction on i:

   - *Basic case*: $i = 1$.
     Notice that $B_1\gamma_0$ is well-moded. Since $B_1\gamma_0 \xrightarrow{\gamma_1}_{P(\mathcal{P})} \Box$ is a derivation of $length < m$, by inductive hypothesis on the length of the derivation, $B_1\gamma_0 \overset{\gamma_1}{\hookrightarrow}$ FACTSTACK.
   - *Inductive case*:
     Assume $(B_1, \ldots, B_{i-1})\gamma_0 \overset{\gamma_1 \cdots \gamma_{i-1}}{\hookrightarrow}$ FACTSTACK. Notice again that, by Corollary 2.5, $B_i\gamma_0\gamma_1 \cdots \gamma_{i-1}$ is well-moded. Since $B_i\gamma_0\gamma_1 \cdots \gamma_{i-1} \xrightarrow{\gamma_i}_{P(\mathcal{P})} \Box$ is a derivation of $length < m$ then, by inductive hypothesis on the length of the derivation, $B_i\gamma_0\gamma_1 \cdots \gamma_{i-1} \overset{\gamma_i}{\hookrightarrow}$ FACTSTACK.
     By composing the substitutions, $(B_1, \ldots, B_i)\gamma_0 \overset{\gamma_1 \cdots \gamma_i}{\hookrightarrow}$ FACTSTACK, so that the claim follows. $\Box$

   Now, from Claim 2 and the fact that $c \in P$, it follows that at some stage of Phase 1 of the algorithm, $H\gamma_0\gamma_1 \cdots \gamma_n$ was added to FACTSTACK. Since $\gamma_0 = mgu(D, H)$, it follows that $D \overset{\theta}{\hookrightarrow}$ FACTSTACK, where $\theta = \gamma_0\gamma_1 \cdots \gamma_n$. $\Box$

Case 2b.

We first prove the following claim:

Claim 3 *For* $i \in [1,k]$, $(B_1,\ldots,B_i)\gamma_0 \overset{\gamma_1\cdots\gamma_i}{\hookrightarrow}$ FACTSTACK.

Proof of Claim 3. The proof is by induction on i:

- *Basic case*: $i = 1$.
  Notice that $B_1\gamma_0$ is well-moded, *mode*$(B_1) = (Out, In)$, and that *recipient*(B_1) = *recipient*(C) = *recipient*(D). Since $C \in$ GOALSTACK, and since $B_1\gamma_0 \xrightarrow{\gamma_1}_{P(\mathcal{P})} \square$ is a derivation of $length < m$, by inductive hypothesis on the length of the derivation, $B_1\gamma_0 \overset{\gamma_1}{\hookrightarrow}$ FACTSTACK.
- *Inductive case*:
  Assume $(B_1,\ldots,B_{i-1})\gamma_0 \overset{\gamma_1\cdots\gamma_{i-1}}{\hookrightarrow}$ FACTSTACK. Notice again that, by Corollary 2.5, $B_i\gamma_0\gamma_1\cdots\gamma_{i-1}$ is well-moded. Since, by inductive hypothesis on the length of the derivation, $B_{i-1}\gamma_0\cdots\gamma_{i-2} \overset{\gamma_{i-1}}{\hookrightarrow}$ FACTSTACK, at some point of Phase 2 of the algorithm the following $dummy$ clause was added to CLSTACK:

  $$dm : dummy(X, \textit{issuer}(B_{i-1}\gamma_0\cdots\gamma_{i-1}))\text{:-}\\ dummy(X, \textit{issuer}(B_{i-1}\gamma_0\cdots\gamma_{i-1})).$$

  Notice that *mode*$(dummy) = (Out, In)$ and that dm is well-moded. Notice also, that $B_i\gamma_0\cdots\gamma_{i-1}$ is connected to $dummy(X, \textit{issuer}(B_{i-1}\gamma_0\cdots\gamma_{i-1}))$. This implies that at some point of Phase 1 of the algorithm all the clauses moded (Out, In) were fetched at *recipient*$(B_i\gamma_0\cdots\gamma_{i-1})$ and added to CLSTACK.
  Now, since $B_i\gamma_0\gamma_1\cdots\gamma_{i-1} \xrightarrow{\gamma_i}_{P(\mathcal{P})} \square$ is a derivation of $length < m$ then, by inductive hypothesis on the length of the derivation, $B_i\gamma_0\gamma_1\cdots\gamma_{i-1} \overset{\gamma_i}{\hookrightarrow}$ FACTSTACK. By composing the substitutions, $(B_1,\ldots,B_i)\gamma_0 \overset{\gamma_1\cdots\gamma_i}{\hookrightarrow}$ FACTSTACK, and the claim follows. $\square$

Notice that as an immediate consequence of Claim 3, at some point of Phase 1 of the algorithm all the (Out, In) clauses from *issuer*$(B_k\gamma_0\cdots\gamma_k)$ were added to CLSTACK. In particular, clause c was added to CLSTACK.

For the remaining atoms of the body we prove the following claim.

Claim 4 *For* $i \in [k+1,n]$, $(B_{k+1},\ldots,B_i)\gamma_0\cdots\gamma_k \overset{\gamma_{k+1}\cdots\gamma_i}{\hookrightarrow}$ FACTSTACK.

Proof of Claim 4. The proof is again by induction on i:

- *Basic case*: $i = k+1$.
  Notice that $B_{k+1}\gamma_0\cdots\gamma_k$ is well-moded. Since $B_{k+1}\gamma_0\cdots\gamma_k \xrightarrow{\gamma_{k+1}}_{P(\mathcal{P})} \square$ is a derivation of $length < m$, by inductive hypothesis on the length of the derivation $B_{k+1}\gamma_0\cdots\gamma_k \overset{\gamma_{k+1}}{\hookrightarrow}$ FACTSTACK.

- *Inductive case*:
  Assume $(B_{k+1}, \ldots, B_{i-1})\gamma_0 \cdots \gamma_k \overset{\gamma_{k+1}\cdots\gamma_{i-1}}{\hookrightarrow}$ FACTSTACK. Notice again that, by Corollary 2.4, $B_i\gamma_0 \cdots \gamma_k\gamma_{k+1} \cdots \gamma_{i-1}$ is well-moded. Since $B_i\gamma_0 \cdots \gamma_k\gamma_{k+1} \cdots \gamma_{i-1} \xrightarrow{\gamma_i}_{P(\mathcal{P})} \Box$ is a derivation of $length < m$ then, by inductive hypothesis on the length of the derivation, $B_i\gamma_0 \cdots \gamma_k \cdots \gamma_{i-1} \overset{\gamma_i}{\hookrightarrow}$ FACTSTACK. By composing the substitutions, $(B_{k+1}, \ldots, B_i)\gamma_0 \cdots \gamma_k \overset{\gamma_{k+1}\cdots\gamma_i}{\hookrightarrow}$ FACTSTACK, so that the claim follows. $\Box$

From Claim 3 and Claim 4 it follows that for $i \in [1, n]$ $(B_1, \ldots, B_i)\gamma_0 \overset{\gamma_1\cdots\gamma_i}{\hookrightarrow}$ FACTSTACK. From this and from the fact that $c \in P$ it follows that at some stage of Phase 1 of the algorithm, $H\gamma_0\gamma_1 \cdots \gamma_n$was added to FACTSTACK. Since $\gamma_0 = mgu(D, H)$, it follows that $D \overset{\theta}{\hookrightarrow}$ FACTSTACK, where $\theta = \gamma_0\gamma_1 \cdots \gamma_n$. $\Box$ $\Box$

Theorem 5.3 *Let $\mathcal{S}$ be an RT$_0$ policy. Then* SP$(\mathcal{S}) \models m(a, r, d)$ *iff* $T(\mathcal{S}) \models r(a, d)$.

Proof. See the Appendix. Take an RT statement $a.r \leftarrow d$. The meaning of this statement is given by the clause *SP*$(a.r \leftarrow d) = m(a, r, d)$. On the other hand, its CoreTuLiP equivalent is given by $T(a.r \leftarrow d) = r(a, d)$. Generalising this, we now define a mapping *sp_to_tulip* which transforms atoms of the form $m(x, y, z)$ in atoms of the form $y(x, z)$ (the mapping is extended to clauses and programs in the obvious way). It is easy to see that if (1) $m/3$ is the only predicate symbol defined in program P, and if (2) each second argument of each atom occurring in P is ground, then *sp_to_tulip* is only a syntactic mapping, and that for each ground atom A, we have that $P \models A$ iff *sp_to_tulip*$(P) \models$ *sp_to_tulip*(A). Now, since for any *SP*$(\mathcal{S})$ we have that (1) and (2) are both satisfied, and since *sp_to_tulip*(*SP*$(\mathcal{S})) = T(\mathcal{S})$, the thesis follows. $\Box$

Proposition 5.4 *Let c be an RT$_0$ credential.*

(a) if c is stored at a then $T(c)$ is also stored at a.

(b) if c is a well typed then $T(c)$ is traceable.

Proof. *(a)* This is a direct consequence of Definition 5.2 and Definition 3.2. Concerning *(b)*, table 2 shows all possible well typed RT$_0$ credentials, their CoreTuLiP counterparts and the corresponding modes. Using Definition 3.2 it is straightforward to check that for each well typed RT$_0$ credential shown in Table 2 the corresponding CoreTuLiP clause is traceable. $\Box$

RT_0 credential (c)	Possible types for r, r_1, and r_2 such that the credential is well typed			Translation to CoreTuLiP ($T(c)$)	Modes		
	r	r_1	r_2		r	r_1	r_2
$a.r \leftarrow d$	ITA			$r(a, d).$	(I,O)		
	STA			$r(a, d).$	(O,I)		
	ITD			$r(a, d).$	(I,I)		
$a.r \leftarrow b.r_1$	ITA	ITA		$r(a, X) :- r_1(b, X).$	(I,O)	(I,O)	
	STA	STA		$r(a, X) :- r_1(b, X).$	(O,I)	(O,I)	
	ITD	ITA		$r(a, X) :- r_1(b, X).$	(I,I)	(I,O)	
	ITD	ITD		$r(a, X) :- r_1(b, X).$	(I,I)	(I,I)	
	ITD	STA		$r(a, X) :- r_1(b, X).$	(I,I)	(O,I)	
$a.r \leftarrow a.r_1.r_2$	ITA	ITA	ITA	$r(a, X) :- r_1(a, Y), r_2(Y, X).$	(I,O)	(I,O)	(I,O)
	STA	STA	STA	$r(a, X) :- r_2(Y, X), r_1(a, Y).$	(O,I)	(O,I)	(O,I)
	ITD	ITA	ITA	$r(a, X) :- r_1(a, Y), r_2(Y, X).$	(I,I)	(I,O)	(I,O)
	ITD	ITA	ITD	$r(a, X) :- r_1(a, Y), r_2(Y, X).$	(I,I)	(I,O)	(I,I)
	ITD	ITA	STA	$r(a, X) :- r_1(a, Y), r_2(Y, X).$	(I,I)	(I,O)	(O,I)
	ITD	ITD	STA	$r(a, X) :- r_2(Y, X), r_1(a, Y).$	(I,I)	(I,I)	(O,I)
$a.r \leftarrow b_1.r_1 \cap b_2.r_2$	ITA	ITA	ITA	$r(a, X) :- r_1(b_1, X), r_2(b_2, X).$	(I,O)	(I,O)	(I,O)
	ITA	ITA	ITD	$r(a, X) :- r_1(b_1, X), r_2(b_2, X).$	(I,O)	(I,O)	(I,I)
	ITA	ITA	STA	$r(a, X) :- r_1(b_1, X), r_2(b_2, X).$	(I,O)	(I,O)	(O,I)
	ITA	ITD	ITA	$r(a, X) :- r_2(b_2, X), r_1(b_1, X).$	(I,O)	(I,I)	(I,O)
	ITA	STA	ITA	$r(a, X) :- r_2(b_2, X), r_1(b_1, X).$	(I,O)	(O,I)	(I,O)
	STA	STA	STA	$r(a, X) :- r_2(b_2, X), r_1(b_1, X).$	(O,I)	(O,I)	(O,I)
	STA	STA	ITA	$r(a, X) :- r_2(b_2, X), r_1(b_1, X).$	(O,I)	(O,I)	(I,O)
	STA	STA	ITD	$r(a, X) :- r_2(b_2, X), r_1(b_1, X).$	(O,I)	(O,I)	(I,I)
	STA	ITA	STA	$r(a, X) :- r_1(b_1, X), r_2(b_2, X).$	(O,I)	(I,O)	(O,I)
	STA	ITD	STA	$r(a, X) :- r_2(b_2, X), r_1(b_1, X).$	(O,I)	(I,I)	(O,I)
	ITD	ITA	ITA	$r(a, X) :- r_1(b_1, X), r_2(b_2, X).$	(I,I)	(I,O)	(I,O)
	ITD	ITA	ITD	$r(a, X) :- r_1(b_1, X), r_2(b_2, X).$	(I,I)	(I,O)	(I,I)
	ITD	ITA	STA	$r(a, X) :- r_1(b_1, X), r_2(b_2, X).$	(I,I)	(I,O)	(O,I)
	ITD	ITD	ITA	$r(a, X) :- r_2(b_2, X), r_1(b_1, X).$	(I,I)	(I,I)	(I,O)
	ITD	STA	ITA	$r(a, X) :- r_2(b_2, X), r_1(b_1, X).$	(I,I)	(O,I)	(I,O)
	ITD	STA	STA	$r(a, X) :- r_2(b_2, X), r_1(b_1, X).$	(I,I)	(O,I)	(O,I)
	ITD	STA	ITA	$r(a, X) :- r_2(b_2, X), r_1(b_1, X).$	(I,I)	(O,I)	(I,O)
	ITD	STA	ITD	$r(a, X) :- r_2(b_2, X), r_1(b_1, X).$	(I,I)	(O,I)	(I,I)
	ITD	ITD	STA	$r(a, X) :- r_2(b_2, X), r_1(b_1, X).$	(I,I)	(I,I)	(O,I)
	ITD	ITD	ITD	$r(a, X) :- r_2(b_2, X), r_1(b_1, X).$	(I,I)	(I,I)	(I,I)

Table 2: Well typed RT_0 credentials and the corresponding CoreTuLiP traceable clauses their modes (I=In, O=Out)