Abstract

SPKI/SDSI is a language for expressing distributed access control policy, derived from SPKI and SDSI. We provide a first-order logic (FOL) semantics for SDSI, and show that it has several advantages over previous semantics. For example, the FOL semantics is easily extended to additional policy concepts and gives meaning to a larger class of access control and other policy analysis queries. We prove that the FOL semantics is equivalent to the string rewriting semantics used by SDSI designers, for all queries associated with the rewriting semantics. We also provide a FOL semantics for SPKI/SDSI and use it to analyze the design of SPKI/SDSI. This reveals some problems. For example, the standard proof procedure in RFC 2693 is semantically incomplete. In addition, as noted before by other authors, authorization tags in SPKI/SDSI are algorithmically problematic, making a complete proof procedure unlikely. We compare SPKI/SDSI with RT\textsuperscript{C1}, which is a language in the RT Role-based Trust-management framework that can be viewed as an extension of SDSI. The constraint feature of RT\textsuperscript{C1}, based on Constraint Datalog, provides an alternative mechanism that is expressively similar to SPKI/SDSI tags, semantically natural, and algorithmically tractable.

1 Introduction

In 1996, Rivest and Lampson [24] proposed a new public-key infrastructure, called the Simple Distributed Security Infrastructure (SDSI), featuring the use of linked local names. Concurrently, Ellison et al. developed the Simple Public Key Infrastructure (SPKI), which emphasizes delegation of authorization. In 1997, the two efforts were merged, leading to a system called SPKI/SDSI. The standard reference on SPKI/SDSI is RFC 2693 [4], with a later paper [3], whose authors include several designers of SPKI/SDSI, providing certificate chain reduction algorithms for SPKI/SDSI, clearer descriptions of many features, and certain minor design changes. SPKI/SDSI can be viewed as a trust-management (TM) language for access control in distributed systems. A TM language generally has a syntax for specifying policy statements and queries and a semantic relation \( \vdash \). Given a set\(^1\) \( \mathcal{P} \) of policy statements and a query \( \mathcal{Q} \), such as a request of access to a resource, \( \mathcal{P} \vdash \mathcal{Q} \) means that the policy \( \mathcal{P} \) supports \( \mathcal{Q} \).

In SPKI/SDSI, policy statements take the form of name-definition certificates (or name certs), authorization certificates (or auth certs), and Access Control List (ACL) entries. Name certs, which originally came from SDSI, are also called 4-tuples. Principals are identified with public keys, and each principal has its own name spaces for names. A local name, which is identified by a principal

\(^1\)In this paper, we always assume that the set of policy statements is finite.
and an identifier, is bound to a set of principals that we call the members of the local name. Only \( K \) can issue name certs that determine the members of \( KA \). \( K \) can define \( KA \) to include a principal, a local name, or a linked local name (also called an extended name). We use SDSI to refer to the sub-language of SPKI/SDSI that just has name certs. Auth certs and ACL entries are also called 5-tuples; they came originally from SPKI. In a 5-tuple, the issuing principal grants certain authorization to a subject, which can be a principal, a local name, an extended name, or a threshold subject. The issuing principal can also specify whether the subject can further delegate the authorization it receives in a 5-tuple.

A set of SPKI/SDSI statements defines a policy, and many properties of a policy are of interest to its authors and users. The most basic query is whether a policy allows a principal to access a resource. However, it may also be important to determine safety and availability properties of a policy [19], for example, whether one principal can issue certificates that change the consequences of certificates issued by another. RFC 2693 does not explicitly specify a class of queries that can be made against a policy, but provides operational rules for using SPKI/SDSI name certs to reduce extended names and local names and for producing new 5-tuples from existing 5-tuples. The 5-tuple reductions implicitly define a class of access queries for SPKI/SDSI as well as a proof procedure for answering these queries.

The semantics of SPKI/SDSI has attracted a lot of interest in the security research community. Beginning with Abadi [1], significant effort has gone into finding a logic-based semantics for both SDSI naming alone and SPKI/SDSI. A logic developed by Howell and Kotz [9] extends the ABLP logic [2, 13] with a restricted form of delegation and provides a semantics for SPKI/SDSI authorization. Halpern and van der Meyden subsequently developed the Logic of Local Name Containment (LLNC) [6] and then extend the logic to deal with 5-tuples in SPKI [7]. These studies all use specialized propositional modal logics. Li [14] provides a logic programming based reading for SPKI/SDSI, which has several drawbacks. In the present paper, we use standard first-order logic (FOL) and to explicitly consider the class of queries supported by the semantics.

In our study of the SDSI naming portion of SPKI/SDSI (Section 2 of this paper), we present a FOL semantics based on translating each name cert into a Datalog\(^2\) clause. Since Datalog is a subset of first-order logic, this leads to a FOL semantics; since Datalog has a computational interpretation, a set \( P \) of name certs induces a Datalog program. Since the semantics of a policy is defined by classical logical implication, this approach allows any FOL formula to be understood as a meaningful query against a policy. We prove that the FOL semantics is equivalent to the string rewriting semantics used by SDSI designers, for all queries associated with the rewriting semantics. The advantages of our approach over previous logics are the following. It captures the set-based semantic intuition of SDSI, which is used in Clarke et al. [3] and Halpern and van der Meyden [6]. It uses classical first-order logic rather than more complex modal logics. The FOL semantics contains more information in the sense that a larger class of queries can be formulated and understood in the semantics. The semantics is easily extended to support useful extensions to SDSI; and, finally, the relationship between the FOL semantics and logic programming provides an efficient method to answer a large class of queries.

In our study of full SPKI/SDSI, we first describe a trust management language \( RT_1^C \) [16], which uses Constraint Datalog as its semantic foundation. We then give a FOL semantics for SPKI/SDSI by treating authorization tags and validity specification as constraints, and examine

\(^2\)Datalog is a limited form of logic programming, which does not have non-constant function symbols. Datalog is also a query language for relational databases.
several design issues of SPKI/SDSI, using $RT_1^C$ for comparison. The language $RT_1^C$ [16], which may be viewed as an extension of SDSI, is a language from the RT family of Role-based Trust-management languages [18, 20]. One characteristic of $RT_1^C$ is the use of various constraint domains to express policies about structured resources such as file hierarchies and standard concepts such as time of day or days of the week. After developing a semantics for SPKI/SDSI that treats authorization tags and validity specification as constraints, we observe that SPKI’s 5-tuple reduction procedure is semantically incomplete. One reason is that reduction does not handle union of tags. For example, if $K_1$ grants to $K_2$ in two certificates with two authorization tags ($\ast$ range numeric ge 1 le 5) and ($\ast$ numeric range ge 4 le 10), it is not possible to derive that $K_1$ grants to $K_2$ ($\ast$ range numeric ge 2 le 7), which is a logical implication of the two 5-tuples. Another problem, pointed out by Howell [8], is that the intersection of two authorization tags may be a set that is not representable by any finite set of authorization tags. This suggests that no rewriting algorithm dealing with SPKI/SDSI tags can be semantically complete. We then discuss several design issues of SPKI/SDSI, referring to $RT_1^C$ for comparison. Among other benefits of a logical point of view, the constraint feature of $RT_1^C$ is expressively similar to SPKI/SDSI tags, but semantically natural and algorithmically tractable.

We give FOL semantics and equivalence results for the SDSI naming portion of SPKI/SDSI in Section 2, with trust management language $RT_1^C$ [16] and Constraint Datalog in Section 3, FOL semantics and analysis of SPKI/SDSI in Section 4. We conclude in Section 5.

2 A First-Order Logic (FOL) semantics for SDSI

We use $\mathcal{K}$ denote the set of all principals and use $K$, often with subscripts or superscripts, to denote a principal. An identifier is a word over some given standard alphabet. The set of all identifiers is denoted by $\mathcal{A}$, and an identifier is denoted by $A$ or $B$ (often with subscripts). We assume that both $\mathcal{K}$ and $\mathcal{A}$ are countable.³

A name string⁴ is a principal followed by 0 or more identifiers. The set of all name strings is denoted by $\mathcal{S}$. $\mathcal{S}$ is a subset of the set of all strings over the alphabet $\mathcal{K} \cup \mathcal{A}$, i.e., $\mathcal{S} \subset (\mathcal{K} \cup \mathcal{A})^*$. A local name is of the form $K.A$, where $K \in \mathcal{K}$ and $A \in \mathcal{A}$. The set of all local names is denoted by $\mathcal{N}_L$. An extended name is of the form of a principal followed by more than one identifier. The set of all extended names is denoted by $\mathcal{N}_E$. Clearly, $\mathcal{S} = \mathcal{K} \cup \mathcal{N}_L \cup \mathcal{N}_E$, i.e., name strings are either principals, local names, or extended names.

A name-definition certificate (name cert) $C$ is a signed 4-tuple $(K, A, S, V)$, where

- $K \in \mathcal{K}$ is a public key called the issuer; the certificate is signed by $K$.
- $A \in \mathcal{A}$ is an identifier.
- $S \in \mathcal{S}$ is a name string, called the subject.
- The validity specification $V$ provides information regarding the validity of the certificate.

³Besides principals and identifiers, SDSI 1.1 [24] also has special roots. A special root is an identifier that is bound to the same principal in every name space. It is not present in SPKI/SDSI, and we do not deal with special roots in this paper. However, it would be straightforward to handle them.

⁴Name strings are called terms in [3, 10]. Since we will be using first-order logic, we avoid the use of terms here.
We say the 4-tuple \((K, A, S, V)\) defines the local name \(K\ A\) and \(S\) is the definition. In this section, we ignore the validity specification \(V\). We assume that the validity of name certs are checked, and only valid certificates are considered. We will discuss how to handle the validity specification in the logical semantics in Section 4.2. When ignoring the validity specification, we write \((K, A, S, \cdot)\) for a 4-tuple.

In the rest of this section, we first describe three semantics for SDSI that have appeared in the literature, sometimes implicitly, then introduce the FOL semantics and prove its equivalence to other semantics; finally, we compare the FOL semantics with other semantics for SDSI.

2.1 A String Rewriting Semantics for SDSI

In RFC 2693, a 4-tuple-reduction mechanism is given. When reducing a name string using a 4-tuple, the rule is to replace a local name in the name string by its definition in the 4-tuple. Clarke et al. [3] made it explicit that 4-tuple reduction is essentially string rewriting. In the string rewriting semantics for SDSI, a 4-tuple \((K,A,S,\cdot)\) is viewed as a rewriting rule \(K\ A \rightarrow S\). The two 4-tuple-reduction rules in Section 6.4 of RFC 2693 become the following rewriting.

Rewriting \(K_1 A_1 A_2 \cdots A_\ell\) using \(K_1 A \rightarrow K_2\) gives \(K_2 A_2 \cdots A_\ell\).

Rewriting \(K_1 A_1 A_2 \cdots A_\ell\) using \(K_1 A \rightarrow K_2 B_1 \cdots B_k\) gives \(K_2 B_1 \cdots B_k A_2 \cdots A_\ell\).

In the rest of this paper, we will often use the rewriting rule format to represent 4-tuples. That is, we will write \(K\ A \rightarrow S\) instead of \((K,A,S,\cdot)\) to represent a 4-tuple.

Definition 1 (Rewriting Semantics for SDSI: RS[\(\mathcal{P}\)]) Given a set \(\mathcal{P}\) of 4-tuples, we use \(\text{RS}[\mathcal{P}]\) to denote the rewriting system generated by viewing each 4-tuple in \(\mathcal{P}\) as a rewriting rule. Given two name strings \(S_1\) and \(S_2\), we write \(\text{RS}[\mathcal{P}] \triangleright S_1 \rightarrow S_2\) if one can rewrite \(S_1\) into \(S_2\) in one step using a rewriting rule from \(\text{RS}[\mathcal{P}]\). We write \(\text{RS}[\mathcal{P}] \triangleright S_1 \stackrel{\ast}{\rightarrow} S_2\) if using rewriting rules from \(\text{RS}[\mathcal{P}]\), one can rewrite \(S_1\) into \(S_2\) in zero or more steps.

A set of 4-tuples defines a binary relation \(\ast\rightarrow\) among name strings. The class of string rewriting systems generated by sets of 4-tuples have the following characteristics: when rewriting a name string, the rewriting replaces the first two symbols, which form a local name, with another name string, and the result is also a name string. Jha and Reps [10] pointed out that SDSI string rewriting systems correspond exactly to the class of string rewriting systems modelled using push-down systems. This connection reduces certain computation problems in SDSI to reachability analysis and model checking in push-down systems, for which efficient algorithms exist.

Given a set of 4-tuples, for any name string \(S\), there exists a set of principals that \(S\) can rewrite into. This determines a valuation for every name string, and leads to the set-theoretic semantics for SDSI.

2.2 A Set-theoretic Semantics

In the set-theoretic semantics for SDSI, every name string has a valuation that is a set of principals. Clarke et al. [3] use an informal description of this semantics as the semantic intuition for SDSI. Halpern and van der Meyden [6] use the same spirit to provide a semantics for LLNC. Li, Winsborough, and Mitchell [20] use a variation as the semantics for \(RT_0\), which can be viewed as SDSI enhanced with the intersection operator in the subject. The following presentation of the semantics follows [20].
Definition 2 (Set-theoretic Semantics for SDSI: \( \mathcal{M}[\mathcal{P}] \)) Given any function \( f : \mathcal{N}_L \to \wp(\mathcal{K}) \) mapping local names to sets of principals, where \( \wp(\mathcal{K}) \) is the power set of \( \mathcal{K} \), we define \( \mathcal{V}_f : \mathcal{S} \to \wp(\mathcal{K}) \) as follows (\( \mathcal{V}_f \) extends \( f \) to give a valuation for every name string):

\[
\begin{align*}
\mathcal{V}_f(K) &= \{K\} \\
\mathcal{V}_f(K A) &= f(K A) \\
\mathcal{V}_f(K_1 A_1 \cdots A_l) &= \bigcup_{K_2 \in f(K_1, A_1)} \mathcal{V}_f(K_2 A_2 \cdots A_l), \text{ where } l > 1
\end{align*}
\]

A set \( \mathcal{P} \) of 4-tuples defines the following system of set containments:

\[ \text{SC}[\mathcal{P}] = \{ f(K A) \supseteq \mathcal{V}_f(S) \mid K A \mapsto S \in \mathcal{P} \} \]

The semantics of \( \mathcal{P}, \mathcal{M}[\mathcal{P}] \), is defined to be the least function\(^5\) \( f : \mathcal{N}_L \to \wp(\mathcal{K}) \) that satisfies \( \text{SC}[\mathcal{P}] \). We use a least-solution definition for \( \mathcal{M}[\mathcal{P}] \) because 4-tuples may define local names recursively. Note that \( \mathcal{V} \) naturally extends \( \mathcal{M}[\mathcal{P}] \) and gives a valuation for each name string. 

We now show that \( \mathcal{M}[\mathcal{P}] \) is well defined and present a straightforward way to construct it finitely. We define the following sequence of functions \( f^i : \mathcal{N}_L \to \wp(\mathcal{K}), i \in \mathbb{N} \)

\[
\begin{align*}
f^0(K A) &= \emptyset \\
f^{i+1}(K A) &= \bigcup_{K A \mapsto S \in \mathcal{P}} \mathcal{V}_f(S)
\end{align*}
\]

It is easy to show by induction that, for any \( K A \) not defined in \( \mathcal{P} \), \( f^i(K A) = \emptyset \) for any \( i \in \mathbb{N} \), and that, for any \( K A \) and any \( i \in \mathbb{N} \), \( f^i(K A) \) only contains principals occurring in \( \mathcal{P} \). It follows that the set of all functions that could occur in the sequence \( f^i \) forms a finite lattice. In addition, the sequence \( f^i \) is nondecreasing. Therefore, the least fixed point \( f^\omega \) of the sequence, given by \( f^\omega = \bigcup_{i \geq 0} f^i(K A) \), exists; in addition, \( f^\omega = \mathcal{M}[\mathcal{P}] \). The function \( f^\omega \) clearly satisfies \( \text{SC}[\mathcal{P}] \), and one can show by induction that any function that satisfies \( \text{SC}[\mathcal{P}] \) is greater than or equal to \( f^\omega \).

This set-theoretic semantics contains less information than the rewriting semantics, in the following sense. We will show that, if \( \text{RS}[\mathcal{P}] \triangleright S_1 \dashv S_2 \), then \( \mathcal{V}_{\mathcal{M}[\mathcal{P}]}(S_1) \supseteq \mathcal{V}_{\mathcal{M}[\mathcal{P}]}(S_2) \). (This follows from Proposition 1 in Section 2.3 and Theorem 3 in Section 2.5.) However, the converse is only guaranteed to be true when \( S_2 \) is a principal, \( i.e. \), \( \mathcal{V}_{\mathcal{M}[\mathcal{P}]}(S) \supseteq K \) if and only if \( \text{RS}[\mathcal{P}] \triangleright S \dashv K \) (This follows from Proposition 1 in Section 2.3 and Theorem 3 in Section 2.5; see also Proposition 2 in Section 2.5.) It may well be the case that \( \mathcal{V}_{\mathcal{M}[\mathcal{P}]}(S_1) \supseteq \mathcal{V}_{\mathcal{M}[\mathcal{P}]}(S_2) \), but there is no rewriting relationship between \( S_1 \) and \( S_2 \). For example, given \( \mathcal{P} \) that contains the two four tuples \( K A_1 \mapsto K_2 \) and \( K A_2 \mapsto K_2 \), the valuation of \( K A_1 \) clearly contains the valuation of \( K A_2 \); however, using \( \mathcal{P} \), one cannot rewrite \( K A_1 \) into \( K A_2 \). In other words, \( \text{RS}[\mathcal{P}] \triangleright S_1 \dashv S_2 \) if not equivalent to \( \forall K((\text{RS}[\mathcal{P}] \triangleright S_1 \dashv K) \iff (\text{RS}[\mathcal{P}] \triangleright S_2 \dashv K)) \); the former is stronger than the latter.

When making access control decisions, the valuations of name strings are necessary, but the rewriting relationship between two name strings are not directly useful. However, this relation is very helpful for understanding the effect of 4-tuples. If \( S_1 \) rewrites into \( S_2 \) from \( \mathcal{P} \), then \( S_1 \) rewrites\(^6\)

\(^5\) One function \( f : \mathcal{N}_L \to \wp(\mathcal{K}) \) is less than another \( g : \mathcal{N}_L \to \wp(\mathcal{K}) \) if \( f(K A) \subseteq g(K A) \) for every local name \( K A \in \mathcal{N}_L \).

\(^6\)
We observe that contains \( \exists \) contains no matter what new policy statements are added. This is especially useful for studying the availability properties of policies when policies may change, e.g., statements may be added and/or removed [19].

### 2.3 A LP Semantics for SDSI

The set-theoretic semantics can be captured naturally using logic programs. This observation was made by Halpern and van der Meyden [6]. It was also used implicitly in the semantics of the RT framework [20, 18]. The semantics of \( RT_0 \), the basic component of the RT framework, was first defined using sets [20]. When additional features, such as internal structures of role terms (role terms correspond to identifiers in SDSI) are added, a logic programming based semantics is used [18]. The advantage of the LP approach over the set approach is that it is modular and easily extended to additional forms of policy statements.

**Definition 3 (LP Semantics for SDSI)** We use one ternary predicate \( m \); intuitively, \( m(K, A, K') \) means that \( K' \) is in the valuation of the local name \( K A \). We define a macro contains, which takes a name string and a logical variable as parameters, and defines a first-order logic formula.

\[
\begin{align*}
\text{contains}[K][z] & \quad \text{is } (K = z) \\
\text{contains}[K A][z] & \quad \text{is } m(K, A, z) \\
\text{contains}[K A_1 A_2 \cdots A_\ell][z] & \quad \text{is } \exists y_1 (m(K, A_1, y_1) \land \text{contains}[y_1 A_2 \cdots A_\ell][z]), \text{ where } \ell > 1
\end{align*}
\]

Given a set \( \mathcal{P} \) of 4-tuples, we define \( \text{LP}[\mathcal{P}] \) as follows:

\[
\text{LP}[\mathcal{P}] = \{ \forall z (\text{contains}[K A][z] \iff \text{contains}[S][z]) \mid K A \rightarrow S \}
\]

To see that \( \forall z (\text{contains}[K A][z] \iff \text{contains}[S][z]) \) is a logic programming clause (i.e., Horn clause), observe that \( \text{contains}[K A_1 A_2 \cdots A_\ell][z] \) is logically equivalent to \( \exists y_1 \exists y_2 \cdots \exists y_{\ell-1} m(K, A_1, y_1) \land m(y_1, A_2, y_2) \land \cdots \land m(y_{\ell-1}, A_\ell, z) \). Further observe that the above definition of \( \text{LP}[\mathcal{P}] \) is equivalent to the following.

\[
\text{LP}[\mathcal{P}] = \left\{ \begin{array}{ll}
m(K, A, K_1) & \mid K A \rightarrow K_1 \in \mathcal{P} \\
m(K, A, z) \land m(K_1, A_1, z) & \mid K A \rightarrow K_1 A_1 \in \mathcal{P} \\
m(K, A, z) \land m(K_1, A_1, y_1) \land \cdots \land m(y_{\ell-1}, A_\ell, z) & \mid K A \rightarrow K_1 A_1 \cdots A_\ell \in \mathcal{P}, \ell > 1
\end{array} \right\}
\]

The semantics of \( \mathcal{P} \) is defined to be the minimal Herbrand model of \( \text{LP}[\mathcal{P}] \). If an atom \( m(K_1, A, K_2) \) is in the minimal Herbrand model of \( \text{LP}[\mathcal{P}] \), we write \( \text{LP}[\mathcal{P}] \models m(K_1, A, K_2) \).

We now show that the LP semantics is equivalent to the set-theoretic semantics. Given a set \( \mathcal{P} \) of 4-tuples, consider the set \( G \) of all functions \( g : \mathcal{N}_L \rightarrow \wp(K) \) that satisfies the following two conditions: (1) \( g(KA) = \emptyset \) if either \( K \) or \( A \) does not appear in \( \mathcal{P} \); (2) for any local name \( K A \), \( g(KA) \) contains only principals in \( \mathcal{P} \). There exists a bijection between \( G \) and the set of all the Herbrand interpretations of \( \text{LP}[\mathcal{P}] \). Given a function \( g \in G \), the corresponding interpretation is obtained by including \( m(KA, K') \) in the interpretation if and only if \( g(KA) \supseteq K' \). Similarly, one obtains a function from each Herbrand interpretation, by assigning \( K A \) to the set that contains each \( K' \) such that \( m(KA, K') \) is in the interpretation. Furthermore, the definition of contains extends the predicate \( m \) to determine members of name strings, in exactly the same way in which
Proposition 1 (Equivalence of LP semantics and set-theoretic semantics) Given a set \( P \) of 4-tuples, a name string \( S \) and a principal \( K' \), \( \text{LP}[P] \models \text{contains}[S][K'] \) if and only if \( \forall M \models_{P}(S) \ni K' \). In particular, for any local name \( K A \), \( \text{LP}[P] \models m(K, A, K') \) if and only if \( M(P)(K A) \ni K' \).

The LP semantics is an attractive logical semantics. It is natural in the sense that it directly captures the set-based semantics intuition. It can also be used for computation purposes. Observe per rule. For example, the clause “\( m(K, A, z); - m(K_1, A_1, y_1) \), \( m(y_1, A_2, y_2) \), \( m(y_2, A_3, z) \)” is equivalent to the two clauses “\( m(K, A, z); - m(K_1, A', y_2) \), \( m(y_2, A_3, z) \)” and “\( m(K_1, A', y_2); - m(K_1, A_1, y_1) \), \( m(y_1, A_2, y_2) \)” where \( A' \) is an identifier not appearing in \( P \). This means that, given a set \( P \) of 4-tuples with total size \( N \), let \( L \) be the length of the longest extended name, any Horn query can be answered in time \( O(N^{3}L) \). This complexity is the same as the complexity bound derived in three papers [3, 20, 10] using three different methods.

2.4 A FOL Semantics

The LP semantics has the same limitation as the set-theoretic semantics, it cannot be used to directly determine whether \( S_1 \) rewrites into \( S_2 \). Such a query cannot be expressed using Horn queries. We now propose a first-order logic semantics to address this issue. The idea is very simple. Each Horn clause can be viewed as a first-order sentence; and the logic program \( \text{LP}[P] \) can be viewed as a first-order theory. The rewriting query can be viewed as a first-order formula, and logical implication defines the semantics.

Definition 4 Given a set \( P \) of 4-tuples, we define \( \text{Th}[P] \) to be the following first-order theory:

\[
\text{Th}[P] = \{ \forall z(\text{contains}[K A][z] \iff \text{contains}[S][z]) \mid K A \rightarrow S \in P \}
\]

A query whether \( \text{RS}[P] \models S_1 \rightarrow S_2 \) can be answered by checking whether \( \text{Th}[P] \models \forall z(\text{contains}[S_1][z] \iff \text{contains}[S_2][z]) \). Note that \( \text{Th}[P] \) is the same as \( \text{LP}[P] \), here we just view it as a FOL theory.

Other first-order logic formulas can also be used as meaningful queries. For example, the formula \( \exists z(m(K_1, A_1, z) \iff m(K_1, A_2, z)) \iff \exists z(m(K_2, A_1, z) \land m(K_2, A_2, z)) \) means that if \( K_2 A_1 \) and \( K_2 A_2 \) shares a common member, then \( K_1 A_1 \) and \( K_1 A_2 \) also shares a common member.

2.5 Equivalence Among SDSI’s Semantics

Proposition 1 says that the set-theoretic semantics is equivalent to the LP semantics for membership queries. The LP semantics can be viewed as a special case of the FOL semantics where only Horn queries are allowed. However, to the best of our knowledge, the relationship between the rewriting semantics and the other three semantics have not been established and proved in literature before.
In this section, we prove the following equivalence: given any set $\mathcal{P}$ of 4-tuples, $RS[\mathcal{P}] \vdash S_1 \rightarrow S_2$ if and only if $Th[\mathcal{P}] \models \forall z(contains[S_1][z] \iff contains[S_2][z])$. In addition, we establish a way to use logic programs to efficiently determine whether $Th[\mathcal{P}] \models \forall z(contains[S_1][z] \iff contains[S_2][z])$, and prove that this approach is correct.

First, we prove a proposition that will be useful in proving the main theorem.

**Proposition 2** Given a set $\mathcal{P}$ of policy statements, if $Th[\mathcal{P}] \models m(K, A, K')$, then $RS(\mathcal{P}) \vdash K A \rightarrow K'$.

See Appendix A for the proof.

We need the following definition, which helps in transforming a query $Th[\mathcal{P}] \models \forall z(contains[S_1][z] \iff contains[S_2][z])$ into a Horn query. This definition gives a canonical way for placing a principal $K'$ in the valuation of a name string $S$.

**Definition 5**

Given a set $\mathcal{P}$ of 4-tuples, a name string $S = K_1 A_1 A_2 \cdots A_\ell$ where $\ell \geq 1$, and a principal $K'$, define

$$add(\mathcal{P}, S, K') = \left\{ \begin{array}{ll}
\mathcal{P} \cup \{K_1 A_1 \rightarrow K'\} & \text{when } l = 1 \\
\mathcal{P} \cup \{K_1 A_1 \rightarrow K_1', K_1' A_2 \rightarrow K_2', \cdots, K_{l-1}' A_\ell \rightarrow K'\} & \text{when } l > 1
\end{array} \right.$$ 

where $K_1', \cdots, K_{l-1}'$ are principals not in $K[\mathcal{P}] \cup \{K_1, K'\}$.

It is easy to see that $RS(add(\mathcal{P}, K_1 A_1 A_2 \cdots A_\ell, K')) \vdash K_1 A_1 A_2 \cdots A_\ell \rightarrow K'$. We are now ready to state the main theorem of this section.

**Theorem 3**

Given a set $\mathcal{P}$ of 4-tuples, and two name strings $S_1$ and $S_2$, the following three statements are equivalent.

1. $RS[\mathcal{P}] \vdash S_1 \rightarrow S_2$.
2. $Th[\mathcal{P}] \models \forall z(contains[S_1][z] \iff contains[S_2][z])$.
3. $Th[\mathcal{P}'] \models m(K, A, K')$, where $K A$ is any local name not defined in $\mathcal{P}$, and $\mathcal{P}'$ and $K'$ depend on the form of $S_2$:
   - when $S_2$ is a principal, $K' = S_2$ and $\mathcal{P}' = \mathcal{P} \cup \{K A \rightarrow S_1\}$
   - when $S_2 = K_1 A_1 \cdots A_\ell$ where $\ell \geq 1$, we set $K'$ to be a principal not appearing in $\mathcal{P}$ and $\mathcal{P}' = add(\mathcal{P}, S_2, K') \cup \{K A \rightarrow S_1\}$.

See Appendix A for the proof. The equivalence of 2 and 3 in Theorem 3 says that to determine whether $Th[\mathcal{P}] \models \forall z(contains[S_1][z] \iff contains[S_2][z])$, one can do the following: create a new principal $K'$, find a local name $K A$ not defined in $\mathcal{P}$, construct $\mathcal{P}'$ by adding to $\mathcal{P}$ the 4-tuple $K A \rightarrow S_1$ and additional 4-tuples to make sure that $S_2$ rewrites into $K'$, then check whether $m(K, A, K')$ follows from $LP[\mathcal{P}]$.

The proof of the equivalence of 2 and 3 in Appendix A uses their relationships with the rewriting semantics. However, this equivalence also follows from general results in proof theory. Horn clauses and queries of the form $\forall z(contains[S_1][z] \iff contains[S_2][z])$ fall into a subclass of Harrop Hereditary
formulas. Nadathur [23] showed that, for this subclass of formulas, classical provability, intuitionistic provability, and uniform provability are equivalent. Uniform proofs [22] are a restricted form of intuitionistic proofs that embody a special form of goal-directedness. Using uniform proofs, in order to determine whether
\[
\text{Th}[\mathcal{P}] \models \forall z \forall y_1 \forall y_2 (m(K,A,z) \iff m(K_1,A_1,y_1) \land m(y_1,A_2,y_2) \land m(y_2,A_3,z)),
\]
one creates new constants \( K', K'_1, K'_2 \) that do not appear in \( \mathcal{P} \), add to \( \mathcal{P} \) three facts \( m(K_1,A_1,K'_1) \), \( m(K'_1,A_2,K'_2) \), and \( m(K'_2,A_3,K') \), and then try to prove \( m(K,A,K') \) from it. This is essentially what we are doing in Definition 5 and Theorem 3. The equivalence of classical provability and uniform provability for this class of queries says that this proof method is sound and complete with respect to classical first order logic. The equivalence of classical provability and intuitionistic provability for this class of queries says that one can also view the semantics as defined in intuitionistic logic rather than classical logic.

2.6 Comparing FOL Semantics with Other Semantics

Other logical semantics for SDSI exist. Both Abadi’s logic [1] and Halpern and van der Meyden’s Logic of Local Name Containment (LLNC) [6] are propositional modal logics. The core relation in their logic is the rewriting into relation. The focus of their study is to get a set of axioms characterizing their logic.

SDSI 1.1 [24] (the version of SDSI before merging with SPKI) gives pseudo-code of a non-deterministic algorithm for resolving a name string into a principal given a set of 4-tuples. Like the set-theoretic semantics, this algorithmic semantics defines a mapping from name strings to sets of principals. This semantics was used as the reference semantic in the logics by Abadi [1] and by Halpern and van der Meyden [6]. Li [14] showed that this semantics is equivalent to the string rewriting semantics when mapping name strings to principals. Li [14] gave another logic program semantics for SDSI; the logic program used there uses lists and is not Datalog; furthermore, it is not as natural as the LP semantics in Definition 3, and is not as easy to extend to FOL.

Comparing with other logic-based semantics for SDSI, such as LLNC, the FOL semantics has the following advantages:

- The FOL semantics directly captures the set-based semantic of SDSI, which is used widely as the underlying semantic intuition of SDSI, e.g., both by SDSI designers in [3] and by Halpern and van der Meyden [6]. In addition, it uses classical first-order logic, instead of more complex logics with modal operators.
- It contains more information in the sense that a larger class of meaningful queries can be formulated and defined in this semantics.
- It is easily extended to support useful extensions of SDSI. There are two important ways of extending the SDSI 4-tuples. The first one is to add an intersection operator, e.g., \( K A \mapsto K A_1 \cap K A_2 \). The meaning of such a statement is immediate given the set-based semantic intuition. Supporting intersection in the FOL semantics is straightforward. Another way to extend SDSI is to allow identifiers to be logical terms, e.g., \( K \text{student}('\text{Stanford}') \mapsto K_{\text{John Smith}} \). In FOL semantics, this can be supported either by making the predicate \( m \) take more parameters or by having a predicate for each identifier function symbol. Extending logics such as LLNC to support these extensions may be possible, but seems less obvious.
• The FOL semantics has computational significance. The equivalence of 2 and 3 in Theorem 3 gives an efficient (in time polynomial in the size of $P$) way to check whether $\mathsf{RS}[P] \triangleright S_1 \xrightarrow{*} S_2$ for any name strings $S_1$ and $S_2$. In LLNC, although a complete axiom system was given, the complexity of using the axiom system to determine whether $\mathsf{RS}[P] \triangleright S_1 \xrightarrow{*} S_2$ is not clear.

Some of these advantages also apply when comparing with the string rewriting semantics, e.g., the FOL semantics contains more information and is more extendable. Because of this, we believe that the FOL semantics should be viewed as the reference semantics. Any semantics for SDSI should be equivalent to the FOL semantics for the class of queries it handles. Some other semantics are useful for computational or understanding purposes, of course. For example, the string rewriting semantics is both helpful for understanding SDSI and for computation, because of its relationship with pushdown systems.

3 $RT^C_1$: A TM Language Extending SDSI

In Section 2, we have seen that SDSI’s semantics can be defined in several equivalent ways and can be computed efficiently. This shows that SDSI’s design is natural and elegant. SDSI was originally designed as a scheme for naming. As such, it lacks important features when used as a full-fledged trust-management language. First, SDSI lacks intersection; one cannot define a local name to contain the intersection of two or more name strings. Many natural security policy statements have the form that a principal has a certain permission (or an attribute) if it has two (or more) other attributes at the same time. Second, SDSI identifiers do not have internal structures. For example, instead of saying “$K$ student $\mapsto K_{\text{John Smith}}$”, it would be more useful to say “$K$ student(dept=’CS’, program=’PhD’, year=’2002’, name=’John Smith’, · · · ) $\mapsto K_{\text{John Smith}}$”, because one can then use, for example, “$K$ perm $\mapsto K$ student(dept=’CS’)” to grant “perm” to all principals that are CS students. Having internal structures in identifiers also enables one to represent relationships among principals. For example, one can then have local names such as “$K$ managerOf(Alice)” and “$K_1$ physicianOf(Bob)” and issue statements such as “$K$ accessRecord(?X) $\mapsto K_1$ physicianOf(?X)”, which means that the physician of any patient can access the record for that patient. (“?X” represents a logical variable.) Having internal structures in identifiers also enables one to represent access permissions that take parameters which identify resources and access modes.

SPKI/SDSI intends to be a full-fledged TM language; the approach taken in designing SPKI/SDSI can be viewed as adding SDSI features to SPKI, e.g., allowing name strings to be used in subjects of auth certs. In Section 4, we will analyze SPKI/SDSI and argue that the design of SPKI/SDSI is problematic in several ways. In particular, authorization tags, which are used to qualify permissions in SPKI/SDSI, are quirky and algorithmically problematic. To make our analysis easier to follow, in this section we first look at another way of extending SDSI. Namely, we look at $RT^C_1$ [16], which is a language in the $RT$ Role-based Trust-management framework [16, 18, 20]. $RT$ consists of a family of TM languages. As pointed out in [18, 20], the design of $RT$ is heavily influenced by SDSI and Delegation Logic [15]. $RT_0$, the most basic language in the $RT$ family, can be viewed as adding intersections to SDSI. $RT_1$ adds to $RT_0$ parameters. $RT^C_1$ further extends $RT_1$ with constraints. The semantics of the $RT$ languages are based on translating statements into logical sentences, in the same style as SDSI’s logic-based semantics.
3.1 Syntax of Policy Statements in $RT^C_1$

Before describing the syntax of policy statements in $RT^C_1$, we first explain the terminology differences between SDSI and $RT$. Identifiers in SDSI are called role terms in $RT$, and local names in SDSI are called roles. We use $R$, often with subscripts, to denote role terms, and we add a dot between a principal and a role term. For example, $K.R$ represents a role in $RT$; it corresponds to the local name $KR$ in SDSI. Policy statements in $RT_0$, $RT_1$, and $RT^C_1$ have the same structures. They only differ in how role terms are formed. In the following statements, the directions of the arrows are the opposite of that used in SDSI rewriting rules. We use this direction because it is the same as the direction of logical implication in the logic based semantics.

- **Type-1**: $K.R \leftarrow K_1$
- **Type-2**: $K.R \leftarrow K_1.R_1$
- **Type-3**: $K.R \leftarrow K.R_1.R_2$
- **Type-4**: $K.R \leftarrow K_1.R_1 \cap K_2.R_2 \cap \cdots \cap K_\ell.R_\ell$

A type-3 statement requires the same $K$ to be used in both the role being defined, $K.R$, and the definition $K.R_1.R_2$. This design is motivated by the need to handle deduction with policy statements that are stored in a distributed manner. Observe that 4-tuples that use long extended names can still be equivalently represented in $RT$, by introducing new role terms and statements. For more details on this, see [20].

In $RT_0$, a role term is simply a role name, which is just like an identifier in SDSI. In $RT_1$, a role term may also contain parameters. These parameters may be constants or variables, and may use constraints in some limited ways. $RT^C_1$ allows more general forms of role terms, which we now describe. In $RT^C_1$, each role name takes the form of $r(h_1, \ldots, h_n)$, in which $r$ is a role name, and for each $i$ such that $1 \leq i \leq n$, $h_i$ takes one of the following three forms: $p = c$, $p \in S$, and $p = \text{ref}$, in which $p$ is the name of one of $r$’s parameters that has type $\tau$, $c$ is a constant of type $\tau$, $S$ is a value set of type $\tau$, and $\text{ref}$ is a reference to another parameter in the same statement, also of type $\tau$. Intuitively, a value set is a constraint-based representation of a set of values, e.g., $[10..800]$ may be a value set of an integer type. Parameters in role terms are specified by name, rather than by position; and they are strongly typed. For example, the following $RT^C_1$ statement “$K_{SA}.socketPerm(host \in \text{descendants(‘stanford.edu’)}), port \in [8000..8443]) \leftarrow K_{Alice}$” means that $K_{SA}$ grants to $K_{Alice}$ the permission to connect to any host in the domain ‘stanford.edu’ at any port between 8000 and 8443.

3.2 Semantics of $RT^C_1$

Without constraints, policy statements in $RT^C_1$ are translated into Datalog clauses (which can be viewed as first-order sentences), in ways very similar to SDSI. $RT^C_1$ statements with constraints are translated into clauses in multi-sorted Constraint Datalog; these clauses can also be viewed as first-order sentences. In the following, we give an brief overview of Constraint Datalog. See [16] for more details of Constraint Datalog and for the translation from $RT^C_1$ statements to clauses in Constraint Datalog.

Constraint Datalog (or DATALOGC) is a restricted form of Constraint Logic Programming (CLP), and is also a class of query languages for Constraint Databases (CDB) [11, 12].
notion of constraint databases, which was introduced by Kanellakis, Kuper, and Revesz [11], grew out of the research on Datalog and CLP and generalizes the relational model of data by allowing infinite relations that are finitely representable using constraints. DATALOGC allows first-order formulas in one or more constraint domains, which may describe file hierarchies, time intervals, and so on, to be used in the body of a rule. Intuitively, a constraint domain is a domain of objects, such as numbers, points in a plane, or files in a file hierarchy, together with a language for speaking about these objects. A constraint domain has a first-order language defined by a set of constants, function symbols, and relation symbols, and a class of quantifier-free formulas in the language, called primitive constraints. The following are some example constraint domains that are used when translating $RT_1^C$ into DATALOGC.

**Tree domains** Each constant of a tree domain takes the form $\langle a_1, \ldots, a_k \rangle$. Imagine a tree in which every node is labelled with a string value. The constant $\langle a_1, \ldots, a_k \rangle$ represents the node for which $a_1, \ldots, a_k$ are the strings on the path from root to this node. A primitive constraint is of the form $x = y$ or $x \theta \langle a_1, \ldots, a_k \rangle$, in which $\theta \in \{=, <, \leq, \prec, \preceq\}$. The constraint $x < \langle a_1, \ldots, a_k \rangle$ means that $x$ is a child of the node $\langle a_1, \ldots, a_k \rangle$, and $x \prec \langle a_1, \ldots, a_k \rangle$ means that $x$ is a descendant of $\langle a_1, \ldots, a_k \rangle$.

**Range domains** The set of all constants in a range domain is linearly ordered. A primitive constraint has the form $x = y$, $x = c$ or $x \in (c_1, c_2)$, in which $c$ is a constant, each of $c_1$ and $c_2$ is either a constant or a special symbol "*", meaning unbounded. And when $c_1$ is not *, "(" can also be "["; similarly, ")" can be "]" when $c_2$ is not *.

**Discrete domains with sets** A basic constraint has the form $x \in \{c_1, \ldots, c_\ell\}$, in which $c_1, \ldots, c_\ell$ are constants.

We say that these constraint domains are unary, because each primitive constraint either has the form $x = y$, where $x$ and $y$ are variables, or contains only one variable. We call a primitive constraint that contains just one variable a basic constraint. The three classes of unary constraint domains described above support the following three operations.

**Conjunction** Given two basic constraints $\phi_1(x)$ and $\phi_2(x)$, determine whether $\phi_1(x) \land \phi_2(x)$ is satisfiable, and if it is, compute $\psi_1(x) \lor \psi_2(x) \lor \cdots \lor \psi_k(x)$ such that $k \geq 1$, each $\psi_i(x)$ is a basic constraint, for $1 \leq i \leq k$, and the disjunction is logically equivalent to $\phi_1(x) \land \phi_2(x)$.

**Constraint Projection** Given any basic constraint, determine whether it is satisfiable.

**Constraint Subsumption** Given the disjunction of a set of basic constraints, determine whether another basic constraint is implied by the disjunction, e.g., determine whether $x \in [2, 8]$ is implied by the disjunction of $x \in [1, 5]$ and $x \in [3, 10]$.

These operations make it possible to evaluate DATALOGC programs using constraints in these domains. Li and Mitchell [16] have shown that tree domains, range domains, and discrete domains with sets have additional properties that make such evaluation tractable.
3.3 Types

Role terms in $RT_1^C$ are strongly typed. One declares what parameters a role takes and the names and data types of these parameters. One also declares new data types using the mechanisms provided by $RT$. Role parameters and data types are declared in application domain specification documents (ADSDs). Each ADSD is globally uniquely identified. One way to uniquely identify an ADSD is to use a collision-free hash of the document as the identifier; one may also include in the identifier an URI pointing to the document so that it can be easily retrieved. An ADSD declares a suite of related data types and role names, called a vocabulary. Policy statements, when using a role name, refer to the ADSD in which the role name is declared. This enables $RT$ to have strongly typed policy statements. ADSDs also provide solutions to the following vocabulary agreement problem. For a statement $K.\text{student} \leftarrow K.\text{university}$.student to make sense; every principal $K'$ that is a member of the role $K.\text{university}$ must agree with $K$ on what student means, e.g., whether it is a student registered in any class, or a student enrolled in a degree program. The use of ADSDs ensure that everyone is talking about the student role declared in one specific ADSD. The use of ADSDs and strongly typed statements helps reduce the possibility of errors in writing policy statements and unintended interaction among policy statements.

The notion of vocabularies is complementary to the notion of localized name spaces for roles. Each addresses a distinct name space issue. For example, an accrediting board might issue an ADSD that declares the role name “student”. This defines the names and data types of the role’s parameters. Such parameters may include university name, student name, program enrolled in, and so on. The ADSD may also contain description of the conditions under which a principal should be made a member of the student role, e.g., it may require a principal be registered in a degree program. Then a university StateU can use this ADSD to issue credentials defining $K_{\text{StateU}.\text{student}}$. Although using a vocabulary created by another principal, StateU is still the authority over who is a member of the role $K_{\text{StateU}.\text{student}}$.

$RT_1^C$ has several categories of types: integer types, float types, enumeration types, string types, tree types. Some system-defined data types of these categories exist, and one can declare new data types of these categories. Each type category has a syntax for defining value sets. Unordered enumeration types and string types correspond to discrete domains with sets; a value set takes the form of a set of constants. Integer types, float types, and ordered enumeration types correspond to range domains; a value set takes the form of a range, e.g., [10, 200], or (10, ∗). Tree types correspond to tree domains; and are used to represent hierarchical structured resources such as file hierarchies and DNS names; a value set takes the form of a predefined function symbol applied to a node, e.g., descendants((edu, stanford)) and currentAndChildren((edu, stanford, cs)).

4 Analyzing SPKI/SDSI using First-Order-Logic (FOL)

In this section, we extend the FOL semantics for SDSI to SPKI/SDSI, which has authorization certificates (or auth certs) and ACL entries, in addition to name certs. Both auth certs and ACL entries are 5-tuples. An auth cert is a signed five tuple $(K, H, D, T, V)$, where

- $K \in \mathcal{K}$ is the issuer principal, which signs the cert. The issuer grants a specific authorization through this 5-tuple.
- $H \in \mathcal{H}$ is called the subject, where $\mathcal{H}$ is defined to be the least set satisfying the following two
conditions: (1) \( S \subseteq H \), where \( S \) is the set of all name strings, and (2) \( \theta_k(H_1, H_2, \cdots, H_n) \in H \), where \( H_1, H_2, \cdots, H_n \in H \). The subject specifies principals that receive authorization from this 5-tuple.

- \( D \in \{0, 1\} \) is called the delegation bit. When \( D = 1 \), the subject may further delegate the authorization it receives from this 5-tuple.
- \( T \) is the authorization tag, each tag represents a (potentially infinite set of) byte strings. \( T \) specifies the authorization that is granted by this 5-tuple.
- \( V \) is the validity specification, which is the same as in the case of a name cert.

An ACL entry is a locally stored 5-tuple (\( \text{Self}, H, D, T, V \)). It is very similar to an auth cert, except that the issuer is a special symbol \( \text{Self} \) instead of a key and that the ACL entry is not signed. We will treat \( \text{Self} \) as a special principal in \( K \).

The 5-tuple reduction rule in Section 6.3 of RFC 2693 is as follows: the two 5-tuples \((K_1, S_1, D_1, T_1, V_1)\) and \((K_2, S_2, D_2, T_2, V_2)\) yield \((K_1, S_2, D_2, \text{AIntersect}(T_1, T_2), \text{VIntersect}(V_1, V_2))\), provided that \( S_1 = K_2 \), \( D_1 = 1 \), and \( \text{AIntersect}(T_1, T_2) \) succeeds.

Observe that the 5-tuple reduction rule only applies when the subject is a principal. When the subject is a name string, the way to use the 5-tuple reduction rule is to first replace a 5-tuple that has a name string \( S \) as the subject with a set of 5-tuples, each of which has one principal in the valuation of \( S \) as the subject. The procedure for handling threshold is described in details in Clarke et al. [3].

4.1 Handling Basic Delegation Relationships in 5-tuples

To understand 5-tuples, we first make some simplifying assumptions. These assumptions will be removed later one by one.

1. We assume that \( T \) does not have any internal structure, and can be viewed as an identifier. As a result, \( \text{AIntersect}(T_1, T_2) \) either fails (when \( T_1 \neq T_2 \)) or equals \( T_1 = T_2 \).
2. We ignore the validity specification \( V \), and write \((K, S, T, D, \cdot)\) for a 5-tuple. We assume that only valid certificates are considered.
3. We assume that the subject does not contain thresholds; in other words, the subject is a name string \( S \).

When \( D = 0 \), a 5-tuple \((K, S, 0, T, \cdot)\) simply means that \( K \) grants the authorization \( T \) to any principal who is in the valuation of \( S \). If we allow identifiers that correspond to authorization specifications to be used in 4-tuples, we can use \( K T \) to denote the set of all principals that \( K \) grants the authorization \( T \) to, and view this 5-tuple as a 4-tuple \( K T \mapsto S \).

When \( D = 1 \), a 5-tuple \((K, S, 1, T, \cdot)\) means that \( K \) grants the authorization to any principal that is in the valuation of \( S \) and to any principal such a principal grants \( T \) to. In other words, \((K, S, 1, T, \cdot)\) can be represented using two 4-tuples \( K T \mapsto S \) and \( K T \mapsto S T \). For example, a 5-tuple \((K, K_1 A_1 A_2, 1, T, \cdot)\) is represented using \( K T \mapsto K_1 A_1 A_2 \) and \( K T \mapsto K_1 A_1 A_2 T \).

Using 5-tuple reduction, one can use \((K_1, K_2, 1, T, \cdot)\) and \((K_2, K_3, 1, T, \cdot)\) to derive \((K_1, K_3, 1, T, \cdot)\). This can be viewed as deriving new rewriting rules from existing ones.
• rewrite $K_1 T \rightarrow K_2 T$ using $K_2 T \rightarrow K_3$ and derive $K_1 T \rightarrow K_3$

• rewrite $K_1 T \rightarrow K_2 T$ using $K_2 T \rightarrow K_3 T$ and derive $K_1 T \rightarrow K_3 T$

These observations show that, under these simplifying assumptions, 5-tuples can be viewed as 4-tuples, and we can use the FOL semantics for SDSI to provide a semantics for the simplified version of SPKI.

In the version of SPKI that existed before merging with SDSI, the subject of a 5-tuple cannot contain names. In this case a 5-tuple $(K, K_1, 1, T, V)$ can be represented using two 4-tuples $K T \rightarrow K_1$ and $K T \rightarrow K_1 T$. This represents very limited delegation relationships, one cannot express $K T_1 \rightarrow K T_2$ or the tuple $K T \rightarrow K T_1 T$, which represents a delegation about $T$ from $K$ to members of $K T_1$.

4.2 Using Constraints to Handle Authorization Tags and Validity Specifications

Having understood the delegation semantics of SPKI, we remove the first simplifying assumption and deal with authorization tags. Howell provided a detailed analysis of authorization tags in his PhD thesis (Chapter 5 of [8]), in which an authorization tag $T$ is viewed as representing an (often infinite) set of strings. Here, we take the same view and use constraints to specify the authorization represented as a tag. We use a unary constraint domain in which a basic constraint has the form $t \in T$, where $t$ is a logical variable and $T$ is an authorization tag. The mapping from tags to sets of strings then determines the truthfulness of formulas in this constraint domain. We will look at details of tags and this constraint domain in Section 4.3.

Definition 6 Given a set $\mathcal{P}$ of 4-tuples and 5-tuples, we define $\text{Th}[\mathcal{P}]$ to be the following Constraint Datalog program (which is also a first-order theory). In addition to the predicate $m$ used for 4-tuples, we also use another ternary predicate $g$: intuitively, $g(K, T, K')$ says that $K$ grants the authorization $T$ to $K'$.

$$
\text{Th}[\mathcal{P}] = \begin{cases}
\forall z (\text{contains}[K A][z] \iff \text{contains}[S][z]) & \text{if } (K, A, S, V) \in \mathcal{P} \\
\forall z \forall t (g(K, t, z) \iff \text{contains}[S][z] \land t \in T) & \text{if } (K, S, 0, T, V) \in \mathcal{P} \\
\forall z \forall t (g(K, t, z) \iff \text{contains}[S][z] \land t \in T) & \text{if } (K, S, 1, T, V) \in \mathcal{P} \\
\forall z \forall y \forall t (g(K, t, z) \iff \text{contains}[S][y] \land g(y, t, z) \land t \in T) & \text{if } (K, S, 1, T, V) \in \mathcal{P}
\end{cases}
$$

RFC 2693 does not explicitly specify the class of queries in SPKI. However, because the 5-tuple reduction rule deduces new 5-tuples from existing ones, SPKI determines whether a 5-tuple follows from a set of statements. This is a reasonable “semantics” because authorization requests are also represented as 5-tuples. Because 5-tuples are represented using first-order formulas in our semantics, first-order logic naturally defines a semantic relation. Given a set $\mathcal{P}$ of 5-tuples and 4-tuples, if one wants to know whether a 5-tuple $(K, S, 0, T, V)$ follows from $\mathcal{P}$, one can ask whether $\text{Th}[\mathcal{P}] \models \forall z \forall t (g(K, t, z) \iff \text{contains}[S][z] \land t \in T)$. If one wants to know whether a 5-tuple $(K, S, 1, T, V)$ follows from $\mathcal{P}$, one should also check whether $\text{Th}[\mathcal{P}] \models \forall z \forall y \forall t (g(K, t, z) \iff \text{contains}[S][y] \land g(y, t, z) \land t \in T)$.

When $\text{AIntersect}(T_1, T_2)$ is sound, i.e., $t \in \text{AIntersect}(T_1, T_2) \Rightarrow t \in T_1 \land t \in T_2$ is a tautology for any pair of tags $T_1$ and $T_2$, the 5-tuple reduction rule is sound with respect to the logical semantic relation. For example, the 5-tuple reduction that uses $(K_1, K_2, 1, T_1, \cdot)$ and $(K_2, K_3, 0, T_2, \cdot)$ to derive $(K_1, K_3, 0, \text{AIntersect}(T_1, T_2), \cdot)$ is essentially the following sound logical deduction: from
\(\forall z \forall t (g(K_1, t, z) \Leftarrow g(K_2, t, z) \land t \in T_1) \) and \(\forall t (g(K_2, t, K_3) \Leftarrow t \in T_2)\), deduce \(\forall t (g(K_1, t, K_3) \Leftarrow t \in A\text{Intersect}(T_1, T_2))\).

Validity specifications can be handled by extending the two ternary predicates \(m\) and \(g\) to take an additional parameter \(v\), which denotes the time during which this tuple is valid and viewing validity specifications as constraints on \(v\). For example, validity specifications that are validity periods are straightforwardly represented using range constraints.

### 4.3 Structure of Tags

In this section, we look at the internal structure of tags. The main point of this section is that SPKI's 5-tuple reduction is an incomplete proof procedure with respect to the FOL semantics of SPKI. In addition, as pointed out before by Howell [8], authorization tags are not well behaved. Specifically, the constraint domain for tags does not support operations that we need to compute the meaning of \(\text{Th}[P]\).

The following descriptions of authorization tags follow RFC 2693. An authorization is a list of byte-strings or sub-lists. Two authorizations intersect by matching, element for element. If one list is longer than the other but matches at all elements where both lists have elements, then the longer list is the result of the intersection. This means that additional elements of a list must restrict the permission granted. For example, \((\text{ftp (host ftp.clark.net)})\) may represent the permission of ftp access to every on the host \(\text{ftp.clarke.net}\). This is more general than \((\text{ftp (host ftp.clark.net) (dir /pub/cme)})\), and the intersection of the two tags results in the latter. SPKI also has a small number of special expressions.

\((\ast)\) stands for the set of all tags and byte-strings. In other words, it will match anything. When intersected with another tag, the result is that other tag.

\(((\ast \text{ set } \text{tag-expr}))\) stands for the set of elements listed in the \(\ast\)-form.

\(((\ast \text{ prefix } \text{byte-string}))\) stands for the set of all byte strings that start with the one given in the \(\ast\)-form.

\(((\ast \text{ range } \text{ordering} \text{? lower-limit}? \text{ upper-limit}?)\) stands for the set of all byte strings lexically (or numerically) between the two limits. The ordering parameter (alpha, numeric, time, binary, date) specifies ordering.

We now show that 5-tuple reduction is incomplete with respect to the logical semantics, when interpreting tags as representing a set of strings. For example, given two 5-tuples \((K, K_1, 0, (\ast \text{ set read write}), V)\) and \((K, K_1, 0, (\ast \text{ set delete}), V)\), then the query \((K, K_1, 0, (\ast \text{ set read delete}), V)\) should be true; but it cannot be inferred from the two 5-tuples by reduction. Intuitively, one should be able to combine authorizations received from multiple 5-tuples. However, the reduction rule uses only tag intersection and does not consider tag union. From this observation and the discussion in Section 4.2, we have the following theorem.

**Theorem 4** The 5-tuple reduction rule is a sound but incomplete procedure with respect to the first-order logic semantics, when tags are viewed as representation of sets of strings and \(A\text{Intersect}(T_1, T_2)\) is sound.
A natural question to ask is whether there exists a sound and complete proof procedure for determining whether one 5-tuple follows from a set of 5-tuples. To do this, we have to look at the internal structures of tags and the operations on them. The short answer is that the constraint domain for representing tags is ill-behaved, and normal Constraint Datalog evaluation procedure does not apply. In [8], Howell pointed out that intersections between some tags result in sets that may not be finitely represented using tags. This happens when intersecting a (* prefix) expression and a (* range) expression, or intersecting two two (* range) expressions that use different ordering. The example Howell gave is intersecting (tag (* range numeric ge 0.5 le 0.5)) with (tag (* prefix 000)). Howell suggested artificially defining intersections in these potentially problematic cases to be empty, for lack of better solutions. This suggests that the constraint domain for authorization tags does not support the conjunction operation. It is also unclear how it can support the subjunction operation.

4.4 Authorization Tags in SPKI vs. Named and Typed Parameters in $RT^C_1$

We have seen that, by viewing tags as representing sets of strings and viewing 4-tuples and 5-tuples as logical sentences, 5-tuple reduction is incomplete, and it seems unlikely that a complete proof procedure exists. This is clearly unsatisfactory. We now look for ways to remedy this problem.

SPKI/SDSI defines tags and their intersection in a syntactical way. Howell [8] provided a semantics for tags by viewing them as representations of sets of strings and found that “special tags cause havoc” (6.5.3 of [8]). The string semantics for tags is still syntactical in that it does not consider what the strings intend to represent. The (* prefix) tag, for example, is used for representing tree-like file hierarchies. If we consider the motivation and application for this construct, it seems more natural to represent statements about file hierarchies using multi-sorted first-order logic with a special sort devoted to files and directories. This approach leads to a constraint domain for tree-like hierarchies, namely, a tree domain, which we discussed in Section 3.2.

The move to constraints over tree domains not only gives us a natural semantics, but leads to more expressive policy options. For example, (* prefix) is not helpful for referring to the DNS hierarchy in an access policy; we need a (* postfix) tag instead. Nor can one use (* prefix) to represent the set of all files and directories that are direct children of a directory, whereas tree domains support these constraints. Similarly, (* range) tags are more naturally viewed as a form of constraints over range domains than sets of strings.

We believe that instead of viewing tags as defining sets of strings, and allowing combinations of string operations that do not meaningfully refer to any controlled resources, it is more informative and semantically appealing to use one special purpose constraint domain for each concept. The language $RT^C_1$ does exactly this, allowing many choices of constraint domains. For example, in $RT^C_1$, one can declare one tree domain for file hierarchies, another tree domain for DNS names, one range domain for time of the day, and another range domain for port numbers, and so on. Each constraint domain has its standard semantic meaning, given by a first-order structure, and is easy to understand and handle algorithmically. In $RT^C_1$, a role term may have multiple named and typed parameters, each parameter from one constraint domain. This syntactic requirement allows different constraint domains to be combined in a single policy, without introducing meaningless statements that, for example, apply range predicates to tree expressions. A statement in $RT^C_1$ can be translated into a Constraint Datalog clause with constraints from tree domains, range domains, or discrete domains with sets. In [16], we have proved that these domains are tractable and that
evaluating the minimal models of Datalog programs with any multi-sorted combination of tractable domains remains tractable.

While we believe the multi-sorted language with multiple constraint domains is more appealing, there are combinations of SPKI/SDSI tags that cannot be translated into $RT_1^C$ in any straightforward way. While many of these combinations do not seem meaningful or desirable, there are some aspects of SPKI/SDSI tag intersection that do seem useful and are not easily expressed in $RT_1^C$ without special provision. The SPKI/SDSI $\text{AIntersect}$ operator treats a longer tag as a more specialized tag. The design rationale for this to enable one to further specialize an authorization by appending new fields at the end. This flexibility seems to be missing in the strongly typed approach. The designers of the initial tags may not be able to foresee all necessary parameters; therefore, the initial type specification may not have all the parameters one wants. In $RT_1^C$, this flexibility is brought back by a feature that we call restrictive inheritance. A role name can be declared to restrictively inherit another role name and adds more parameters. Statements about the original role also implies statements about the extended role. For more details on this, see [17].

4.5 Threshold Subject in SPKI vs. the Intersection Operator in $RT_1^C$

SPKI/SDSI lacks support for conjunction. As a result, it is not possible to directly grant a permission to any principal who has two or more attributes at the same time. Another use of conjunction is to allow a principal to manage a permission (by further delegating to other principals) without being able to use the permission. In $RT_1^C$, which has an intersection operator providing logical conjunction, $K$.perm $\rightarrow$ $K_1$.perm $\cap$ K.student can be read as saying that $K$ allows $K_1$ to delegate the authorization perm, but only to members of $K$.student. If $K_1$ is not a member of $K$.student, then $K_1$ cannot make itself a member of $K$.perm, even if it issues $K_1$.perm $\rightarrow$ $K_1$.

Threshold subjects are part of SPKI/SDSI, but have proven difficult to characterize in other approaches. If threshold subjects, which SPKI/SDSI only allows in 5-tuples, are also allowed in 4-tuples, they can be used to implement conjunction. For example, $\theta_2(K_1 A_1, K_2 A_2)$ can represent $K_1 A_1 \cap K_2 A_2$. Threshold subjects in 4-tuples are allowed in [4] and earlier versions of [5], but are not allowed in [3, 5], because they are viewed as “too convoluted to be useful in practice” [3]. As observed by Li [14], the meaning of threshold subjects in 5-tuples is different from that in 4-tuples. In a 4-tuple, $k$ of the $n$ subjects in a threshold subject must be resolved to a single principal. In a 5-tuple, $k$ subjects can be resolved to different principals, which can further delegate to a single principal.

Coming up with a declarative semantics for threshold subjects in 5-tuples is not an easy task. Clarke et al. ([3], Section 10) uses a highly operational approach to handle thresholds. Halpern and van der Meyden [7] avoids threshold subjects in their logical reconstruction of SPKI. Intuitively, with threshold subjects, it is not a single principal that has some attributes (e.g., being granted an authorization); instead, a set of principals together have some attributes. This set of principals can all delegate to one principal; they can also submit a joint access request.

In order to provide a logical approach to threshold subjects, the $RT$ framework has the $RT_T$ component, which handles threshold using manifold roles. Manifold roles generalize normal roles to allow each member to be a set of principals rather than a single principal. The $RT_T$ component in the $RT$ framework supports more expressive threshold structures and separation-of-duties policies, and has a logical semantics. See [18] for details on $RT_T$. An example policy that one can easily express in $RT_T$ is requiring two different cashiers together to complete a transaction. To express
this in SPKI, one has to explicitly list all the cashier principals in the threshold subject, and so the policy statement needs to change each time a new cashier is added or removed. Note that delegating to $\theta_2(K \text{ cashier}, K \text{ cashier})$ is incorrect, since one single cashier principal satisfies the threshold in SPKI’s semantics.

5 Conclusion

SPKI/SDSI is a language for expressing distributed access control policy, derived from SPKI and SDSI. Significant effort has gone into finding a logic-based semantics for both SDSI naming alone and SPKI/SDSI. We presented a FOL semantics for SDSI based on translating each name cert into a Datalog clause, viewed as a first-order logic sentence. We also proved that the FOL semantics is equivalent to the string rewriting semantics used by SDSI designers, for all queries associated with the rewriting semantics. The advantages of our approach over previous logics are the following. First-order logic captures the set-based semantic intuition of SDSI, and requires only classical first-order logic rather than more complex modal logics. The FOL semantics contains more information than rewriting semantics in the sense that a larger class of queries can be formulated and understood. This opens the door to more possibilities of safety and availability analysis, along the lines developed in [19]. The FOL semantics is easily extended to support useful extensions to SDSI and, finally, the relationship between the FOL semantics and logic programming provides an efficient method to answer a large class of queries.

In our study of full SPKI/SDSI, we used a trust management language $RT^C_1$ [16], which has Constraint Datalog as its semantic foundation. $RT^C_1$ is a language in the RT family of Role-based Trust-management languages [18, 20], and can be viewed as extending SDSI by adding several features. By adopting specific constraint domains tailored to SPKI/SDSI, we provided a FOL semantics for SPKI/SDSI in which authorization tags and validity specification are interpreted as logical constraints. This interpretation of SPKI/SDSI helped us examine several design issues, using $RT^C_1$ for comparison. Our analysis shows that SPKI’s 5-tuple reduction procedure is semantically incomplete. One reason is that reduction does not handle union of tags. In addition, authorization tags in SPKI/SDSI are algorithmically problematic, making a complete proof procedure unlikely. The constraint feature of $RT^C_1$ provides an alternative mechanism that is often more expressive than SPKI/SDSI tags, semantically natural, and algorithmically tractable. The translation of SPKI/SDSI into logic with constraints, and experience with $RT^C_1$, suggests that Constraint Datalog is an appropriate foundation for trust management languages, subsuming SPKI/SDSI and possessing several algorithmic and expressiveness advantages.
A Proofs

Proposition 2 Given a set $\mathcal{P}$ of policy statements, if $\text{Th}[\mathcal{P}] \models m(K, A, K_1)$, then $\text{RS}(\mathcal{P}) \triangleright K \ A \leftarrow K_1$.

Proof. From standard result in logic programming, $\text{Th}[\mathcal{P}] \models m(K, A, K_1)$ if and only if $m(K, A, K_1)$ is in the minimal Herbrand model of $\text{Th}[\mathcal{P}]$. We now summarize a standard fixpoint characterization of the minimal Herbrand model, which we will use in this proof. For a Datalog program $\mathcal{DP}$, let $\mathcal{DP}^{\text{inst}}$ be the ground instantiation of $\mathcal{DP}$ using constants in $\mathcal{DP}$, the immediate consequence operator, $\mathcal{T}$, is defined as follows. Given a set of ground logical atoms $K$, $\mathcal{T}(K)$ consists of all logical atoms, $a$, such that $a \in \mathcal{DP}^{\text{inst}}$ and $b_j \in K$ for $1 \leq j \leq n$. The least fixpoint of $\mathcal{T}$ can be constructed as follows. Define $\mathcal{T}^0 = \emptyset$ and $\mathcal{T}^{i+1} = \mathcal{T}(\mathcal{T}^i)$ for $i \geq 0$. This defines an increasing sequence of subsets of a finite set. Thus there exists an $N$ such that $\mathcal{T}^N = \mathcal{T}^{N+1}$. $\mathcal{T}^N$ is easily shown to be the least fixpoint of $\mathcal{T}$, which we denote by $\mathcal{T}^\omega$. $\mathcal{T}^\omega$ is identical to the minimal Herbrand model of $\mathcal{DP}$ [21]; therefore, $\text{LP}[\mathcal{P}] \models m(X, u, Z)$ if and only if $m(X, u, Z) \in \mathcal{T}^{\omega}$.

We prove this proposition by using induction on $i$ to show that if $m(K, A, K') \in \mathcal{T}^i$, then $\text{RS}(\mathcal{P}) \triangleright K \ A \leftarrow K'$. The basis is trivially satisfied because $\mathcal{T}^0 = \emptyset$. In the step, $m(K, A, K') \in \mathcal{T}^{i+1}$, one of the following three cases apply.

Case one: $m(K, A, K') \in \mathcal{L}[\mathcal{P}]$, this means that $K \ A \leftarrow K'$ is in $\mathcal{P}$. Clearly, $\text{RS}[\mathcal{P}] \triangleright K \ A \leftarrow K'$.  

Case two: $m(K, A, z)$, $m(K, A, z) \in \mathcal{L}[\mathcal{P}]$, and $m(K, A, K') \in \mathcal{T}^{i+1}$. In this case, $K \ A \leftarrow K_1 A_1$ is in $\mathcal{P}$, and by induction hypothesis, $\text{RS}[\mathcal{P}] \triangleright K_1 A_1 \leftarrow K'$. Using rewriting rules in $\text{RS}[\mathcal{P}]$, one can rewrite $K A$ first to $K_1 A_1$, and then to $K'$; so $\text{RS}[\mathcal{P}] \triangleright K \ A \leftarrow K'$.  

Case three: $m(K, A, z) \in \mathcal{L}[\mathcal{P}]$, $m(K, A, z) \in \mathcal{L}[\mathcal{P}]$, and $m(K, A, K') \in \mathcal{T}^{i+1}$. In this case, $K \ A \leftarrow K_1 A_1 \cdots A_{\ell} \in \mathcal{P}$ and by induction hypothesis, $\text{RS}[\mathcal{P}] \triangleright K_1 A_1 \leftarrow K'_1, \ K'_2 \ A_2 \leftarrow K'_2, \cdots , K'_{\ell-1} A_{\ell-1} \leftarrow K'_\ell$. Using rewriting rules in $\text{RS}[\mathcal{P}]$, one can rewrite $K A$ first to $K_1 A_1 \cdots A_{\ell}$, then into $K'_1 A_2 \cdots A_{\ell}$, and so on, and finally into $K'$.

Theorem 3 Given a set $\mathcal{P}$ of 4-tuples, and two name strings $S_1$ and $S_2$, the following three statements are equivalent.

1. $\text{RS}[\mathcal{P}] \triangleright S_1 \leftarrow S_2$.
2. $\text{Th}[\mathcal{P}] \models \forall z(\text{contains}(S_1[z]) \iff \text{contains}(S_2[z]))$.
3. $\text{Th}[\mathcal{P'}] \models m(K, A, K')$, where $K A$ is any local name not defined in $\mathcal{P}$, and $\mathcal{P'}$ and $K'$ depend on the form of $S_2$:
   - when $S_2$ is a principal, $K' = S_2$ and $\mathcal{P'} = \mathcal{P} \cup \{K A \leftarrow S_1\}$
   - when $S_2 = K_1 A_1 \cdots A_{\ell}$ where $\ell \geq 1$, we set $K'$ to be a principal not appearing in $\mathcal{P}$ and $\mathcal{P'} = \text{add}(\mathcal{P}, S_2, K') \cup \{K A \leftarrow S_1\}$.  

20
Proof. 1 implies 2: We use induction on the number of rewriting steps. Base case, $S_1 = S_2$, the formula $\forall z(\text{contains}[S_1][z] \iff \text{contains}[S_2][z])$ is a tautology. Consider the step, suppose that $RS[P] \triangleright S_1 \rightarrow S' \rightarrow S_2$. We will prove that $Th[P] \models \forall z(\text{contains}[S_1][z] \iff \text{contains}[S'][z])$. Induction hypothesis gives us $Th[P] \models \forall z(\text{contains}[S'][z] \iff \text{contains}[S_2][z])$. Combining them, we have $Th[P] \models \forall z(\text{contains}[S_1][z] \iff \text{contains}[S_2][z])$.

We now prove that $Th[P] \models \forall z(\text{contains}[S_1][z] \iff \text{contains}[S'][z])$ when $RS[P] \triangleright S_1 \rightarrow S'$. $S_1$ is either a principal, a local name, or an extended name. $S_1$ cannot be a principal, since then the rewriting from $S_1$ to $S'$ is impossible. If $S_1$ is a local name, then $S_1 \rightarrow S' \in P$, and so $\forall z(\text{contains}[S_1][z] \iff \text{contains}[S'][z]) \in Th[P]$. If $S_1$ is an extended name $K A_1 \cdots A_\ell$ for some $\ell \geq 2$, we assume that the 4-tuple used to rewrite $S_1$ into $S'$ is $K A_1 \rightarrow K_1 B_1 \cdots B_k$ for some $k \geq 0$, then $S' = K_1 B_1 \cdots B_k A_2 \cdots A_\ell$. Consider any model of $Th[P]$ and any principal $z$ in the model, if $\text{contains}[S'][z]$ is true in the model, then there exists principals $y_1', \cdots, y_k'$ and $y_2, \ldots, y_\ell$ such that $m(K_1, B_1, y_1'), m(y_1', B_2, y_2'), m(y_{k-1}', B_k, y_k'), m(y_k', A_2, y_2), \ldots, m(y_{\ell-1}, A_\ell, z)$ are true in the model. Since $K A_1 \rightarrow K_1 B_1 \cdots B_k$, then $Th[P] \models \forall z(\text{contains}[K A_1][z] \iff \text{contains}[K_1 B_1 \cdots B_k][z])$, therefore $m(K, A_1, y_k')$ is true in the model. Then it follows that $\text{contains}[K A_1 \cdots A_\ell][z]$ is true in the model. Therefore, $Th[P] \models \forall z(\text{contains}[S_1][z] \iff \text{contains}[S'][z])$.

2 implies 3: Assume, without loss of generality, that $S_1 = K A_1 \cdots A_\ell$ for some $\ell \geq 0$ and $S_2$ is $K_1 B_1 \cdots B_k$ from some $k \geq 0$. Observe that by definition of $P'$, $Th[P'] \models \text{contains}[S_2][K']$. When $S_2$ is a principal, $S_2 = K'$. When $S_2$ is a local name or an extended name, $Th[P']$ contains the following $k$ atoms $m(K_1, B_1, K_1'), m(K_1', B_2, K_2'), \ldots, m(K_{k-1}', B_k, K')$. If $Th[P] \models \forall z(\text{contains}[S_1][z] \iff \text{contains}[S_2][z])$, then $Th[P'] \models \text{contains}[S_1][K']$. Also observe that $Th[P'] \models \forall z(\text{contains}[K A_1][z] \iff \text{contains}[S_2][z])$; therefore, $Th[P'] \models m(K, A, K')$.

3 implies 1 From Proposition 2, we know that if $Th[P'] \models m(K, A, K')$, then $RS[P'] \triangleright K A \rightarrow K'$. Consider the rewriting sequence, the first rule applied has to be $K A \rightarrow S_1$, since it is the only rule that can apply. (Recall that by definition $K A$ is not defined in $P$.) Consider the last step, the rule applied has to be $K_{\ell-1} A_{\ell} \rightarrow K'$, since that is the only rule having a $K'$ on its right hand side. The rule applied in the second to last step has to be $K_{\ell-2} A_{\ell-1} \rightarrow K'_{\ell-1}$, since that is the only rule having $K_{\ell-1}$ on its right-hand side, and so on. Therefore, the rewriting sequence must contains in its middle a sequence rewriting from $S_1$ to $S_2$. Further observe that the rules in $P'$ but not in $P$ cannot be applied in this middle sequence. Therefore, $RS[P] \triangleright S_1 \rightarrow S_2$. ■
References


