A Note On SPKI's Authorisation Syntax

Olav Bandmann*
Industrilogik L4i AB
Odengatan 87, SE–113 22 Stockholm, Sweden
olav@L4i.se

Mads Dam†
LECS/IMIT, Royal Institute of Technology (KTH)
KTH Electrum 229, SE–164 40 Kista, Sweden
mfd@kth.se

Abstract

Tuple reduction is the basic mechanism used in SPKI to make authorisation decisions. A basic problem with the SPKI authorisation syntax is that straightforward implementations of tuple reduction are quadratic in both time and space. In the paper we introduce a restricted version of the SPKI authorisation syntax, which appears to conform well with practice, and for which authorisation decisions can be made in nearly linear time.

1 Introduction

SPKI [3, 4] is a framework for authorisation intended particularly for networked applications. In SPKI, authority is bound to principals primarily identified by public keys. An SPKI authorisation certificate <I,S,D,A,V> specifies the following items of information:

- I: An issuer as a public key.
- S: A subject which is identified primarily through a public key.
- D: A delegation flag, indicating whether or not the authorisation at hand is delegable.
- A: A "tag", or authorisation, determining the authority assigned to the subject by the certificate.
- V: A validity field determining optional intervals and online conditions for validity.

Authorisations are given in the form of S-expressions, following on from the work of Rivest [8]. S-expressions are essentially parenthesized list expressions in the style of LISP. To give an example, the right for subjects in the group `admin`, belonging to unit `finance`, to read the `income` attribute of all objects of type `person` might be given as a nested list structure

```
X :  (obj person
     (conds (grp admin)
            (unit finance))
     (op income read))
```

Authorisations and requests are given in the same syntax. If we consider X as a request a corresponding authorisation might have the shape e.g.

```
Y :  (obj person
     (conds (grp admin))
     (op income read))
```

meaning that all members of the group `admin` are permitted read access, not only members

*Work done while at SICS, Swedish Institute of Computer Science. Project at SICS supported by a grant from Microsoft Research, Cambridge, U.K.

†Partially supported by the Swedish Agency for Innovation Systems, project "Policy-Based Network Management", and by the Swedish Research Council grant 621-2001-2637, "Semantics and Proof of Programming Languages"

of the `finance` unit. Or, as another example, the authorisation might have the shape:

```
Z :  (obj person
     (conds (grp admin)
            (unit finance))
     (op income))
```

intended to be interpreted such that now the `income` attribute can be read *and* written. In both cases X should be granted, since, in an intuitive sense which we make precise in the paper, X is "more specific than", or, "authorised by", both Y and Z. The example gives the game away: An authorisation expression becomes more specific by extending lists to the right.

In order to be able to specify more complex authorisations in a concise manner, SPKI adds a number of auxillary constructions to be interpreted, essentially, as abbreviating sets of basic S-expressions. The following extensions are considered:

- `(*)` is the wildcard.
- `(* set` $X_1 \ \cdots \ X_n$`)` is the union of the sets $X_1, \ldots, X_n$, $n \geq 1$.
- `(* range` $R \ l \ u$`)` is the set of all X in the interval determined by the ordering R, lower limit l and upper limit u.
- `(* prefix` w`)` is the set of all strings having w as prefix.

Thus, to give an example, the authorisation

```
X' :  (obj person
      (conds (grp admin)
        (* set (unit finance)
               (unit personnel)))
      (op income (* set read write)))
```

is just an abbreviation for the obvious size 4 set.

In SPKI, authorisation decisions are made through a process of "tuple reduction". Authorisations and requests are compared by computing their intersection using the operation `AIntersect`. As an example, with X and X' as defined above, `AIntersect(`X'`,`X`)` $= X$. The intersection of X' and X is the most permissive authorisation granted by both X' and X. If `AIntersect(`X'`,`X`)` $= X$ then the most permissive authorisation granted by both X' and X is X itself, or in other words, all authorisation granted by X is also granted by X', i.e. X is authorised by X'.

Computation of the `AIntersect` function is in many cases quite unproblematic, in particular when one of the arguments lack one of the special `*` forms. In general, however, `AIntersect` may cause a quadratic blowup, and this is the basic problem we address in this note.

The problem arises when comparing `* set` forms. The naive algorithm simply expands an S-expression involving `* set` forms to one without them. In many applications this procedure is in fact quite adequate. First, it will often be the case that one of the arguments to `AIntersect` is without `*` forms. Second, requests will often be small, and a quadratic blow-up will be without much consequence. The SPKI standard opens up for implementors to provide set-to-set transformations to alleviate the problems that may remain, but no concrete suggestions are given.

On the other hand one will in fact sometimes want to compute using complex authorisations. For instance, one will want to subject authorisations to simple analyses of the type:

Q: Is authorisation X stronger than authorisation Y?

where X and Y are general S-expressions. Observe that Q is just a different way of saying that `AIntersect(`X`,`Y`)` $= X$. Secondly, simply by providing the tools to describe complex authorisations, users may eventually want to use them, for instance to precompute sets of authorisations, or to use the `* set` notation as a macro facility.

This is discussed in slightly more detail in section 9. As another example we have, in the Amanda project at SICS, been exploring a general mechanism for delegation based on a modelling of delegation as the constrained issuance of new authorisations [6, 1]. The resulting S-expression can become quite complex, and furthermore the need arises, in the decision making process, to compare authorisations of a general shape.

For these reasons we have found a need to subject the SPKI authorisation syntax to a deeper analysis. In the paper we obtain the following main results:

1. A characterisation of the SPKI entailment relation in terms of a partial ordering $\leq_S$.
2. A weak version of $\preceq_S$, which is *sound*, so that $x \preceq_S y$ implies $x \leq_S y$.
3. A restricted S-expression syntax for which the weak relation $\preceq_S$ is *complete*, i.e. coincides with $\leq_S$.
4. An efficient algorithm to compute `AIntersect`, and a proof that `AIntersect` is the greatest lower bound with respect to the $\leq_S$ ordering.

The key idea for the restricted S-expression syntax is simply to require that non-atomic elements of `* set` expressions are tagged with a unique tag (or, in SPKI terminology, type). On the evidence we have so far gathered this is nothing more than a formalisation of existing SPKI practice, and all examples in the SPKI documents [3, 4, 5] stay within the restricted syntax.

The paper is organised in the following way: In the first sections we describe authorisation trees as the basic form of $*$-free S-expressions, and then the syntax and semantics of S-expressions is given as sets of authorisation trees. The syntax is given in slightly abstract terms; instead of the concrete range and prefix constructions we just assume a set of primitive set constants, as this makes the presention less cluttered. In section 5 we proceed to introduce the partial orders $\leq_S$ and $\preceq_S$, and in section 6 we relate $\leq_S$ and $\preceq_S$ by showing first soundness, and then pinpointing the condition in the definition of the weak partial order which causes completeness to fail for general expressions. Then, in section 7 we turn to `AIntersect`, to establish the results (4) above.

2 Authorisation Trees

We start by defining authorisation trees, used to give semantics to the complete SPKI authorisation element. Let A be a denumerable set of "atomic" elements ranged over by a of one or several data types such as strings or integers. The set T of *authorisation trees*, ranged over by t, is determined by the following BNF style grammar:

$$t \ ::= \ a \ \mid \ (a \ t_1 \ \cdots \ t_n)$$

where $n \geq 0$.

The intention is that authorisation trees are positional. Types, in particular the type of an atom a' appearing as a subtree t_i of the tree $(a \ t_1 \ \cdots \ t_n)$, are determined by two pieces of information:

- The position i
- The label a

Types are determined by some external means; here it suffices to assume some fixed binding of types to labelled tuple positions. We define a partial order $\leq_T$ on T inductively as follows. Let $x, y \in T$.

1. If $x \in A$ or $y \in A$ then $x \leq_T y$ if and only if $x = y$.
2. If $x = (x_1 \ \cdots \ x_m) \in T$ and $y = (y_1 \ \ldots \ y_n) \in T$, then $x \leq_T y$ if and only if $m \geq n$ and $x_i \leq_T y_i$ for $i = 1, \ldots, n$.

A simple proof by induction shows that $\leq_T$ is indeed a partial order.

Elements in T represent authorisations, and the partial order $\leq_T$ represents the "is authorised by" relation, which in SPKI normally is represented in terms of the `AIntersect` operation.

Example 1 Consider the authorisation trees X, Y, and Z of section 1. We obtain that $X \leq_T Y$ and $X \leq_T Z$, but not $Y \leq_T Z$ and neither $Z \leq_T Y$. If we let

U :
```
(obj person
(conds (grp admin))
(op income))
```

then $Y \leq_T U$ and $Z \leq_T U$.

In terms of the partial ordering $\leq_T$, the intended use of authorisation trees is as follows. Assume that a certain principal p wants to perform an action a requiring the authorisation x. Then p has the authorisation for a if (and only if) p has some authorisation y satisfying $x \leq_T y$.

A problem here is that the language is too restricted to be very useful. The solution is to use *sets* of authorisation trees instead of singletons. In the example above, p has the authorisation for a if p has some authorisation Y (a set of authorisation trees) such that there exists a $y \in Y$ satisfying $x \leq_T y$.

For this reason SPKI extends the basic S-expression syntax by notation for sets of authorisation trees.

3 Syntax of S–expressions

S–expressions represent *sets* of authorisation trees. Essentially, authorisation trees are extended with notation for set unions, in addition to primitive range and prefix constructions. To cater for these primitives we assume a denumerable set B of set constants, and a mapping $Val : B \to 2^A \setminus \{\emptyset\}$ assigning to each constant in B the nonempty set of atoms it represents.

Definition 1 (S–expressions) The set S of *S–expressions*, ranged over by X, Y, is determined as follows:

$$X \quad ::= \quad \texttt{(*)} \mid a \mid b \mid \texttt{(}a\ X_1\ \cdots\ X_n\texttt{)} \mid \texttt{(* set }X_1\ \cdots\ X_m\texttt{)}$$

where $a \in A$, $b \in B$, and $n \geq 0$, $m \geq 1$.

So, an S-expression can be either an atom (in A), a primitive set of atoms, a tuple, or a `(* set ...)` form, used to denote unions. We assume, of course, that A does not contain the special wildcard symbol `(*)`. S-expressions of the form either $a \in A$ or $b \in B$ are called *atomic*. In SPKI, two types of set constants are considered:

1. Elements representing *ranges* of elements in A. E.g. all strings in A between "bird" and "fish", alphabetically, or all integers in A greater than 5. There are many options here including type of interval and type of order. Note that, by the definition of *Val* above, we do not allow empty ranges.
2. Elements representing sets of strings in A which have a certain strings as *prefixes*. E.g. all strings in A beginning with "/pub/".

4 Semantics of S–expressions

An element X of S represents a non empty subset of T: the set of trees that are authorised by X.

Definition 2 (S–expression Semantics) We define the function $\|\cdot\| : S \to 2^T \setminus \{\emptyset\}$ as follows:

1. $\|\texttt{(*)}\| = T$
2. $\|a\| = \{a\}$ for all $a \in A$

3. $\|b\| = Val(b)$ for all $b \in B$
4. $\|(X_1 \ \cdots \ X_m)\| = \{(t_1 \ \cdots \ t_l) \mid l \geq m, \forall_{i:1\leq i\leq m} \quad t_i \in \|X_i\|\}$
5. $\|(\texttt{*}\ \texttt{set}\ X_1 \ \cdots \ X_m)\| = \|X_1\| \cup \ldots \cup \|X_m\|$

Note that, in (4), X_1 and t_1 are constrained to be atoms, by definition 1. We expect $\|X\|$ to be lower closed, so that if $t \in \|X\|$ and $t' \leq_T t$ then also $t' \in \|X\|$, or in other words, if t is authorised by X and t' is authorised by t then t' should be authorised by X as well. This property is easily verified.

Proposition 1 *For all $X \in S$, $\|X\|$ is lower closed.*

Proof A trivial induction. □

The naive way of deciding whether or not $t \in \|X\|$ is to rewrite X to a normal form where all occurrences of the `* set` construction are pushed to the outermost level, thus reducing questions of the form $t \in \|X\|$ to the case where X does not have occurrences of the `* set` construction. To make this clear, say that X_1 and X_2 are equivalent, $X_1 \cong X_2$, if $\|X_1\| = \|X_2\|$.

Proposition 2

$$(X_1 \cdots (\texttt{*}\ \texttt{set}\ X_{i,1} \ \cdots \ X_{i,n}) \cdots X_m) \cong (\texttt{*}\ \texttt{set}\ (X_1 \cdots X_{i,1} \cdots X_m)) \ \cdots \ (X_1 \cdots X_{i,n} \cdots X_m))$$

Example 2 Let

$$X = (a\ (\texttt{*}\ \texttt{set}\ b\ (c\ (\texttt{*}\ \texttt{set}\ d\ e))))$$

where all a, b, etc. are atoms in A. This represents the set of authorisation trees which are lists of length at least two beginning with a and having either b or another list t of length at least two as its second element, where t begins with c and has d or e as its second component. Using (2) along with the obvious idempotency law we obtain:

$$\begin{aligned} X \ &\cong \ (\texttt{*}\ \texttt{set}\ (a\ (\texttt{*}\ \texttt{set}\ b\ (c\ d))) \\ &\qquad (a\ (\texttt{*}\ \texttt{set}\ b\ (c\ e)))) \\ &\cong \ (\texttt{*}\ \texttt{set}\ (a\ b)\ (a\ (c\ d)) \\ &\qquad (a\ b)\ (a\ (c\ e))) \\ &\cong \ (\texttt{*}\ \texttt{set}\ (a\ b)\ (a\ (c\ d)) \\ &\qquad (a\ (c\ e))) \end{aligned}$$

Note that, according to def. 2, the set $\|X\|$ includes not only a list such as $t = (a\ (c\ e))$, but also any authorisation tree t' for which $t' \leq_T t$. As an example, t' can have the shape $(a\ (c\ e\ f)\ g\ h)$.

5 Preorder on S–expressions

Clearly, calculations like the one in example 2 are not very efficient. To circumvent this, we need to be able to decide the following problems *without* actually calculating $\|\cdot\|$:

1. Given $t \in T$ (an authorisation request) and an S-expression X (stored, perhaps, as the authorisation element of some certificate), does $t \in \|X\|$ hold?
2. Given S-expressions X and Y, is every authorisation request granted by X also granted by Y?

Observe that both questions can be put in the same form, since t is trivially represented as an S-expression denoting the lower closure of $\{t\}$. We thus define a preorder, $\leq_S$, on S-expressions to reflect the semantics of 2. above:

Definition 3 (S–expression Preorder) The preorder $\leq_S$ on S is defined by

$$X \leq_S Y \iff \|X\| \subseteq \|Y\|$$

In other words, whatever is authorised by X is also authorised by Y. The difficulty in computing $\leq_S$ is illustrated by the following example, which also shows why $\leq_S$ is not a partial order.

Example 3 Let $X =$ `(a (* set b c))` and $Y =$ `(* set (a b) (a c))`. By definition 3, $X \leq_S Y$ and $Y \leq_S X$, even though $X \neq Y$ (X and Y are syntactically different). It is easy to deduce that $Y \leq_S X$ since `(a b)` $\leq_S X$ and `(a c)` $\leq_S X$ both hold. To verify $X \leq_S Y$, on the other hand, essentially requires the computation of $\|X\|$, to realize that $\|X\|$ is the lower closure of the set containing `(a b)` and `(a c)`.

This example shows the case which is to be avoided, namely where the right hand side of the equality is a set expression with at least two elements. In order to ameliorate the worst case behaviour we propose a *weaker* preorder on S, which is reasonably efficient to compute, and which does not rely on computing $\|\cdot\|$ (but it does rely on the computation of *Val*, since this function has not been explicitly defined).

The definition of the weak preorder uses the operation *flt*, which uses the equivalences such as

$$\texttt{(* set } X_1 \texttt{ (* set } X_{2,1}\ X_{2,2}\texttt{) } X_3\texttt{)} \cong \texttt{(* set } X_1\ X_{2,1}\ X_{2,2}\ X_3\texttt{)}$$

to flatten all *immediate* nestings of the `* set` constructor.

Definition 4 (Weak Preorder) Define the preorder $\preceq_S$ on S by induction in the following way. Let $X, Y \in S$. Then $X \preceq_S Y$ if and only if one of the following cases hold:

1. $Y = (*)$
2. $X, Y \in A$ and $X = Y$
3. $X = a \in A$, $Y = b \in B$, and $a \in \mathit{Val}(b)$
4. $X = b \in B$, $Y = a \in A$, and $\mathit{Val}(b) = \{a\}$ (a rather unusual situation)
5. $X, Y \in B$ and $\mathit{Val}(X) \subseteq \mathit{Val}(Y)$
6. $X =$ `(`$X_1 \cdots X_m$`)`, $Y =$ `(`$Y_1 \cdots Y_n$`)`, $m \geq n$, and $X_i \preceq_S Y_i$ for $i = 1, \ldots, n$
7. $X =$ `(* set` $X_1 \cdots X_m$`)` and $X_i \preceq_S Y$ for $i = 1, \ldots, m$
8. $X = b \in B, \mathit{flt}(Y) =$ `(* set` $Y_1 \cdots Y_n$`)`, and $\mathit{Val}(X) \subseteq \bigcup\{\|Y_i\| \mid 1 \leq i \leq n$ and Y_i is either atomic, or $Y_i =$ `(*)`$\}$.
9. X is of the form neither b nor `* set`, $Y =$ `(* set` $Y_1 \ \ldots \ Y_n$`)`, and $\exists_i\ X \preceq_S Y_i$

Referring to example 3 note that $Y \preceq_S X$ holds, but $X \preceq_S Y$ does not. The clause 4.9 is the cause of incompleteness. The problematic case is when X is a list and Y a `* set` expression, as in example 3. Observe also that 4.8 does in fact appeal to the function $\|\cdot\|$. However this is only a convenience, and does not introduce extra computational overhead, since all Y_i in that case are either atoms or sets of atoms. The reason for using the *flt* operation is to avoid otherwise pathological cases such as $b \preceq_S$ `(* set (* set b))`.

Since this is not completely apparent we check that $\preceq_S$ indeed defines a preorder.

Theorem 1 *The relation $\preceq_S$ is a preorder.*

Proof We must prove that

1. $X \preceq_S X$ for all $X \in S$, and
2. $X \preceq_S Y$ and $Y \preceq_S Z$ implies $X \preceq_S Z$ for all $X, Y, Z \in S$.

The first part is proved by a simple induction over the definition of $\preceq_S$. We'll skip the details.

The second part is a rather tedious induction over the structure of first Y, and then X and Z, as needed. So, assume $X \preceq_S Y$ and $Y \preceq_S Z$:

$Y =$ `(*)`: Since $Y \preceq_S Z$ the only cases that can apply are $Z =$ `(*)` (which is trivial) and $Z =$ `(* set` $Z_1 \cdots Z_m$`)` such that, in the latter case, $Y \preceq_S Z_i$ for some $i : 1 \leq i \leq m$. By the induction hypothesis, $X \preceq_S Z_i$ whence $X \preceq_S Z$ as well, completing the case.

$Y =$ `(`$Y_1 \cdots Y_n$`)`: In this case Z has one of the forms $Z =$ `(*)`, $Z =$ `(`$Z_1 \cdots Z_m$`)`, or $Z =$ `(* set` $Z_1 \cdots Z_m$`)`. In each case the proof is easily completed.

Y = `(* set` $Y_1 \cdots Y_n$`)`: We may assume that *flt*$(Y) = Y$. One of the following subcases apply:

- X = `(* set` $X_1 \cdots X_l$`)` and $X_i \preceq_S Y$ for all $i : 1 \leq i \leq l$.
- $X = b$ and $Val(X) \subseteq \cup\{\|Y_i\| \mid 1 \leq i \leq n, Y_i$ atomic, or $Y_i =$ `(*)`$\}$
- $X \preceq_S Y_i$ for some $i : 1 \leq i \leq n$

The first and third subcases are immediately dismissed by the induction hypothesis. For the second subcase we know that $Y_i \preceq_S Z$ for each $i : 1 \leq i \leq n$. We proceed then by cases on Z, noting that we need only consider the case of Y_i atomic or $Y_i =$ `(*)`. Thus, *flt*(Z) has one of the forms a, b, `(*)`, or `(* set` $Z_1 \cdots Z_m$`)` such that, for each choice of i we find a j such that $Y_i \preceq_S Z_j$. The former three cases are resolved by a little calculation. For the latter we may assume that Z_j is either atomic, or $Z_j =$ `(*)`. Thus, since $\preceq_S$ is sound for atomic expressions, we know that $\|Y_i\| \subseteq \|Z_j\|$. This suffices to establish the conclusion.

The remaining cases for Y atomic are quite simple and left to the reader. □

6 Soundness and Completeness

In this section we relate the definitions of $\leq_S$ and $\preceq_S$. First we show soundness.

Theorem 2 (Soundness of $\preceq_S$) *For all* $X, Y \in S$

$$X \preceq_S Y \implies X \leq_S Y \ . \qquad (1)$$

Proof By induction over the definition of $\preceq_S$ (def. 4). We begin with the base cases 1–5. Assume that $X \preceq_S Y$ and that one of the cases 1–5 in definition 4 applies. We want to show that $\|X\| \subseteq \|Y\|$. Consider the five cases:

1. $Y =$ `(*)`:
   $\|X\| \subseteq T = \|Y\|$
2. $X = Y = a \in A$:
   $\|X\| = \|Y\|$
3. $X = a \in A$, $Y = b \in B$, and $a \in Val(b)$:
   $\|X\| = \|a\| = \{a\} \subseteq Val(b) = \|b\| = \|Y\|$
4. $X = b \in B$, $Y = a \in A$, and $Val(b) = \{a\}$:
   $\|X\| = Val(b) = \{a\} = \|a\| = \|Y\|$
5. $X = b_1 \in B$, $Y = b_2 \in B$, and $Val(b_1) \subseteq Val(b_2)$:
   $\|X\| = Val(b_1) \subseteq Val(b_2) = \|Y\|$

Hence, cases 1 to 5 are proved. We continue with the inductive step in cases 6–9:

6. $X =$ `(`$X_1 \cdots X_m$`)`, $Y =$ `(`$Y_1 \cdots Y_n$`)`, $m \geq n$, and $X_i \preceq_S Y_i$ for $i = 1, \ldots, n$:
   Let $t \in \|X\|$. Then t has the shape
   $$t = (t_1 \ \cdots \ t_l)$$
   $l \geq m$, and $t_i \in \|X_i\|$ for all $i : 1 \leq i \leq m$. By the induction hypothesis, $t_j \in \|Y_j\|$ whenever $1 \leq j \leq n$ and it follows that $t \in \|Y\|$.
7. $X =$ `(* set` $X_1 \cdots X_m$`)`, $Y \neq$ `(*)`, and $X_i \preceq_S Y$ for all $i : 1 \leq i \leq m$. By the induction hypothesis, $X_i \leq_S Y$ as well, so $X \leq_S Y$ follows.
8. $X = b \in B$, *flt*$(Y) =$ `(* set` $Y_1 \cdots Y_n$`)`, and $Val(X) \subseteq \bigcup\{\|Y_i\| \mid 1 \leq i \leq n$ and Y_i is atomic, or $Y_i =$ `(*)`$\}$. By calculation.
9. X is of the form neither b nor `* set`, $Y =$ `(* set` $Y_1 \ \ldots \ Y_n$`)`, and $\exists_i \ X \preceq_S Y_i$. By the induction hypothesis, $X \leq_S Y_i$ hence also $X \leq_S Y$. □

As we have pointed out, $\preceq_S$ is incomplete in general. To attain completeness the only change required is to make the final clause of 4 more inclusive.

Definition 5 *Define the preorder $\preceq'_S$ on S by replacing the clause 9 of def. 4 by the following condition:*

9′. *X is of the form neither b nor* `* set`, *$Y =$* `(* set` $Y_1 \ \ldots \ Y_n$`)`, *and* $\|X\| \subseteq \|Y\|$.

So, the source of incompleteness is clause 9, i.e. that there should exists a universal i such that every element in $\|X\|$ is bounded from above by some element from $\|Y_i\|$. The result is that this completely explains the difference between $\leq_S$ and $\preceq_S$.

Theorem 3 (Soundness and Completeness for $\preceq'_S$)
For all $X, Y \in S$

$$X \preceq'_S Y \iff X \leq_S Y \ . \qquad (2)$$

Proof The implication $\Longrightarrow$ is a simple extension of the soundness proof, taking the modified clause 9' into account. This is an easy exercise.

The completeness argument hinges on the following auxillary observation, namely that if $Y = (\texttt{*}\ \texttt{set}\ Y_1\ \cdots\ Y_n)$ and $(\texttt{*}) \leq_S Y$ then $(\texttt{*}) \leq_S Y_i$ for some $i : 1 \ldots n$. For a contradiction suppose that for all i, $(\texttt{*}) \leq_S Y_i$ does not hold. We may assume that Y is flattened. Each Y_i will be either atomic or have the shape $(a_i\ \ldots)$. Pick some a distinct from all the a_i. No authorisation tree of the shape $(a\ t_1\ \cdots\ t_l)$ is in $\|Y\|$, so $(\texttt{*}) \leq_S Y$ cannot hold.

We now assume $X \leq_S Y$ and proceed by induction over the structure of Y. First, however, note using clause 7 of def. 4 we may assume that X is not a set expression.

1. $Y = (\texttt{*})$: Since $\|(\texttt{*})\| = T$ the result is immediate.
2. $Y = a$. Either $X = a$ as well, or $X = b$ and $\mathit{Val}(b) = \{a\}$. In either case the proof is complete.
3. $Y = b$. Either $X = a$ and $a \in \mathit{Val}(b)$ or else $X = b'$ and $\mathit{Val}(b') \subseteq \mathit{Val}(b)$. Either cases are immediate.
4. $Y = (Y_1\ \cdots\ Y_n)$. The only possibility is $X = (X_1\ \cdots\ X_m)$, $m \geq n$, and $X_i \leq_S Y_i$ for all $i : 1 \leq i \leq n$. The result then follows directly from the induction hypothesis.
5. $Y = (\texttt{*}\ \texttt{set}\ Y_1\ \cdots\ Y_n)$. By the above observation we can assume that $X \neq (\texttt{*})$. If $X = a$ then $X \preceq_S Y_i$ for some i and we are done. If $X = b$ then clause 8 can be seen to hold. The final case, then, is for X of the shape $(X_1\ \cdots\ X_m)$, and in this case the modified clause 9' applies. The proof is thus completed. □

7 Restricted S-Expressions

We then turn to the identification of a syntax fragment for which the weak preorder, even without the modification of Theorem 3, is complete. The idea is to use tagging: Every authorisation tree appearing in a set expression must contain a leading a, making it distinct from trees appearing in other elements of that set. Formally, the restricted syntax can be defined thus:

Definition 6 (Restricted S-expressions) The set R of *restricted S-expressions*, ranged over by r, along with the set of *a-restricted S-expressions*, ranged over by r^a, $a \in A$, is defined by the following grammar:

$$\begin{array}{rcl} r & ::= & (\texttt{*}) \mid a \mid b \mid (a\ r_1\ \cdots\ r_n) \mid \\ & & (\texttt{*}\ \texttt{set}\ r^{a_1}\ \ldots\ r^{a_m}) \\ r^a & ::= & a' \mid b \mid (a\ r_1\ \cdots\ r_n) \end{array}$$

where $a, a' \in A$, $b \in B$, $n \geq 0$, $m \geq 1$, and where all a_i, $1 \leq i \leq m$ are distinct.

The purpose of the r^a form is to ensure that if r^a is actually a list then it is tagged by a. Choices of r^a as atoms or set constants can be done freely.

Example 4 The S-expression

$$r = (a_1\ (\texttt{*}\ \texttt{set}\ (a_2\ c)\ (a_2\ d)\ a_2))$$

is not restricted. The S-expression

$$s = (a_1\ (\texttt{*}\ \texttt{set}\ (a_2\ c)\ (a_3\ d)\ a_2)),$$

on the other hand, is restricted, as is the S-expression

$$r' = (a_1\ (\texttt{*}\ \texttt{set}\ (a_2\ (\texttt{*}\ \texttt{set}\ c\ d))\ a_2)).$$

Note that $r \cong r'$.

In fact, the restriction appears to merely codify existing SPKI practice. All the examples of [3, 4, 5] fit the restricted syntax, and indeed it is not hard to show that that any S-expression can be rewritten into restricted form, by flattening nested `* set`'s and pushing tags out of `* set`'s, as in example 4. Thus, whenever a "real" set union (as opposed to the disjoint union provided by the restricted syntax) is needed, it suffices to use atomic S-expressions only, which is permitted.

We obtain that the weak preorder is actually complete for the restricted fragment.

Theorem 4 (Completeness, Restricted S-expressions)
For all restricted S-expressions $r_1, r_2 \in R$,

$$r_1 \leq_S r_2 \quad \Longrightarrow \quad r_1 \preceq_S r_2$$

Proof By 3 it suffices to show $r_1 \preceq'_S r_2 \Longrightarrow r_1 \preceq_S r_2$. To establish this by induction it is sufficient to show that, for restricted expressions, condition 3.9' implies condition 4.8. We may thus assume that r_2 has the form `(* set` $r^{a_1} \cdots r^{a_m}$`)`, and for r_1 there are three cases to consider:

- $r_1 =$ `(*)`. Since r_2 is restricted the only possibility is that $r^{a_i} =$ `(*)` as well for some i.
- $r_1 = a'$. Either $r^{a_i} = a'$ for some i, or else $r^{a_i} = b$ for some i and $b \in B$ such that $a' \in \mathit{Val}(b)$. In either case we are done.
- $r_1 =$ `(`$a\ r_{1,1} \cdots r_{1,n}$`)`, $n \geq 0$. Since all a_i are distinct, we can infer that `(`a`,`$r_{1,1}$`,...,`$r_{1,n}$`)` $\leq_S r^{a_i}$ for some i : $1 \leq i \leq m$, and we are done by 4.9. □

8 SPKI's `AIntersect`

In this section we show that SPKI's AIntersect behaves as we expect when $\leq_S$ is interpreted as set containment, and when applied to the restricted syntax.

Since `AIntersect` is not completely defined in the SPKI documents we define this operation ourselves below. It is quite straigthforward to verify that our version fits the examples given in the draft standards.

To define `AIntersect` in the present slightly abstracted setting we need to assume that intersections exist at least on the level of set constants $b \in B$. That is, for all $b_1, b_2 \in B$ there is a b, denoted $b_1 \cap b_2$, such that $\mathit{Val}(b) = \mathit{Val}(b_1) \cap \mathit{Val}(b_2)$. We assume that $b_1 \cap b_2$ can be computed in time linear in the size of representation of b_1 and b_2.

Now, to define the `AIntersect` operation the set S is extended by the special constant $\perp$, denoting failure. For lists, if one of the argument positions is $\perp$, the entire list is $\perp$. For unions, if one of the argument positions is $\perp$ that argument is ignored. With these comments, the definition is given on fig. 1. In the figure a few symmetric cases are left out, in order not to clutter up the picture unnecessarily. Note that `AIntersect` is indeed well-defined as an operation on $S \cup \perp$. For time complexity we obtain:

Proposition 3 `AIntersect(`r_1`,`r_2`)` *is computable in time* $\mathcal{O}(n \log n)$ *where* n *is the sum of the lengths of* r_1 *and* r_2.

Proof Start by sorting the input such that elements of set expressions appear in order. This can be done in time $\mathcal{O}(n \log n)$. Once ordered, the computation of `AIntersect` is linear. □

Observe that proposition 3 applies to the restricted syntax only. Notice also that if authorisations can be assumed to be already sorted, a linear scan of the expressions suffices.

Finally we need to show that `AIntersect` is indeed the greatest lower bound with respect

$\texttt{AIntersect}((*),r) = r$

$\texttt{AIntersect}(r,(*)) = r$

$\texttt{AIntersect}(\bot,r) = \bot$

$\texttt{AIntersect}(r,\bot) = \bot$

$\texttt{AIntersect}(a,a) = a$

$\texttt{AIntersect}(a,b) = a$, if $a \in \mathit{Val}(b)$

$\texttt{AIntersect}(a,b) = \bot$, if $a \notin \mathit{Val}(b)$

$\texttt{AIntersect}(a,(a'\ r_1\ \cdots\ r_n)) = \bot$

$\texttt{AIntersect}(a,(*\ \texttt{set}\ r_1\ \cdots\ r_i = a\ \cdots\ r_n)) = a$

$\texttt{AIntersect}(a,(*\ \texttt{set}\ r_1\ \cdots\ r_i = b\ \cdots\ r_n)) = a$, if $a \in \mathit{Val}(b)$

$\texttt{AIntersect}(a,(*\ \texttt{set}\ r_1\ \cdots\ r_i\ \cdots\ r_n)) = \bot$, if none of above two cases apply

$\texttt{AIntersect}(b,b') = b \cap b'$

$\texttt{AIntersect}(b,(a,r_1\ \cdots\ r_n)) = \bot$

$\texttt{AIntersect}(b,(*\ \texttt{set}\ r_1\ \cdots\ r_n))$

$=$ $(*\ \texttt{set}\ \texttt{AIntersect}(b,r'_1)\ \cdots\ \texttt{AIntersect}(b,r'_m))$,
where $r'_1,\ldots,r'_m$ is the sequence of atomic elements in $r_1,\ldots,r_n$

$\texttt{AIntersect}((a\ r_1\ \cdots\ r_n),(a\ r'_1\ \cdots\ r'_n\ r'_{n+1}\ \cdots\ r'_m))$

$=$ $(\texttt{a}\ \texttt{AIntersect}(r_1,r'_1)\ \cdots\ \texttt{AIntersect}(r_n,r'_n)\ r'_{n+1}\ \cdots\ r'_m)$,
where $m \geq n$

$\texttt{AIntersect}((a\ r_1\ \cdots\ r_n),(a'\ r'_1\ \cdots\ r'_m)) = \bot$, if $a \neq a'$

$\texttt{AIntersect}((a\ r_1\ \cdots\ r_n),(*\ \texttt{set}\ r'_1\ \cdots\ r'_i\ \cdots\ r'_k))$

$=$ $\texttt{AIntersect}((a\ r_1\ \cdots\ r_n),r'_i)$, if r'_i has tag a

$\texttt{AIntersect}((a\ r_1\ \cdots\ r_n),(*\ \texttt{set}\ r'_1\ \cdots\ r'_m)) = \bot$,
if no r'_i has tag a

$\texttt{AIntersect}((*\ \texttt{set}\ r_1\ \cdots\ r_n),\ r\ \ \texttt{as}\ (*\ \texttt{set}\ r'_1\ \cdots\ r'_m))$

$=$ $(*\ \texttt{set}\ \texttt{AIntersect}(r_1,r)\ \cdots\ \texttt{AIntersect}(r_n,r))$

Figure 1: Definition of `AIntersect`

to $\leq_S$ for the restricted syntax. This verifies that

- The operation `AIntersect` behaves as we expect of an intersection operation
- The preorder $\leq_S$ behaves as we expect with respect to `AIntersect`

For this purpose recall that a semilattice is a structure with a binary operation which is idempotent, commutative, and associative. Further, we extend $\|\cdot\|$ to the domain $S \cup \bot$ by $\|\bot\| = \emptyset$.

Theorem 5 (Correctness of `AIntersect`)

1. $(S, \texttt{AIntersect})$ *is a semilattice.*
2. *For all* $r_1, r_2 \in R$, $\|\texttt{AIntersect}(r_1, r_2)\| = \|r_1\|$ *iff* $r_1 \leq_S r_2$.

Proof Both proofs are routine inductions. We leave out the proof of (1) altogether. For (2) we proceed by induction on the structure of r_1. We cover a couple of representative cases:

$r_1 = (a\ r_{1,1}\ \cdots\ r_{1,n})$: We proceed by cases in r_2. The cases where r_2 is one of `(*)`, $\bot$, or atomic are resolved by symmetric counterparts of equations in fig. 1. Remaining are:

- $r_2 = (a'\ r_{2,1}\ \cdots\ r_{2,m})$: If $a \neq a'$ then $\|\texttt{AIntersect}(r_1, r_2)\| = \emptyset \neq \|r_1\|$ and $\|r_1\| \not\subseteq \|r_2\|$. If $a = a'$ we can assume that $m \geq n$ the case otherwise is symmetric. The conclusion now follows directly by the induction hypothesis.
- $r_2 = (\texttt{* set}\ r_{2,1}\ \cdots\ r_{2,m})$: We obtain $\|\texttt{AIntersect}(r_1, r_2)\| \neq \emptyset$ just in case exactly one $r_{2,i}$ has tag a, which is sufficient to establish the case. □

9 Conclusion

We have shown how a restricted syntax for the SPKI authorisation element can be defined such that general authorisations and entailments between authorisations can be decided in almost linear time. Moreover, the restricted syntax appears to follow existing SPKI practice, so no real restriction in expressive power or usage is incurred.

To which extent our results are important in practice can be discussed. The computation of `AIntersect` is simplified when queries do not involve unions, i.e. the `* set` construct. This is the assumption made, for instance, in the Pisces implementation (see url: www.cnri.reston.va.us/software/pisces/). At any rate, as long as authorisation expressions and certificate chains remain small, the overhead may be negligible. Moreover, SPKI's simple delegation model enables chaining to be decided in polynomial time [2].

So one may argue that the problem is in practice negligible. We do not think this point of view is necessarily valid. First, we have not found such a thing as a clear and well-established SPKI practice. Nothing in the draft standards prohibits the use of unions in requests, and this capability might very well be used in practice. Several examples can be given. For instance, an application programmer might wish to exploit the revocation predictability built into the SPKI framework by computing a set of requests in advance. Or, as another example, it might be deemed useful to use the union construction to introduce macros. For instance, `USLocs`, `MidWestLocs`, etc., might be introduced as macros (at the application level) representing S-expressions of the form e.g.

```
MidWestLocs =
(* set
...
(location Nebraska Lincoln)
(location Kansas Topeka Centre)
(location Kansas Topeka North)
(location Kansas Wichita)
...  )
```

There is no prior reason why such a macro might not appear as part of a request, say, to determine whether access to Midwestern branch office sales statistics is permitted or not. The result, however, can be serious per-

formance degradation at request time.

Going beyond SPKI as it currently stands there is also the possibility that new mechanisms, for instance for delegation (cf. [1, 6, 7]), will be introduced which require comparisons to be made between authorisations of a general shape. An important purpose of the present paper is to set the stage for further studies in this direction, in terms of an evaluation model with good computational properties.

Acknowledgements Thanks to Dieter Gollmann, Microsoft Research, Cambridge, also to Babak Sadighi and Roland Hedberg, SICS, and to Thom Birkeland at IMIT/KTH.

References

[1] O. Bandmann, M. Dam, and B. Sadighi Firozabadi. Constrained delegation. In *Proc. 23rd Annual Symp. on Security and Privacy*, 2002. To appear.

[2] Dwaine Clarke, Jean-Emile Elien, Carl Ellison, Matt Fredette, Alexander Morcos, and Ronald L. Rivest. Certificate chain discovery in SPKI/SDSI, 1999.

[3] Carl M. Ellison, Bill Frantz, Butler Lampson, Ron Rivest, Brian M. Thomas, and Tatu Ylonen. SPKI Certificate Theory, May 1999. RFC 2693, expired. URL: ftp://ftp.isi.edu/in-notes/rfc2693.txt.

[4] Carl M. Ellison, Bill Frantz, Butler Lampson, Ron Rivest, Brian M. Thomas, and Tatu Ylonen. Simple public key certificate, July 1999. Internet Draft, expired. URL: http://world.std.com/ cme/spki.txt.

[5] Carl M. Ellison, Bill Frantz, Butler Lampson, Ron Rivest, Brian M. Thomas, and Tatu Ylonen. SPKI examples, March 1998. Internet Draft, expired. URL: http://world.std.com/ cme/examples.txt.

[6] B. Sadighi Firozabadi, M. Sergot, and O. Bandmann. Using Authority Certificates to Create Management Structures. To appear in Proc. 9th Security Protocols Workshop, Cambridge, UK, April 2001.

[7] Jon Howell and David Kotz. A formal semantics for SPKI. In *Proc. 6th European Symposium on Research in Computer Security*, 2000.

[8] Ron Rivest. S-expressions, May 1997. Internet Draft, expired. URL: http://theory.lcs.mit.edu/ rivest/sexp.txt.