Regular SPKI

Mads Dam[⋆]

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

Abstract. SPKI is a certificate-based framework for authorisation in distributed systems. The SPKI framework is extended by an iteration construct, essentially Kleene star, to express constraints on delegation chains. Other possible applications, not explored in the paper, include multidomain network routing path constraints. The main decision problems for the extended language are shown to correspond to regular language membership and containment respectively. To support an efficient decision algorithm in both cases we give a sound and complete inference system for a fragment of the language which is decidable in polynomial time. We finally show how to use the extended syntax to represent constrained delegation in SPKI.

1 Introduction

SPKI (Simple Public-Key Infrastructure) [EFL+99] is a framework for authorisation based on the idea that authorisations and names are bound to public keys by signed certificates. SPKI uses LISP-like S-expressions [Riv97] to express authorisations. As an example, the S-expression

```
(object document (attributes (name doc1) (loc EU))
                 (op read) (subject (users orgA)))      (1)
```

might express the authority of orgA users to read objects of type document which have name doc1 and are located in the EU[1].

The treatment of delegation in SPKI is rudimentary. Authorisations are equipped with a flag which, when set, enable holders of authorisations unconstrained delegation rights for that authority. In [BDF02] we argued that such unconstrained delegation right are not always desirable, and we proposed a mechanism, *constrained delegation*, which uses regular expressions to control the way authority is propagated between principals along a delegation chain. It is the objective of the present paper to examine how the constrained delegation approach can be handled within the SPKI framework.

[⋆] Work supported by Vinnova, the Swedish Agency for Innovation Systems, project "policy-based network management".

[1] In fact, proper SPKI treats such authorisations using a specialised 5-tuple syntax. This, though, is of little consequence for the present discussion.

In [BDF02], delegation chain constraints are expressed by regular expressions. As a simple example, the chain constraint `admin`*`users` expresses the set of delegation chains which have a prefix in `admin` and terminate in `users`. This chain constraint would capture the situation where a group of administrators are given authority to create some management structure within the group for administering, say, the access rights of users.

Our proposal for handling this within the SPKI framework is to extend the SPKI syntax by a new primitive, `(* path ...)`, denoting, roughly, Kleene star. To see how this might be done, consider the following S-expression:

```
(object document (attributes (name doc1) (loc EU))
            (op read) (delegation (* path (admin)) (users)) (2)
```

In this example[2] the `subject` component of (1) is replaced by a component `delegation` that represents delegation chain constraints. Here delegation is licensed through `admin` in any number of steps, but the read permissions that are ultimately granted must be under users control.

Built in to SPKI is the idea of possibly refining, at each step of delegation, the authority received in the previous step. The basic mechanism for refinement is to recursively append further constraints to each list. In the presence of `path` expressions, unfolding must be added to this mechanism. As an example, (2) could be used to justify:

```
(object document (attributes (name doc1)
   (loc EU France)) (op read) (delegation (admin domain1)
                          (* path (admin domain2)) (users)))
```

restricting documents to the location France, and allowing administrators in domain1 to delegate through administrators in domain2 an arbitrary number of steps. In turn, this S-expression can be further refined, and the delegated authority discharged, to ultimately result in authorisations similar to (1) above. Below, in section 7, we give an example showing this process of delegation in more detail.

In the paper we discuss mechanisms which could be used for such a regular language extension to SPKI. At the basic level two extensions are required: S-expression concatenation, and Kleene star (`path`). We show how this can be done, and how basic questions concerning S-expressions can then be reduced to corresponding questions for regular languages. This is sufficient to efficiently answer simple authorisation queries. Refinement, however, corresponding to regular language containment, will require exponential time. Chain discovery, then, will also be exponential. To address this we introduce a restricted syntax for which refinement is decidable in polynomial time. We present an inference system for entailment (containment) for this fragment which is based on fixed point induction. The inference system is shown to be sound and complete. The completeness

[2] Which is actually wrong: Using the notation introduced in the paper, the delegation component should properly be written `(delegation (* path (admin));((users)))` where `;` is list concatenation.

proof, in particular, allows the decision procedure to be extracted. We then show how to use the regular expression facility to represent constrained delegation, and how name resolution and delegation can be handled in this framework.

2 S-Expressions

A SPKI expression denotes a set of S-expressions [Riv97]. Let A be a denumerable set of "atomic" elements ranged over by a of one or several data types such as strings, octets, or integers. The set S of *S-expressions*, ranged over by s, is determined by the following BNF style grammar:

$$s \ ::= \ a \ | \ (s_1 \ \cdots \ s_n)$$

where $n \geq 0$. In other words, S is the tuple algebra over A, and, e.g., $(a_1 \ a_2)$ is the tuple with left hand component $a_1 \in A$ and right hand component $a_2 \in A$. To account for authorisation we introduce a partial ordering $\leq$ on S. Let $s_1, s_2 \in S$.

1. If $s_1 \in A$ or $s_2 \in A$ then $s_1 \leq s_2$ if and only if $s_1 = s_2$.
2. If $s_1 = (s_{1,1} \ \cdots \ s_{1,m}) \in S$ and $s_2 = (s_{2,1} \ \ldots \ s_{2,n}) \in S$, then $s_1 \leq s_2$ if and only if $m \geq n$ and $s_{1,i} \leq s_{2,i}$ for $i = 1, \ldots, n$.

That is, $s_1 \leq s_2$ just in case s_1 is more specific (more constrained, or authorised by) s_2, and that the process of becoming "more specific" is by appending more information to the end of sublists.
Example 1.

```
  (object document (attributes (name doc1) (loc EU France)
                                                      (author NN)))
    ≤ (object document (attributes (name doc1) (loc EU)))
```

If s represents a policy authorised for a principal x, and s' represents a request of x such that $s' \leq s$, then and only then should the request s' be granted. In the SPKI literature this idea is usually treated not using the partial order $\leq$, but through the associated operation of glb, or intersection, such that $s' \leq s$ iff $s \cap s' = s'$ (cf. [BD02]).

In SPKI, S-expressions are usually required to begin with an atom. The leading atom, which we refer to as a "tag" below, serves as a type indicator. That is, the type of an element s_i of a list $(a \ s_1 \ \cdots \ s_n)$ will in standard SPKI be determined by the tag a and the position i. These type associations are determined by some external means; here it suffices to assume some fixed such binding, when it applies.

3 Syntax of Regular SPKI Expressions

S-expressions provide a basic syntax for expressing constrained authorisation, but the notation is not really versatile enough for practical use. For this reason the SPKI authorisation syntax extends S-expressions with the following features:

– Constructs to denote sets of atoms (the SPKI prefix and range constructs). These constructs are left out of the present treatment, but they can be added without substantial complications.
– Constructs to denote sets of lists (the wildcard `(*)` and the `* set` construct).

To this constructs we add the following two:

– An iterator, `* path`, basically Kleene star.
– Composition of S-expressions, denoted by semicolon.

Definition 1 (Regular SPKI Expressions). The set $\mathcal{S}$ of *regular SPKI expressions*, ranged over by σ, is determined as follows:

$$\sigma ::= \texttt{(*)} \mid a \mid \texttt{(}\sigma_1 \cdots \sigma_n\texttt{)} \mid \sigma_1\texttt{;}\sigma_2 \mid \\ \texttt{(* set } \sigma_1 \cdots \sigma_n\texttt{)} \mid \texttt{(* path } \sigma\texttt{)}$$

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

Essentially, SPKI expressions can be regarded as abbreviations of sets of S-expressions. This is brought out by the semantics, fig. 1.

$$\begin{aligned}
&\|\texttt{(*)}\| = S \\
&\|a\| = \{a\} \quad \text{for all } a \in A \\
&\|\texttt{(}\sigma_1 \cdots \sigma_n\texttt{)}\| = \{\texttt{(}s_1 \cdots s_n\texttt{)} \mid \forall i : 1 \ldots n . s_i \in \|\sigma_i\|\} \\
&\|\sigma_1\texttt{;}\sigma_2\| = \\
&\quad \{\texttt{(}s_1\texttt{,}\ldots\texttt{,}s_n\texttt{)} \mid \exists i : 1 \ldots n . \texttt{(}s_1 \cdots s_i\texttt{)} \in \|\sigma_1\|, \texttt{(}s_{i+1} \cdots s_n\texttt{)} \in \|\sigma_2\|\} \\
&\|\texttt{(* set } \sigma_1 \cdots \sigma_m\texttt{)}\| = \|\sigma_1\| \cup \ldots \cup \|\sigma_m\| \\
&\|\texttt{(* path } \sigma\texttt{)}\| = \{\texttt{(}s_1 \cdots s_n\texttt{)} \mid \forall i : 1 \ldots n . s_i \in \|\sigma\|\}
\end{aligned}$$

Fig. 1. Regular SPKI expressions, semantics

Let $\sigma_1 \cong \sigma_2$ iff $\|\sigma_1\| = \|\sigma_2\|$. General lists are definable in terms of composition (`;`) and singleton lists, since

$$\texttt{(}\sigma_1 \cdots \sigma_n\texttt{)} \cong \texttt{(}\sigma_1\texttt{)} \ \texttt{;} \ \cdots \ \texttt{;} \ \texttt{(}\sigma_n\texttt{)}. \tag{3}$$

Definability the other direction does not hold. This is easily seen, as the list constructor strictly increases depth of nesting which composition does not. As a consequence we only need to consider the empty list (`()`) and singletons (`(`σ`)`) as primitive. This is exploited heavily below. It is important, however, to bear equation (3) in mind, since it will allow the composition operator to be eliminated in favour of the more standard list syntax in all “reasonable” cases, except those that specifically involve path expressions.

Example 2. Let $\sigma = \texttt{(}a\ \texttt{(* path}\ b\texttt{)}\ c\texttt{)}$. We compute:

$$\begin{aligned}\|\sigma\| &= \{\texttt{(}s_1\ \ s_2\ \ s_3\texttt{)} \mid s_1 \in \|a\|, s_2 \in \|\texttt{(* path}\ b\texttt{)}\|, s_3 \in \|c\|\} \\ &= \{\texttt{(}a\ \ \texttt{(}s_1\ \ \cdots\ \ s_n\texttt{)}\ \ c\texttt{)} \mid \forall i.s_i \in \|b\|\} \\ &= \{\texttt{(}a\ \ \texttt{(}b\ \ \cdots\ \ b\texttt{)}\ \ c\texttt{)}\}\end{aligned}$$

Compare with $\sigma' = \texttt{(}a\texttt{);(* path}\ b\texttt{);(}c\texttt{)}$:

$$\begin{aligned}\|\sigma'\| = \{\texttt{(}s_1\ \ \cdots\ \ s_n\texttt{)} \mid n \geq 2, s_1 \in \|a\|,& \\ \texttt{(}s_2\texttt{,}\ldots\texttt{,}s_{n-1}\texttt{)} \in \|\texttt{(* path}\ b\texttt{)}\|, s_n \in \|c\|\}& \\ = \{\texttt{(}a\ \ s_2\ \ \cdots\ \ s_{n-1}\ \ c\texttt{)} \mid n \geq 2, \forall i : 1 \ldots n.s_i \in \|b\|\}& \\ = \{\texttt{(}a\ \ b\ \ \cdots\ \ b\ \ c\texttt{)}\} \quad .&\end{aligned}$$

The semantics of fig. 1 is not the only one possible. A different semantics, $\|\sigma\|'$, would introduce ; as concatenation of s-expression lists, and then define $\|\sigma\|'$ as in fig. 1 except that:

$$\|\texttt{(* path}\ \sigma\texttt{)}\|' = \{s_1\texttt{;}\cdots\texttt{;}s_n \mid \forall i : 1 \leq i \leq n.s_i \in \|\sigma\|\}. \tag{4}$$

The two semantics are easily related, since obviously

$$\|\texttt{(* path}\ \sigma\texttt{)}\| = \|\texttt{(* path (}\sigma\texttt{))}\|'.$$

We prefer the semantics of fig. 1 since the notation in the latter case seems to contribute not much more than the need to add a few extra parentheses.

The partial ordering $\leq$ on S-expressions is extended to regular SPKI expressions in the following way:

$$\sigma_1 \leq \sigma_2 \text{ iff } \forall s_1 \in \|\sigma_1\|.\exists s_2 \in \|\sigma_2\|.s_1 \leq s_2 \tag{5}$$

To see that this definition makes sense, let

$$\downarrow\!\sigma = \{s \mid \exists s' \in \|\sigma\|.s \leq s'\} \quad .$$

The set $\downarrow\!\sigma$ is the "downwards closure" of $\|\sigma\|$ according to $\leq$. In the intuitive sense of section 2 it is the set of all S-expressions which are authorised by some element in $\|\sigma\|$. The following is standard:

Proposition 1. *$\sigma_1 \leq \sigma_2$ iff $\downarrow\!\sigma_1 \subseteq \downarrow\!\sigma_2$* □

In other words, $\sigma_1 \leq \sigma_2$ just in case every S-expression authorised by σ_2 is also authorised by σ_2.

Two problems are of central interest:

1. P1, membership: Given a request formulated as an S-expression s and an authorisation policy σ, is $s \in \downarrow\!\sigma$?
2. P2, entailment: Given authorisation policies σ_1, σ_2, is σ_1 authorised by σ_2 (i.e. does $\sigma_1 \leq \sigma_2$ hold)?

It is not very difficult to cast these problems in terms of regular languages. Define an ancillary ordering on S by $s_1 \sqsubseteq s_2$ iff either $\exists a \in A. s_1 = a = s_2$ or else $s_1 = (s_{1,1} \ \cdots \ s_{1,m})$, $s_2 = (s_{2,1} \ \cdots \ s_{2,m})$, and $s_{1,i} \leq s_{2,i}$ for all $i : 1 \leq i \leq m$. That is, $\sqsubseteq$ acts just as $\leq$ except that appending rightmost list elements to the outermost list is not permitted. Let then

$$\Downarrow\sigma = \{s \mid \exists s' \in \|\sigma\|. s \sqsubseteq s'\}$$

Now, consider S-expressions as given in the form $(s_1)\texttt{;}\cdots\texttt{;}(s_m)$ instead of $(s_1 \ \cdots \ s_m)$, and consider `(;)` and Kleene star closure $(\cdot)^*$ as operations on sets C of S-expressions as follows:

$$\begin{aligned} C_1\texttt{;}C_2 &= \{s_1\texttt{;}s_2 \mid s_1 \in C_1, s_2 \in C_2\} \\ C^* &= \{s_1\texttt{;}\cdots\texttt{;}s_n \mid \forall i : 1 \leq i \leq n. s_i \in C\} \end{aligned}$$

We obtain the following characterisation of closure sets (proof is given in the appendix):

Proposition 2.

1. $\Downarrow\texttt{(*)} = \|\texttt{(*)}\|$
2. $\Downarrow\texttt{()} = \texttt{()}$
3. $\Downarrow\texttt{(}\sigma\texttt{)} = \{\texttt{(}s\texttt{)} \mid s \in \downarrow\sigma\}$
4. $\Downarrow\sigma_1\texttt{;}\sigma_2 = \Downarrow\sigma_1\texttt{;}\Downarrow\sigma_2$
5. $\Downarrow\texttt{(* set }\sigma_1 \ \cdots \ \sigma_m\texttt{)} = \Downarrow\sigma_1 \cup \cdots \cup \Downarrow\sigma_m$
6. $\Downarrow\texttt{(* path }\sigma\texttt{)} = (\Downarrow\sigma)^*$
7. $\downarrow\sigma = \Downarrow\sigma\texttt{;}\Sigma^*$ $\square$

This proposition provides a direct representation of regular SPKI expressions as "ordinary" regular expressions, and so we obtain:

– $\downarrow\sigma$ is a regular language
– P1 is regular language membership. Thus P1 is decidable in time $\mathcal{O}(|s||\sigma|)$. Moreover, since there is a trivial logspace reduction of regular language membership to P1, P1 is also complete for NLOGSPACE.
– P2 is regular language containment. This follows directly from fact 1. Thus P2 is in EXPTIME and complete for PSPACE.

4 EOL Markers

There is a basic tension between the introduction of path expressions and the basic S-expression syntax. In particular, S-expressions are intended to be positional in the sense explained in section 2. But this positionality breaks down in the context of path expressions. Consider the following example:

$$\sigma = \texttt{(mysequence);((start));(* path (hop));((end))} \qquad (6)$$

An S-expression in $\downarrow\sigma$ can have a shape like

$$\begin{aligned} s = {} & \texttt{(mysequence (start here) (hop there) (hop and-there)} \\ & \texttt{(end over-here) (unintended bit))}. \end{aligned} \tag{7}$$

The SPKI authorisation discipline will admit s as authorised by σ, since the extra component `(unintended bit)` is appended to the right of the list and so just represents one further constraint. But in the context of path expressions this is counter-intuitive, since we may not have a preconceived idea of what the last element of a path is, and so we may not know whether `(end over-here)` or `(Unintended bit)` represents that element. For instance, were the `((end))` item to be missing from σ, an attacker could insert new hops at the end of σ at will. The example also gives away the solution: Simply assume elements with some given tag to represent the end of the list. Any application-dependent choice will do, but we may also introduce a general-purpose atom `EOL` to represent the end of the list. In this manner we will represent σ as

$$\sigma' = \texttt{(mysequence);((start));(* path (hop));((end));(EOL)}. \tag{8}$$

Observe that the addition of the `EOL` atom does not interfere with the semantics in any way.

5 An Efficient Syntax for Entailment

Through the characterisation of SPKI expression in terms of regular languages we obtain a reasonably efficient procedure for deciding the problem P1, is a given S-expression s authorised by the regular SPKI expression σ. The problem P2, however, remains intractable. It may be argued, as is sometimes done in the SPKI literature, that requests will in practice not need to involve the problematic constructions (in the absence of path expressions this means set expressions), but a closer examination of this issue reveals this to be false in many applications (cf. [BD02] for a brief discussion). At any rate the entailment problem is in our view of independent interest, for instance to allow users to efficiently determine the effects of their policy decisions. To address this problem we introduce in this section a restricted syntax for which an efficient decision procedure also for P2 is possible.

In [BD02] we addressed this issue for the basic SPKI authorisation syntax, and obtained an $n \log n$ asymptotic complexity for P2 in this case. The idea was to restrict occurrences of `set` expressions to those of the form

$$\sigma = \texttt{(* set } a_1\texttt{;}\sigma_1 \;\cdots\; a_n\texttt{;}\sigma_n\texttt{)}$$

where all a_i are required to be distinct atoms. This allows queries of the form $a;\sigma' \le \sigma$ to be directly reduced to the query $\sigma' \le \sigma_i$ where $a = a_i$, if such an a_i exists, and if it does not, the query is rejected. This syntactical restriction is just a formalisation of existing SPKI practice (to the extent such a thing exists): It does not in any way reduce the expressiveness of the basic SPKI authorisation syntax.

To extend this approach also to path expressions the idea is simply to tag path expressions as well as sets in such a manner that it becomes immediate how to match a path expression with its unfoldings. For instance, the presence of the `hop` atom makes it trivial to determine that s in (7) is authorised by σ of (6), as is the regular SPKI expression

`(mysequence);((start));((hop));(* path (hop));((end))` (9)

whereas an expression such as

`(mysequence);((start));(* path (hop));(* path (hop));((end))` (10)

would be more difficult to accomodate in principle. This solution we propose is to replace uniqueness at the level of tags with uniqueness at the level of initial segments, as in the following expression:

```
(mysequence);((start));(* path (hop domain1));
        (* path (hop domain2));((end)) .
```
(11)

We proceed to introduce the restricted syntax which makes such a tagging discipline enforceable.

Definition 2 (Restricted Expressions). The set R of restricted expressions, ranged over by r, is given as follows:

$$\begin{array}{rcl} r & ::= & \texttt{(}a\texttt{);}p \mid \texttt{(* set } r^{a_1} \cdots r^{a_m}\texttt{)} \\ r^a & ::= & \texttt{(}a\texttt{);}p \\ p & ::= & \texttt{()} \mid q\texttt{;}p \\ q & ::= & \texttt{(}a\texttt{)} \mid \texttt{(}r\texttt{)} \mid \texttt{(* path } r\texttt{)} \end{array}$$

where $a \in A$, $m \geq 1$, and all a_i, $1 \leq i \leq m$, are distinct.

We generally let $\texttt{(}a\texttt{)}$ abbreviate $\texttt{(}a\texttt{);()}$.

Example 3. Keep in mind the definition of list expressions, def. 3. The following regular SPKI expressions are restricted:

- $\texttt{(}a\ \texttt{(}b\ c\texttt{))} = \texttt{(}a\texttt{);((}b\texttt{);(}c\texttt{))}$
- $\texttt{(* set (}a\ \texttt{foo) (}b\texttt{);(bar))} = \texttt{(* set (}a\texttt{);(foo) (}b\texttt{);(bar))}$
- $\texttt{(}a\texttt{);(* path (}b\texttt{);(* path (}c\texttt{)))}$

The following regular SPKI expressions are *not* restricted:

- a, b, `(*)`
- $\texttt{((}a\ b\texttt{))} = \texttt{((}a\texttt{);(}b\texttt{))}$
- $\texttt{(* set (}a\ b\texttt{) (}a\ c\texttt{))} = \texttt{(* set (}a\texttt{);(}b\texttt{) (}a\texttt{);(}c\texttt{))}$
- $\texttt{(}a\ \texttt{(* path }b\texttt{))} = \texttt{(}a\texttt{);((* path }b\texttt{))}$
- $\texttt{(}a\texttt{);(* path (* path }b\texttt{))}$

The function *tags* computes the set of tags of expressions r and q respectively:

- *tags* $(\texttt{(}a_1\texttt{)};\cdots;\texttt{(}a_m\texttt{)};q_1;\ldots;q_n) = \{\texttt{(}a_1\ \cdots\ a_m\texttt{)}\}$, where q_1 is not of the shape $\texttt{(}a\texttt{)}$ for any a
- *tags* $(\texttt{(* set } r^{a_1},\ldots,r^{a_m}\texttt{)}) = \mathit{tags}\ (r^{a_1}) \cup \cdots \cup r^{a_m}$
- *tags* $(\texttt{(}a\texttt{)}) = \emptyset$
- *tags* $(\texttt{(}r\texttt{)}) = \mathit{tags}\ (r)$
- *tags* $(\texttt{(* path r)}) = \mathit{tags}\ (r)$

Definition 3 (Well-formed Restricted Expressions). The restricted expression r is *well-formed* if whenever r contains a subexpression of the shape $r' = \texttt{(}a\texttt{)};q_1;\cdots;q_n$ and q_i has the shape $\texttt{(* path } r''\texttt{)}$ then for all $j > i$,

$$\mathit{tags}\ (q_j) \cap \mathit{tags}\ (r'') = \emptyset \quad .$$

Example 4. The expressions (9) and (11) are well-formed. The expression (10) is ill-formed, as is the expression

$$\texttt{(mysequence);((start));(* path (hop));((hop));((end))} \qquad (12)$$

6 Inference System

We proceed to give, in fig. 2, an inference system for proving entailments of the form $r_1 \preceq r_2$, intended as syntactical correlates of the entailment relation $r_1 \leq r_2$. Call an expression e of one of the forms $r_1 \preceq r_2$ or $p_1 \preceq p_2$ an *entailment expression*. Judgments have the shape $\Gamma \vdash e$ where e is an entailment expression and Γ is a set of entailment expressions. The proof system implements a form of fixed point induction, in the style of Kozen [Koz83]. It is designed to be used in a bottom-up fashion, and can in fact be read just as a logic program. To show the proof system in action we give a couple of example derivations.

Proposition 3. *The following entailments are derivable:*

1. $\vdash \texttt{(* path (}a\ b\texttt{))} \preceq \texttt{(* path (}a\texttt{))}$.
2. $\vdash \texttt{(* path (}a\ b\texttt{));(* path (}a\ c\texttt{))} \preceq \texttt{(* path (}a\texttt{))}$.

Proof. 1. Reduce first using IX to obtain the subgoals

$$\vdash \texttt{()} \preceq \texttt{(* path (}a\texttt{))} \qquad (13)$$

$$\texttt{(* path (}a\ b\texttt{))} \preceq \texttt{(* path (}a\texttt{))} \vdash$$
$$\texttt{((}a\ b\texttt{));(* path (}a\ b\texttt{))} \preceq \texttt{(* path (}a\texttt{))}. \qquad (14)$$

Subgoal (13) is resolved using VII and II. Subgoal (14) is resolved by VIII first to

$$\texttt{(* path (}a\ b\texttt{))} \preceq \texttt{(* path (}a\texttt{))} \vdash$$
$$\texttt{((}a\ b\texttt{));(* path (}a\ b\texttt{))} \preceq \texttt{((}a\texttt{));(* path (}a\texttt{))} \qquad (15)$$

$$\text{I}\quad \frac{\cdot}{\Gamma, p_1 \preceq p_2 \vdash p_1 \preceq p_2}$$

$$\text{II}\quad \frac{\cdot}{\Gamma \vdash p \preceq \texttt{()}}$$

$$\text{III}\quad \frac{\Gamma \vdash p_1 \preceq p_2}{\Gamma \vdash \texttt{(}a\texttt{)}\texttt{;}p_1 \preceq \texttt{(}a\texttt{)}\texttt{;}p_2}$$

$$\text{IV}\quad \frac{\Gamma \vdash \texttt{(}a\texttt{)}\texttt{;}p \preceq r^{a_i}}{\Gamma \vdash \texttt{(}a\texttt{)}\texttt{;}p \preceq \texttt{(* set } r^{a_1}\texttt{,}\ldots\texttt{,}r^{a_n}\texttt{)}}\ 1 \leq i \leq n$$

$$\text{V}\quad \frac{\Gamma \vdash r^{a_1} \preceq r \quad \cdots \quad \Gamma \vdash r^{a_n} \preceq r}{\Gamma \vdash \texttt{(* set } r^{a_1}\texttt{,}\ldots\texttt{,}r^{a_n}\texttt{)} \preceq r}$$

$$\text{VI}\quad \frac{\Gamma \vdash r_1 \preceq r_2 \qquad \Gamma \vdash p_1 \preceq p_2}{\Gamma \vdash \texttt{(}r_1\texttt{)}\texttt{;}p_1 \preceq \texttt{(}r_2\texttt{)}\texttt{;}p_2}$$

$$\text{VII}\quad \frac{\Gamma \vdash p_1 \preceq p_2}{\Gamma \vdash p_1 \preceq \texttt{(* path } r\texttt{)}\texttt{;}p_2}$$

$$\text{VIII}\quad \frac{\Gamma \vdash p_1 \preceq (r)\texttt{;(* path } r\texttt{)}\texttt{;}p_2}{\Gamma \vdash p_1 \preceq \texttt{(* path } r\texttt{)}\texttt{;}p_2}$$

$$\text{IX}\quad \frac{\Gamma \vdash p_1 \preceq p_2 \qquad \Gamma, \texttt{(* path } r\texttt{)}\texttt{;}p_1 \preceq p_2 \vdash \texttt{(}r\texttt{)}\texttt{;(* path } r\texttt{)}\texttt{;}p_1 \preceq p_2}{\Gamma \vdash \texttt{(* path } r\texttt{)}\texttt{;}p_1 \preceq p_2}$$

Fig. 2. Inference system

then VI to

$$\texttt{(* path (}a\ b\texttt{))} \preceq \texttt{(* path (}a\texttt{))} \vdash \texttt{(}a\ b\texttt{)} \preceq \texttt{(}a\texttt{)} \tag{16}$$

$$\texttt{(* path (}a\ b\texttt{))} \preceq \texttt{(* path (}a\texttt{))} \vdash \texttt{(* path (}a\ b\texttt{))} \preceq \texttt{(* path (}a\texttt{))} \tag{17}$$

which are resolved using III and II, respectively I.
2. The proof reduces, using IX, to

$$\vdash \texttt{(* path (}a\ c\texttt{))} \preceq \texttt{(* path (}a\texttt{))} \tag{18}$$

$$\texttt{(* path (}a\ b\texttt{));(* path (}a\ c\texttt{))} \preceq \texttt{(* path (}a\texttt{))} \vdash \texttt{((}a\ b\texttt{));(* path (}a\ b\texttt{));(* path (}a\ c\texttt{))} \preceq \texttt{(* path (}a\texttt{))} \tag{19}$$

Subgoal (18) is an instance of 1. The proof of (19) follows that of (14). □

The proof system is well-behaved with respect to restricted syntax in the sense that, when used in a bottom-up fashion, if the initial judgment is well-formed then subsequent judgments will be well-formed as well.

Proposition 4. *Let*

$$\frac{\Gamma_1 \vdash e_1 \qquad \cdots \qquad \Gamma_n \vdash e_n}{\Gamma \vdash e}$$

be any instance of one of the proof rules I–IX*. If $\Gamma \vdash e$ is well-formed then so are all $\Gamma_i \vdash e_i$, $1 \le i \le n$.* □

We proceed to consider soundness and completeness. Say that Γ is *valid* if $r_1 \le r_2$ whenever $r_1 \preceq r_2 \in \Gamma$ ($p_1 \preceq p_2 \in \Gamma$), and say that $\Gamma \vdash r_1 \preceq r_2$ is *valid*, written $\Gamma \models r_1 \preceq r_2$, if Γ valid implies $r_1 \le r_2$. Similar definitions apply to terms of the form p_1, p_2. Soundness holds for arbitrary regular SPKI expressions, not only for restricted ones. This is readily apparent from the soundness proof given in the appendix.

Theorem 1 (Soundness). *If $\Gamma \vdash r_1 \preceq r_2$ then $\Gamma \models r_1 \le r_2$* □

Completeness, however, holds only for well-formed, restricted expressions. Problematic cases are the rules for sets (rule IV) and paths (rule VII and VIII) which make use of the restricted format in an essential way.

Theorem 2 (Completeness). *Suppose that r_1, r_2 are well-formed restricted expressions. If $r_1 \le r_2$ then $r_1 \preceq r_2$.* □

The completeness proof is constructive, and provides an algorithm which can be used to decide entailments. If used as-is this algorithm is quadratic: In the worst case, at each step as the input expressions r_1 and r_2 are scanned, set expressions must be scanned against each other. This can easily be brought down to $\mathcal{O}(n \log n)$ if set expressions are sorted according to their tags. So we obtain:

Theorem 3 (Worst-Case Complexity). *The relation $r_1 \le r_2$ is decidable in time $\mathcal{O}(n \log n)$ where n is the sum of the lengths of r_1 and r_2.*

Proof The completeness proof provides an $\mathcal{O}(n \log n)$ algorithm, provided the expressions are sorted. If the input expressions are unsorted, a preprocessing phase of $\mathcal{O}(n \log n)$ brings them into sorted form first. □

7 Constrained Delegation in SPKI

In this section we discuss how path expressions can be used to represent constrained delegation in SPKI.

SPKI has both a naming and an authorisation component. If we ignore validity checking issues we can, for the purpose of the present discussion, view a SPKI name certificate as a triple

$$\texttt{(}k\texttt{,}n\texttt{,}s\texttt{)} \tag{20}$$

where k is an S-expression representing a key, n is a string atom, and s is an S-expression representing a key or a SDSI name, an S-expression of the form

$$\texttt{(name } k_1 \ n_1 \ \cdots \ n_m\texttt{)} \ .$$

For instance, if n is the atom `personnel-dept` and s is the SDSI name `(name` k_1 `head-office personnel-dept section1)` then the certificate (20) should be read as

"k's `personnel-dept` is k_1's `head-office`'s `personnel-dept`'s `section1`".

Delegation chains will refer to principals, as keys or as SDSI names. Hence entailment must be extended to take name resolution into account. Curiously, this aspect is ignored in standard SPKI. There, authorisation expressions [3] are taken as primitive S-expressions, and there is no defined mechanism for resolving a name appearing as part of a standard SPKI authorisation expression.

Entailment is easily extended to take name resolution into account, using a rewriting approach akin to that of [CEE+01]. One rule and a rule schema needs to be added:

$$\text{X} \quad \frac{\cdot}{\Gamma \vdash s\texttt{;(EOL)} \preceq \texttt{(name } k \ n \texttt{ EOL)}} \ (k\texttt{,}n\texttt{,}s) \texttt{ is valid}$$

$$\text{XI} \quad \frac{\Gamma \vdash s_1\texttt{;(EOL)} \preceq s_1'\texttt{;(EOL)} \qquad \Gamma \vdash s_1 \preceq s_1\texttt{;}s_2'}{\Gamma \vdash s_2 \preceq s_1'\texttt{;}s_2'}$$

Observe that rule `(XI)`, with the rules of fig. 2 but in the absence of `(X)`, is admissible, because of completeness. Thus, the only new entailments provable are those arising because of naming (schema `(X)`).

A SPKI authorisation certificate (auth cert) can be viewed as a 4-tuple (again we ignore validity checking):

$$(k\texttt{,}s\texttt{,}d\texttt{,}t) \tag{21}$$

where k is a key, s is a SDSI name, d is a delegation flag, and t is a SPKI tag, a SPKI authorisation expression. There are several ways to adapt the SPKI certificate format to constrained delegation. In this section we consider the case of replacing the d flag with a regular SPKI expression determining a delegation constraint. An alternative would be to include the delegation constraint in the authorisation tag t, as indicated in section 1. This would gain some backwards compatibility at the expense of some notational clarity.

An *extended auth cert* would thus be a certificate as (21) except that d and t would both be regular SPKI expressions, and d, in particular, would represent lists of a form such as

$$\texttt{(delegation } \tau_1\texttt{;(}s_1\texttt{)} \ \cdots \ \tau_n\texttt{;(}s_n\texttt{))} \tag{22}$$

where $\tau_1, \ldots, \tau_n$ are tags (in the sense of section 5) and $s_1, \ldots, s_n$ are SDSI names, or possibly the empty list `()` in case no further delegation is possible.

The latter situation arises when k authorises s for t directly, and the former applies when s receives from k the power to pass an authorisation down the delegation chain. This chaining relation can be accounted for in terms of a rewrite relation $\rightarrow$ on auth certs such that if

$$(k\texttt{,}s\texttt{,}d\texttt{,}t) \rightarrow (k'\texttt{,}s'\texttt{,}d'\texttt{,}t')$$

[3] Tags, in SPKI terminology, not to be confused with tags as used above

then the validity of (k', s', d', t') follows from the validity of (k, s, d, t) (if they have been issued). The single rule governing delegation chaining will be the following:

$$\frac{\vdash k'\texttt{;(EOL)} \preceq s\texttt{;(EOL)} \qquad \vdash \texttt{(delegation }\tau\texttt{;(}s'\texttt{));}d' \preceq d \qquad \vdash t' \preceq t}{(k, s, d, t) \to (k', s', d', t')}$$

where τ is a tag.

Example 5. We give an example based on delegated facility access administration. Two organisations are involved, `orgA` and `orgB` with associated keys K_A and K_B. The task is for `orgA` to grant its administrator the right to engage `orgB` to perform some access management on behalf of `orgA` staff. Assume the following name certs:

$(K_A,\texttt{orgB},K_B)$
$(K_A,\texttt{admin},K_{A,1})$
$(K_A,\texttt{staff},K_{A,2})$
$(K_B,\texttt{staff},K_{B,1})$
$(K_{A,2},\texttt{somebody},K_{A,3})$

Let

$$\begin{aligned} d &= \texttt{(delegation (* path (contractor (name } K_A \texttt{ orgB)));} \\ &\qquad\qquad\qquad\qquad \texttt{((target (name } K_A \texttt{ staff))))} \\ d' &= \texttt{(delegation (* path (contractor (name } K_B \texttt{ staff)));} \\ &\qquad\qquad\qquad\qquad \texttt{((target (name } K_A \texttt{ staff))))} \end{aligned}$$

Assume the initial auth cert

$$(K_A\texttt{,(name } K_A \texttt{ admin),}d\texttt{,(tag access))}$$

Then the following is a valid certificate chain authorising access for $K_{A,3}$:

$(K_{A,1}\texttt{,(name } K_B\texttt{),}d\texttt{,(tag access))}$
$(K_B\texttt{,(name } K_B \texttt{ staff),}d'\texttt{,}t\texttt{)}$
$(K_{B,1}\texttt{,}K_{A,2}\texttt{,(),(tag access))}$

8 Concluding Remarks

We have suggested extending the basic syntax of SPKI with a facility for expressing regular languages, and explored its application in the context of constrained delegation. Constrained delegation is by no means the only conceivable application of such a regular language extension. Another example could be constraints on multidomain routing paths. Also, the extension may open up for more complex authorisation schemes, useful, for instance, in the context of web services orchestration. An example is sequential constraints on authorisation whereby one authorisation (to enter some facility, say) can be made subject to another authorisation having been previously enacted (say, to have been granted some ticket).

The question remains if the extension is too rich, and whether there are other, equally valid ways of achieving the same ends. We doubt, for instance, whether there is much use for nested path expressions. Also, some of the effects which can be obtained with constrained delegation can be obtained equally well by threshold certificates. For instance, for example 5, a similar effect (to avoid `orgB` delegating outside `orgA`) could be achieved by `orgA` initially issuing a threshold cert ensuring that final authorisations can only be applied to `orgA staff`. A systematic investigation of this issue is left for future work.

We proposed also a restricted syntax for which chaining can be decided in polynomial time, as opposed to the exponential worst case running time obtained by a straightforward reduction to regular language containment. Most examples above remain within the restricted fragment, most notably example 5. The algorithm given in the paper does not take name resolution into account. We expect that the techniques of either [CEE+01] or [JR02] can be applied to address the more general problem without substantial problems.

References

[BD02] O. Bandmann and M. Dam. A note on SPKI's authorisation syntax. In *Proc. 1st Annual PKI Research Workshop*, 2002.

[BDF02] O. Bandmann, M. Dam, and B. Sadighi Firozabadi. Constrained delegation. In *Proc. 23rd Annual Symp. on Security and Privacy*, 2002.

[CEE+01] D. Clarke, J.-E. Elien, C. Ellison, M. Fredette, A. Morcos, and R. L. Rivest. Certificate chain discovery in spki/sdsi. *Journal of Computer Security*, 9:285–322, 2001.

[EFL+99] C. M. Ellison, B. Frantz, B. Lampson, R. Rivest, B. M. Thomas, and T. Ylonen. SPKI Certificate Theory, May 1999. RFC 2693, expired. URL: ftp://ftp.isi.edu/in-notes/rfc2693.txt.

[JR02] S. Jha and T. Reps. Analysis of SPKI/SDSI certificates using model checking. In *Proc. IEEE Computer Security Foundations Workshop*, pages 129–146, 2002.

[Koz83] D. Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[Riv97] R. Rivest. S-expressions, May 1997. Internet Draft, expired. URL: http://theory.lcs.mit.edu/ rivest/sexp.txt.

Appendix

Proposition 5.

1. $\Downarrow\texttt{(*)} = \|\texttt{(*)}\|$
2. $\Downarrow\texttt{()} = \texttt{()}$
3. $\Downarrow\texttt{(}\sigma\texttt{)} = \{\texttt{(}s\texttt{)} \mid s \in \downarrow\sigma\}$
4. $\Downarrow\sigma_1\texttt{;}\sigma_2 = \Downarrow\sigma_1\texttt{;}\Downarrow\sigma_2$
5. $\Downarrow\texttt{(* set } \sigma_1 \ \cdots \ \sigma_m\texttt{)} = \Downarrow\sigma_1 \cup \cdots \cup \Downarrow\sigma_m$
6. $\Downarrow\texttt{(* path } \sigma\texttt{)} = (\Downarrow\sigma)^*$
7. $\downarrow\sigma = \Downarrow\sigma\texttt{;}\Sigma^*$

Proof. 1. Trivial
2. $s \in \Downarrow\texttt{()}$ iff $s \leq \texttt{()}$ iff $s \in \Sigma^*$.
3. $s \in \Downarrow\texttt{(}\sigma\texttt{)}$ iff $\exists s_1 \in \|\texttt{(}\sigma\texttt{)}\|$ such that $s \leq s_1$ iff $\exists s_2 . s \leq \texttt{(}s_2\texttt{)}$ and $s_2 \in \|\sigma\|$ iff $\exists s_2', s_3 . s = \texttt{(}s_2'\texttt{)}\texttt{;}s_3$, $s_2' \in \downarrow\sigma$, and $s_3 \in \Sigma^*$ iff $s \in \{\texttt{(}s'\texttt{)} \mid s' \in \downarrow\sigma\}\texttt{;}\Sigma^*$.
4. $s \in \Downarrow(\sigma_1\texttt{;}\sigma_2)$ iff $\exists s_1 \in \|\sigma_1\|, s_2 \in \|\sigma_2\| . s \leq s_1\texttt{;}s_2$ iff $\exists s_1 \in \Downarrow\sigma_1, s_2 \in \Downarrow\sigma_2 . s = s_1\texttt{;}s_2$ iff $s \in \Downarrow\sigma_1\texttt{;}\Downarrow\sigma_2$
5. $s \in \Downarrow\texttt{(* set } \sigma_1 \ \cdots \ \sigma_m\texttt{)}$ iff $s \in \Downarrow\sigma_1 \cup \cdots \cup \Downarrow\sigma_m$
6. $s \in \Downarrow\texttt{(* path } \sigma\texttt{)}$ iff $s = \texttt{(}s_1\texttt{)}\texttt{;}\cdots\texttt{;}\texttt{(}s_n\texttt{)}\texttt{;}s', \forall i : 1 \leq i \leq n . s_i \in \Downarrow\sigma, s' \in \Sigma^*$ iff $s \in (\Downarrow\sigma)^*\texttt{;}\Sigma^*$
7. Immediate by the definition. □

Theorem 4 (Soundness). *If $\Gamma \vdash r_1 \preceq r_2$ then $\Gamma \models r_1 \leq r_2$*

Proof. We prove the statement for both sequents of the form $\Gamma \vdash r_1 \preceq r_2$ and $\Gamma \vdash p_1 \preceq p_2$ by induction on the structure of proof. All cases except IX are routine. We go through the non-trivial cases one by one.

III Suppose Γ and $\Gamma \vdash p_1 \preceq p_2$ are both valid. Let $s \in \downarrow(\texttt{(}a\texttt{)}\texttt{;}p_1)$. By proposition 2, $s = \texttt{(}a \ s_1 \ \cdots \ s_n\texttt{)}$, and $s' = \texttt{(}s_1 \ \cdots \ s_n\texttt{)} \in \downarrow p_1$. By the assumption and fact 1, $s' \in \downarrow p_2$, so $s \in \downarrow\texttt{(}a\texttt{)}\texttt{;}p_2$, as desired.

VI. Suppose that Γ, $\Gamma \vdash r_1 \preceq r_2$ and $\Gamma \vdash p_1 \preceq p_2$ are all valid, and assume that $s \in \downarrow(\texttt{(}r_1\texttt{)}\texttt{;}p_1)$. By proposition 2, s has the shape $\texttt{(}s_1 \ s_2 \ \cdots \ s_n\texttt{)}$ such that $s_1 \in \downarrow r_1$ and $\texttt{(}s_2 \ \cdots \ s_n\texttt{)} \in \downarrow p_1$. But then $s_1 \in \downarrow r_2$ and $\texttt{(}s_2 \ \cdots \ s_n\texttt{)} \in \downarrow p_2$ as well, so $s \in \downarrow(\texttt{(}r_2\texttt{)}\texttt{;}p_2)$ as we wanted.

VII. Suppose that Γ and $\Gamma \vdash p_1 \preceq p_2$ are both valid. Assume also that $s \in \downarrow p_1$. Then $s \in \downarrow(\texttt{(* path } r\texttt{)}\texttt{;}p_2)$ as well, since $\texttt{()} \in \downarrow\texttt{(* path } r\texttt{)}$ for any r and k.

VIII. Assume that Γ and $\Gamma \vdash p_1 \preceq_m \texttt{(}r\texttt{)}\texttt{;}\texttt{(* path } r\texttt{)}\texttt{;}p_2$ are both valid, and that $s \in \downarrow p_1$. Then $s = \texttt{(}s_1 \ \cdots \ s_h \ s_{h+1} \ \cdots \ s_k \ s_{k+1} \ \cdots \ s_l\texttt{)}$ such that $\texttt{(}s_1 \ \cdots \ s_h\texttt{)} \in \downarrow r$, $\texttt{(}s_{h+1} \ \cdots \ s_k\texttt{)} \in \downarrow(\texttt{(* path } r\texttt{)}\texttt{;}p_2)$, and $\texttt{(}s_{k+1} \ \cdots \ s_l\texttt{)} \in p_2$. It follows, by proposition 2, that $\texttt{(}s_1 \ \cdots \ s_h \ s_{h+1} \ \cdots \ s_k\texttt{)} \in \downarrow(\texttt{(* path } r\texttt{)}\texttt{;}p_2)$, so $s \in \texttt{(* path } r\texttt{)}\texttt{;}p_2$ as desired.

IX. This is the only slightly tricky case. We now assume that a proof is given, but that the conclusion of the proof is false. From these assumptions a contradiction is derived. Define

$$C^0 = \texttt{()}$$
$$C^{n+1} = C^n \cup C^n\texttt{;}C$$

Obviously, $C^* = \bigcup_{n\in\omega} C^n$. We use these n's to annotate path expressions in the proof, in this way deriving the contradiction. The annotation uses expressions $\texttt{(* path } r\texttt{)}^n$ which denote $(\downarrow r)^n$.

Assume now that $\Gamma \vdash p_1 \preceq p_2$ and

$$\Gamma, \texttt{(* path } r\texttt{);}p_1 \preceq p_2 \vdash \texttt{(}r\texttt{);(* path } r\texttt{);}p_1 \preceq p_2$$

are all valid, but $\Gamma \vdash \texttt{(* path } r\texttt{);}p_1 \preceq p_2$ is not. Then Γ must valid and $\texttt{(* path } r\texttt{);}p_1 \not\preceq p_2$, i.e. $(\downarrow r)^*;\downarrow p_1 \not\subseteq \downarrow p_2$. Then we find an $n \in \omega$ such that $(\downarrow r)^n;\downarrow p_1 \subseteq \downarrow p_2$ but not $(\downarrow r)^{n+1};\downarrow p_1 \subseteq \downarrow p_2$. The application of IX we are considering is now annotated as follows:

$$\frac{\Gamma \vdash p_1 \preceq p_2 \quad \Gamma, \texttt{(* path } r\texttt{)}^n\texttt{;}p_1 \preceq p_2 \vdash \texttt{(}r\texttt{);(* path } r\texttt{)}^n\texttt{;}p_1 \preceq p_2}{\Gamma \vdash \texttt{(* path } r\texttt{)}^{n+1}\texttt{;}p_1 \preceq p_2}$$

The annotation of the proof is completed simply by letting annotations propagate, using the annotated version of IX in place of IX proper. Now, for the annotated proof, except possibly for instances of I, if the parent (the conclusion) of a rule instance is invalid, then so is one of the children. As a consequence we can trace a path from the invalid proof node $\Gamma \vdash \texttt{(* path } r\texttt{);}p_1 \preceq p_2$ to a leaf, an instance of I, using only invalid sequents. We may assume that there are no further applications of rule IX along this path (otherwise it suffices to consider a proper suffix). It follows that the invalid leaf node must have the shape $\Gamma, \texttt{(* path } r\texttt{)}^n\texttt{;}p_1 \preceq p_2 \vdash \texttt{(* path } r\texttt{)}^n\texttt{;}p_1 \preceq p_2$, but this node is valid, a contradiction. It follows that no proof can lead to a false conclusion, which is what we had to show. □

Theorem 5 (Completeness). *Suppose that r_1, r_2 are well-formed restricted expressions. If $r_1 \leq r_2$ then $r_1 \preceq r_2$*

Proof. We assume that $r_1 \leq r_2$ and give a bottom-up strategy for building a proof of $r_1 \preceq r_2$.

$r_1 = \texttt{()}$. Refine using II.

$r_2 = \texttt{()}$. Any other case than $r_1 = \texttt{()}$ is a contradiction.

$r_1 = \texttt{(}a\texttt{);}p_1$, and $r_2 = \texttt{(}a'\texttt{);}p_2$. The case where $a \neq a'$ is a contradiction. Otherwise we must have $p_1 \leq p_2$, and we refine using III.

$r_1 = \texttt{(}a\texttt{);}p$, $r_2 = \texttt{(* set } r^{a_1} \cdots r^{a_n}\texttt{)}$. We must have $a = a_i$ for exactly one i, and $r_1 \leq r^a$. Refine using IV.

$r_1 = \texttt{(* set } r^{a_1} \cdots r^{a_n}\texttt{)}$, $r_2 = \texttt{(}a\texttt{);}p_2$. This case is not very interesting. We must have $n = 1$, $a_1 = a$ and $r^a \leq r_2$, and we refine according to V.

$r_1 = \texttt{(* set } r_1^{a_{1,1}} \cdots r_1^{a_{1,n_1}}\texttt{)}$, $r_2 = \texttt{(* set } r_2^{a_{2,1}} \cdots r_2^{a_{2,n_2}}\texttt{)}$. In this case let $\alpha_i = \{a_{i,1}, \cdots, a_{i,n_i}\}$, $i \in \{1,2\}$. We must have $\alpha_1 \subseteq \alpha_2$ and for each $a \in \alpha_1$ it will be the case that $r_1^a \leq r_2^a$. Consequently we refine using first V, then IV.

We then proceed by assuming $p_1 \leq p_2$. Consider first the cases where p_1 has the shape $\texttt{(}a\texttt{);}p_1'$, and p_2 has one of the shapes $\texttt{(}r\texttt{);}p_1'$ or $\texttt{(* path } r\texttt{);}p_1'$, or vice versa. These cases are contradictions and so cannot occur. We proceed:

$p_1 = \texttt{(}r_1\texttt{)}\texttt{;}p'_1$, $p_2 = \texttt{(}r_2\texttt{)}\texttt{;}p'_2$. In this case we obtain $r_1 \leq r_2$ and $p'_1 \leq p'_2$, and we refine using VI.

$p_1 = \texttt{(}r_1\texttt{)}\texttt{;}p'_1$, $p_2 = \texttt{(* path }r_2\texttt{)}\texttt{;}p'_2$. Since $p_1 \leq p_2$, either $\mathit{tags}\ (r_1) \cap \mathit{tags}\ (r_2) = \emptyset$ or else $\mathit{tags}\ (r_1) \subseteq \mathit{tags}\ (r_2)$. In the first case we obtain that $p_1 \leq p'_2$ directly, and refine using VII. In the second case note first that we may assume that $\mathit{tags}\ (r_1) \neq \emptyset$ since otherwise the first subcase applies. Assume that $p'_2 = q_1\texttt{;}\cdots\texttt{;}q_n$. If $n = 0$, i.e. $p'_2 = \texttt{()}$, then we obtain directly that $p_1 \leq \texttt{(}r_2\texttt{)}\texttt{;}\texttt{(* path }r_2\texttt{)}\texttt{;}p'_2$ and so refine by VIII. Thus we can assume that $n > 0$. By the properties of Kleene star we know that $\downarrow p_1 \subseteq (\downarrow p'_2) \cup ((\downarrow r_2)\texttt{;}(\downarrow r_2)^*\texttt{;}(\downarrow p'_2))$. By well-formedness we know that $\mathit{tags}\ (r_2) \cap \mathit{tags}\ (q_1) = \emptyset$. Then $\mathit{tags}\ (r_1) \cap \mathit{tags}\ (q_1) = \emptyset$ too, so $\downarrow p_1 \cap \downarrow p'_2 = \emptyset$, whence we may conclude that $p_1 \leq \texttt{(}r_2\texttt{)}\texttt{;}\texttt{(* path }r_2\texttt{)}\texttt{;}p'_2$ and refine by VIII.

$p_1 = \texttt{(* path }r_1\texttt{)}\texttt{;}p'_1$ and either $p_2 = \texttt{(}r_2\texttt{)}\texttt{;}p'_2$ or $p_2 = \texttt{(* path }r_2\texttt{)}\texttt{;}p'_2$. If we find that $p_1 \preceq p_2$ is in the current set Γ proof construction terminates. Otherwise, by $p_1 \leq p_2$ we obtain directly that $p'_1 \leq p_2$ and that $\texttt{(}r_1\texttt{)}\texttt{;}\texttt{(* path }r_1\texttt{)}\texttt{;}p'_1 \leq p_2$, and so refine by IX.

We have thus defined a procedure which constructs a proof from a valid sequent. Our task is to show that the procedure terminates. This is not difficult to see. Let a proof structure (i.e. a tree, possibly infinite, rooted in a sequent, say, $\Gamma \vdash r_1 \preceq r_2$, and constructed according to the proof rules) be given. Let N be the set of all expressions r and p appearing somewhere in this proof structure. This set will be finite. Now, if the procedure fails to terminate it will be possible to trace a path through the (infinite) proof structure starting from the root which infinitely often visits a sequent $\Gamma \vdash p_1 \preceq p_2$ for fixed choices of p_1 and p_2. We can assume that p_1 has the shape $p_1 = \texttt{(* path }r_1\texttt{)}\texttt{;}p'_1$, and that in each case the next rule applied is rule IX. But then, already at the second occurrence of this sequent, I will be applicable, a contradiction. $\square$