

Understanding Trust Management Systems

Stephen Weeks
Strategic Technologies and Architectural Research Laboratory
InterTrust Technologies Corporation
4750 Patrick Henry
Santa Clara, CA 95054, USA
sweeks@intertrust.com

Abstract

This paper presents a mathematical framework for expressing trust management systems. The framework makes it easier to understand existing systems and to compare them to one another, as well as to design new systems. The framework defines the semantics of a trust management engine via a least fixpoint in a lattice, which, in some situations, leads to an efficient implementation. To demonstrate its flexibility, we present KeyNote and SPKI as instantiations of the framework.

1 Introduction

Systems in which multiple entities share resources often use an access control mechanism. The problem of access control can be broken into two subproblems: determining whether or not a request should be allowed, and enforcing the decision. *Trust management systems* solve the first subproblem by defining languages for expressing authorizations and access control policies, and by providing a *trust management engine* for determining when a particular request is authorized. Traditional access control mechanisms are centralized and operate under a closed world assumption in which all of the parties are known. Trust management systems generalize traditional mechanisms by operating in distributed systems and eliminating the closed world assumption. Over the last ten years, a number of trust management systems have been developed, some focusing on authentication [20, 21, 22], others for specialized purposes [3, 8, 18], others for general purpose authorization [4, 6, 13], and others based on logics [1, 2, 17]. Because of the wide range in precision in the specification of these systems and the wide variety of trust management languages, it is difficult to compare the systems in order to intelligently decide which to apply to a new situation. Because of the lack of formality in many of the specifications,

it is difficult to understand their weaknesses, which is especially troubling since the domain of interest is often security related. Finally, because there is no common conceptual framework underlying the systems, it is difficult to reason about the tradeoffs made in their design. This makes the design of new trust management systems more of an art than a science.

This paper presents a mathematical framework for expressing trust management systems. The framework can express many well-known systems, including KeyNote [4] and SPKI [13]. Trust management systems can be concisely specified, which helps in comparing current systems and in analyzing the design tradeoffs of new systems. The framework defines the semantics of a trust management engine as a least fixpoint in a lattice, which for many instantiations directly leads to an implementation. In the case of SPKI, the paper shows how an implementation of the semantics can make a trust management decision more simply and efficiently than the combination of certificate path discovery [9, 11] and tuple reduction [13].

This paper is not concerned with the cryptographic underpinnings of trust management. It assumes that the appropriate integrity checking and signature validation happens before the trust management engine begins computation. It also does not discuss the important issue of revocation. Finally, it does not present a proposal for a new trust management system, although it lays the foundation for exploration of a range of new systems.

Section 2 defines the framework and presents an instantiation via an extended example of a toy trust management system. Sections 3 – 5 show how to express SPKI, Keynote, and logic-based systems as instantiations of the framework. Section 6 formalizes several concepts generally applicable to trust management systems. Section 7 concludes and presents several unanswered questions that this work raises. Appendix A reviews mathematical background and defines notation. Readers unfamiliar with lambda expressions or lattices should refer to the appendix as necessary.

2 Framework

This section defines the framework for expressing trust management systems and presents an extended example. The idea behind the framework is to leave open the kinds of authorizations that can be made by the system, only requiring that they satisfy a few natural mathematical properties. The framework defines the kinds of assertions that can be made by entities in the system and gives a precise semantics specifying what a collection of assertions means and when a request should be granted. In many cases, a direct implementation of this semantics leads to a viable trust management engine.

The elements of the framework are principals, authorizations, authorization maps (abbreviated authmaps), licenses, and assertions. A *principal* is an atomic entity that may make or authorize requests. We use p to range over principals.

$$p \in \text{Principal}$$

For the framework, the only necessary property of principals is that they are distinguishable. In a real implementation, for cryptographic reasons, a principal might correspond to a public key. In the example of this section, we will deal with the principals Alice, Bob, Carl, and Dave.

An *authorization* expresses the permissions granted by a principal. Authorizations form a lattice, $Auth$, where $u \sqsubseteq u'$ means that u' permits more operations than u .

$$u \in \text{Auth}$$

For a given trust management decision, each principal in the system will be associated with a single authorization that describes the permissions granted by that principal. The least upper bound operator \sqcup defines how to sensibly combine multiple authorizations made by the same principal into a single authorization. The $Auth$ lattice can be instantiated in varying ways to express different trust management systems.

As an example, imagine that $Auth$ specifies whether or not Alice may read some particular file, write that file, or both. Let R denote that Alice may read the file, W denote she may write the file, RW denote she may read and write the file, and N denote she may do neither. So, $Auth = \{N, R, W, RW\}$. In order to make $Auth$ into a lattice, define $N \sqsubseteq R$, $N \sqsubseteq W$, $R \sqsubseteq RW$, $W \sqsubseteq RW$ and $\sqcup\{R, W\} = RW$. Note that this authorization lattice only specifies the authorizations granted to Alice for a particular file, not for any other principal or any other file. To represent such authorizations, we could use a more complicated lattice like $\text{Principal} \times \text{File} \longrightarrow \{N, R, W, RW\}$, but we will stick with the simple lattice as the example for this section.

An *authmap* is a function mapping principals to authorizations that describes the authorizations made by each

principal in a trust management decision.

$$m \in \text{AuthMap} = \text{Principal} \longrightarrow \text{Auth}$$

Recall from Appendix A that $AuthMap$ is a lattice under the pointwise ordering because $Auth$ is a lattice. For our example lattice, an authmap expresses the authorization each principal grants to Alice to read or write the file. Such a map might be m , where $m(\text{Bob}) = R$ and $m(\text{Carl}) = RW$.

A *license* grants authorization, expressed as a monotone function from authmaps to authorizations.

$$l \in \text{License} = \text{AuthMap} \longrightarrow_m \text{Auth}$$

Intuitively, the meaning of license l is that if principals grant authorizations as given by authmap m , then l grants the authorization $l(m)$. Figure 1 shows some licenses for the example lattice. License 2, which delegates to Bob, shows how the dependence of licenses on authmaps expresses delegation. The monotonicity requirement means that the authorizations granted by a license can only increase as a consequence of an increase in other principals' authorizations. The reader should verify that all of the licenses in Figure 1 are monotone. In particular, in licenses 6, 7, and 8, note that $\text{if} =$ were used instead of \sqsubseteq , the license would not be monotone. Monotonicity is required in order to give a sensible meaning to a collection of assertions so that the trust management engine will be well-defined.

An *assertion* is an expression of authorization made by a principal. Assertions are the framework's abstraction of digital certificates. Formally, an assertion consists of a principal and a license.

$$a \in \text{Assertion} = \text{Principal} \times \text{License}$$

Assertion $\langle p, l \rangle$ should be read as “ p authorizes l ”. Principal p is referred to as the *issuer*. Any principal could issue an assertion with any of the licenses from Figure 1. In a real implementation, assertions would typically be signed by the public key of the issuing principal and would require signature checking before being processed by the trust management engine.

A trust management engine must take a set of assertions made by various principals, some of whom may delegate to each other, and find a coherent authmap representing the authorizations of those principals. The assertions may reside in the local machine, or may be presented by the principal making the request, and may have been created in various places. The act of finding a coherent authmap from various assertions is the component that distinguishes trust management from traditional access control mechanisms. We define the semantics of a set of assertions as follows (as we do throughout the document, we use \mathcal{M} with a subscript to denote a semantic function).

$$\begin{aligned} \mathcal{M}_{\text{Assertions}} &: \mathcal{P}(\text{Assertion}) \longrightarrow_m \text{AuthMap} \\ \mathcal{M}_{\text{Assertions}}(A) &= \text{lfp}(\lambda m. \lambda p. \sqcup \{l(m) \mid \langle p, l \rangle \in A\}) \end{aligned}$$

$l \in License$	intended meaning
1) $\lambda m. W$	Alice may write the file.
2) $\lambda m. m(\mathbf{Bob})$	Alice may do whatever Bob allows.
3) $\lambda m. \sqcup\{W, m(\mathbf{Bob})\}$	Alice may write and do anything else Bob allows.
4) $\lambda m. \prod\{W, m(\mathbf{Bob})\}$	Alice may write if Bob allows her to write.
5) $\lambda m. \prod\{W, m(\mathbf{Bob}), m(\mathbf{Carl})\}$	Alice may write if both Bob and Carl say she can.
6) $\lambda m. \text{if } card\{p \in \{\mathbf{Bob}, \mathbf{Carl}, \mathbf{Dave}\} \mid R \sqsubseteq m(p)\} \geq 2$ then R else N	Alice may read if any two of Bob , Carl , and Dave say she can.
7) $\lambda m. \text{if } W \sqsubseteq m(\mathbf{Bob})$ then R else N	Alice may read if Bob says she may write.
8) $\lambda m. \text{if } card\{p \mid R \sqsubseteq m(p)\} \geq 2$ then R else N	Alice may read if any two principals say she can.

Figure 1. Example licenses

The intuition behind the definition of $\mathcal{M}_{\text{Assertions}}$ is that it combines all of the assertions issued by each principal into one authorization for that principal, taking delegation into account. In more detail, the set of assertions made by a principal p is $\{l \mid \langle p, l \rangle \in A\}$. Given the authorization map m , the set of authorizations granted by p is $\{l(m) \mid \langle p, l \rangle \in A\}$. By taking a least upper bound, we can combine the authorizations into a single authorization granted by p , namely $\sqcup\{l(m) \mid \langle p, l \rangle \in A\}$. Finally, to find an authorization map that is consistent with all of the licenses, we take a least fixpoint in the *AuthMap* lattice, which relies on the fact that licenses are monotone. Because the licenses in assertions are monotone, so is $\mathcal{M}_{\text{Assertions}}$. That is, if more assertions are input to $\mathcal{M}_{\text{Assertions}}$, then more authorizations will be made by the resulting authmap. The definition of $\mathcal{M}_{\text{Assertions}}$ as a least fixpoint also makes it clear why the framework cannot handle revocation, since that would require non-monotone assertions.

Figure 2 shows some example computations of $\mathcal{M}_{\text{Assertions}}$. Each row has a set of assertions in the left column, a fixpoint computation of an authmap in the middle, and a comment about the example in the right column. The order of assertions in the left column is irrelevant. The fixpoint computation (see Appendix A for an explanation) is shown as a sequence of authmaps, one per row, where each column contains the authorizations made by a principal. Each row contains an authmap consisting of the least upper bound of all of the assertions applied to the authmap in the previous row. The final row is the least fixpoint authmap of the set of assertions. The examples give some idea of how least

fixpoint computation can express concepts like path validation [20], chaining and five-tuple reduction [13], and inter-assertion communication [7].

A trust management engine makes a decision based on a request, an authorizing principal, and a set of assertions. The authorizing principal reflects the policy governing the request. A request is expressed as an element of the *Auth* lattice. The semantics is defined by $\mathcal{M}_{\text{Engine}}$, where $\mathcal{M}_{\text{Engine}}(p, u, A)$ means that principal p authorizes request u according to the assertions in A .

$$\begin{aligned} \mathcal{M}_{\text{Engine}} &: \text{Principal} \times \text{Auth} \times \mathcal{P}(\text{Assertion}) \longrightarrow \text{Bool} \\ \mathcal{M}_{\text{Engine}}(p, u, A) &= u \sqsubseteq \mathcal{M}_{\text{Assertions}}(A)(p) \end{aligned}$$

The trust engine computes the authmap corresponding to the provided assertions A and determines if the request requires less authorization than is granted by the authorizing principal p . For the example lattice, if Alice would like to write the file (which is denoted by the request W) under **Bob**'s control and the assertions a_1 , a_2 and a_3 are available, the trust engine would compute $\mathcal{M}_{\text{Engine}}(\mathbf{Bob}, W, \{a_1, a_2, a_3\})$ to determine if **Bob** authorizes Alice to write.

The trust management engine does not necessarily need to compute the entire fixpoint as defined by $\mathcal{M}_{\text{Assertions}}$. To prove that a request is justified, all the engine must find is an authmap m such that $u \sqsubseteq m \sqsubseteq \mathcal{M}_{\text{Assertions}}(A)(p)$. Finding such an m may be easier than computing the fixpoint for two reasons. First, the trust engine may be able to work in a lattice in which elements are more compactly represented and licenses are more efficiently computed. Second, the

$A \in \mathcal{P}(\text{Assertion})$	$lfp(A)$ computation			Comment
	Bob	Carl	Dave	
$\langle \text{Bob}, \lambda m. W \rangle$	N	N	N	A direct authorization.
	W	N	N	
$\langle \text{Bob}, \lambda m. W \rangle$	N	N	N	Auths combined using \sqcup .
$\langle \text{Bob}, \lambda m. R \rangle$	RW	N	N	
$\langle \text{Bob}, \lambda m. W \rangle$	N	N	N	Unconstrained delegation.
$\langle \text{Carl}, \lambda m. m(\text{Bob}) \rangle$	W	N	N	
	W	W	N	
$\langle \text{Bob}, \lambda m. m(\text{Carl}) \rangle$	N	N	N	Least fixpoint.
$\langle \text{Carl}, \lambda m. m(\text{Bob}) \rangle$				
$\langle \text{Bob}, \lambda m. W \rangle$	N	N	N	Constrained delegation.
$\langle \text{Carl}, \lambda m. \sqcap\{R, m(\text{Bob})\} \rangle$	W	N	N	
$\langle \text{Bob}, \lambda m. RW \rangle$	N	N	N	Constrained delegation.
$\langle \text{Carl}, \lambda m. \sqcap\{R, m(\text{Bob})\} \rangle$	RW	N	N	
	RW	R	N	
$\langle \text{Bob}, \lambda m. W \rangle$	N	N	N	Chained delegation.
$\langle \text{Carl}, \lambda m. m(\text{Bob}) \rangle$	W	N	N	
$\langle \text{Dave}, \lambda m. m(\text{Carl}) \rangle$	W	W	N	
	W	W	W	
$\langle \text{Bob}, \lambda m. W \rangle$	N	N	N	Multiway delegation.
$\langle \text{Carl}, \lambda m. RW \rangle$	W	RW	N	
$\langle \text{Dave}, \lambda m. \sqcap\{m(\text{Bob}), m(\text{Carl})\} \rangle$	W	RW	W	
$\langle \text{Bob}, \lambda m. W \rangle$	N	N	N	Inter-assertion communication.
$\langle \text{Bob}, \lambda m. m(\text{Carl}) \rangle$	W	N	N	
$\langle \text{Carl}, \lambda m. \text{if } W \sqsubseteq m(\text{Bob}) \text{ then } R \text{ else } N \rangle$	W	R	N	
	RW	R	N	

Figure 2. Example least fixpoint computations of $\mathcal{M}_{\text{Assertions}}$

$p \in \text{Principal}$
 $u \in \text{Auth}$
 $m \in \text{AuthMap} = \text{Principal} \longrightarrow \text{Auth}$
 $l \in \text{License} = \text{AuthMap} \longrightarrow_m \text{Auth}$
 $a \in \text{Assertion} = \text{Principal} \times \text{License}$

$\mathcal{M}_{\text{Assertions}} : \mathcal{P}(\text{Assertion}) \longrightarrow_m \text{AuthMap}$
 $\mathcal{M}_{\text{Assertions}}(A) = lfp(\lambda m. \lambda p. \sqcup\{l(m) \mid \langle p, l \rangle \in A\})$

$\mathcal{M}_{\text{Engine}} : \text{Principal} \times \text{Auth} \times \mathcal{P}(\text{Assertion}) \longrightarrow \text{Bool}$
 $\mathcal{M}_{\text{Engine}}(p, u, A) = u \sqsubseteq \mathcal{M}_{\text{Assertions}}(A)(p)$

Figure 3. Framework

engine may terminate the fixpoint computation early, once a large enough m has been found. The sections on KeyNote and SPKI will take advantage of this fact.

Figure 3 summarizes the entire framework. An instantiation of the framework defines a trust management system by providing *Auth* lattice, a language for expressing licenses, and a means of computing \mathcal{M}_{Engine} . The art in designing a trust management system lies in choosing a lattice of authorizations and a subset of the monotone functions to be used as licenses. This controls the expressiveness of the system. The engineering comes in choosing a representation for authorizations and a language for licenses so that the desired monotone functions can be expressed concisely and so that \mathcal{M}_{Engine} can be computed efficiently. The rest of this section is devoted to a toy language for expressing licenses in the example lattice of this section. The following sections will show how to express more realistic trust management systems.

2.1 A toy trust management language

Using all of mathematics to write licenses is fine for expository purposes, but in order to build a practical trust management engine, one must define a language for expressing licenses and give a way for the trust management engine to compute the authorization expressed by a license. Here is an example language of expressions for *Auth* = {N, R, W, RW}.

$$\begin{array}{l}
 e ::= \text{N} \mid \text{R} \mid \text{W} \mid \text{RW} \\
 \quad \mid p \\
 \quad \mid (\text{glb } e \dots) \\
 \quad \mid (\text{lub } e \dots)
 \end{array}$$

A license expression is either a constant denoting the corresponding lattice element, or a principal (in unspecified format) denoting delegation to a principal, or the greatest lower bound (glb), or the least upper bound (lub) of a sequence of other expressions. Let *ExampleLicense* be the set of expressions generated by the above grammar. We can give a semantics to expressions by defining a function \mathcal{M}_{EL} inductively on the structure of expressions.

$$\mathcal{M}_{EL} : \text{ExampleLicense} \longrightarrow \text{License}$$

$$\begin{array}{l}
 \mathcal{M}_{EL}(\text{N}) = \lambda m. \text{N} \\
 \mathcal{M}_{EL}(\text{R}) = \lambda m. \text{R} \\
 \mathcal{M}_{EL}(\text{W}) = \lambda m. \text{W} \\
 \mathcal{M}_{EL}(\text{RW}) = \lambda m. \text{RW} \\
 \mathcal{M}_{EL}(p) = \lambda m. m(p) \\
 \mathcal{M}_{EL}((\text{glb } e \dots)) = \lambda m. \prod \{ \mathcal{M}_{EL}(e)(m), \dots \} \\
 \mathcal{M}_{EL}((\text{lub } e \dots)) = \lambda m. \sqcup \{ \mathcal{M}_{EL}(e)(m), \dots \}
 \end{array}$$

Observe that \mathcal{M}_{EL} always produces a monotone function. Given a representation of *AuthMap* in which the authorization of a principal can be found in constant time, it is clear

that for any expression e , we can compute $\mathcal{M}_{EL}(e)(m)$ in time linear in the size of the expression. Hence, we can compute $\mathcal{M}_{Assertions}(A)$ and \mathcal{M}_{Engine} in time proportional to the sum of the sizes of the licenses in A .

Using *ExampleLicense*, we can write expressions denoting the first six examples in Figure 1.

- 1) W
- 2) Bob
- 3) (lub W Bob)
- 4) (glb W Bob)
- 5) (glb W Bob Carl)
- 6) (glb R (lub (glb Bob Carl) (glb Carl Dave) (glb Bob Dave)))

The last expression shows that this language may be verbose in expressing some monotone functions. Even worse, the last two examples in Figure 1 can not be expressed at all in this language. Of course, by adding more constructs, one could express those examples as well, but at the cost of additional complexity in the trust management engine and possibly additional time taken in computing \mathcal{M}_{EL} . The last example would require an existential quantification operator, similar to that of [17, 18].

3 Simple Public Key Infrastructure (SPKI)

Figure 4 shows how to express SPKI [12, 13, 14], including SDSI [19] local names, in the framework of Section 2. The upper part of the figure defines the lattice of SPKI authorizations and the representation of SPKI assertions. The lower part gives the semantics of SPKI assertions, by showing how to map them to assertions in the sense of Section 2. The figure should be read along with Figure 3, which defines the supporting infrastructure of the framework.

The SPKI *Auth* lattice is a powerset lattice with two kinds of authorizations, *name* and *action*. For authmap m , if name authorization $\langle p', n, t \rangle$ is in $m(p)$, then principal p authorizes principal p' to act as name n at time t . Similarly, if action authorization $\langle p', y, t \rangle$ is in $m(p)$ then principal p authorizes principal p' to perform operation y at time t . Elements of *Name* are arbitrary byte strings. Elements of *Sexp* are s-expressions, and denote operations. For this paper, we will leave *Time* unspecified, other than to note that it is totally ordered by \leq .

The meaning of a name is always taken relative to a given principal. A *full name* is a principal and a sequence of names, where $\langle p, t \rangle \in \mathcal{M}_{FullP}(f, m)$ means that principal p can act on behalf of full name f at time t . \mathcal{M}_{FullP} is extended to subjects by \mathcal{M}_{SubjP} and to name assertions by \mathcal{M}_{Name} . A name assertion is represented by a 4-tuple $\langle p, n, s, \langle t_1, t_2 \rangle \rangle$, which means that principal p authorizes subject s to act on

$n \in \text{Name}$
 $y \in \text{Sexp}$
 $t \in \text{Time}$
 $u \in \text{Auth} = \mathcal{P}(\text{Principal} \times (\text{Name} + \text{Sexp}) \times \text{Time})$
 $f \in \text{FullName} = \text{Principal} \times \text{Name}^*$
 $s \in \text{Subject} = \text{FullName} + (\text{Int} \times \mathcal{P}(\text{FullName}))$
 $d \in \text{Delegate} = \text{Bool}$
 $x \in \text{Action}$
 $\text{TimePeriod} = \text{Time} \times \text{Time}$
 $\text{NameAssertion} = \text{Principal} \times \text{Name} \times \text{Subject} \times \text{TimePeriod}$
 $\text{AuthAssertion} = \text{Principal} \times \text{Subject} \times \text{Delegate} \times \text{Action} \times \text{TimePeriod}$
 $\text{SPKIAssertion} = \text{NameAssertion} + \text{AuthAssertion}$

$\mathcal{M}_{\text{Action}} : \text{Action} \longrightarrow \mathcal{P}(\text{Sexp})$ (omitted)

$\mathcal{M}_{\text{FullP}} : \text{FullName} \times \text{AuthMap} \longrightarrow \mathcal{P}(\text{Principal} \times \text{Time})$

$\mathcal{M}_{\text{FullP}}(\langle p, [] \rangle, m) = \{ \langle p, t \rangle \mid t \in \text{Time} \}$

$\mathcal{M}_{\text{FullP}}(\langle p, [n_0, n_1, \dots] \rangle, m) = \left\{ \langle p', t \rangle \mid \exists p''. \begin{array}{l} \langle p'', n_0, t \rangle \in m(p) \text{ and} \\ \langle p', t \rangle \in \mathcal{M}_{\text{FullP}}(\langle p'', [n_1, \dots] \rangle, m) \end{array} \right\}$

$\mathcal{M}_{\text{SubjP}} : \text{Subject} \times \text{AuthMap} \longrightarrow \mathcal{P}(\text{Principal} \times \text{Time})$

$\mathcal{M}_{\text{SubjP}}(f, m) = \mathcal{M}_{\text{FullP}}(f, m)$

$\mathcal{M}_{\text{SubjP}}(\langle k, F \rangle, m) = \{ \langle p, t \rangle \mid k \leq \text{card} \{ f \in F \mid \langle p, t \rangle \in \mathcal{M}_{\text{FullP}}(f, m) \} \}$

$\mathcal{M}_{\text{Name}} : \text{NameAssertion} \longrightarrow \text{Assertion}$

$\mathcal{M}_{\text{Name}}(p, n, s, \langle t_1, t_2 \rangle) = \langle p, \lambda m. \{ \langle p', n, t \rangle \mid t_1 \leq t \leq t_2 \text{ and } \langle p', t \rangle \in \mathcal{M}_{\text{SubjP}}(s, m) \} \rangle$

$\mathcal{M}_{\text{FullA}} : \text{FullName} \times \text{AuthMap} \longrightarrow \text{Auth}$

$\mathcal{M}_{\text{FullA}}(f, m) = \{ \langle p, y, t \rangle \mid \exists p'. \langle p', t \rangle \in \mathcal{M}_{\text{FullP}}(f, m) \text{ and } \langle p, y, t \rangle \in m(p') \}$

$\mathcal{M}_{\text{SubjA}} : \text{Subject} \times \text{AuthMap} \longrightarrow \text{Auth}$

$\mathcal{M}_{\text{SubjA}}(f, m) = \mathcal{M}_{\text{FullA}}(f, m)$

$\mathcal{M}_{\text{SubjA}}(\langle k, F \rangle, m) = \{ \langle p, y, t \rangle \mid k \leq \text{card} \{ f \in F \mid \langle p, y, t \rangle \in \mathcal{M}_{\text{FullA}}(f, m) \} \}$

$\mathcal{M}_{\text{Auth}} : \text{AuthAssertion} \longrightarrow \text{Assertion}$

$\mathcal{M}_{\text{Auth}}(p, s, d, x, \langle t_1, t_2 \rangle) = \langle p, l \rangle$, where

$l(m) = \left\{ \langle p', y, t \rangle \mid \begin{array}{l} y \in \mathcal{M}_{\text{Action}}(x) \text{ and } t_1 \leq t \leq t_2 \\ \text{and if } d \text{ then } \langle p', y, t \rangle \in \mathcal{M}_{\text{SubjA}}(s, m) \text{ else } \langle p', t \rangle \in \mathcal{M}_{\text{SubjP}}(s, m) \end{array} \right\}$

Figure 4. SPKI instantiation

behalf of name n at any time between t_1 and t_2 . The subject field can either directly identify a full name, or specify a threshold subject of the form $\langle k, \{f_1, f_2, \dots\} \rangle$. A principal can only act for a threshold subject if at least k of the names f_1, f_2, \dots denote that principal.

An authorization assertion is represented by a 5-tuple $\langle p, s, d, x, \langle t_1, t_2 \rangle \rangle$, which means that principal p authorizes subject s to perform (and delegate, if $d = \text{true}$) the operations allowed by action x at any time between t_1 and t_2 . We will not specify actions in more detail – abstractly, they denote sets of s-expressions, so we can assume a function \mathcal{M}_{Action} that gives their meaning. The semantics of authorization assertions is specified by \mathcal{M}_{Auth} , which relies on \mathcal{M}_{SubjA} to define the authorizations granted by a subject. The definition of \mathcal{M}_{Auth} formalizes the intuition above by requiring the action x to include the requested operation y and the time period to contain the request time t . If the delegate flag is set, then \mathcal{M}_{Auth} allows requests authorized by the subject; otherwise, it requires the requestor p' to be able to act on behalf of the subject.

Everything is now in place to see how a trust management computation is carried out. Suppose we want to know if principal p authorizes principal p' to perform the operation denoted by y at time t according to the assertions A . We express the request as $u = \langle p', y, t \rangle \in Auth$. We want to compute $\mathcal{M}_{Engine}(p, u, A)$, which is equivalent to $u \subseteq \mathcal{M}_{Assertions}(A)(p)$. In order to avoid computing the entire least fixpoint, we can specialize the assertions for the given request. Define the restriction of an authorization u' as follows.

$$\mathcal{R}(u') = u' \cap (u \cup \{ \langle p'', n, t \rangle \mid p'' \in Principal, n \in Name \})$$

Thus the restriction of an authorization includes no more than current request u and name authorizations at the current time. Extend \mathcal{R} to sets of assertions so that the licenses only produce restricted authorizations as follows.

$$\mathcal{R}(A) = \{ \langle p, \lambda m. \mathcal{R}(l(m)) \rangle \mid \langle p, l \rangle \in A \}$$

We can prove that $u \subseteq \mathcal{M}_{Assertions}(A)(p)$ if and only if $u \subseteq \mathcal{M}_{Assertions}(\mathcal{R}(A))(p)$. We can also observe in Figure 4 that name assertions affect the meaning of authorization assertions, but not vice-versa. The implication of these two facts is that instead of computing a least fixpoint over the *AuthMap* lattice, we can first compute a fixpoint over the lattice $Principal \rightarrow \mathcal{P}(Principal \times Name)$ and then over the lattice $Principal \rightarrow Bool$. The first fixpoint computes for each principal p the pairs $\langle p', n \rangle$ such that p authorizes p' to act on behalf of name n at time t . The second fixpoint computes for each principal p whether or not the request u is authorized by p . If there are c assertions, the first fixpoint requires $O(c^2)$ space and the second requires only $O(c)$ space. Xavier Serret-Avila and I have built an

engine for the second fixpoint using the approach described here. In our experience, the time for trust management computation is dwarfed by the cryptographic costs of signature checking.

3.1 Related work

The SPKI standard [13] presents 4-tuple and 5-tuple reduction as a means of implementing \mathcal{M}_{Engine} . 5-tuple reduction is weaker than the least fixpoint approach described above because it relies on action intersection, which takes two actions x_1 and x_2 and attempts to compute an action x such that $\mathcal{M}_{Action}(x) = \mathcal{M}_{Action}(x_1) \cap \mathcal{M}_{Action}(x_2)$. Unfortunately, this is not always possible, since the action language is not expressive enough. As was observed in [15], it is not always possible to represent the the intersection of a **range* and **prefix* action. One way to fix this problem would be to add an intersection operator to the action language. In [15], they instead show that one can always find an action satisfying the weaker condition $\mathcal{M}_{Action}(x) \subseteq \mathcal{M}_{Action}(x_1) \cap \mathcal{M}_{Action}(x_2)$. This guarantees that if their trust management engine returns *true* then so does \mathcal{M}_{Engine} , but not the converse.

A more important weakness of tuple reduction is that it only works on ordered lists of assertions, not unordered sets. To find the right order, the client must perform certificate path discovery [9, 11], something that is at least as complicated as reduction itself, and requires $O(c^3)$ space. The least fixpoint computation above automatically and efficiently performs certificate path discovery and subsumes the reductions necessary to make the trust management decision.

Another approach to specifying SPKI is presented in [16], which gives a simple Prolog program in which the rules correspond to tuple reduction. For a given trust management decision, each assertion is translated to a fact and added to the logic program. The least fixpoint semantics of Prolog then includes all authorizations derivable by combining the facts using the rules, which is roughly equivalent to all tuples derivable using tuple reduction. With a Prolog implementation that uses an appropriate search strategy, the program can be run directly as an implementation of the trust management engine.

4 KeyNote

Figure 5 shows how to express KeyNote [4, 5] in the framework of Section 2. The figure should be read along with Figure 3, which defines the supporting infrastructure of the framework. The KeyNote authorizations *Auth* form a function lattice, where an authorization maps a *request* to a *compliance value*. Compliance values are

$$\begin{aligned}
v &\in \text{Value} \\
x &\in \text{Action} &= \mathcal{P}(\text{String} \times \text{String}) \\
&\text{Request} &= \mathcal{P}(\text{Principal}) \times \text{Action} \\
u &\in \text{Auth} &= \text{Request} \longrightarrow \text{Value} \\
z &\in \text{Licensees} \\
c &\in \text{Conditions} \\
\text{KeyNoteAssertion} &= \text{Principal} \times \text{Licensees} \times \text{Conditions}
\end{aligned}$$

$$\mathcal{M}_{\text{Licensees}} : \text{Licensees} \times (\text{Principal} \longrightarrow \text{Value}) \longrightarrow_m \text{Value} \quad (\text{omitted})$$

$$\mathcal{M}_{\text{Conditions}} : \text{Conditions} \times \text{Request} \longrightarrow \text{Value} \quad (\text{omitted})$$

$$\mathcal{M}_{\text{Keynote}} : \text{KeyNoteAssertion} \longrightarrow \text{Assertion}$$

$$\mathcal{M}_{\text{Keynote}}(p, z, c) = \langle p, \lambda m. \lambda \langle P, x \rangle. \sqcap \{ \mathcal{M}_{\text{Conditions}}(c, x), \mathcal{M}_{\text{Licensees}}(z, \lambda p. m(p)(P, x)) \} \rangle$$

Figure 5. KeyNote instantiation

strings, totally ordered, intended to denote levels of authorization. To keep a simple example in mind, imagine that $\text{Value} = [\text{false}, \text{true}]$, where $\text{false} \sqsubseteq \text{true}$. The partial order on Auth is derived from the order on Value viewed as a lattice. A request $\langle P, x \rangle$ consists of a set of requesting principals P (called `ACTION_AUTHORIZERS` in [4]) and an action, which is represented as a list of pairs of arbitrary strings. The action describes the request; for example, an action to delete a file might be $\{\langle \text{operation}, \text{delete} \rangle, \langle \text{file}, / \text{tmp} / \text{foo} \rangle\}$.

A KeyNote assertion $\langle p, z, c \rangle$ means that issuing principal p authorizes the requests specified by the conditions c , possibly delegating to licensees z . The licensees language is similar to the toy language of Section 2. It has operators for greatest and least upper bound of the compliance values in the Value order, as well as an operation for finding the k -th largest of a set of compliance values. The semantics is summarized by a function $\mathcal{M}_{\text{Licensees}}$, which takes a licensees expression and a map giving the a compliance value for each principal, and gives the value of the expression. The conditions language has floating point, integer, string, and relational operators for inspecting the request and computing a compliance value. The semantics is summarized by $\mathcal{M}_{\text{Conditions}}$, which takes a conditions expression and the request, and computes a compliance value. The semantics of KeyNote assertions is given by $\mathcal{M}_{\text{Keynote}}$, which says that the license corresponding to a KeyNote assertion returns the greatest lower bound (in the Value order) of the meaning of the conditions and licensees fields. Delegation is possible via the licensees field, which is only allowed to query the authmap to find other principals' authorizations of the same request $\langle P, x \rangle$. The license field in the assertion is monotone because $\mathcal{M}_{\text{Licensees}}$ is monotone in its second argument.

A KeyNote trust management engine is responsible for computing a compliance value given a set of assertions

and a request. Suppose that the principals P present to principal p the assertions A to prove that the action x is justified, i.e. has a compliance value of at least v . We can express the request as the (partial) function $[\langle P, x \rangle \mapsto v]$. Then, the trust management engine must decide if $\mathcal{M}_{\text{Engine}}(p, [\langle P, x \rangle \mapsto v], A)$, which is equivalent to $[\langle P, x \rangle \mapsto v] \sqsubseteq \mathcal{M}_{\text{Assertions}}(A)(p)$, which is equivalent to $v \sqsubseteq \mathcal{M}_{\text{Assertions}}(A)(p)(P, x)$. Thus, the trust management engine need only compute a fixpoint in the lattice $\text{Principal} \longrightarrow \text{Value}$, recording for each principal its compliance value on the given request.

5 Logic-based approaches

In the logic-based approach to trust management, the trust management engine is responsible for constructing [1, 3, 17] or checking [2] a proof that the desired request is valid. The logic is defined by a set of formulas, Formula , and a binary relation $F \vdash f$, which should be read “formula f can be proved given the formulas F as axioms”. The logic contains a primitive constructor to express authorizations of principals: if p is a principal and f is a formula, then p **says** f means that principal p authorizes whatever formula f does. Finally, the logic contains (among other rules) inference rules that ensure that each principal is logically consistent.

- If $F \vdash f$ then $F \vdash p$ **says** f .
- If $F \vdash p$ **says** $f \Rightarrow f'$ and $F \vdash p$ **says** f then $F \vdash p$ **says** f'

An assertion is represented as a (signed) formula of the form p **says** f . A request is also represented by a formula. The trust management system says that principal p authorizes request f according to the assertions F if $F \vdash p$ **says** f .

$$\begin{aligned}
f &\in \text{Formula} \\
u &\in \text{Auth} &= \{F \subseteq \text{Formula} \mid F = \overline{F}\} \\
p \text{ says } f &\in \text{LogicAssertion} &= \text{Principal} \times \text{Formula} \\
\mathcal{M}_{\text{Logic}} &: \text{LogicAssertion} \longrightarrow \text{Assertion} \\
\mathcal{M}_{\text{Logic}}(p \text{ says } f) &= \langle p, \lambda m. \overline{f \cup \{p' \text{ says } f' \mid f' \in m(p')\}} \rangle
\end{aligned}$$

Figure 6. The logic-based approach

Figure 6 shows how to model this approach in the framework of Section 2. The figure should be read along with Figure 3, which defines the supporting infrastructure of the framework. The elements of the *Auth* lattice are deductively closed sets of formulas ordered by \subseteq . A set of formulas f is deductively closed if $F = \overline{F}$, where $\overline{F} = \{f \mid F \vdash f\}$. The least upper bound operation is union, except that elements must be deductively closed: $\bigsqcup \{F_i \mid i \in I\} = \overline{\bigcup_{i \in I} F_i}$. A request f is represented as an element of *Auth* as $\{f\}$. Let a set of logical assertions F be given and let $A = \{\mathcal{M}_{\text{Logic}}(f) \mid f \in F\}$. The semantics of assertions, $\mathcal{M}_{\text{Logic}}$, is defined so that $f \in \mathcal{M}_{\text{Assertions}}(A)(p)$ if and only if $F \vdash p \text{ says } f$. The trust engine says that p authorizes a request f according to the assertions A if $\mathcal{M}_{\text{Engine}}(p, \overline{\{f\}}, A)$, which is equal to $\overline{\{f\}} \subseteq \mathcal{M}_{\text{Assertions}}(A)(p)$, which is equivalent to $f \in \mathcal{M}_{\text{Assertions}}(A)(p)$, which by the above is equivalent to $F \vdash p \text{ says } f$. Thus, the definition of the trust engine in the framework coincides with the logic-based definition. Depending on the logic, a direct least fixpoint computation may or may not be feasible. If the set of consequences of a finite set of formulas is finite (as in [3, 17]), then the least fixpoint can be directly computed. If, on the other hand, the logic is undecidable (as in [2]), then the trust management computation can either use an incomplete proof search or require the client to provide a proof that $f \in \mathcal{M}_{\text{Assertions}}(A)(p)$.

6 Applications

6.1 Certificate reduction

This section formalizes the notion of certificate reduction, in which a trust management system provides a mechanism to combine several certificates into a single certificate that summarizes their meaning (e.g. SPKI tuple reduction). For example, if we have the assertions

$$\begin{aligned}
a_{A1} &= \langle \text{Alice}, \lambda m. m(\text{Bob}) \rangle \\
a_B &= \langle \text{Bob}, \lambda m. m(\text{Carl}) \rangle
\end{aligned}$$

then we can create the assertion

$$a_{A2} = \langle \text{Alice}, \lambda m. m(\text{Carl}) \rangle$$

and prove that we have not increased the authorizations granted by Alice. Certificate reduction can make the overall system more efficient by reducing the number of certificates carried around and used in subsequent trust engine decisions. It can also provide anonymity to some of the participants. In our example, Bob no longer needs to be mentioned when Alice grants authorizations by delegating to Carl.

We must be careful, however, since it is not the case that for all sets of assertions A that

$$\mathcal{M}_{\text{Assertions}}(A \cup \{a_{A1}, a_B\}) \stackrel{?}{=} \mathcal{M}_{\text{Assertions}}(A \cup \{a_{A2}\})$$

The following example shows one reason why.

$$\begin{aligned}
A &= \{\langle \text{Carl}, \lambda m. W \rangle\} \\
\mathcal{M}_{\text{Assertions}}(A \cup \{a_{A1}, a_B\})(\text{Bob}) &= W \\
\mathcal{M}_{\text{Assertions}}(A \cup \{a_{A2}\})(\text{Bob}) &= \perp
\end{aligned}$$

Obviously, because we have removed Bob's delegation to Carl, we have reduced Bob's authorizations. The following example shows why even the authorizations granted by Alice may change.

$$\begin{aligned}
A &= \{\langle \text{Bob}, \lambda m. W \rangle\} \\
\mathcal{M}_{\text{Assertions}}(A \cup \{a_{A1}, a_B\})(\text{Alice}) &= W \\
\mathcal{M}_{\text{Assertions}}(A \cup \{a_{A2}\})(\text{Alice}) &= \perp
\end{aligned}$$

Since we have removed Alice's delegation to Bob, we have reduced Alice's authorizations.

To simplify the formalization, we will only consider certificate reduction of two certificates. Suppose that we have two assertions $a_1 = \langle p_1, l_1 \rangle$ and $a_2 = \langle p_2, l_2 \rangle$. Define the reduction of a_1 and a_2 as

$$\mathbf{R}(a_1, a_2) = \langle p_1, \lambda m. l_1(m \sqcup [p_2 \mapsto l_2(m)]) \rangle$$

It is possible to prove that the reduced assertion grants no more authorizations than were granted by the original two assertions. That is, for all sets of assertions A ,

$$\mathcal{M}_{\text{Assertions}}(A \cup \{\mathbf{R}(a_1, a_2)\}) \subseteq \mathcal{M}_{\text{Assertions}}(A \cup \{a_1, a_2\})$$

Our earlier examples showed that equality does not hold for arbitrary sets of assertions. However, if none of the assertions mention p_2 , then equality does hold for all principals

except for p_2 . More formally, if for all $\langle p, l \rangle \in A$, $p \neq p_2$ and for all authmaps m , $l(m) = l(m \sqcap [p_2 \mapsto \perp])$, then

$$\begin{aligned} & \mathcal{M}_{\text{Assertions}}(A \cup \{\mathbf{R}(a_1, a_2)\}) \\ &= [p_2 \mapsto \perp] \sqcap \mathcal{M}_{\text{Assertions}}(A \cup \{a_1, a_2\}) \end{aligned}$$

For a particular trust management system, the language for writing licenses might not be expressive enough to allow certificate reduction. That is, there may be license expressions e_1 and e_2 denoting licenses l_1 and l_2 , but no license expression denoting l , where $\langle p_1, l \rangle = \mathbf{R}(\langle p_1, l_1 \rangle, \langle p_2, l_2 \rangle)$. In the case of SPKI, with the minor exception of the intersection of `*range` and `*prefix` actions, the license language is expressive enough. One principle designers of new trust management systems should keep in mind is making the license language powerful enough to express all of the monotone functions that are needed for certificate reduction.

6.2 Proof checking vs. proof construction

One way to view an implementation of $\mathcal{M}_{\text{Engine}}$ is that when $\mathcal{M}_{\text{Engine}}(p, u, A) = \text{true}$, the engine has constructed a proof that $u \sqsubseteq \mathcal{M}_{\text{Assertions}}(A)(p)$. For several reasons, it may make more sense to require the client of the trust engine to provide the proof and have the trust engine simply check that the proof is valid (as is done in [2]). First, the trust engine can be smaller and faster because proof checking is easier than computing the entire fixpoint. Also, third-party modules can be introduced that find the proof using efficient, complex, or new methods. Finally, as part of a trusted computing base, having a simpler and more reliable engine increases overall security.

We can model this approach within our framework using a slightly modified trust engine, $\mathcal{M}_{\text{EngineProof}}$, that takes as input a finite sequence of assertions instead of an unordered set of assertions.

$$\mathcal{M}_{\text{EngineProof}} : \text{Principal} \times \text{Auth} \times \text{Assertion}^* \longrightarrow \text{Bool}$$

To decide if $\mathcal{M}_{\text{EngineProof}}(p, u, [a_1, \dots, a_n])$, where $a_i = \langle p_i, l_i \rangle$, the trust engine computes an increasing sequence of authmaps m_0, \dots, m_n in the *AuthMap* lattice:

$$\begin{aligned} m_0 &= \perp \\ m_i &= m_{i-1} \sqcup [p_i \mapsto l_i(m_{i-1})] \end{aligned}$$

The engine returns `true` if $u \sqsubseteq m_n(p)$. It is easy to see that for all i , $m_i \sqsubseteq \mathcal{M}_{\text{Assertions}}(A)$, where $A = \{a_1, \dots, a_n\}$. That is, $\mathcal{M}_{\text{EngineProof}}$ computes a lower bound to the fixpoint needed by $\mathcal{M}_{\text{Engine}}$. By the pointwise ordering on authmaps, if $m \sqsubseteq \mathcal{M}_{\text{Assertions}}(A)$ and $u \sqsubseteq m(p)$ then $u \sqsubseteq \mathcal{M}_{\text{Assertions}}(A)(p)$. From this it follows that if $\mathcal{M}_{\text{EngineProof}}(p, u, [a_1, \dots, a_n])$ then $\mathcal{M}_{\text{Engine}}(p, u, \{a_1, \dots, a_n\})$. Thus, $\mathcal{M}_{\text{EngineProof}}$ never authorizes a request unless $\mathcal{M}_{\text{Engine}}$ would have authorized it.

7 Conclusion

The framework in this paper can be used to explain existing trust management systems and to help design new ones. It can provide a precise specification of the semantics of a trust management system, which is important for building correct, interoperable implementations. The least fixpoint semantics leads to implementations of trust management engines. The framework can concisely specify trust management systems by an authorization lattice and language for licenses. This makes it possible to compare the expressiveness of systems. It also makes it easier to assess the applicability of a system to a given situation and to analyze design tradeoffs among current and new systems. The framework can also help to improve existing languages for expressing licenses by making them more regular and more expressive.

There are some aspects of trust management systems that do not fit well within the framework. One example is REFERENCE [8], in which the trust management engine directly interprets policies and credentials, without finding a fixpoint meaning. It is also impossible to express non-monotonic systems. It would be interesting to explore generalizations of the framework that could encompass these systems.

The most interesting technical question that this work raises is what are the right *Auth* lattices to choose and what collections of monotone license functions should be used so that the trust management system is expressive enough and so that the trust management engine can efficiently approximate the least fixpoint. In answering this question, there are likely to be useful insights from the field of abstract interpretation [10], a program analysis technique based on fixpoint computation.

8 Acknowledgements

I would like to thank Jim Donahue, Neal Glew, Stuart Haber, Jim Horning, Umesh Maheshwari, Jeff McDow, Radek Vingralek, and Andrew Wright for helpful comments on this work and on early drafts of the paper. I would also like to thank Xavier Serret-Avila for helping me to learn SPKI and for collaborating on the SPKI implementation. Finally, I would like to thank the anonymous referees for helpful suggestions that provided the basis of Section 6.

A Notation and mathematical background

This section introduces the notation and mathematical background used throughout the paper. Capitalized names in italic font indicate sets, for example, *Int* is the set of integers. Lower case letters typically denote elements of sets, as in $i \in \text{Int}$. The power set of the integers is denoted $\mathcal{P}(\text{Int})$.

Upper case letters typically range over elements of a power set, as in $I \in \mathcal{P}(Int)$. The cardinality of a set A is denoted $card(A)$. If A and B are sets, then $A + B$, $A \times B$, and $A \rightarrow B$ are the disjoint sum, cartesian product, and set of partial functions from A to B , respectively. If A is a set, then A^* denotes the set of finite sequences of elements of A . In expressions denoting sets, the operator precedence (in decreasing order) is $*$, \times , \rightarrow , $+$. An element of $A \times B \times C$ is denoted $\langle a, b, c \rangle$. An element of A^* is denoted $[a_0, a_1, a_2]$. The expression $[a_1 \mapsto b_1, a_2 \mapsto b_2, \dots]$ denotes the function f in $A \rightarrow B$ such that $f(a_1) = b_1$, $f(a_2) = b_2, \dots$. The expression $\lambda a. e$ denotes the function in $A \rightarrow B$ that when applied to an argument $a \in A$, returns the result of expression e , which may refer to a , and denotes an element of B . For example, $\lambda i. i + 1$ is the function that adds one to its argument. The expression $f(a)$ denotes the application of function f to argument a . The inner $\langle \rangle$ is dropped in $f(\langle a, b, c \rangle)$, which is written as $f(a, b, c)$. Function application associates to the left and \rightarrow associates to the right. For example, if $f = \lambda i. \lambda j. i - j$, then $f \in Int \rightarrow Int \rightarrow Int$ and $f(7)(5) = 2$.

A binary relation \sqsubseteq on A is a subset of $A \times A$. The expression $a \sqsubseteq a'$ denotes $\langle a, a' \rangle \in \sqsubseteq$. The relation \sqsubseteq is reflexive if $a \sqsubseteq a$ for all $a \in A$, transitive if $a \sqsubseteq a'$ whenever $a \sqsubseteq a'$ and $a' \sqsubseteq a''$, and anti-symmetric if $a = a'$ whenever $a \sqsubseteq a'$ and $a' \sqsubseteq a$. A partial order is a set A and a relation \sqsubseteq that is reflexive, transitive, and anti-symmetric. In a partial order, a is an upper bound of a subset $A' \subseteq A$ if for all $a' \in A'$, $a' \sqsubseteq a$. A least upper bound of A' , denoted $\bigsqcup A'$, is an upper bound a of A' such that $a \sqsubseteq a_0$ whenever a_0 is an upper bound of A' . The greatest lower bound, denoted \bigsqcap , is defined analogously. A lattice is a partial order in which every subset $A' \subseteq A$ has a least upper bound. The least element of a lattice is denoted by \perp , and is equal to $\bigsqcup \{\}$. If A is a set then $\mathcal{P}(A)$ is a lattice, where $A' \sqsubseteq A''$ iff $A' \subseteq A''$ and $\bigsqcup \{A_i \mid i \in I\} = \bigcup \{A_i \mid i \in I\}$. If A and B are lattices then $A \times B$ is a lattice, where $\langle a, b \rangle \sqsubseteq \langle a', b' \rangle$ iff $a \sqsubseteq a'$ and $b \sqsubseteq b'$, and $\bigsqcup \{\langle a_i, b_i \rangle \mid i \in I\} = \langle \bigsqcup \{a_i \mid i \in I\}, \bigsqcup \{b_i \mid i \in I\} \rangle$. If A is a set and B is lattice, then $A \rightarrow B$ is a lattice under the pointwise ordering, where $f \sqsubseteq g$ iff $f(a) \sqsubseteq g(a)$ for all $a \in A$ and where $\bigsqcup \{f_i \mid i \in I\} = \lambda a. \bigsqcup \{f_i(a) \mid i \in I\}$.

A function f from a partial order A to a partial order B is *monotone* if $f(a) \sqsubseteq f(a')$ whenever $a \sqsubseteq a'$. The set of monotone functions from A to B is written $A \rightarrow_m B$. If $f \in A \rightarrow A$, then a is a fixpoint of f if $f(a) = a$. If A is a partial order, then a *least fixpoint* of f , written $lfp(f)$, is a fixpoint a such that $a \sqsubseteq a'$ whenever a' is a fixpoint of f . If f has a least fixpoint, it is unique. If A is a lattice and $f \in A \rightarrow_m A$, then f always has a least fixpoint.

A *chain* in a partial order (A, \sqsubseteq) is a sequence of $a_i \in A$ such that for all i , $a_i \sqsubseteq a_{i+1}$. If A and B are partial orders and $f \in A \rightarrow_m B$, then f is *continuous* if for every chain

a_i in A , $f(\bigsqcup \{a_i \mid i \in I\}) = \bigsqcup \{f(a_i) \mid i \in I\}$. Define $f^1(a) = f(a)$ and for all i , $f^{i+1}(a) = f(f^i(a))$. Then, if f is continuous, the least fixpoint of f is given by $lfp(f) = \bigsqcup \{f^i(\perp) \mid i \in Int\}$. If the elements of the lattice A are representable and f is computable and for some i , $f^i(\perp) = f^{i+1}(\perp)$, then this gives us a method for computing $lfp(f)$; namely, compute $f^1(\perp)$, $f^2(\perp)$, $f^3(\perp)$, etc. until the sequence converges.

References

- [1] M. Abadi, M. Burrows, B. Lampson, and G. Plotkin. A calculus for access control in distributed systems. In J. Feigenbaum, editor, *Advances in Cryptology - CRYPTO '91, 11th Annual International Cryptology Conference*, pages 1–23, Aug. 1991. LNCS 576.
- [2] A. W. Appel and E. W. Felten. Proof-carrying authentication. In *6th ACM Conference on Computer and Communications Security*, Nov. 1999.
- [3] D. Balfanz, D. Dean, and M. Spreitzer. A security infrastructure for distributed Java applications. In *Proceedings of the 21st IEEE Symposium on Security and Privacy*, May 2000.
- [4] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. D. Keromytis. The KeyNote trust-management system version 2. RFC 2704, Sept. 1999.
- [5] M. Blaze, J. Feigenbaum, and A. D. Keromytis. KeyNote: Trust management for public-key infrastructures. *Lecture Notes in Computer Science*, 1550:59–63, 1999.
- [6] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, Research in Security and Privacy, Oakland, CA, May 1996. IEEE Computer Society, Technical Committee on Security and Privacy, IEEE Computer Society Press.
- [7] M. Blaze, J. Feigenbaum, and M. Strauss. Compliance-checking in the PolicyMaker trust-management system. In *Proc. 2nd Financial Crypto Conference*, pages 251–265. Springer-Verlag, 1998. LNCS 1465.
- [8] Y.-H. Chu, J. Feigenbaum, B. LaMacchia, P. Resnick, and M. Strauss. REFEREE: Trust management for Web applications. *Computer Networks and ISDN Systems*, 29(8–13):953–964, Sept. 1997.
- [9] D. Clarke, J.-E. Elien, C. M. Ellison, M. Fretette, A. Morcos, and R. L. Rivest. Certificate chain discovery in SPKI/SDSI. Available at <http://theory.lcs.mit.edu/~rivest/>, Nov. 1999.
- [10] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM Symposium on Principles of Programming Languages*, pages 238–252, Jan. 1977.
- [11] J.-E. Elien. Certificate discovery using SPKI/SDSI 2.0 certificates. Master's thesis, Massachusetts Institute of Technology, May 1998.
- [12] C. M. Ellison. SPKI requirements. RFC 2692, Sept. 1999.

- [13] C. M. Ellison, B. Frantz, B. Lampson, R. L. Rivest, B. M. Thomas, and T. Ylonen. SPKI certificate theory. RFC 2693, Sept. 1999.
- [14] C. M. Ellison, B. Frantz, B. Lampson, R. L. Rivest, B. M. Thomas, and T. Ylonen. Simple public key certificate, July 99. Available at <http://www.pobox.com/~cme/spki.txt>.
- [15] J. Howell and D. Kotz. A formal semantics for SPKI. Technical Report 2000-363, Department of Computer Science, Dartmouth College, Hanover, NH, Mar. 2000.
- [16] N. Li. Local names in SPKI/SDSI 2.0. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop*, July 2000.
- [17] N. Li, B. N. Grosz, and J. Feigenbaum. A logic-based knowledge representation for authorization with delegation (extended abstract). In *Proceedings of the 12th IEEE Computer Security Foundations Workshop*, pages 162–174. IEEE Computer Society Press, 1999.
- [18] Y. Mass and A. Herzberg. Access control meets public key infrastructure or: Assigning roles to strangers. In *Proceedings of the 21st IEEE Symposium on Security and Privacy*, May 2000. Available at <http://www.hrl.il.ibm.com/TrustEstablishment/default.asp>.
- [19] R. L. Rivest and B. Lampson. SDSI — A simple distributed security infrastructure, Oct. 1996. Available at <http://theory.lcs.mit.edu/~cis/sdsi.html>.
- [20] I. T. Union. ITU-T recommendation X.509 (08/97) – information technology – open systems interconnection – the directory: Authentication framework, Aug. 1997.
- [21] E. Wobber, M. Abadi, M. Burrows, and B. Lampson. Authentication in the Taos operating system. *ACM Transactions on Computer Systems*, 12(1):3–32, Feb. 1994.
- [22] P. Zimmerman. *The Official PGP User's Guide*. MIT Press, Cambridge, 1995.