Reconstructing Trust Management

Ajay Chander [*]
STANFORD UNIVERSITY
ajayc@cs.stanford.edu

Drew Dean [†]
SRI INTERNATIONAL
ddean@csl.sri.com

John C. Mitchell[*]
STANFORD UNIVERSITY
mitchell@cs.stanford.edu

Abstract

We present a trust management kernel that clearly separates authorization and structured distributed naming. Given an access request and supporting credentials, the kernel determines whether the request is authorized. We prove soundness and completeness of the authorization system without names and prove that naming is orthogonal to authorization in a precise sense. The orthogonality theorem gives us simple soundness and completeness proofs for the entire kernel. The kernel is formally verified in PVS, allowing for the automatic generation of a verified implementation of a reference monitor. By separating naming and authorization primitives, we arrive at a compositional model and avoid concepts such as "speaks-for" that have led to anomalies in logical characterizations of other trust management systems.

1 Introduction

Access control in distributed systems is challenging. Unlike centralized systems, a resource owner may not know the identity of an access requester, and thus identity-based access control in such systems can be very limiting. Authentication is not the only issue in distributed systems; the dynamic nature of such systems encourages the use of policy constructs like delegation and local naming, which must be suitably supported in any practical distributed system. One early strand of work in this area occurred in the development of the Taos operating system in the late 1980s and early 1990s, and included characterizations of policy language [23], logical frameworks for understanding access control [2], and system integration [15].

[*]Computer Science Department, Stanford University, Stanford, CA 94305 U.S.A. Supported in part by DARPA contract N66001-00-C-8015 and ONR grant N00014-01-1-0795.

[†]Computer Science Laboratory, SRI International, Menlo Park, CA 94025 U.S.A. Supported in part by DARPA contract N66001-00-C-8015 and ONR grant N00014-01-1-0837.

Concurrent proposals such as Neuman's proxy-based architecture [19] and the Digital distributed system security architecture [9] of Gasser, et al. also introduced the notion of delegation in a world of autonomous, cooperating, peer entities. In the mid 1990s, Blaze, Feigenbaum, and Lacy coined the term "trust management" [6] to treat similar issues in an application-independent way.

In this paper, we will use the term "trust management" to refer to distributed access control based on cryptographic keys, signed credentials, and local policies [6, 5, 4]. One characteristic of trust management is *delegation:* the owner of a resource can empower another principal to grant bounded access rights to the resource. For example, owner Alice [10] may allow principal Bob to delegate access, Bob may delegate this right to Charlie, and Charlie may then exercise this right. Another characteristic of trust management is distributed naming. When Alice expresses her policy about resources, she may refer to so-called *local names* that she defines herself, or refer to names that are globally known or defined by other principals. For example, an employee at Company *Z* may refer to *Z's CEO's assistant*, meaning the principal that *Z's CEO* defines to be her *assistant*, where *Z's CEO* is again the principal that *Z* designates as *CEO*.

One fundamental source of complexity in trust management is the interaction between deduction, signing, and distributed naming. Since trust management authorization is based on logical deduction, it is natural to try to formulate name definitions in a logical manner and appeal to deduction for name resolution. Also, trust management policies involve assertions by various principals, with digital signatures used to verify that assertions were actually issued by authorized principals. However, previous efforts to harness logical characterizations of naming and signing have led to puzzling interactions and anomalies. Much ink has been spilled examining the naming portion of SDSI [1, 11], for example, and more recently its interaction with SPKI [12]. It has been pointed out by several authors [1, 3, 12] that "surprising" conclusions may result from the addition of seemingly reasonable logical access control rules to a system.

We believe, though, that the interaction between authorization, naming, and correct signing of certificates, originally presented in SDSI [21] in a purely operational way, should submit to a simple modular solution. Hence, we present here a clean sheet design, with a simple formalization. Our state-transition model borrows from our earlier work [7], where we developed a general access control framework based on abstract system states, state transitions, and logical deduction of access judgments. This framework was used to compare the expressive power of trust management systems to access control list and capability systems, reaching the conclusion that trust management combines the strong points of access control list and capability systems by allowing subjects to delegate rights in a revokable manner. As a labeled transition system, the state-transition model is also inherently *operational* as opposed to the *axiomatic* specifications presented in previous work. A direct benefit of this fact is an intuitive operational semantics for the transition system, which is also close in spirit to actual implementations of access control mechanisms in distributed systems.

Having found the state transition modeling a useful comparative tool, in this paper we address the interaction of naming and authorization primitives. What does a name mean? How is authorization granted? Can we define a clear semantics for each of these constructs, and then combine them so that there are "no surprises"?

This paper makes several technical contributions. First, we present *access graphs* as a semantic model for reasoning about access; the interpretation of the state transition model in an access graph is algebraic in flavor. We start by establishing the robustness of our trust management authorization model by proving soundness and completeness theorems. These theorems show that a simple deductive characterization of authorization corresponds exactly to an access-graph-based semantics of authorization assertions. Second, we address the semantics of names within the state-transition model, specifying the relevant system states, transitions, and judgments relevant to naming. A key idea in our treatment of names is to identify the meaning of a name with an environment or a name space. A name binding within a name space, on the other hand, assigns to a name a key or another name. Our notion of environment is similar to its use in a programming language-theoretic setting; we evaluate policy statements and local name definitions *w.r.t.* specific environments, and define properties like globality in terms of quantification over environments. Our naming proposal is expressive enough to capture, for example, SDSI/SPKI naming [21].

With a clear modeling of both authorization and naming, we extend our authorization mechanism to allow for structured distributed names, thus producing a trust management kernel. We model the kernel as a state-transition system, combining the relevant parts of the previous models for authorization and naming. Our formal statement of the "no surprises" result is the commutativity theorem, which informally states that the access decisions made in the kernel can be viewed as a sequential application of name resolution rules, followed by authorization rules that reason with keys. This result also provides soundness and completeness theorems for the entire kernel, thereby confirming our hypothesis that naming and authorization compose cleanly. As a complementary result, we formally specify the power of the naming abstraction by using simulation relations that compare authorization actions in the full kernel with authorization actions referring only to keys. Finally, we have formally verified the trust management kernel within the automated theorem-proving system PVS, with the ability to generate proofs corresponding to allowed accesses. Our work thus forms the basis for the automatic generation of a verified implementation of a reference monitor in distributed systems.

Our analysis presented us with some informal guidelines that we found useful in evaluating ad-hoc logical rules. The first is captured by the commutativity theorem. The other concerns adherence to scope; we found several examples of rules in the literature where unclear scope extrusion results in unintended consequences. Our insistence on tracking scope in name and action definitions enabled us to disallow problematic rules [1] within our semantics, and to express concepts like globality.

The rest of the paper is organized as follows. Section 2 discusses past work on the modeling of names. Section 3 summarizes the state-transition model, TM, of trust management without names. Section 4 presents a graph-based semantics and proves soundness and completeness of the access judgment rules. Section 5 presents the structure and semantics of names used for naming resources and in access control constructs. We also mention how SPKI/SDSI naming can be expressed within our system. Section 6 presents the model TM_{N}, extending the original state-transition model with structured names. We formally show the orthogonality of name and authorization resolution as a commutativity theorem, and use it to prove soundness and completeness of the entire system in Section 7. We discuss some inconsistencies that arise from incomplete semantics in Section 7.1. Section 8 describes the PVS specification of our theorems, and of decision procedures to decide access. Section 9 presents our final conclusions.

2 Related Work

The most closely related work to this is the study of SPKI/SDSI naming and authorization. Lampson and Rivest proposed SDSI in 1996 [21], introducing the notion of privately defined names and name resolution of linked names. SDSI's names were adopted, among other things, into the merged SDSI/SPKI 2.0 proposal [8]. Abadi did the first formal study [1] of the SDSI name resolution algorithm in 1998 by formally specifying the resolution rules in a logic, and showing that any name resolved by the SDSI resolution algorithm corresponded to a proof using the logical resolution rules. The logic was based on a possible world semantics that interpreted the name definition $p \mapsto q$ as $[\![p]\!] \subseteq [\![q]\!]$, where $[\![p]\!]$ informally is the set of keys associated with $[\![p]\!]$. Abadi also used the logical relation "speaks-for" as an interpretation of name binding. He pointed out that the addition of apparently reasonable rules might result in undesirable conclusions from a security standpoint. Halpern and van der Meyden followed Abadi's work with two influential papers [11, 12] in which they introduced LLNC, the logic of local name containment that interprets $p \mapsto q$ as the more intuitive$[\![p]\!] \supseteq [\![q]\!]$ instead. In addition, they separated "speaks-for" from the semantics of naming, associating speaks-for with delegation of authority. Their axiomatic characterization provides a useful specification of naming but is not directly connected to an actual implementation. Li [16] presented a logic programming formulation of name resolution by encoding the process as a standard logic program. However, like other previous logical characterizations, it lacks a methodology for evaluating a proposed set of access control rules that involve both naming and authorization.

Authorization, on the other hand, has a longer history. One of the earliest models of access control was the *access matrix* of Lampson [14], used for safety analysis by Harrison, Ruzzo, and Ullman [13]. Lipton and Snyder [17] introduced the Take-Grant model of access control, and our graph-based model is in some ways similar to it. The granularity in these models was at the level of single objects and subjects; the introduction of roles [22] provided a newer level of abstraction in access control. In parallel, authorization was also formally studied within operating systems in the late 1980s and 1990s [23, 2], while delegation constructs have been more recently studied in the context of distributed [19, 9] and peer-to-peer systems.

3 A State-Transition Model of Authorization

In earlier work [7], we proposed a new model of access control, based on abstract system states, state transitions, and logical deduction of access control judgments. The main idea is to identify a set of abstract system states, each containing the kind of information that would be maintained in an access control system. The important property of each state is the set of access requests that will be allowed in it, and the access requests that will be allowed after subsequent actions such as the transfer of a capability. The set of allowed access requests may be recorded directly in the state, as in access control lists, or derived from properties of the state by some form of logical inference. In this framework, we compare access control mechanisms underlying trust management, access control lists, and two flavors of capabilities, by comparing the resulting labeled transition systems (see Appendix A) using traditional forms of simulation relations from programming language and concurrency theory.

The general state-transition-based approach to the modeling of access control mechanisms specifies the following:

1. A *world state*, the part of the system configuration that is relevant to the access control mechanism,
2. A set of possible *actions*, each defining a transition function from world states to world states,

3. An *access judgment*, which states when one object can access another. This may be specified in the form of logical inference, equivalent to some implementable algorithm.

Given a world state *WS*, the judgment that subject *s* can exercise the right *r* on object *o* is written as $WS \vdash s \rightarrow (o,r)$.

For the case of trust management without local name spaces, the model for which we shall denote by TM, the world state consists of a set *O* of objects, a set *R* of rights, and two maps *A* (bounded root-ACL) and *D* (bounded Delegate):

$$\begin{array}{rcl} A & : & O \times R \rightarrow \mathcal{P}(O \times \mathbb{N}) \\ D & : & O \times R \times O \rightarrow \mathcal{P}(O \times \mathbb{N}) \end{array}$$

The maps *A* and *D* capture the two ways through which access privileges are propagated in this framework — either they are granted directed by the host object via the root ACL map *A*, or they are delegated through other objects that hold the access right, via the delegation map *D*. Each of these methods provides for a delegation depth that bounds further granting privileges, and we capture this by the set $\mathbb{N} = \{0,1,\ldots,\infty\}$ in the co-domain of the two functions. Thus, if $(o_s,n) \in A((o,r))$, then o_s can access right *r* on object *o*, and can delegate that access right to another object o_d that can then delegate it to a maximum effective depth of $n-1$. The delegation action by o_s would be modeled as $(o_d,n-1) \in D(o_s,r,o)$. In general if an object o_s delegates its access right *r* on object *o*, then the set of such delegations is captured by $D(o_s,r,o)$.

An action is specified by how it changes the components of the world state *w* in which it is invoked. Table 1 summarizes the effects of actions for the trust management model. The action Create(o_c,o) denotes the creation of object *o* by the creating object o_c, and affects the components O,A, and *D* of the world state. Object o_c is given the right r_e to edit *o*'s root-ACL, and the ability to delegate that right to anyone it wishes to. No one else holds any other rights to *o* at this point. The action Delete(o) removes all instances of the object *o* from the system, thereby removing it from the set of objects *O*, its root-ACLs from the map *A*, and all delegated access rights to it from the map *D*. In other words, the maps *A* and *D* are updated by restricting their domains to the sets $(O-\{o\}) \times R$ and $(O-\{o\}) \times R \times (O-\{o\})$, respectively. The Add$(o,r,o_s,d)$ action gives subject o_s the right *r* on object *o* with further delegation powers *d* and therefore affects only the *A*-map component of the world state. Since this newly obtained right has not yet been delegated, the map *D* and other state components remain the same. Similarly, the Remove(o,r,o_s,d) action removes subject o_s from the root-ACL corresponding to right *r* on object *o*. Finally, the actions Delegate(o_s,o,r,o_d,d) and Revoke(o_s,o,r,o_d,d) capture the delegation or revocation, respectively, by object o_s of its access right (o,r) to delegatee object o_d, with further delegation privileges *d*. Note that none of the actions

	O	A	D
Create(o_c, o)	$\cup\{o\}$	$(o, r_e) \mapsto (o_c, 1)$ $(o, r) \mapsto \emptyset$	$(s, r, o) \mapsto \emptyset$
Add(o, r, o_s, d)		$(o, r) \mapsto A(o, r)$ $\cup\{o_s, d\}$	
Remove(o, r, o_s, d)		$(o, r) \mapsto A(o, r)$ $-\{o_s, d\}$	
Delegate(o_s, o, r, o_d, d)			$(o_s, r, o) \mapsto D(o_s, r, o)$ $\cup\{o_d, d\}$
Revoke(o_s, o, r, o_d, d)			$(o_s, r, o) \mapsto D(o_s, r, o)$ $-\{o_d, d\}$
Delete(o)	$-\{o\}$	$\vert$	$\vert$

Table 1: Trust Management

changes the set of rights R, which is assumed to be a fixed part of the system specification.

We specify the access judgment as a logical judgment in a proof system with the four inference rules given below. To bootstrap the inference process, we translate the world state into predicates. The maps A and D of the world state can be interpreted as set-membership predicates; $\mathsf{ACL}(s, o, r, d)$ is true iff $(s, d) \in A(o, r)$, and $\mathsf{Del}(s, o, r, r_s, d)$ is true iff $(r_s, d) \in D(s, r, o)$. In other words, $\mathsf{ACL}(s, o, r, d)$ is true iff subject s belongs on the root-ACL for the access right (o, r), and $\mathsf{Del}(s, o, r, r_s, d)$ is true iff subject s has delegated its access right (o, r) to subject r_s, with further delegation allowed up to depth d in both cases. In the system TM, subject s can access the (o, r) pair iff it can produce a proof of $\mathsf{Access}(s, o, r, d)$ for some d, from the predicate equivalent of the world state and the following four inference rules:

$$\begin{array}{lr}
(\mathit{RootACL}) & \mathsf{ACL}(A, B, r, d) \vdash \mathsf{Access}(A, B, r, d) \\
(\mathit{Delegation}) & \mathsf{Access}(A, B, r, d+1), \mathsf{Del}(A, B, r, C, d) \\
 & \vdash \mathsf{Access}(C, B, r, d) \\
(\mathit{Ord1}) & \mathsf{Access}(A, B, r, d+1) \vdash \mathsf{Access}(A, B, r, d) \\
(\mathit{Ord2}) & \mathsf{Del}(A, B, r, C, d+1) \vdash \mathsf{Del}(A, B, r, C, d)
\end{array}$$

The first two rules capture the root ACL and delegation chain mechanisms of obtaining access, and the last two capture the “downward closure” property of delegation depths. The idea of “downward closure” is very simple: if a subject can delegate a right up to depth n, then clearly that subject can also delegate the right up to depths $0, 1, \ldots, n-1$. Essentially, rules (*Ord1*) and (*Ord2*) produce appropriate predicates for an application of the (*Delegation*) rule.

Notice that the A and D maps specify a simple access control policy language for the system TM, and the other components of the modeling provide the supporting data structures and valid transitions thereof, and a logical representation of the access semantics. Extending this to a more expressive policy language would entail augmenting the "state" in this state machine, and providing additional transitions and logical access judgment rules. However, our policy language is already sufficient to capture access mechanisms in some significant trust management systems like SPKI/SDSI [21, 8] and KeyNote [4]. Since our focus in this paper is on the interaction between authorization and naming in general trust management systems, we will use TM as a representative example for the analysis, which can be extended to more expressive trust management systems. For further details on the model TM and its relationship to other access control mechanisms, the interested reader may look at [7].

4 Semantics

We present an intuitive semantics for the system TM that has an algebraic flavor: the idea is to map elements of the world state to appropriate carrier sets and functions on them. In particular, we will have the carrier sets $O^{\mathcal{M}}$, $R^{\mathcal{M}}$, and the functions

$$\begin{array}{rcl} A^{\mathcal{M}} & : & O^{\mathcal{M}} \times R^{\mathcal{M}} \rightarrow \mathcal{P}(O^{\mathcal{M}} \times \mathbb{N}) \\ D^{\mathcal{M}} & : & O^{\mathcal{M}} \times R^{\mathcal{M}} \times O^{\mathcal{M}} \rightarrow \mathcal{P}(O^{\mathcal{M}} \times \mathbb{N}) \end{array}$$

We will represent these mappings as edges in a graph, so that allowed accesses may be represented simply as valid access paths.

More formally, we define a model $\mathcal{M}$ as a directed labeled *access graph* $(O^{\mathcal{M}}, E)$, where the vertex set $O^{\mathcal{M}}$ corresponds to the set of objects in the model, together with an auxiliary set $R^{\mathcal{M}}$ that corresponds to the set of rights. Edges in the access graph have labels of the form (o, r, d), where $o \in O^{\mathcal{M}}, r \in R^{\mathcal{M}}$ and $d \in \mathbb{N}$. Thus, $E \subseteq O^{\mathcal{M}} \times (O^{\mathcal{M}} \times R^{\mathcal{M}} \times \mathbb{N}) \times O^{\mathcal{M}}$. We require that every access graph satisfy the downward closure property (see Section 3):

$$(x, (o, r, n+1), y) \in E \Rightarrow (x, (o, r, n), y) \in E.$$

Given a world state *WS*, we say that a model $\mathcal{M}$ satisfies *WS*, written $\mathcal{M} \models WS$, if the following conditions hold:

1. For each element $o \in O, r \in R$, there exist unique elements $o^{\mathcal{M}} \in O^{\mathcal{M}}, r^{\mathcal{M}} \in R^{\mathcal{M}}$, respectively.

2. The maps A and D are captured by the edges of the access graph. Formally,

$$\begin{aligned}(s,d) \in A(o,r) &\Rightarrow (s^{\mathcal{M}},(o^{\mathcal{M}},r^{\mathcal{M}},d),o^{\mathcal{M}}) \in E \\ (r_s,d) \in D(s,r,o) &\Rightarrow ({r_s}^{\mathcal{M}},(o^{\mathcal{M}},r^{\mathcal{M}},d),s^{\mathcal{M}}) \in E\end{aligned}$$

   In other words, there is an edge from vertex u to v if u is present on the root-ACL for the corresponding right on vertex v, or if v delegates a right on some other vertex to u. The edge itself is labeled with a tuple (o,r,d) specifying the object/right pair and further delegation powers d.

Given a model $\mathcal{M}$, a subject s can access right r on object o, written $\mathcal{M} \models s \longrightarrow (o,r)$, iff there exists a directed *access path* from $s^{\mathcal{M}}$ to $o^{\mathcal{M}}$ marked with labels of the form $(o^{\mathcal{M}},r^{\mathcal{M}},d)$ such that for every prefix of the path (including the entire path), the number of edges in the prefix is $\leq 1+d_i$, the delegation bound in the last edge of the prefix.

$$o^{\mathcal{M}} \xleftarrow{(o^{\mathcal{M}},r^{\mathcal{M}},d_1)} {s_1}^{\mathcal{M}} \leftarrow \ldots \leftarrow {s_i}^{\mathcal{M}} \xleftarrow{(o^{\mathcal{M}},r^{\mathcal{M}},d_{i+1})} {s_{i+1}}^{\mathcal{M}} = s^{\mathcal{M}}$$

The above condition is equivalent to requiring that the labels $(o^{\mathcal{M}},r^{\mathcal{M}},d_i)$ on the access path from $s^{\mathcal{M}}$ to $o^{\mathcal{M}}$ be such that if d_i and d_{i+1} are the depths associated with any two consecutive edges on the path, then $d_{i+1} \leq d_i - 1$. We say that $\mathcal{M} \models \mathsf{Access}(s,o,r,d)$ iff the first edge of some access path from $s^{\mathcal{M}}$ to $o^{\mathcal{M}}$ is marked with $(o^{\mathcal{M}},r^{\mathcal{M}},d)$.

Note that for any given world state, we can always construct a minimal model that provides concrete representations of exactly the objects and rights in the world state, and has no more edges in the access graph than those required by the A and D maps.

4.1 Theorems

We show that the logical characterization of access judgments in TM is sound and complete *w.r.t.* the access graph semantics. These theorems have been formally verified as part of our PVS effort; for more details see Section 8. Here we present sketches of the main proof strategies used in the PVS proofs.

THEOREM 4.1 *Given an arbitrary world state WS, subject $s \in O$, object $o \in O$ and right $r \in R$, $WS \vdash \mathsf{Access}(s,o,r,d) \Rightarrow \forall \mathcal{M}.\mathcal{M} \models WS \Rightarrow \mathcal{M} \models \mathsf{Access}(s,o,r,d')$ for all $d' \leq d$.*

Proof sketch. We proceed by induction on the structure of the proof of $WS \vdash \mathsf{Access}(s,o,r,d)$; for each proof rule, we combine the access path (in $\mathcal{M}$'s access graph) of the antecedent provided by the induction hypothesis with the fact that $\mathcal{M} \models WS$ to obtain the access path of the consequent. The downward closure property of the models allows us to extend the proof of each subcase to an arbitrary $d' \leq d$, while also directly proving cases (*Ord1*) and (*Ord2*). In PVS, we carry out this induction using PVS's support for induction over inductively defined datatypes. For each proof tree corresponding to a proof of access, we provide an inductive construction of the corresponding path in the access graph as a part of the proof.

COROLLARY 4.2 (Soundness) *Given an arbitrary world state WS, subject $s \in O$, object $o \in O$ and right $r \in R$, $WS \vdash s \rightarrow (o,r) \Rightarrow \forall \mathcal{M}.[\mathcal{M} \models WS \Rightarrow \mathcal{M} \models s \rightarrow (o,r)]$.*

Proof. Immediate from Theorem 4.1, and the definition of $\mathcal{M} \models s \rightarrow (o,r)$.

THEOREM 4.3 (Completeness) *Let WS be an arbitrary world state and let $s, o \in O, r \in R$. Then, $[\forall \mathcal{M}.(\mathcal{M} \models WS) \Rightarrow (\mathcal{M} \models s \rightarrow (o,r))] \Rightarrow WS \vdash s \rightarrow (o,r)$.*

Proof sketch. Each model $\mathcal{M}$ that satisfies *WS* contains the sets $O^{\mathcal{M}}$ and $R^{\mathcal{M}}$, and has edges in the access graph corresponding to the maps A and D of the world state. Since $\mathcal{M} \models s \rightarrow (o,r)$, there exists a valid access path in the graph, whose edges correspond to instances of the ACL and Del predicates that hold in the world state. These can be combined in a straightforward manner using the logical rules $\vdash$ to construct a proof of $WS \vdash s \rightarrow (o,r)$. Once again, we use PVS induction over inductively defined paths. For each well-formed path — that is, a path corresponding to valid access in the graph we construct the corresponding proof tree. The base case for paths of length 1 uses the rule *(RootACL)*, possibly with some applications of (*Ord1*). For paths of greater length, the induction step uses the *(Delegation)* rule, possibly with prior applications of rules (*Ord1*) and (*Ord2*).

5 Structured Names

Any practical trust management system must provide for some version of linked private name spaces. Since naming constructs are independent of authorization primitives, it should be possible to present a semantics for them separately. Moreover, it would be desirable that this semantics compose well with the semantics for authorization, so that there are no "surprises" in a system that uses both. Before we construct the combined system, we must consider the question: what does a name represent?

In real systems, names are ubiquitous: we use them to refer to variables and blocks in a program, to files on disk, to users on a machine, machines on a network, and so on, so that we may conveniently refer to them again. Each of these is a memorable abstraction for the underlying representation in terms of memory addresses (for variables), file and user identifiers in an operating system, numeric IP addresses, and so on. In real life, too, we use names as a means to *refer to* things such as people, streets, and cities. Each name is meant to be unambiguous within a presumed domain of discourse: for example, Portland may denote one city in Oregon and another in Maine. Similarly, the same variable may be used more than once in a program, as long as it is unique within a block. In addition to being associated with an identity, a name provides a *context* for referring to other things; for example, "Powell Books, W Burnside St., Portland, Oregon" refers to a specific bookstore on "W. Burnside St., Portland, Oregon." We conclude that at the very least names are an abstraction for things, intended to be memorable and robust (*e.g.*, Joe's PhoneNumber), and define a context that may be used to refer to other things.

Any practical access control policy must, then, be able to associate privileges with names, instead of only with keys. This is desirable for a few important reasons:

1. Policies are often written in terms of people, and humans think about people by name, not key.

2. A person's key is likely to change over time. Keys get both inadvertently lost (recoverable by no one) and maliciously compromised by attackers. The process of a person getting a new key should not require policy changes.

3. Names can represent nonatomic entities such as groups, making the policy language expressive without compromising on succinctness.

These reasons all argue for policies to be expressed in terms of names, not keys. However, actual execution environments for access control utilize unambiguous identifiers like user- and file-ids, public keys, and IP addresses to make decisions. As in most trust management proposals, we focus on names as being placeholders for keys in policies. Hence, we need a mapping from the surface syntax of the policy language (which may reference names) to the keys used to decide access requests.

We now add structured distributed names to our model as an independent primitive, and present its semantics as separate from that of authorization. In Section 6, we show how the semantics of the two constructs compose. We first present the syntax of the new constructs.

5.1 Keys

We denote the set of keys by $\mathcal{K}$. Our modeling assumes that keys are globally unique identifiers, used to authenticate principals in a distributed system. Keys can be part of symmetric or public-key cryptography systems, and should satisfy security requirements of the same. In particular, it should be hard to guess the key associated with a principal; this is usually inversely proportional to the exponential of the key length in bits. We will use this fact in our specification of the *AddKey* action in Section 5.5 by assuming that a newly generated key is unique. The intention is that every name in the system eventually resolves to a key. We assume the existence of a distinguished key $\perp_k$, which we shall use to denote a nonterminating name resolution process. Name resolution is an injective map and is formally defined in Section 5.5.

5.2 A Simple Grammar for Names

We denote the set of names by $\mathcal{N}$, which is a union of the set of local names $\mathcal{N}_\mathcal{L}$, and the set of global names $\mathcal{N}_\mathcal{G}$. The set of names $\mathcal{N}$ is distinguished from the set of keys $\mathcal{K}$, $\mathcal{K} \cap \mathcal{N} = \emptyset$.

Global Names We assume the existence of a set of universally recognized global names $\mathcal{N}_\mathcal{G}$. Such mechanisms exist in other naming schemes, for example, SDSI provides for distinguished global names. A global name g is meant to have context-independent semantics; we may assume that there exists a global name oracle, GNO, that returns the unique meaning associated with a global name.

Local Names Any principal in a distributed system can define its own set of names, creating a private *name space*. Such names are called *local names*. The set of local names consists of *base names* chosen from an appropriate language ($\mathcal{N}_\mathcal{B}$, say), and *compound names* created through *linking*; we use '.' to denote the linking operation. A local name l is therefore defined by

$$l ::= b | l.b$$

where $b \in \mathcal{N}_\mathcal{B} \bigcup \mathcal{N}_\mathcal{G}$. For example, Joe, Joe.PhoneNumber, Joe.PepsiCo, are all local names. Note that the grammar is left recursive, and indicates the natural left-to-right operational semantics of parsing a linked local name. To avoid confusion with global names, we assume that the set of base names excludes the set $\mathcal{N}_\mathcal{G}$, that is, $\mathcal{N}_\mathcal{B} \bigcap \mathcal{N}_\mathcal{G} = \emptyset$.

Fully Qualified Names Local names, by definition, require a context (name space) in which to resolve them. We say that a name is *fully qualified* if there is enough information such that it may be resolved without any further help. Fully qualified names are formally defined in Section 5.4.

5.3 Groups and Roles

While names provide a useful level of abstraction over keys, some policies become further succinct and robust with extended naming abstractions. For example, security policies may refer to a collection of named individuals, or *groups*. A group is simply a set of names. Groups are *static* entities; the members of a group do not change with time. For example, the group WITS_Attendees_2002 refers to a fixed set of named individuals. In this sense, a group expression can simply be viewed as a macro expansion into a list of names.

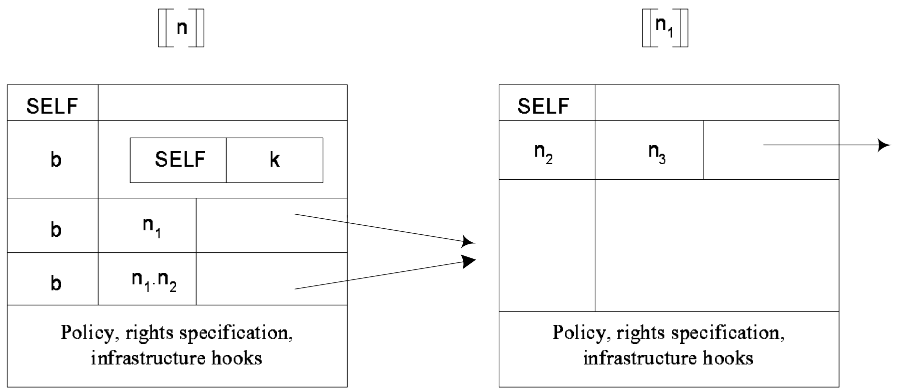
In addition, access patterns in many real-world situations cluster around the notion of a *role* [22]. A role, as opposed to a group, is a *dynamic* collection of named individuals. For example, the members of the group WITS_Attendees change every year. A role thus provides another layer of abstraction between the entity appearing in a policy statement and a key. We will see one concrete expression of this with RoleObjects in the next section.

We note that there are several constructs that may be used to derive either group or role expressions. Example constructs include linking, globality, conjunction, disjunction and other propositional connectives and functional operators. While each of these may be used to generate a group or a role, the semantic difference arises from their *evaluation order*: groups are static, while roles are dynamic.

5.4 Name Spaces

In the “real world”, the entities that names represent have both identity and the authority to name other resources. Other pieces of information, like its security policy, may also be “owned” by the entity. We model the collection of information held by an entity that is relevant to the name resolution process, by its *name space*. The entity itself has a name, which we consider to be semantically synonymous with the entity’s name space.

From the point of view of naming, what information should an entity’s name space contain? At the very least, it should be self-aware, that is, contain the name of the entity and the key to which it corresponds. In addition, it should contain local name definitions for that entity. Therefore, we may think of a name space as a record, which we call a NameObject (Figure 1), with fields for these different kinds of information. We assume that every name space has a distinguished



[[n]]
[[n1]]
SELF
b
SELF
k
b
n1
b
n1.n2
Policy, rights specification, infrastructure hooks
SELF
n2
n3
Policy, rights specification, infrastructure hooks

Figure 1: The structure of a NameObject

element, SELF, that resolves to the current key of the owner of the name space. While the SELF field can have further structure (*e.g.*, it can be a set of keys), our analysis focuses on the interaction between naming and authorization and avoids unnecessary complexity by assuming that it is a single key.

We denote the name space associated with name $n \in \mathcal{N}$ by $[\![n]\!]$. We have that, for each $n \in \mathcal{N}$,

$$\begin{aligned} [\![n]\!] &: \mathsf{NameObject} \\ [\![n]\!].\textsc{self} &: \mathcal{K} \end{aligned}$$

We use $n\downarrow$ as shorthand for $[\![n]\!].\textsc{self}$, to denote the key to which n resolves.

We now have the machinery to define fully qualified names (FQNs): a FQN is a $([\![N]\!], N.\vec{m}) \in \mathsf{NameObject} \times \mathcal{N}$ pair, where $\vec{m}$ is shorthand for a local name composed of base or global names via linking. A FQN is thus simply a local name together with the NameObject context in which to resolve it. For example, in Figure 1, $([\![n]\!], n.n_1.n_2)$ is a FQN. We denote the set of fully qualified names by $\mathcal{F}$. Also, we adopt the convention that $([\![N]\!], N)$ refers to the name space $[\![N]\!]$ itself. We will use the terms $([\![N]\!], N.\vec{m})$ and $[\![N]\!].\vec{m}$ interchangeably.

Local name definitions in a name space can now be formally modeled as elements of the set $(\mathcal{N}_{\mathcal{B}} \bigcup \mathcal{N}_{\mathcal{G}}) \times (\mathcal{K} \bigcup \mathcal{F})$. A local name definition $b \stackrel{\text{def}}{=} n$ represents the element (b, n), and defines the base name b to be n, where n is a key or a FQN; these are the only two sensible choices within our system. Requiring n to be a FQN provides the starting point of the name resolution process, and models the expectation that a local name is defined in a name space only if the name space owner knows how to resolve it. There are some restrictions on local name definitions: first, each base name may be defined only once within each name space. Circular definitions within the same name space, which by definition must involve a self-referencing set of base names, are disallowed. Finally, if b is a global name, then it can be defined only in the name space $[\![\textsf{GNO}]\!]$. For robust policies, and without losing expressive power, it is generally beneficial for the right-hand side of a name definition to be of the form $b.b$.

We note that the concept of a NameObject can be generalized to other entities that appear in local policies and credentials, such as keys, groups, or roles. Keys present no difficulty, whereas a GroupObject can simply be a collection of NameObjects. A RoleObject, which captures the dynamic nature of a role, is a structure similar to the one in Figure 1, with pointers to other RoleObjects, and provides another level of abstraction between the local policy element and the key. Local name definitions may now also be expressed in terms of these new entities.

A NameObject or RoleObject would be most naturally implemented as a small program that answers queries via Remote Procedure Call. Cryptographic integrity protection is required, either via a secure RPC channel, or via digitally signed responses to name queries for such an implementation to be secure.

5.5 A State-Transition Model of Naming

In order to extend the authorization model with names, we must first capture naming as an independent process within the state-transition model. Accordingly, we specify abstract system states that are relevant to name resolution, actions upon those states, and the name resolution process.

World State We define the world state WS_n to be the tuple

$$WS_n = (K, B, G, L, NS)$$

where K is the set of keys, B is the set of base names, G is the set of global names and L is the set of local names constructed from the sets B and G using the grammar in Section 5.2. These sets correspond directly to the sets $\mathcal{K}$, $\mathcal{N_B}$, $\mathcal{N_G}$, and $\mathcal{N_L}$, and the allowed actions in this model will maintain this correspondence. The set of name spaces, NS, consists of records, each of which has a SELF field that takes on a value in the set K, and a set of mappings $(B \bigcup G) \rightarrow (K \bigcup (NS \times L))$ that capture local name definitions. We also denote the set of fully qualified names by F, which corresponds to the set $\mathcal{F}$ described in Section 5.4; $F = NS \times L$. The world state thus captures the data structures of interest, namely, those that can represent names as per our grammar, as well as local name definitions and identity within a name space.

Actions An action specifies a transition from one world state to another. The set of possible actions corresponds to valid ways of modifying the data structures that make up a world state. Moreover, each action is *local*, and is carried out within a specific name space. For a given world state w, we specify the results of action α that takes a given vector of arguments $\vec{v}$, and is carried out in name space $[\![n]\!]$, by $[\![n]\!]\alpha(\vec{v};w)$. We assume that the components of the world state before the action are given by K,B,G,L, and NS, and by K',B',G',L', and NS' afterwards.

The set of allowed actions comprises adding and removing a global name (equivalently, changing the name space of the global name oracle appropriately), creating or deleting a name space, and modifying the SELF field or the set of mappings (local name definitions) in an element of NS. Actions that correspond to local name definitions are the most interesting, as they allow for the creation of new local names and enable name resolution; we also examine some of the other actions:

• *AddKey*: Subject n_s updates its identifying key to be k.

$$[\![n_s]\!]\text{AddKey}(k;w) = (K \cup \{k\}, B, G, L, NS'),\ \text{where}$$

$$NS' = NS[[\![n_s]\!].\text{SELF} \mapsto k]$$

Notice that by definition this action is allowed only in the name space of the subject n_s. This captures the fact that the private key (k^{-1}, say) is known only to n_s. We assume that the generated key k is not present in K; the chance of this happening in a real system is usually exponentially low in the size of the key. The notation used above for NS' states that it is the same as NS, except at the SELF field of the element $[\![n_s]\!]$, which now has the value k.

• *AddGlobalName*: The global name g is added to the system, and is associated with the key k_g.

$$[\![\mathsf{GNO}]\!]\text{AddGlobalName}(g,k_g;w) = (K \cup \{k_g\}, B, G \cup \{g\}, L', NS'),\ \text{where}$$

$$NS' = NS[[\![\mathsf{GNO}]\!].g \mapsto k_g],$$

and L' is the set of local names generated using the sets B and $G \bigcup \{g\}$. Note that a global name can be defined only by the global name oracle, and thus the action above is well-defined only in the name space $[\![\mathsf{GNO}]\!]$.

• *AddLND*: Subject n_s creates or updates the local name definition $b \overset{\text{def}}{=} n$. There are two cases, based on whether n is a key or a fully qualified name. If $n = k \in \mathcal{K}$,

$$[\![n_s]\!]\text{AddLND}(b,k;w) = (K \cup \{k\}, B \cup \{b\}, G, L', NS'),\ \text{where}$$

$$NS' = NS[[\![n_s]\!].b \mapsto k],$$

and L' is the set of local names generated using the sets $B \bigcup \{b\}$ and G. The action creates (or updates) the field b in the name space $[\![n_s]\!]$, and gives it the value k. If $n = f \in \mathcal{F}$,

$$[\![n_s]\!]\text{AddLND}(b, f; w) = (K, B \cup \{b\}, G, L', NS'), \text{ where}$$

$$NS' = NS[[\![n_s]\!].b \mapsto (\mathbf{Proj_1} f, \mathbf{Proj_2} f)],$$

and L' is as above. Consider parsing NS' for the FQN $([\![N]\!], N.\vec{m})$. The name space $[\![n_s]\!]$ is updated at the value of the field b, which is a pair of the linked name $N.\vec{m}$, and a pointer to the name space $[\![N]\!]$; this is exactly the information behind the local name definition.

Notice that name definitions are not defined without the name space that provides the evaluation environment, and thus we avoid scope extrusion problems that in conjunction with primitives like "speaks-for" have led to logical inconsistencies [1]. Moreover, there is a clear and direct interpretation of the logical predicate "says", used in other logical modelings [1] of local names, in this model. If a local name definition $b \overset{\text{def}}{=} n$ belongs in a name space n_s, this may be represented as the predicate $[\![n_s]\!].\text{SELF}$ says $(b \overset{\text{def}}{=} n)$.

Naming Judgment Given a world state WS_n, we are interested here in the following naming judgment: what key does a name resolve to? This may be captured as a logical judgment starting from a predicate equivalent of NS; here we define the semantics of the *name resolution* process, $\rightsquigarrow: \mathcal{F} \rightarrow \mathcal{K}$, operationally. Given a FQN $([\![n]\!], n.n_1.n_2 \ldots n_k)$, we say that it resolves to the following key:

$$([\![n]\!], n_1.n_2 \ \ldots \ n_k) \rightsquigarrow [\![\ldots[\![[\![n]\!].n_1]\!].n_2 \ldots]\!].n_k]\!].\text{SELF}$$

Notice that this corresponds to a left-to-right traversal of pointers corresponding to $n_1, n_2 \ldots, n_k$ in the appropriate NameObjects. In case this process encounters dangling pointers, or if one of the names $n_1, \ldots, n_k$ does not exist, we assume that the evaluation process returns the key $\bot_k$. Given a world state WS_n, the naming judgment that a fully qualified name f resolves to the key k is written $WS_n \vdash_n f \rightsquigarrow k$. In this case, we also use $f \downarrow$ for k.

In addition to associating entities with keys, names may be used to refer to passive resources like files. Since we use NameObjects to represent name spaces, we say that the *meaning* of a name is the value of the corresponding instance variable in the appropriate NameObject. Thus, the *meaning* of a key-valued name is a key, of a linked name-valued local name is the corresponding name space, and of a resource-valued name is the actual resource in question. Notice that with this identification, we may use $[\![n]\!]$ to denote the meaning of a name as well. This allows us to answer the naming judgment: what does a name mean?

Global Names A linked local name can also reference global names; we discuss this special case of name resolution here. Global names have context independent semantics, and thus are interpreted identically in all name spaces, that is,

$$\forall [\![n_1]\!], [\![n_2]\!] \in NS : g \in \mathcal{N}_{\mathcal{G}} \Rightarrow [\![n_1.g]\!] = [\![n_2.g]\!].$$

As a corollary, we get that the meaning of a global name in the empty environment, $[\![g]\!]$, is well-defined. Since we assume the existence of a global name oracle, we have that

$$\forall [\![n]\!] \in NS : g \in \mathcal{N}_{\mathcal{G}} \Rightarrow [\![g]\!] = [\![\mathsf{GNO}.g]\!] = [\![n.g]\!].$$

Thus, any FQN of the form $([\![N]\!], N.\vec{l}.g.\vec{m}), g \in \mathcal{N}_{\mathcal{G}}$ is equivalent to $([\![\mathsf{GNO}]\!], \mathsf{GNO}.g.\vec{m})$. The following facts directly follow from the above definitions:

FACT 5.1 *A FQN evaluates to the same key in any environment (name space).*

FACT 5.2 *Global names and compound names containing global names are FQNs. For a global name g and compound name $\vec{b}.g.n$, the corresponding FQN is $([\![g]\!], n)$.*

We note that previous proposals for modeling local names have focused on the meaning of a local name *definition*, which is a separate construct from the name itself. In our modeling, a local name definition $b \overset{\text{def}}{=} l$ in name space $[\![n]\!]$ corresponds to the fact that $([\![n]\!], n.b) \rightsquigarrow l.\text{SELF}$; it is an action that modifies a name space. However, the name itself denotes a name space or *evaluation context*, where evaluation may be the resolution of linked local names to keys, or certificate processing for deciding access. As described above, the context includes the identity of the name (corresponding key pairs, say) and local name definitions. It may also include information about trusted certificate authorities and other information that guides the process of creating, distributing, and verifying certificate chains.

We make a final point regarding the modeling of groups and roles in the state-transition model. We may capture the difference between them by allowing certain actions in one but not in the other. In particular, a change action (add,remove) may not be allowed on the data structure corresponding to groups, while being permitted for roles.

5.6 Embedding SPKI/SDSI

The syntactical machinery to define names in our framework is general enough to capture a variety of trust management language proposals; here we focus on expressing SPKI/SDSI 2.0 [8]. SPKI derives its specification of local names from the original SDSI proposal; names can be either “basic” or “compound”. A basic SDSI name b in n's namespace is captured by the declaration $n : (\text{name } b)$. The corresponding name in our model would be $n.b$. Compound SDSI names, referenced through chaining, can be expressed by the "." operator in our system. The compound SDSI name $n : (\text{name } b\ n_1 \ldots n_k)$ corresponds to $n.b.n_1 \ldots n_k$ in our model.

SPKI certificates capture identity (name $\mapsto$ key), attribute (permission $\mapsto$ name), or authorization (permission $\mapsto$ key) mappings. Each certificate contains an issuer, subject, delegation depth (either 0 or ∞), "authorization" expression, and validity interval. Each certificate may be signed by an entity, and we consider the body of the certificate to be a statement made inside the name space of that entity. An identity certificate $(n \mapsto k)$ can be expressed simply by associating the SELF field with k in the name space $[\![n]\!]$. SPKI does allow a local name to be bound to more than one key; as mentioned earlier, we can model this by allowing the SELF field to be a set of keys. Attribute and permission certificates are special instances of the authorization constructs in the model TM_N (Section 6), namely, $s \longrightarrow (o,r)$ and $\mathsf{Access}_\mathsf{N}(s,o,r,d)$ respectively, where subjects and object can be names or keys. The SPKI authorization expression corresponds to the permission language used to express the abstract (o,r) right, and may be application dependent. Validity intervals specify whether a SPKI certificate may be used in the current computation for deciding access, and are independent of naming and authorization concerns.

6 Authorization and Naming

We now extend the basic trust management model TM to allow for distributed, structured names. Recall that the basic access control judgment checks whether, in a given world state, a subject s can access right r on object o. The extension to naming constructs effectively replaces the occurrences of subjects with corresponding NameObjects that contain their identifying keys. We assume that objects are referenced unambiguously through subjects that own them. There are no global name spaces, and all actions and judgments take place in a particular context, or name space, of the appropriate subject. We present the different parts of the state transition modeling below.

World State The world state WS_N of the trust management system TM_N is defined to be the tuple

$$WS_\mathsf{N} = (WS_n, R, A_\mathsf{N}, D_\mathsf{N})$$

where WS_n captures the data structures relevant to naming (Section 5.5.) Note the parallel between this world state and the one for the model TM, in Section 3. The maps A_N and D_N are counterparts of the maps A and D of the system TM, and capture root-ACL and delegation mechanisms of granting access, respectively (here F is the set of fully qualified names in WS_n):

$$\begin{array}{lcl} A_{\mathsf{N}} & : & F \times R \to P(F \times \mathbb{N}) \\ D_{\mathsf{N}} & : & F \times R \times F \to P(F \times \mathbb{N}) \end{array}$$

We note the change in the domain and co-domain sets from the set of objects O (in the system TM) to F. The context-independent semantics of fully qualified names ensures that these global maps are meaningful.

Actions The actions of TM_{N} can reference local names and are therefore interpreted in the context of a name space. As before, for a given world state w, we specify the results of action α carried out in name space $[\![n]\!]$, that takes a given vector of arguments $\vec{v}$, by $[\![n]\!]\alpha(\vec{v};w)$. We assume that the components of the world state before the action are given by WS_n, R, A_{N}, and D_{N}, and by $WS_n{}', R', A'_{\mathsf{N}}$, and D'_{N} afterwards. Recall that $[\![N]\!].m$ is shorthand for the FQN $([\![N]\!], N.m)$. We focus here on the actions that modify the A_{N} and D_{N} maps:

• *Add*(to root-ACL): Subject n_s adds subject s' to the ACL associated with right r on object o, with further delegation privileges up to depth d.

$$[\![n_s]\!]\mathrm{Add}_{\mathsf{N}}(o,r,s',d;w) = (WS_n, R, A'_{\mathsf{N}}, D_{\mathsf{N}}), \text{ where}$$

$$A'_{\mathsf{N}} = A_{\mathsf{N}}[([\![n_s]\!].o, r) \mapsto A_{\mathsf{N}}([\![n_s]\!].o, r) \cup \{([\![n_s]\!].s', d)\}]$$

Note that the fully qualified names ensure that this local action updates the global map A_{N} meaningfully.

• *Remove*(from root-ACL): Subject n_s removes subject s' access to right r on object o, given that n_s owns object o.

$$[\![n_s]\!]\mathrm{Remove}_{\mathsf{N}}(o,r,s',d;w) = (WS_n, R, A'_{\mathsf{N}}, D_{\mathsf{N}}), \text{ where}$$

$$A'_{\mathsf{N}} = A_{\mathsf{N}}[([\![n_s]\!].o, r) \mapsto A_{\mathsf{N}}([\![n_s]\!].o, r) - \{([\![n_s]\!].s', d)\}]$$

• *Delegate*(access right): Subject n_s delegates its access right r on object $([\![s']\!], s'.o)$ to delegatee subject n_d, with further delegation powers d.

$$[\![n_s]\!]\mathrm{Del}_{\mathsf{N}}([\![s']\!].o, r, n_d, d; w) = (WS_n, R, A_{\mathsf{N}}, D'_{\mathsf{N}}), \text{ where}$$

$$D'_{\mathsf{N}} = D_{\mathsf{N}}[(([\![n_s]\!], \textsc{self}), r, [\![s']\!].o) \mapsto D_{\mathsf{N}}(([\![n_s]\!], \textsc{self}), r, [\![s']\!].o) \cup \{([\![n_s]\!].n_d, d)\}]$$

• *Revoke*(delegated access right): Subject n_s revokes a delegated right r on $([\![s']\!], s'.o)$, from subject n_d. The map D_{N} is appropriately modified:

$$[\![n_s]\!]\mathrm{Revoke}_{\mathsf{N}}([\![s']\!].o, r, n_d, d; w) = (WS_n, R, A_{\mathsf{N}}, D'_{\mathsf{N}}), \text{ where}$$

$D'_{\mathsf{N}} = D_{\mathsf{N}}[((\llbracket n_s \rrbracket, \textsc{self}), r, \llbracket s' \rrbracket.o) \mapsto D_{\mathsf{N}}((\llbracket n_s \rrbracket, \textsc{self}), r, \llbracket s' \rrbracket.o) - \{(\llbracket n_s \rrbracket.n_d, d)\}]$

Other actions allow for the modification of the name definitions in a name space, but these do not affect the authorization decision until they are referenced via the actions above. The actions corresponding to modifications of WS_n (see Section 5.5) are included in the set of actions in TM_{N}.

Access Judgment Given a world state w and a fully qualified object $(\llbracket n_s \rrbracket, n_s.o)$, we would like to decide whether a subject s can access right r on this object. We specify the access judgment in the system TM_{N} as a logical judgment, given the following four inference rules. These access judgment rules are derived from the rules of the base system TM (Section 3), with additional support for names. First, we transform the world state WS_{N} into its (signed) predicate equivalent:

1. $\mathsf{ACL}_{\mathsf{N}}(\llbracket B \rrbracket.A, \llbracket B \rrbracket.o, r, d)_{B\downarrow}$ holds iff $(\llbracket B \rrbracket.A, d) \in A_{\mathsf{N}}(\llbracket B \rrbracket.o, r)$, and
2. $\mathsf{Del}_{\mathsf{N}}(\llbracket A \rrbracket, \llbracket B \rrbracket.o, r, \llbracket A \rrbracket.C, d)_{A\downarrow}$ holds iff $(\llbracket A \rrbracket.C, d) \in D_{\mathsf{N}}((\llbracket A \rrbracket, \textsc{self}), r, \llbracket B \rrbracket.o)$.

The first construct above expresses the addition of A to the root ACL associated with the right r on B; notice that this action is signed by B's key. The second allows A to delegate its right r on object o to C; again this statement is signed with A's key.

$$\begin{array}{ll}
(\mathit{RootACL}_{\mathsf{N}}) & \mathsf{ACL}_{\mathsf{N}}(\llbracket B \rrbracket.A, \llbracket B \rrbracket.o, r, d)_{B\downarrow} \vdash_{\mathsf{N}} \mathsf{Access}_{\mathsf{N}}(\llbracket B \rrbracket.A, \llbracket B \rrbracket.o, r, d)_{B\downarrow} \\
(\mathit{Delegation}_{\mathsf{N}}) & \mathsf{Access}_{\mathsf{N}}(\llbracket A \rrbracket, \llbracket B \rrbracket.o, r, d+1)_{B\downarrow}, \mathsf{Del}_{\mathsf{N}}(\llbracket A \rrbracket, \llbracket B \rrbracket.o, r, \llbracket A \rrbracket.C, d)_{A\downarrow} \\
 & \vdash_{\mathsf{N}} \mathsf{Access}_{\mathsf{N}}(\llbracket A \rrbracket.C, \llbracket B \rrbracket.o, r, d)_{B\downarrow} \\
(\mathit{Ord1}_{\mathsf{N}}) & \mathsf{Access}_{\mathsf{N}}(\llbracket A \rrbracket.s, \llbracket B \rrbracket.o, r, d+1)_{B\downarrow} \vdash_{\mathsf{N}} \mathsf{Access}_{\mathsf{N}}(\llbracket A \rrbracket.s, \llbracket B \rrbracket.o, r, d)_{B\downarrow} \\
(\mathit{Ord2}_{\mathsf{N}}) & \mathsf{Del}_{\mathsf{N}}(\llbracket A \rrbracket, \llbracket B \rrbracket.o, r, \llbracket A \rrbracket.C, d+1)_{A\downarrow} \vdash_{\mathsf{N}} \mathsf{Del}_{\mathsf{N}}(\llbracket A \rrbracket, \llbracket B \rrbracket.o, r, \llbracket A \rrbracket.C, d)_{A\downarrow}
\end{array}$$

The first two rules capture the root-ACL and delegation chain mechanisms of obtaining access to a right; the second rule in particular combines a signed delegation to C made by A, with access rights granted to A by B, into an access statement for C vouched for by B. This rule therefore also acts as the equivalent of a logical inference rule for the says predicate. The signatures on the predicates provide integrity assurances by matching the signing key with the identity of the originating name space. The last two rules capture "downward closure" of delegation depths, and produce appropriate predicates for application of the delegation rule. We say that subject s' can access right r on object $(\llbracket n_s \rrbracket, n_s.o)$ iff it can produce a proof of $\mathsf{Access}_{\mathsf{N}}(\llbracket n_s \rrbracket.s', \llbracket n_s \rrbracket.o, r, d)_{n_s\downarrow}$ for some d.

6.1 Compositionality

The system TM_{N} adds names to the authorization primitives in TM, and provides logical rules that refer to both constructs. If we removed what we added — that is, if we resolved all names to keys — the system TM_{N} should behave in the same way as TM. On the other hand, the naming abstraction is powerful because it makes policies robust. We formalize these intuitions with the following two theorems: the first states that any authorization action of TM_{N} can be simulated by an action of TM, given name resolution as an auxiliary single-step operation. The second shows that changes made in a policy that references names cannot be captured with a bounded set of actions in a policy that references only keys. We make the formal comparison using simulation relations between the two labeled transition systems for TM and TM_{N}; some definitions can be found in Appendix A. For more details on labeled transition systems and simulation relations, see [18]. We start by defining a relation between world states of the two systems.

If WS_{N} is a world state of the system TM_{N}, we use $WS_{\mathsf{N}}\downarrow$ to denote the corresponding world state of the system TM where NameObjects have been replaced by their identifying keys, and the FQNs in the domain and co-domain of the $A_{\mathsf{N}}, D_{\mathsf{N}}$ maps resolved to keys to obtain the A and D maps of $WS_{\mathsf{N}}\downarrow$. We specify the translation formally below: given world state $WS_{\mathsf{N}} = (WS_n, R, A_{\mathsf{N}}, D_{\mathsf{N}})$ of TM_{N}, the corresponding world state $WS_{\mathsf{N}}\downarrow$ of TM is given by (O, R, A, D) where

1. For each $o \in O$, there exists a NameObject $[\![n]\!] \in NS$ such that $[\![n]\!].\mathrm{SELF} = o$. Recall that $WS_n = (K, B, G, L, NS)$. Also, for each NameObject $[\![n]\!] \in NS$, there exists an $o \in O$ such that $[\![n]\!].\mathrm{SELF} = o$.

2. For each $s, o \in O, r \in R, d \in \mathbb{N}$, if $(s, d) \in A((o, r))$, then there exists NameObjects $[\![n_s]\!]$, $[\![n_o]\!]$ such that $([\![n_s]\!]\downarrow, d) \in A_{\mathsf{N}}(([\![n_o]\!]\downarrow, r))$, and vice versa.

3. For each $s, o, r_s \in O, r \in R, d \in \mathbb{N}$, if $(r_s, d) \in D((s, r, o))$, then there exist NameObjects $[\![n_s]\!], [\![n_o]\!], [\![n_{r_s}]\!] \in NS$ such that $([\![n_{r_s}]\!]\downarrow, d) \in D(([\![n_s]\!]\downarrow, r, [\![n_o]\!]\downarrow))$, and vice versa.

The theorems of Section 7 tell us that the states WS_{N} and $WS_{\mathsf{N}}\downarrow$ effectively allow the same accesses in the two systems. The following theorem correlates transitions in one system to transitions in the other.

THEOREM 6.1 *The system* TM *can strongly simulate authorization actions of system* TM_{N}*, given name resolution as a one-step operation.*

Proof. Note that in order to define a simulation (Appendix A), we need to define a correspondence functor from world states of TM_{N} to TM. Since well-formed world states are created by sequences of actions, we define f inductively as follows:

$$\begin{array}{lcl} f([\![n_s]\!]\mathrm{Add}_{\mathsf{N}}(o,r,s',d;w)) & = & \mathrm{Add}([\![n_s]\!].o\downarrow,r,[\![n_s]\!].s'\downarrow,d;f(w)) \\ f([\![n_s]\!]\mathrm{Remove}_{\mathsf{N}}(o,r,s',d;w)) & = & \mathrm{Remove}([\![n_s]\!].o\downarrow,r,[\![n_s]\!].s'\downarrow,d;f(w)) \\ f([\![n_s]\!]\mathrm{Del}_{\mathsf{N}}([\![s']\!].o,r,n_d,d;w)) & = & \mathrm{Delegate}([\![n_s]\!]\downarrow,[\![s']\!]\downarrow,r,[\![n_s]\!].n_d\downarrow,d;f(w)) \\ f([\![n_s]\!]\mathrm{Revoke}_{\mathsf{N}}([\![s']\!].o,r,n_d,d;w)) & = & \mathrm{Revoke}([\![n_s]\!]\downarrow,[\![s']\!]\downarrow,r,[\![n_s]\!].n_d\downarrow,d;f(w)) \end{array}$$

The correspondence functor defines the required binary relation between world states of the two systems, as well as the translation between actions.

As a corollary, since corresponding states allow the same accesses, no unexpected authorizations are allowed by the authorization actions of TM_{N}.

THEOREM 6.2 *The system* TM *cannot simulate naming actions of* TM_{N}.

Proof. We show that a weak simulation cannot exist between the two systems. Consider a world state w_{N} of TM_{N}, such that $[\![n]\!] \in w_{\mathsf{N}}.NS$ and $[\![n]\!].\textsc{self} = k$. Also, let $[\![n]\!]$ have right r on every object in w_{N} through the A_{N} map. Let there be N objects in w_{N}. Let w be the equivalent state of TM, where $[\![n]\!]$ is replaced with k, and k has right r on every object through the A map. In order to simulate the TM_{N} action $[\![n]\!]\mathrm{AddKey}(k';w_{\mathsf{N}})$ within the system TM, we have to perform N *Remove* actions for k, one *Create* action for k', and N subsequent *Add* actions. The actions involved depend on the privileges granted to the key k in state w, and thus no weak simulation between w and w_{N} can exist.

THEOREM 6.3 (Commutativity) *For any world state WS_{N} of the system* TM_{N}, *and access formula Q_{N}, the following commutative diagram holds ($\rightsquigarrow$ represents name resolution):*

$$\begin{tikzcd} WS_{\mathsf{N}} \arrow[r, "\vdash_{\mathsf{N}}"] \arrow[d, "\rightsquigarrow"'] & Q_{\mathsf{N}} \arrow[d, "\rightsquigarrow"] \\ WS_{\mathsf{N}}\downarrow \arrow[r, "\vdash"'] & Q_{\mathsf{N}}\downarrow \end{tikzcd}$$

Proof. We proceed by induction on the structure of the proof p of $WS_{\mathsf{N}} \vdash_{\mathsf{N}} Q_{\mathsf{N}}$. We consider the last rule used in p and show that the diagram holds for each possible case. The proof of each of the four cases proceeds identically; we work out the case for $(RootACL_{\mathsf{N}})$ in detail. Suppose $\mathsf{ACL}_{\mathsf{N}}([\![B]\!].A,[\![B]\!].o,r,d)_{B\downarrow} \vdash_{\mathsf{N}}$ $\mathsf{Access}_{\mathsf{N}}([\![B]\!].A,[\![B]\!].o,r,d)_{B\downarrow}$, and that $WS_n \vdash_n ([\![B]\!],B.A) \rightsquigarrow k_1,([\![B]\!],B.o) \rightsquigarrow k_2$. By the translation from world states in TM_{N} to world states in TM, the predicate $\mathsf{ACL}_{\mathsf{N}}([\![B]\!].A,[\![B]\!].o,r,d)_{B\downarrow}$ is true of WS_{N} iff $\mathsf{ACL}(k_1,k_2,r,d)$ is true of $WS_{\mathsf{N}}\downarrow$. Now we may prove $Q_{\mathsf{N}} \downarrow = \mathsf{Access}(k_1, k_2, r, d)$ using the rule *RootACL* of the proof system $\vdash$, and we are done. We note that this inductive proof is essentially equivalent to a horizontal ladder-like constructive proof of the commutativity diagram above.

The commutativity theorem tells us that the application of name and authorization resolution rules in any order produces the same final result; it is a statement about no "surprises" from the rules $\vdash_{\mathsf{N}}$. In particular, we may resolve all names first to keys, and then use the rules $\vdash$ to decide access, yielding the final result as a composition of the two operations. More generally, we posit the commutativity theorem as a sanity benchmark for rules in access control systems that combine authorization and naming primitives: it should be possible to *separate* any access proof into name and access resolution.

7 Semantics

We extend the semantic model defined in Section 4 to include distributed, structured names. Therefore, our semantics again has an algebraic flavor, where we map elements of the world state to carrier sets and functions on them. We will denote elements of the carrier sets by nodes in a graph, and the mappings as graph edges. This is conceptually close to a basis for an actual implementation for such a system. Formally, we define a model as a directed labeled *access graph* $({O_{\mathsf{N}}}^{\mathcal{M}}, E)$, where the vertex set corresponds to the set of (named) objects and keys in the model. An access graph has two sorts of labeled edges, one related to name resolution (E^{name}) and the other to the delegation of authorization (E^{auth}). The intention is that the vertices correspond to name spaces; E^{name} edges incident on a graph vertex corresponding to $[\![N]\!]^{\mathcal{M}}$ will describe name resolution provided by the name space $[\![N]\!]$, and the incident E^{auth} edges will correspond to the A_{N} and D_{N} maps that reference $[\![N]\!]$. Following our treatment for the system TM, a semantic model also consists of the carrier sets $R^{\mathcal{M}}$, $B^{\mathcal{M}}$, $G^{\mathcal{M}}$, and $L^{\mathcal{M}}$ which correspond to the rights, and base, global, and local names allowed in the model.

Labels on graph edges allow us to combine them in order to resolve names or make an access decision:

$$\begin{array}{rcl} E^{\text{auth}} & \subseteq & {O_{\mathsf{N}}}^{\mathcal{M}} \times ((B^{\mathcal{M}} \times L^{\mathcal{M}}) \times R^{\mathcal{M}} \times \mathbb{N}) \times {O_{\mathsf{N}}}^{\mathcal{M}} \\ E^{\text{name}} & \subseteq & {O_{\mathsf{N}}}^{\mathcal{M}} \times L^{\mathcal{M}} \times {O_{\mathsf{N}}}^{\mathcal{M}} \end{array}$$

The label on an E^{auth} edge tracks the fully qualified object on which the right is being authorized, together with delegation depth d. The label on an E^{name} edge tracks a portion of a linked local name that is yet to be traversed from the destination vertex in order to resolve a local name definition in the name space of the source vertex. Formally, E is the disjoint union of edge sets E^{name} and E^{auth}. We

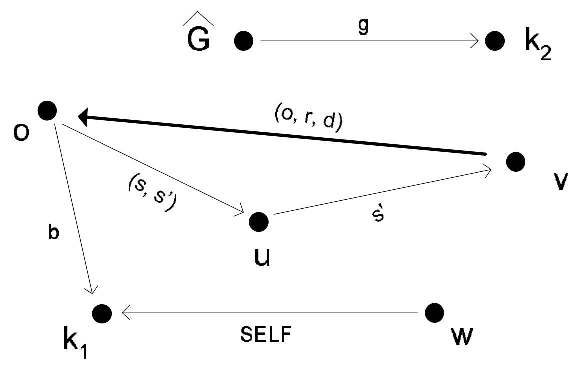


Figure 2: An example access graph $\mathcal{M}$; the heavy arrow is an E^{auth} edge, while the others are in E^{name}. Here, $\mathcal{M} \models (o.s = u.s' = v) \rightarrow (o,r,d)$. Also, the global name g maps to the key k_2, and $w \downarrow= k_1$. Notice that E^{name} edges flow in the direction of name resolution while E^{auth} edges flow in the direction of authorization resolution.

require that models satisfy the downward closure property on all edges in E^{auth}, that is, $\forall x,y \in O_{\mathsf{N}}{}^{\mathcal{M}}, o \in B^{\mathcal{M}} \times L^{\mathcal{M}}, r \in R^{\mathcal{M}}, n \in \mathbb{N}:$

$$(x,(o,r,n+1),y) \in E^{\text{auth}} \Rightarrow (x,(o,r,n),y) \in E^{\text{auth}}.$$

Given a world state WS_{N}, we say that a model $\mathcal{M}$ satisfies WS_{N}, written $\mathcal{M} \models WS_{\mathsf{N}}$, if the following conditions hold:

1. For each element $k \in K, r \in R, b \in B, g \in G, l \in L$, there exist unique elements $k^{\mathcal{M}} \in O_{\mathsf{N}}{}^{\mathcal{M}}, r^{\mathcal{M}} \in R^{\mathcal{M}}, b^{\mathcal{M}} \in B^{\mathcal{M}}, g^{\mathcal{M}} \in G^{\mathcal{M}}, l^{\mathcal{M}} \in L^{\mathcal{M}}$ respectively.

2. For each name space $[\![N]\!] \in WS_{\mathsf{N}}.NS$, there exists a vertex $[\![N]\!]^{\mathcal{M}}$ in the access graph.

3. The access graph has a distinguished vertex $\hat{G}$ such that for each element $g \in G$ where $\mathsf{GNO}(g) = k$,

$$(\hat{G}, g, k) \in E^{\text{name}}.$$

   In other words, the distinguished vertex $\hat{G}$ of the access graph resolves the global name g in a manner consistent with the world state, and in particular with the name space of the global name oracle. We note that the evaluation of global names can be considered separately from the access graph, but providing a distinguished vertex allows us to capture the entirety of name resolution within the framework of E^{name} edges in the access graph, and thus we include it here.

4. The local name definitions of WS_{N} are captured by the edge set E^{name} of the access graph. Formally,

$$\begin{aligned}
(b \overset{\text{def}}{=} k) \in [\![N]\!] \quad &\Rightarrow \quad ([\![N]\!]^{\mathcal{M}}, b^{\mathcal{M}}, k^{\mathcal{M}}) \in E^{\text{name}} \\
(b \overset{\text{def}}{=} ([\![N']\!], N'.l)) \in [\![N]\!] \quad &\Rightarrow \quad ([\![N]\!]^{\mathcal{M}}, (b^{\mathcal{M}}, l^{\mathcal{M}}), [\![N']\!]^{\mathcal{M}}) \in E^{\text{name}} \\
(b \overset{\text{def}}{=} ([\![N']\!], N')) \in [\![N]\!] \quad &\Rightarrow \quad ([\![N]\!]^{\mathcal{M}}, b^{\mathcal{M}}, [\![N']\!]^{\mathcal{M}}) \in E^{\text{name}}
\end{aligned}$$

   In other words, a local name b defined to be a key k must correspond to an E^{name} edge in the access graph with label $b^{\mathcal{M}}$. Here, we also allow b to be SELF, and thus each vertex in the graph must have an E^{name} edge, labeled with SELF, to its identifying key. A local name definition for b in terms of another name $N'.l$ is represented by an E^{name} edge between the corresponding vertices labeled by the pair $(b^{\mathcal{M}}, l^{\mathcal{M}})$. The name resolution process is guided further from $[\![N']\!]^{\mathcal{M}}$ by the structure of $l^{\mathcal{M}}$. Notice that if $l = \varepsilon$, the empty string, then the label $b^{\mathcal{M}}$ is used instead as shorthand.

5. The maps A_{N} and D_{N} are captured by the edge set E^{auth} of the access graph. Formally,

$$\begin{aligned}
([\![n_s]\!].s', d) \in A_{\mathsf{N}}([\![n_s]\!].o, r) &\Rightarrow (\hat{s'}, (\hat{o}, r^{\mathcal{M}}, d), [\![n_s]\!]^{\mathcal{M}}) \in E^{\text{auth}} \\
([\![n_s]\!].n_d, d) \in D_{\mathsf{N}}(([\![n_s]\!], \text{SELF}), r, [\![s']\!].o) &\Rightarrow (\hat{n_d}, (\hat{o}, r^{\mathcal{M}}, d), [\![n_s]\!]^{\mathcal{M}}) \in E^{\text{auth}},
\end{aligned}$$

   where $\hat{s'}, \hat{n_d}$ and $\hat{o}$ are the vertices obtained by following the appropriate E^{name} edges starting from $[\![n_s]\!]^{\mathcal{M}}$ or $[\![s']\!]^{\mathcal{M}}$ as the case may be.

We say that a vertex s can access right r on another vertex o, written $\mathcal{M} \models_{\mathsf{N}} s \to (o, r)$, iff there exists a directed *access path* of E^{auth} edges from s to o labeled with the tuple (o, r, d) such that for every prefix of the path (including the entire path), the number of edges in the prefix is $\leq 1 + d_i$, the delegation bound in the last edge of the prefix.

$$o \xleftarrow{(o,r,d_1)} s_1 \leftarrow \ldots \leftarrow s_i \xleftarrow{(o,r,d_{i+1})} s_{i+1} = s$$

LEMMA 7.1 (Model Compilation) *For any world state WS_{N} of TM_{N} and model M,*

$$\exists M'.[M \models WS_{\mathsf{N}} \iff M' \models WS_{\mathsf{N}} \downarrow] \wedge [M \models_{\mathsf{N}} s \to (o, r) \iff M' \models s \downarrow \to (o \downarrow, r)]$$

Proof. Construct the access graph of the model M' from the access graph of the model M by replacing the vertices corresponding to name spaces with the keys in their SELF fields, and the labels $([\![n_s]\!].o, r, d)$ on the E^{auth} edges by $([\![n_s]\!].o \downarrow, r, d)$. The calculation of $[\![n_s]\!].o \downarrow$ consists of traversing the appropriate E^{name} edges. Since the world state $WS_{\mathsf{N}}\downarrow$ is a replacement of the names in WS_{N} by the keys to which they resolve, the lemma follows immediately.

THEOREM 7.2 (Soundness) *Given an arbitrary world state WS_{N}, fully qualified subject $s \in F$, object $o \in O$ and right $r \in R$, $WS_{\mathsf{N}} \vdash s \rightarrow (o,r) \Rightarrow \forall \mathcal{M}.\mathcal{M} \models WS_{\mathsf{N}} \Rightarrow \mathcal{M} \models s \rightarrow (o,r)$.*

Proof. Given $WS_{\mathsf{N}} \vdash s \rightarrow (o,r)$, it follows from Theorem 6.3 that $WS_{\mathsf{N}} \downarrow \vdash s \downarrow\rightarrow (o\downarrow,r)$. Suppose $\mathcal{M} \models WS_{\mathsf{N}}$, and let $\mathcal{M}'$ be the model given by Lemma 7.1. Then, $\mathcal{M}' \models WS_{\mathsf{N}} \downarrow$, and by soundness of $\vdash$ (Corollary 4.2), we get that $\mathcal{M}' \models s \downarrow\rightarrow$ (o↓, r). The result now follows from Lemma 7.1.

THEOREM 7.3 (Completeness) *Let WS_{N} be an arbitrary world state and let $s \in F, o \in O$ and $r \in R$. Then, $[\forall \mathcal{M}.(\mathcal{M} \models WS_{\mathsf{N}}) \Rightarrow (\mathcal{M} \models s \rightarrow (o,r))] \Rightarrow WS_{\mathsf{N}} \vdash s \rightarrow (o,r)$.*

Proof. Given any $\mathcal{M} \models WS_{\mathsf{N}}$, it follows from Lemma 7.1 and the hypothesis of the theorem that $\exists \mathcal{M}'.\mathcal{M}' \models WS_{\mathsf{N}} \downarrow \wedge \mathcal{M}' \models s \downarrow\rightarrow (o\downarrow,r)$. From the completeness of the rules $\vdash$(Theorem 4.3), we get that $WS_{\mathsf{N}} \downarrow\vdash s \downarrow\rightarrow (o\downarrow,r)$. Theorem 6.3 now gives us $WS_{\mathsf{N}} \vdash s \rightarrow (o,r)$.

7.1 Avoiding Inconsistencies

A key benefit of a separate understanding of naming and authorization is the ability to evaluate new rules that may refer to either or both of these constructs, for consistency.

A case in point is Abadi's "converse of globality" axiom, which when combined with linking [1] leads to undesirable inferences about names. If g is a global name, and p is any local name, then the axiom states that $g \mapsto (p'\text{s}\ g)$. In other words, it allows p the ability to bind the global name g to $p'\text{s}\ g$. This axiom is not true under our semantics since global names cannot be redefined in local name spaces other than that of the Global Name Oracle (see Section 5.4.)

Another problematic axiom in Abadi's formulation is the "symmetry" axiom: $(p \mapsto q) \implies (q \mapsto p)$. Again, this axiom is not true in our semantics (Section 5) since two local name definitions $p \overset{\text{def}}{=} q$ and $q \overset{\text{def}}{=} p$ can never occur within the same name space.

Since we associate the meaning of a name consistently with a name space, all our actions are meaningful only within a name space. In addition, all statements used as predicates in the access judgment are vouched for by specific entities, perhaps as signatures with the keys associated with their names. The principle of remembering scope seems to have been violated surprisingly often in logical rules for access control. For example, in designing an access control logic for distributed documents [3], the authors point out two rules, *(Cont)* and *(Del)*, earlier candidates for which were found to have undesirable consequences. In both cases, an appropriately modified version of the rule was substituted. Both rules contained a scope violation; this could have been detected directly instead of in an ad-hoc manner. We strongly advocate adherence to scope as a guideline in designing logical rules for distributed systems.

8 Formally Verified Executable Specifications in PVS

So far, we have described a theoretical model of trust management with distributed, structured name spaces for subjects and shown soundness, completeness, and compositionality. We would like to carry over this clean modeling to a trusted implementation with the help of formal verification tools that provide guarantees on the properties of real functions used in the implementation. Our goals are twofold: first, to increase assurance by providing machine-checkable proofs of our main results, so that a logical inference engine like Prolog may use the inference rules as input in general decision procedures to decide access; second, to provide specialized executable procedures for deciding access for which correctness guarantees can be made within a formal system. We thus provide the basis for a formally verified reference monitor that sits at the heart of a larger system that has support for policy languages, revocation mechanisms, and certificate storage and distribution.

Our choice to use the Prototype Verification System (PVS) [20] was motivated by two reasons: from version 2.3 onward, PVS provides the ability to execute functions on ground terms; these functions can be shown to satisfy desired properties within PVS, and the compiled LISP code linked to C or Java. Second, access to local expertise accelerated the process of using a new tool. While total correctness relies on the correctness of PVS, the Common LISP compiler, and its associated runtime system, we believe that a completely automatic translation from specification to code will still yield substantially greater assurance than a manual transcription. In addition, our implementation in PVS produces proofs as witnesses of allowed accesses, which may be independently audited.

Figure 3 shows an architecture schematic, where the client (Alice, say) wishes to access file "data.doc" on the resource host. She presents her request with certificates that, in conjunction with the host's policy, will constitute a proof of access. The host application provides access control by consulting a locally running trust management daemon that manages the local name space and policy, and has information about certificate directories to query for fresh certificates, and so on. This daemon combines Alice's certificates with some of its own and produces a proof of access, or decides that access should be denied. The proof is passed through the PVS access checker, which makes sure that the proof is well-formed. In the case

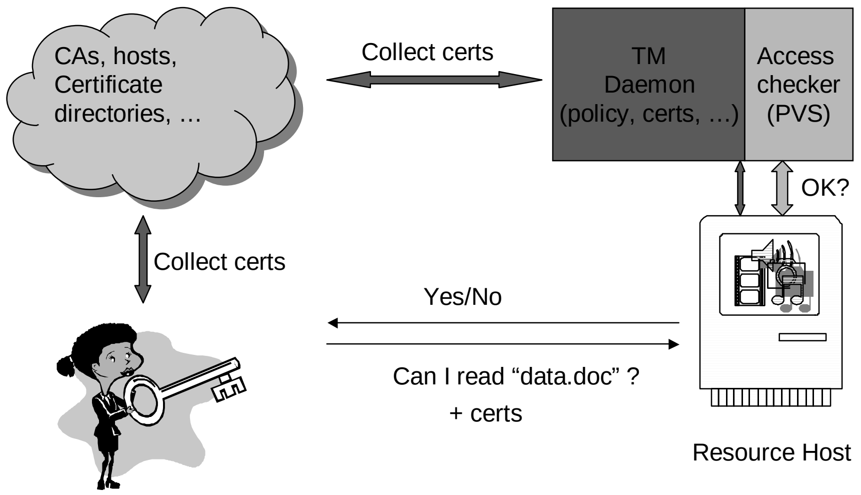


Figure 3: System Architecture and PVS

of our example trust management system TM_{N}, this proof would use the rules $\vdash_{\mathsf{N}}$, and PVS would check that the proof was valid. In addition, we have shown the commutativity theorem for the rules $\vdash_{\mathsf{N}}$ in PVS, which increases our confidence in the meta-validity of the proof, and therefore in the system itself.

Specification We make use of PVS parametrized theories and datatypes to encode world states, proofs, and graphs. The parameters are the type of objects and rights, so that specific instantiations of our theory can be used with any desired representation of real objects and rights. A world state is represented as a record with fields ob, *r*, *A*, and *D* corresponding to its syntactic components *O*, *R*, *A*, and *D* respectively (Section 3). We further condition the *A* and *D* maps such that the image of any element referencing an object in the world state specifies only objects within the world state. In the PVS code below, “object” and “right” specify types containing at least one element. The notation $[\#\ldots\#]$ denotes that “preWS” is a record with the given fields; PVS uses the ‘‘’ operator to reference elements of a record. Set membership is denoted using parentheses, for example, w‘ob(t‘1) is equivalent to $t.1 \in w.\mathrm{ob}$. The PVS type “WS” represents world states, as a subtype of preWS, and contains elements *w* of preWS such that the objects referenced in the co-domain of the *A* and *D* maps lie in *w*.ob.

```
object: TYPE+
right: TYPE+
preWS: TYPE =
[# ob: set[object],
   r: set[right],
   A: [[object, right] → set[[object, nat]]],
   D: [[object, right, object] → set[[object, nat]]] #]
Dom(w:preWS): set[[object,nat]] = {t: [object,nat] | w‘ob(t‘1)}
WS: TYPE = {w: preWS | ∀ (o_1, o_2: (w‘ob)), (r: (w‘r)):
       (w‘A(o_1, r) ⊆ Dom(w)) ∧ (w‘D(o_1, r, o_2) ⊆ Dom(w))}
```

We specify semantic models as directed labeled graphs in PVS; for this purpose we created a small parameterized theory of such graphs. A well-formed path in a model, captured by the inductively defined predicate “wfp” on paths, corresponds to legitimate access in the semantic model. In the following PVS code, “xgraph_model” is part of a theory of labeled graphs parameterized on the type of vertices and labels in the graph. The type “preG” is a record consisting of the set of vertices, edges, and labels in the graph. The PVS type “graph” denotes those elements g of preG where the edges join elements of g.vertices with labels from the set g.labels. A model is simply a graph with objects for vertices and labels of the form (o,r,d).

```
xgraph_model[V: TYPE+, L: TYPE+]: THEORY
preG: TYPE = [# vertices:set[V], edges:set[edge], labels:set[L] #]
graph: TYPE = {g: preG | ∀ (e: (g‘edges)):
          (g‘vertices)(e‘1) ∧ (g‘vertices)(e‘2) ∧ (g‘labels)(e‘3)}
model: TYPE = {G: graph |
  ∀ (u: (G‘vertices), r: right, d: nat): G‘labels(u, r, d)}
wfp(p:path, M:model, w: WS, s,O:(w‘ob), r:(w‘r), d:nat):
  INDUCTIVE bool
```

We specify proofs within our logical system as the abstract datatype of proof trees, with constructors corresponding to the empty proof, the application of the four access rules (Section 3), and the definition of the ACL predicate.

```
proof_tree[object: TYPE+, right: TYPE+]: DATATYPE
 BEGIN
  EmptyProof: empty?
  pred_acl(s: object, O: object, r: right, d: nat): acl?
  pred_del(s: object, O: object, r: right, r_s: object, d: nat): del?
  rule_rootacl(rootacl_premiss: proof_tree): rootacl?
  rule_delegation(l_del: proof_tree, r_del: proof_tree): delegation?
  rule_ord1(ord1_premiss: proof_tree): ord1?
  rule_ord2(ord2_premiss: proof_tree): ord2?
 END proof_tree

wft(t: proof_tree, w: WS, s, O: (w`ob), r: (w`r), d: nat):
  INDUCTIVE bool
```

As the reader may see, this datatype has constructors for predicate definitions and for the logical inference rules used to combine them. A well-formed proof tree, defined by the predicate "wft" on proof trees, corresponds to a valid proof of the predicate Access(s,o,r,d) using our proof rules. The predicates "models_access?" and "infer?" check whether there exist valid proofs of access within the semantic and syntactic models, respectively.

```
models_access?(M:model, w:WS, s,O:(w`ob), r:w`r, d:nat): bool
    = ∃ (p: path): wfp(p, M, w, s, O, r, d)

infer?(w: WS, s,O: (w`ob), r: (w`r), d: nat): bool
    = ∃ (t: proof_tree): wft(t, w, s, O, r, d)
```

The PVS soundness and completeness theorems will relate well-formed paths in the access graph and well-formed proof trees.

Soundness and Completeness Since soundness and completeness theorems refer to models that satisfy a given world state, we provide the function "construct_model" that constructs the minimal model satisfying a world state, and the function "models_ws?" that checks if a given model satisfies a given world state.

```
construct_model(w: WS): model =
        (# vertices := w`ob,
           edges    := construct_edges(w),
           labels   := construct_labels(w) #)
L1: LEMMA ∀(w: WS): models_ws?(construct_model(w), w)
```

For soundness, given a well-formed proof tree, we show the existence of the corresponding well-formed path in any model satisfying the world state (which must contain the constructed model.) This is done by using PVS support for induction over inductively defined data types, "proof_tree" in this case.

```
Soundness: THEOREM
    ∀ (w: WS, s, O: (w`ob), r: (w`r), d: nat):
      infer?(w, s, O, r, d) ⇒
      (∀ (M: model): models_ws?(M, w) ⇒
         models_access?(M, w, s, O, r, d))
Completeness: THEOREM
  ∀ (w: WS, s, O: (w‘ob), r: (w‘r), d: nat):
     (∀ (M: model): models_ws?(M, w) ⇒
        models_access?(M,w,s,O,r,d)) ⇒ infer?(w,s,O,r,d)
```

For completeness, given a model satisfying the world state and a well-formed path in it, we prove the existence of the corresponding well-formed proof tree, showing that access is derivable within the proof rules. The proofs proceed by using PVS induction on the structure of paths in the graph. For both these theorems, the predicates "models_access?" and "infer?" provide the existential witnesses (path and proof, respectively) to induct over.

Specialized decision procedures as PVS functions Though the soundness and completeness procedures give us guarantees about our proof rules, and their use in an automated logical inference system, in practice we would like to direct the search for a proof (or a path in the corresponding access graph) using specialized heuristics. PVS's ability to generate executable code does not presently work for specifications that use finite sets. Hence, we chose lists as an alternative representation for the world state components *O*, *R*, and maps *A* and *D*, in order to define executable functions that use them as arguments.

```
eval_on_list(w: WS_list, l: list[object] | sublist(l, w‘ob),
 O: in_objects(w), r: in_rights(w), d: nat):RECURSIVE proof_tree =
CASES l OF null: EmptyProof,
      cons(x, y): LET p: proof_tree =
         LET pA: proof_tree = in_A(delete_objects(w,y),x,O,r,d) IN
         IF empty?(pA) THEN
           eval_on_list(delete_objects(w, cons(x,y)),
             find_delegators(delete_objects(w,y),x,O,r,d), O, r, d+1)
         ELSE pA
         ENDIF
         IN IF empty?(p) THEN eval_on_list(Delete(w,x), y, O, r, d)
              ELSE p ENDIF
ENDCASES
MEASURE length(l)
```

The PVS function above searches for a proof of access given a world state and an access request. It is defined as a recursive function on the size of the world state, length(l), which decreases with each recursive call. It first tries to see ("in_A") if the requested access is simply an application of the *(RootACL)* and (*Ord1*) rules, else it recurses on potential delegators ("find_delegators"). An "EmptyProof" proof tree is returned if neither of these cases finds a proof. Essentially, the function performs backward search from the goal, and returns a proof tree that can be checked for well-formedness, thereby showing soundness of the decision procedure. Completeness of this procedure follows from an inductive proof on the structure of paths, which shows that if no path is returned by the procedure, then no well-formed path can exist in the access graph.

Actions In order to use PVS functions on ground terms that represent actual world states, we may either write them out by hand, or create them using sequences of actions. The latter method is more robust, and has the benefit of creating well-formed world states. Therefore, we created a separate theory to specify the effect of actions on the world state (Table 1), as well as an "initial" world state containing only a "supervisor" object. A reachable world state is one that can be reached only through an arbitrary sequence of actions, and it makes sense to apply evaluation functions only on such states.

There were two main technical difficulties while doing the PVS proofs: first, since access in a model happens either via a root-ACL mechanism, or through a delegation chain ending in a root-ACL, we needed a way to distinguish edges in the model that corresponded to the A and D maps, respectively. A more challenging issue was to deal with the "downward-closure" of the delegation depth (see rules (*Ord1*) and (*Ord2*), Section 3). This corresponds to a nested induction on delegation depths within the outer induction on proof trees and paths, which we factored out into two separate lemmas.

In addition, we experimented with various representations of the data structures and lemmas in our theory to optimize the human effort involved in proving the main theorems. This included trying out dependent types for the A and D maps in the world state, alternate inductive definitions for proof-trees and paths, varying quantification order on the lemmas so that built-in PVS proof commands could be used optimally, and factoring out common subproofs. The valuable tool experience gained with the proof of the soundness theorem directly helped us to achieve a clean, modular decomposition of the completeness theorem, and its proof was also accomplished much more quickly.

Our soundness and completeness theorems make no assumptions on the size of world states or models. Our induction hypothesis uses the correspondence between subpaths and subproof trees, and does not constrain their size. The exacting nature of these proofs, however, informed our theory by requiring us to specify our informal assumptions, such as the downward closure property on access graphs (see

Section 4.1), which we had initially left unspecified.

Incorporating names in PVS The second part of our PVS implementation extends the base system with naming constructs, and its specifications and proof techniques are very similar to those described above. We encode the world state for names, WS_n, using the PVS record type with five fields corresponding to the syntactic components of the world state. Name resolution is implemented as a recursively defined function that uses the local name definitions (represented as maps) in elements of the set of name spaces. The world state for the combined system TM_{N} is then defined by adding a set of rights, and the maps A_{N} and D_{N} on fully qualified names. The proof rules $\vdash_{\mathsf{N}}$ are represented as an inductive datatype that captures the predicates used in constructing proofs and the inference rules for combining them. Semantic models are constructed as access graphs, suitably instantiated with name spaces and keys as vertices, and a tag on the edges to distinguish them. The commutativity theorem uses PVS induction on proof trees in TM_{N} and constructs corresponding proof trees of TM.

The interested reader may find further details on our PVS effort, including complete theory specifications, at `http://www.csl.sri.com/programs/security/jcs/toc.html`.

9 Conclusion

We have designed and formally verified a trust management kernel, for which the separation of naming and authorization led to a clean and simple formalization. This kernel acts as a reference monitor that determines whether a given access request is authorized based on policy and supporting credentials. Using access graphs as a semantic model, we show soundness and completeness of the authorization system without names. The orthogonality of naming and authorization is captured precisely in a commutativity theorem, which also gives us simple soundness and completeness proofs for the entire kernel. The kernel is formally verified in PVS, allowing for the automatic generation of a verified implementation of a reference monitor. By separating naming and authorization primitives, we arrive at a compositional model and avoid primitives such as “speaks-for” that have proven troublesome in the past. The simplicity of the soundness and completeness proofs for the full system suggests that a composite approach can be extended to reasoning about other features of a trust management system such as policy language constructs and revocation.

Using PVS, we can automatically generate executable code for the reference monitor core from our formal specification. We believe that NameObjects can be effectively implemented via small Java programs that act as XML RPC servers. Much of this code will be mechanically generated. For cryptographic security, there are off-the-shelf TLS implementations in Java as well. While there is still room for error, the assurance level of this system should be substantially higher than most: we have a formally verified specification, safe programming languages, a well-analyzed cryptographic protocol, and only a small amount of handwritten code.

References

[1] Martín Abadi. On SDSI's linked local name spaces. *Journal of Computer Security*, 6(1,2):3–21, 1998.

[2] Martín Abadi, Michael Burrows, Butler Lampson, and Gordon Plotkin. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(4):706–734, September 1993.

[3] Dirk Balfanz, Drew Dean, and Mike Spreitzer. Security infrastructure for distributed Java applications. In *Proceedings of the 2000 IEEE Symposium on Research in Security and Privacy*, pages 15–26. IEEE Computer Society press, May 2000.

[4] M. Blaze, J. Feigenbaum, and A. D. Keromytis. KeyNote: Trust management for public-key infrastructures. In *Proceedings of the 1998 Security Protocols Workshop*, volume 1550 of *Lecture Notes in Computer Science*, pages 59–63, 1999.

[5] M. Blaze, J. Feigenbaum, and M. Strauss. Compliance checking in the PolicyMaker trust management system. In *FC: International Conference on Financial Cryptography*, volume 1465 of *Lecture Notes in Computer Science*, pages 254–274, 1998.

[6] Matt Blaze, Joan Feigenbaum, and Jack Lacy. Decentralized trust management. In *Proceedings of the 1996 IEEE Symposium on Research in Security and Privacy*, pages 164–173. IEEE Computer Society Press, May 1996.

[7] Ajay Chander, Drew Dean, and John C. Mitchell. A state-transition model of trust management and access control. In *Proceedings of the 14th IEEE Computer Security Foundations Wokshop*, pages 27–43, June 2001.

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

[9] M. Gasser, A. Goldstein, C. Kaufman, and B. Lampson. The digital distributed system security architecture. In *Proceedings of the 12th NIST-NCSC National Computer Security Conference*, pages 305–319, 1989.

[10] John Gordon. The Alice and Bob after-dinner speech. Invited lecture given at the Zurich Seminar, April 1984. Available at `http://www.conceptlabs.co.uk/alicebob.html`.

[11] J. Halpern and R. van der Meyden. A logic for SDSI's linked local name spaces. *Journal of Computer Security*, 9(1,2):75–104, 2001.

[12] J. Halpern and R. van der Meyden. A logical reconstruction of SPKI. *Proceedings of the 14th IEEE Computer Security Foundations Workshop*, pages 59–70, June 2001.

[13] M. Harrison, W. Ruzzo, and J. Ullman. Protection in operating systems. In *Communications of the ACM*, pages 461–471. ACM, August 1976.

[14] Butler Lampson. Protection. In *Proceedings of the 5th Annual Princeton Conference on Information Sciences and Systems*, pages 437–443, Princeton University, 1971.

[15] Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber. Authentication in distributed systems: Theory and practice. In *William Stallings, Practical Cryptography for Data Internetworks*. IEEE Computer Society Press, 1996. Based on article in ACM Transactions on Computer Systems, November 1992.

[16] Ninghui Li. Local names in SPKI/SDSI. *Proceedings of the 13th Computer Security Foundations Workshop*, pages 2–15, July 2000.

[17] Richard J. Lipton and Lawrence Snyder. A linear time algorithm for deciding subject security. *Journal of the ACM*, 24(3):455–464, 1977.

[18] Robin Milner. *Operational and Algebraic Semantics of Concurrent Programs*, volume B of *Handbook of Theoretical Computer Science*, chapter 19, pages 1201–1242. Elsevier Science Publishers B.V., Amsterdam, The Netherlands, 1990.

[19] B. Clifford Neuman. Proxy-based authorization and accounting for distributed systems. In *International Conference on Distributed Computing Systems*, pages 283–291, 1993.

[20] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Language Reference, Version 2.3.* SRI International, September 1999. Available from `http://pvs.csl.sri.com/`.

[21] Ron Rivest and Butler Lampson. SDSI–A Simple Distributed Security Infrastructure. Available at `http://theory.lcs.mit.edu/~rivest/sdsi11.html`, October 1996.

[22] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, February 1996.

[23] Edward Wobber, Martín Abadi, Michael Burrows, and Butler Lampson. Authentication in the Taos operating system. *ACM Transactions on Computer Systems*, 12(1):3–32, 1994.

A Simulation Relations

A Labeled Transition System (LTS) over a set of actions *Act* is a pair $(\mathcal{Q}, \mathcal{T})$ consisting of

1. A set of states $\mathcal{Q}$
2. A ternary relation $\mathcal{T} \subseteq (\mathcal{Q} \times Act \times \mathcal{Q})$ called the transition relation

Elements (p, α, p') of the transition relation are also denoted by $p \xrightarrow{\alpha} p'$.

DEFINITION A.1 *Strong (one-step) simulation relation*

Let $\mathcal{L} = (\mathcal{Q}, \mathcal{T})$ be an LTS over a set of actions *Act*, and let $\mathcal{L}' = (\mathcal{Q}', \mathcal{T}')$ be another LTS over the set of actions Act'. Let S be a binary relation between $\mathcal{Q}$ and $\mathcal{Q}'$, $S \subseteq \mathcal{Q} \times \mathcal{Q}'$. Further, let t be a mapping from actions in *Act* to those in Act'. Then S is called a strong simulation relation over $\mathcal{L}$ and $\mathcal{L}'$ if, whenever pSp', if $p \xrightarrow{\alpha} q$, then there exists $q' \in \mathcal{Q}'$ such that $p' \xrightarrow{t(\alpha)} q'$ and qSq'. We say that p' strongly simulates p if there exists a strong simulation S such that pSp'.

DEFINITION A.2 *Weak (many-step) simulation relation*

Let $\mathcal{L} = (\mathcal{Q}, \mathcal{T})$ be an LTS over a set of actions *Act*, and let $\mathcal{L}' = (\mathcal{Q}', \mathcal{T}')$ be another LTS over the set of actions Act'. Let S be a binary relation between $\mathcal{Q}$ and $\mathcal{Q}'$, $S \subseteq \mathcal{Q} \times \mathcal{Q}'$. Further, let t be a mapping from actions in *Act* to *sequences of* one or more actions in Act'. Then S is called a weak simulation relation over $\mathcal{L}$ and $\mathcal{L}'$ if, whenever pSp', if $p \xrightarrow{\alpha} q$, then there exists $q' \in \mathcal{Q}'$ such that $p' \xrightarrow{t(\alpha)} q'$ and

qSq'. It is assumed that $t(\alpha)$ depends only on α and is independent of p and p'. In other words, the action α in the first LTS is always simulated by the sequence of actions $t(\alpha)$ in the second LTS. We say that p' weakly simulates p if there exists a weak simulation S such that pSp'.