Distributed Programming with Distributed Authorization *

Kumar Avijit
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
kavijit@cs.cmu.edu

Anupam Datta
CyLab
Carnegie Mellon University
Pittsburgh, PA 15213
danupam@cmu.edu

Robert Harper
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
rwh@cs.cmu.edu

Abstract

We propose a programming language, called PCML$_5$, for building distributed applications with distributed access control. Target applications include web-based systems in which programs must compute with stipulated resources at different sites. In such a setting, access control policies are *decentralized* (each site may impose restrictions on access to its resources without the knowledge of or cooperation with other sites) and *spatially distributed* (each site may store its policies locally). To enforce such policies PCML$_5$ employs a distributed proof-carrying authorization framework in which sensitive resources are governed by reference monitors that authenticate principals and demand logical proofs of compliance with site-specific access control policies. The language provides primitive operations for *authentication*, and *acquisition* of proofs from local policies. The type system of PCML$_5$ enforces locality restrictions on resources, ensuring that they can only be accessed from the site at which they reside, and enforces the authentication and authorization obligations required to comply with local access control policies. This ensures that a well-typed PCML$_5$ program cannot incur a runtime access control violation at a reference monitor for a controlled resource.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Type structure; D.4.6 [*Operating Systems*]: Security and Protection—Access Controls, Authentication

General Terms Languages, Design, Theory, Security

Keywords Distributed programming, proof-carrying authorization, authorization logic, logical frameworks, phase distinction

1. Introduction

We present a programming language for writing distributed programs that compute with resources spread across multiple sites on a network. Each site on the network is assumed to be administered by a principal who wishes to control access to all the resources at that site.

We consider an approach based on distributed proof-carrying authorization (DPCA), which was first introduced by Appel and Felten (Appel and Felten 1999), and developed further in subsequent work (Bauer et al. 2001, 2005; Garg 2009). There are two key ingredients to the DPCA approach. First, the policies specifying the conditions under which a principal may access a resource are formulated as theories in a formal logic. The exact form of the logic varies from one setting to another, but must be expressive enough to admit reasoning from the perspective of a principal using judgments of the form "k `affirms` p". This expressivity is required when not all resources are controlled by the same principal. The second component of DPCA is the use of proof-carrying reference monitors to control access to resources. With each access request to a resource, its reference monitor demands a formal proof stating that the controlling principal at the site affirms that the accessing principal is permitted to access the resource. The burden of producing such a proof from the local policies is thus on the accessing principal, and not on the reference monitor. The reference monitor simply verifies the proof before granting access.

The security of the DPCA approach relies on the integrity of reference monitors and the soundness of the authorization logic. It is assumed that an attacker cannot access the resource in any manner other than having to go through the reference monitor. It is further assumed that the proof-checker of the reference monitor is sound in the sense that it accepts only those proofs that are valid under the host's policies. The effectiveness of the reference monitor depends, in turn, on the expressiveness of the authorization logic. In particular, the `affirms` judgment is essential to stating distributed authorization policies. For example, if k controls access to a resource r, then an access control entry for r has to be of the form "k `affirms` $\texttt{mayrd}(r, k')$", stating that the principal k' is permitted by the principal k to access r. The logic should not allow for deductions of "k `affirms` p" for propositions p that were not intended by k. Such properties have been investigated by Garg and Pfenning in context of a particular authorization logic (Garg and Pfenning 2006).

To facilitate writing non-malicious code in such a setting, we propose an extension to ML$_5$ (Murphy 2008; Murphy et al. 2007), which is a distributed programming language, with support for distributed proof-carrying authorization. The ML$_5$ type system supports distributed programming by using a modal interpretation to distribution of resources. In particular, the ML$_5$ type system ensures that resources on a network can only be accessed at the site where they reside, even though references to these resources may be distributed freely. This is achieved through the typing judgment $m : A@w$ stating that the computation m may be executed at the site w. ML$_5$ itself only supports distribution, but not authorization.

* The material is based on work supported in part by NSF grants 0716469, and CCR-0424422 (TRUST). The second author was also supported in part by the Army Research Office through grant DAAD19-02-1-0389 to Carnegie Mellon CyLab.

TLDI'10, January 23, 2010, Madrid, Spain.

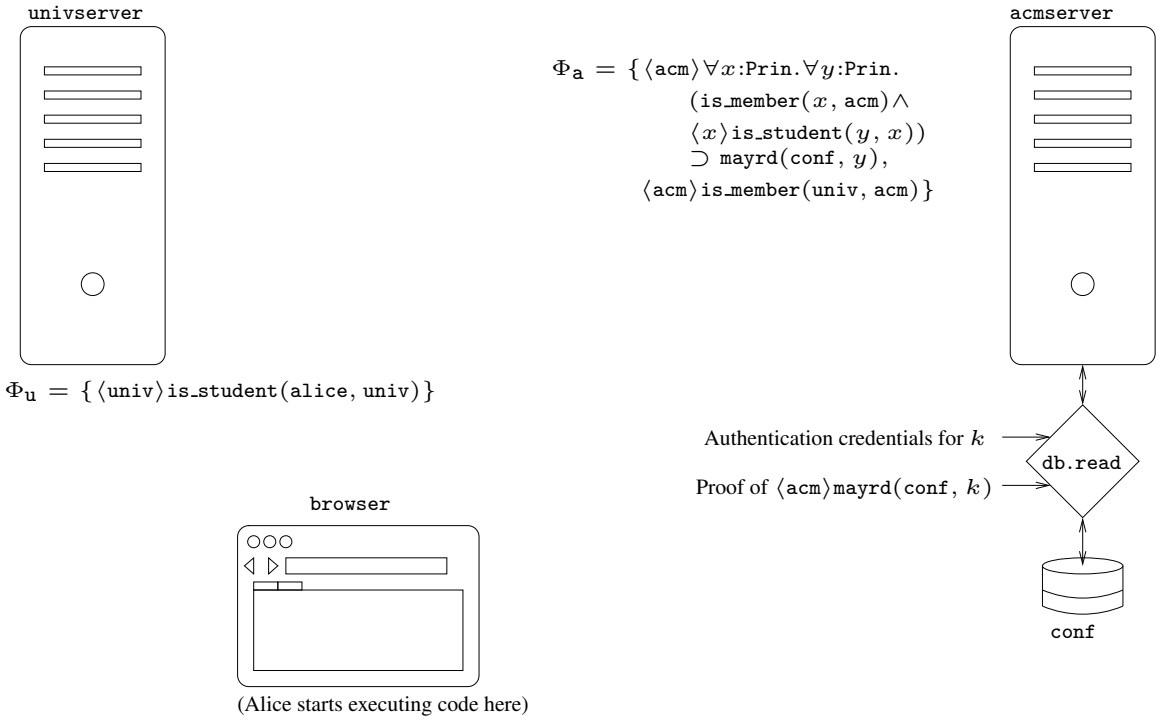

Figure 1. An example scenario for distributed PCA

The contribution of this paper is an extension of the ML_5 type system, to statically enforce the proof requirements imposed by proof-carrying reference monitors that control the resources on the network. There are three main ingredients in this extension:

1. **Authentication**: PCML_5 provides a primitive operation that allows the executing program to authenticate as a principal, obtaining credentials that are presented to the reference monitor in order to identify the accessing principal.
2. **Distributed policy acquisition**: We assume that access control policies are distributed on the network, and must be acquired dynamically to construct the proofs demanded by the reference monitors at a site. As a simple example, a program must consult the access control list for a resource at a site to determine whether a specified principal is permitted to access a specified resource in a particular manner. We enrich PCML_5 with a primitive for doing deductions using local authorization policies at the sites.
3. **Specification of proof obligations imposed by reference monitors**: Each resource is governed by an API whose type specifies all the proof demands imposed by the reference monitor. In particular, the type for a resource r specifies that the reference monitor must be presented with the following:
   - A credential that witnesses authentication of the accessing principal k.
   - A proof in the authorization logic that k is permitted by the governing authority to access r.
   - Any other arguments appropriate to the resource and the particular mode of access.

The main result of the paper is the authorization theorem which states that well-typed programs cannot incur authorization failure at the reference monitor governing the resource. This theorem holds because the type system ensures that the accessing principal must provide the proof required by the reference monitor, which therefore cannot fail at execution time. This imposes the obligation on the programmer to collect sufficient credentials and derivations prior to arriving at the API call-site. This process of collecting credentials might itself fail. For instance, the program may query the host policy to determine if the principal k is allowed access to a resource r, i.e., attempt to prove that k' `affirms` $\mathtt{mayrd}(r, k)$, where k' is the controlling authority of r. This query can fail, if the indicated access is not permitted. However, if the query succeeds, the program acquires evidence that can be used in further deductions that are presented to the reference monitor. An analogy with array bounds check may be helpful here. One may envision a language in which the ability to perform a subscript operation on an array demands a proof that the given index is within bounds. It is incumbent on the program to acquire, through a combination of run-time checks and logical deductions, a proof that the proposed access is legitimate. The run-time checks used to assemble the ingredients of these proofs may fail, but the language can ensure that the proof presented at the access site will always be valid, and hence proof-checking at reference monitor cannot fail.

Figure 1 illustrates the use of PCML_5 for authorized, distributed computation. There are three sites on the network, viz., `browser`, `acmserver` and `univserver`, and `acmserver` hosts a database `conf`. Access to `conf` at `acmserver` is governed by the principal `acm`. The site `univserver` is administered by `univ`. The local policy of each site w is specified as a logical theory Φ_w, which is a set of propositions of the form $\langle k\rangle P$, where k is the governing authority at w. We use the modal proposition $\langle k\rangle P$ as the internalization of the judgment k `affirms` P. The database `conf` is protected by a reference monitor that demands an authentication credential, and a proof of the proposition $\langle \mathtt{acm}\rangle \mathtt{mayrd}(\mathtt{conf}, k)$ with k being the principal authenticated.

We wish to write a distributed program that starts execution at `browser`, travels to the site `acmserver`, and accesses the database `conf` with appropriate authentication and authorization proofs. The

first step is to authenticate the program as running on behalf of a principal using PCML$_5$'s authentication primitive. This generates a credential that will be used to prove the identity of accessing principal at the reference monitor. The rest of the program can be written as if executing on behalf of this principal, say k. The next step is assembling a proof for $\langle \texttt{acm} \rangle \texttt{mayrd}(\texttt{conf}, k)$. The program does this by first proving that k is a student of `univ`, i.e., by obtaining a proof for $\langle \texttt{univ} \rangle \texttt{is\_student}(k, \texttt{univ})$, and then using this proof to deduce that `acm` permits k to read `conf`. First of all, the program moves to `univ` to do a local proof search for $\langle \texttt{univ} \rangle \texttt{is\_student}(k, \texttt{univ})$. For the particular authorization theories in the example, this proof search fails unless k is `alice`. In case of failure, the program simply aborts. However if the query is successful, the program moves to `acmserver` to deduce

$$\langle \texttt{univ} \rangle \texttt{is\_student}(k, \texttt{univ}) \supset \langle \texttt{acm} \rangle \texttt{mayrd}(\texttt{conf}, k)$$

from the local policy. This proof check succeeds under the theory $\Phi_{\texttt{a}}$. The two locally deduced proofs are moved to `acmserver` where they are combined using an $\supset$-elimination to deduce

$$\langle \texttt{acm} \rangle \texttt{mayrd}(\texttt{conf}, k)$$

This proof, along with the authentication credential for k is passed to the API for reading `conf`. Provided that the previous local deductions at `univserver` and `acmserver` succeed, it is guaranteed that the proof is accepted by the reference monitor.

2. Authorization logic

The presentation of PCML$_5$ in this paper uses the GP authorization logic (Garg and Pfenning 2006), which we summarize in this section. However, we emphasize that we pick a specific logic for illustration only. Since the logic is integrated into the type system using higher-order abstract syntax (as described in Section 5), the PCML$_5$ language can be easily coupled with other authorization logics that can be encoded in the framework.

Sorts	s	::=	$\texttt{prin} \mid \texttt{db}$
First-order terms	a	::=	$\alpha \mid \mathbf{a}$
Predicate symbols	p		
Propositions	P	::=	$P_1 \supset P_2 \mid P_1 \wedge P_2$
		$\mid$	$\forall \alpha{:}s.P \mid \langle a \rangle P \mid p(a_1, \ldots, a_n)$
Basic judgments	J	::=	$a{:}s \mid P\ \texttt{true} \mid a\ \texttt{affirms}\ P$
Proposition context	Φ	::=	$\cdot \mid \Phi, P\ \texttt{true}$
Term context	$\mathbf{\Delta}$	::=	$\cdot \mid \mathbf{\Delta}, \alpha{:}s$
Signature	$\mathbf{\Sigma}$	::=	$\cdot \mid \mathbf{\Sigma}, \mathbf{a}{:}s \mid \mathbf{\Sigma},\ p{:}(s_1, \ldots, s_n)$

The logic is parameterized by a signature $\mathbf{\Sigma}$ that fixes the arity of the predicates; $p{:}(s_1, \ldots, s_n)$ denotes that p is a predicate of arity n with s_i being the sort of its ith argument. There are atleast two sorts, viz., `prin` and `db`, representing principals and databases respectively. The signature also fixes principal and database constants $\mathbf{a}$.

Apart from the standard propositional connectives, the logic features a family of modalities indexed by principals, to express propositions affirmed by principals. The modal proposition $\langle a \rangle P$ (read as "a says P") expresses that proposition P is endorsed by principal a. There are three forms of basic judgments: the sorting judgment $a{:}s$, which means that the first-order term a is of the sort s; the truth judgment $P\ \texttt{true}$; and the affirmation judgment $a\ \texttt{affirms}\ P$, which means that a (a term of the sort `prin`) endorses P.

We use $\mathbf{\Delta}; \Phi \vdash^{\mathcal{L}}_{\mathbf{\Sigma}} J$ to mean the hypothetical judgment J from assumptions $\mathbf{\Delta}$ and Φ, under the signature $\mathbf{\Sigma}$. The superscript $\mathcal{L}$ differentiates entailment in the logic from that in hypothetical judgments of PCML$_5$ which we shall introduce later. We collect hypotheses about sorting judgments $\alpha{:}s$ under the context $\mathbf{\Delta}$; hypotheses of truth judgments $P\ \texttt{true}$ are written as Φ. We do not need to form hypotheses about affirmation judgments in this logic.

$$\frac{\mathbf{\Delta}; \Phi \vdash^{\mathcal{L}}_{\mathbf{\Sigma}} P\ \texttt{true} \qquad \mathbf{\Delta}; \Phi \vdash^{\mathcal{L}}_{\mathbf{\Sigma}} a : \texttt{prin}}{\mathbf{\Delta}; \Phi \vdash^{\mathcal{L}}_{\mathbf{\Sigma}} a\ \texttt{affirms}\ P}(\texttt{truaff})$$

$$\frac{\mathbf{\Delta}; \Phi \vdash^{\mathcal{L}}_{\mathbf{\Sigma}} a\ \texttt{affirms}\ P}{\mathbf{\Delta}; \Phi \vdash^{\mathcal{L}}_{\mathbf{\Sigma}} \langle a \rangle P\ \texttt{true}}(\langle\rangle\text{-I})$$

$$\frac{\mathbf{\Delta}; \Phi \vdash^{\mathcal{L}}_{\mathbf{\Sigma}} \langle a \rangle P_1\ \texttt{true} \qquad \mathbf{\Delta}; \Phi, P_1\ \texttt{true} \vdash^{\mathcal{L}}_{\mathbf{\Sigma}} a\ \texttt{affirms}\ P_2}{\mathbf{\Delta}; \Phi \vdash^{\mathcal{L}}_{\mathbf{\Sigma}} a\ \texttt{affirms}\ P_2}(\langle\rangle\text{-E})$$

Figure 2. Selected rules from natural deduction for the authorization logic. (Reproduced from (Garg and Pfenning 2006)).

In order to illustrate the *says* modality, we present selected rules from a natural deduction for the logic in Figure 2. Rule (`truaff`) says that true propositions are affirmed by all principals. Rule ($\langle\rangle$-I) internalizes the judgment $a\ \texttt{affirms}\ P$ as the proposition $\langle a \rangle P$. Rule($\langle\rangle$-E) eliminates the modality in $\langle a \rangle P_1$ by using a proof of $\langle a \rangle P_1$ to cut a hypothesis $P_1\ \texttt{true}$ in a derivation of affirmation $a\ \texttt{affirms}\ P_2$ by the same principal a.

3. Distributed policies and proof-checking

In our setup, each site is governed by an administrative principal who decides the authorization policy at that site. We shall use the notation $\bar{w}$ to denote the governing principal at site w. We model the local policy as a context of propositions declared to be true by the governing principal. From a global perspective, a proposition P declared by the principal $\bar{w}$ has the force of the proposition $\langle \bar{w} \rangle P$. Thus the local policy at w, called Φ_w, is a context of the form:

$$\Phi_w ::= \langle \bar{w} \rangle P_1\ \texttt{true}, \ldots, \langle \bar{w} \rangle P_n\ \texttt{true}$$

A proof presented to a reference monitor may contain facts from local policies at various sites. Logically, such a proof is valid under the union of local policies at all sites. We refer to this union as the *amalgamated* authorization policy and denote it by

$$\Phi = \bigcup_{w \in \texttt{Wld}} \Phi_w$$

where `Wld` is the set of all sites. We use the same notation for amalgamated policy as a proposition context from Section 2 in order to highlight that a policy is just a context in the logic. In implementations, evidence for the local policy at a site is provided by digitally signed certificates, which can be verified by reference monitors at any site. A certificate for proposition P signed by a principal a is unforgeable and establishes the judgment $\langle a \rangle P\ \texttt{true}$ irrefutably. Verification of a proof constructed from such evidences requires checking the validity of these certificates. We view these evidences as implementing the amalgamated authorization policy. Therefore, our formalism directly uses the full policy Φ for the purpose of proof-checking at reference monitors.

In practical distributed authorization systems for trust management (Blaze et al. 1996) and other applications (Bauer et al. 2005) policy statements made by an administrator at its site w are often cached at another site w' in order to aid distributed proof construction. For example, consider the authorization policy at `univserver` from Figure 1. The evidence for $\langle \texttt{univ} \rangle \texttt{is\_student}(\texttt{alice}, \texttt{univ})$ may be cached at Alice's machine `browser`. In this paper, we do not consider caching of proofs

at sites. The policy available at a site is restricted to only consist of declarations made by the principal governing that site.

4. Overview of PCML$_5$

We shall now informally discuss the key ideas behind PCML$_5$. There are three main aspects of PCML$_5$:

1. PCML$_5$ is a language for writing distributed programs that interact with resources which are distributed across different administrative domains. The type system tracks the site where a computation is executed. This allows the language to enforce the requirement that a resource can be accessed only from the site where it is situated.
2. PCML$_5$ uses an authorization logic as a part of its type system. The logic is used to express within the language authorization proof obligations that are imposed by reference monitors.
3. PCML$_5$ allows construction of proofs by combining the results of a distributed proof search with proof constructors from authorization logic. The type system statically ensures that proofs passed to a reference monitor satisfy its proof specifications. This ensures that a call made to a reference monitor from a well-typed PCML$_5$ program never incurs a runtime error due to failure of verification of proof.

We now elaborate on each of these aspects.

4.1 Distribution of computation

PCML$_5$ is based on ML$_5$ (Murphy 2008; Murphy et al. 2004), which is a distributed programming language based on propositions-as-types interpretation of the modal logic Intuitionistic S5 where modal worlds are interpreted as sites in a distributed system. The central idea that PCML$_5$ inherits from ML$_5$ is that terms are classified using types *relative* to sites. The typing judgment $\Gamma \vdash m : A@w$ means that under the hypotheses about variables in Γ, the term m is typed as A for the site w where it is to be evaluated.

The idea of typing relative to worlds is central to writing distributed programs since it allows the type system to track where different pieces of code are meant to execute. A term m for the site w' can be remotely executed from a world w by doing a $\texttt{get}[w']m$ at w. This causes computation to move to w' to execute m, and the result is brought back to w, provided type of the result is mobile. We discuss mobility later in Section 5.3 after having introduced the formalism.

4.2 Authorization logic

PCML$_5$ incorporates an authorization logic as static constructors using higher-order abstract syntax. Constructors are classified by kinds. Propositions belong to the kind `Prop` and proofs of a proposition P are kinded as $\texttt{Prf}(A)$ where A is the representation of the proposition P. Proof of an affirmation judgment a `affirms` P is represented as a constructor of kind $\texttt{Affirms}(A_1, A_2)$, where A_1 represents the principal a, and A_2 is the representation of P. As we shall see later, this translation provides a compositional bijection between propositions, proofs, and affirmations, on one hand, and constructors of kind families `Prop`, `Prf`, and `Affirms`, on the other.

The main requirement from this translation is to be able to reflect the consequence relation of logic in an adequate manner using PCML$_5$ as is done in LF (Harper et al. 1993). Thus if there is a proof of P_2 hypothetical in a proof of P_1 in the logic, one can write a constructor of kind $\texttt{Prf}(A_2)$ with a free variable of kind $\texttt{Prf}(A_1)$ in the language, and vice versa, where A_i's are the representations of propositions P_i's. We shall make this relation precise once we introduce the formalism.

In order to ensure that adequacy is without doubt, we follow a phase distinction (Harper et al. 1990) between static and dynamic phases of PCML$_5$. The constructor level is constrained to be pure in the sense that constructors do not depend on runtime terms. This way terms can be arbitrarily effectful, but they do not affect the types in any way.

4.3 Representation of principals and resources

Principals and resources appear in propositions and proofs, and also occur as runtime values. In order to maintain a phase distinction, we differentiate between the runtime and the compile-time representations of principals and resources, yielding a dual representation for them:

1. They appear as static constructors of kinds `Prin` and `Db` resp. These constructors appear in types and propositions, such as $\texttt{mayrd}(d, p)$.
2. They are also represented as runtime values that mediate access to them. The type system tracks the identities of these runtime values by linking them to their static counterparts using their types. The type $\texttt{db}(A)$ is a singleton type that represents runtime database values indexed by A, which has the kind `Db`. We refer to the static representations, like A, as indices.

The two representations are created together. For databases, this is done using a primitive operation `db.open`: $\exists\alpha{::}\texttt{Db.db}(\alpha)$.

In the case of principals, we need runtime representations of only those principals that are authenticated. The type $\texttt{Iam}(A)$ represents the type of authentication tokens for $A{::}\texttt{Prin}$.

4.4 Runtime acquisition of proofs

PCML$_5$ provides a primitive operation $\texttt{acquire}[A]\{\alpha.m_1 \mid m_2\}$ to enable querying for proofs using local security policies. When executed at a site w, the primitive does a local proof search for a proof of A using only Φ_w, the policy at w. The language itself does not specify or depend on any particular proof search strategy. It is however reasonable to assume that any strategy used in an implementation should at least succeed in finding a proof when one is present as an atomic fact in the policy.

The key idea behind `acquire` is that it enables acquiring facts from security policies and incorporating them in the program at runtime. The results of the runtime query, if successful, discharge the hypothesis α bound in the branch m_1. In case the query succeeds, the result is substituted for α in m_1 and execution proceeds with m_1. In case the query fails, m_2 is executed.

5. Syntax and static semantics

Figure 3 presents the syntax of PCML$_5$. We divide the syntax into three levels: runtime terms, constructors, and kinds. Runtime terms are classified by types, which are a subset of constructors, and constructors are classified by kinds. The language is parameterized by two forms of signatures:

1. Σ_c: is the signature that introduces constructor constants, $\mathbf{a}{\Rightarrow}K$. Besides constant principals and databases, this signature introduces the encoding of authorization logic.
2. Σ_t: is the term signature and it gives types to externally implemented API constants $\mathbf{c}{:}\forall\langle\alpha_1{\Rightarrow}K_1, \ldots, \alpha_n{\Rightarrow}K_n\rangle.A \mathbin{@} w$.

In order to avoid normalization at the constructor level, we present only the normal (canonical) forms in the style of Canonical LF (Harper and Licata 2007). The idea is to exclude β and η redexes altogether by rearranging the constructors as *neutral* (N) and *normal* (A) forms so that all occurrences of eliminations appear before any introduction. Since we admit only canonical constructors,

$$
\begin{array}{rlll}
\text{Kinds} & K & ::= & \texttt{Type} \mid \texttt{Wld} \mid \texttt{Prop} \mid \texttt{Prin} \mid \texttt{Prf}(A) \\
& & \mid & \texttt{Affirms}(A_1, A_2) \mid \texttt{Db} \mid \Pi\alpha{::}K_1.K_2 \\
\text{Constructors} & & & \\
\text{(Neutral forms)} & N & ::= & \alpha \mid \mathbf{a} \mid A_1 \to A_2 \mid A_1 \times A_2 \mid A_1 + A_2 \mid \texttt{unit} \\
& & \mid & \exists\alpha{::}K.A \mid \texttt{Iam}(A) \mid (N\ A) \\
\text{(Normal forms)} & A, w, k & ::= & N \mid \lambda\alpha{::}K.A \\
\text{Polytypes} & \tau & ::= & \forall\langle\alpha_1{:}K_1, \ldots, \alpha_n{:}K_n\rangle A \\
\text{Terms} & m & ::= & x \mid \lambda x{:}A.m \mid (m_1\ m_2) \mid \langle m_1, m_2\rangle \mid \pi_1 m \mid \pi_2 m \mid \texttt{inl}\ m \mid \texttt{inr}\ m \\
& & \mid & \texttt{case}\ m\ \texttt{of}\ x.(m_1 \mid m_2) \mid \langle\rangle \\
& & \mid & \texttt{let}\ x = m_1\ \texttt{in}\ m_2 \\
& & \mid & \texttt{get}[w]m \\
& & \mid & \texttt{acquire}[A] \mid \texttt{authenticate} \\
& & \mid & \mathbf{c}[A_1, \ldots, A_n](m) \\
& & \mid & \{\alpha = A; m : A'\} \\
& & \mid & \texttt{open}\ \{\alpha, x\} = m_1\ \texttt{in}\ m_2 \\
& & \mid & \texttt{iam}[\mathbf{a}] \\
\text{Constructor signature} & \Sigma_c & ::= & \cdot \mid \Sigma_c, \mathbf{a}{:}K \\
\text{Term signature} & \Sigma_t & ::= & \cdot \mid \Sigma_t, \mathbf{c}{:}\tau@w \\
\text{Constructor context} & \Delta & ::= & \cdot \mid \Delta, \alpha{\Rightarrow}K \\
\text{Term context} & \Gamma & ::= & \cdot \mid \Gamma, x{:}A@w
\end{array}
$$

Figure 3. PCML$_5$ syntax: Shaded parts are PCML$_5$'s additions to ML$_5$

we use hereditary substitution (Watkins et al. 2002) to keep constructors canonical upon substitution, details of which are omitted here.

5.1 Judgment forms

The basic judgment forms used in static semantics are:

$K\ \texttt{kind}$	K is a well-formed kind
$A \Leftarrow K$	A is checked for the kind K
$N \Rightarrow K$	K is synthesized as the kind of N
$m : A@w$	m is of type A at the world w

We maintain a phase-distinction between static and dynamic phases in PCML$_5$. This means that runtime terms do not appear in constructors. Accordingly, we divide the context into a static part, Δ, which gives kinds to constructor variables α, and a dynamic part Γ which types term variables x. Because of the phase-distinction, the dynamic context is needed only for term typing judgments. Selected rules defining well-formed kinds are shown here:

$$\frac{\Delta \vdash_{\Sigma_c} A \Leftarrow \texttt{Prop}}{\Delta \vdash_{\Sigma_c} \texttt{Prf}(A)\ \texttt{kind}}(\texttt{Proof})$$

$$\frac{\Delta \vdash_{\Sigma_c} A_1 \Leftarrow \texttt{Prin} \qquad \Delta \vdash_{\Sigma_c} A_2 \Leftarrow \texttt{Prop}}{\Delta \vdash_{\Sigma_c} \texttt{Affirms}(A_1, A_2)\ \texttt{kind}}(\texttt{Affirms})$$

The kind $\texttt{Prf}(A)$ classifies proofs of proposition A, and the kind $\texttt{Affirms}(A_1, A_2)$ classifies evidences for the judgment $A_1\ \texttt{affirms}\ A_2$ from Section 2.

All types except one in PCML$_5$ are inherited from ML$_5$. PCML$_5$ adds an abstract type $\texttt{Iam}(A)$ of authentication credentials of the principal A. We shall return to this type in Section 5.4 when we discuss authentication.

$$\frac{\Delta \vdash_{\Sigma_c} A \Leftarrow \texttt{Prin}}{\Delta \vdash_{\Sigma_c} \texttt{Iam}(A) \Rightarrow \texttt{Type}}(\texttt{auth})$$

5.2 Representing authorization logic

The kind level of PCML$_5$ is sufficiently rich to encode the authorization logic from Section 2 using higher-order abstract syntax. For instance, the proposition $P_1 \supset P_2$ can be encoded as the constructor $\texttt{imp}\ A_1\ A_2$, where $\texttt{imp} \Rightarrow \texttt{Prop}{\to}\texttt{Prop}{\to}\texttt{Prop}$ is a constructor constant. Propositions in the logic become constructors of kind $\texttt{Prop}$ in PCML$_5$. Proofs, i.e., derivations of the judgment $\mathbf{\Delta}; \Phi \vdash^{\mathcal{L}}_{\mathbf{\Sigma}} P\ \texttt{true}$ get translated into open constructors of kind $\texttt{Prf}(A)$, where A is translation of P, under the context Δ that represents $\mathbf{\Delta}, \Phi$. Affirmations, i.e., derivations of judgment $\mathbf{\Delta}; \Phi \vdash^{\mathcal{L}}_{\mathbf{\Sigma}} a\ \texttt{affirms}\ P$ are translated into open constructors of kind $\texttt{Affirms}(A_1, A_2)$ where A_1 and A_2 are representations of a and P respectively.

The central principle behind the translation of logic is that hypothetical reasoning in PCML$_5$ represents the consequence relation of the logic. This enables constructing proofs in a program that are hypothetical in proofs of certain propositions (such as those where a runtime policy query is involved), and later discharging those hypotheses using facts from authorization policies. In order to state an adequacy theorem, we first need some terminology: We use the notation Σ_c^+ to denote the signature consisting of constants in Σ_c and the constants defining the embedding of authorization logic in PCML$_5$. The following judgments show representation of an entity from the logic into a construct in the language (rules are omitted here but can be found in our full report (Avijit et al. 2009)):

$$
\begin{array}{rl}
\mathbf{\Delta} \vdash^{\mathcal{L}}_{\mathbf{\Sigma}} P\ \texttt{prop} \gg \Delta \vdash_{\Sigma_c^+} A \Leftarrow \texttt{Prop} & \text{Propositions} \\
\mathbf{\Delta}; \Phi \vdash^{\mathcal{L}}_{\mathbf{\Sigma}} P\ \texttt{true} \gg \Delta \vdash_{\Sigma_c^+} A_1 \Leftarrow \texttt{Prf}(A_2) & \text{Proofs} \\
\mathbf{\Delta}; \Phi \vdash^{\mathcal{L}}_{\mathbf{\Sigma}} a\ \texttt{affirms}\ P \gg & \\
\Delta \vdash_{\Sigma_c^+} A \Leftarrow \texttt{Affirms}(A_1, A_2) & \text{Affirmations}
\end{array}
$$

We lift the translation to contexts and signatures in the straightforward manner, representing a hypothesis $P\ \texttt{true}$ as $\alpha{\Rightarrow}\texttt{Prf}(A)$ where A is the representation of P, and α is a fresh variable. In the

following theorem, we shall assume that Σ_c is a representation of $\mathbf{\Sigma}$ and Δ_1, Δ_2 represent the contexts $\mathbf{\Delta}$ and Φ resp..

Theorem 5.1 (Adequacy)**.** *Let* $\mathbf{\Delta} \vdash^{\mathcal{L}}_{\mathbf{\Sigma}} P_1\ \mathtt{prop} \gg \Delta_1 \vdash_{\Sigma_c^+} A_1 \Leftarrow \mathtt{Prop}$*, and* $\mathbf{\Delta} \vdash^{\mathcal{L}}_{\mathbf{\Sigma}} P_2\ \mathtt{prop} \gg \Delta_1 \vdash_{\Sigma_c^+} A_2 \Leftarrow \mathtt{Prop}$*. Then* $\mathbf{\Delta}; \Phi, P_1\ \mathtt{true} \vdash^{\mathcal{L}}_{\mathbf{\Sigma}} P_2\ \mathtt{true}$ *iff there exists* A_3 *such that* $\Delta_1, \Delta_2, \alpha{\Rightarrow}\mathtt{Prf}(A_1) \vdash_{\Sigma_c^+} A_3 \Leftarrow \mathtt{Prf}(A_2)$.

In the rest of the paper, we freely use logical notation instead of its translation for readability. For instance, we use $A_1 \supset A_2$ to mean $\mathtt{imp}\ A_1\ A_2$. The notation $\hat{\Phi}$ denotes the representation of the authorization policy Φ in the language.

5.3 Situated typing and distributed computation

PCML$_5$ inherits the notion of typing relative to worlds from ML$_5$. Here we briefly recap the central ideas from ML$_5$: distribution of computation and mobility of data.

The main motivation behind situated typing in ML$_5$ is to express locality of resources, and use it to restrain where a piece of code can run, with the idea being that a resource can only be accessed by code running at the same location. Distribution of execution is achieved by moving result of a computation from one location to another using a `get`. This is expressed in the following typing rule:

$$\frac{\Delta; \Gamma \vdash_{\Sigma_c;\Sigma_t} m : A@w' \qquad \Delta \vdash_{\Sigma_c} A\ \mathtt{mobile}}{\Delta; \Gamma \vdash_{\Sigma_c;\Sigma_t} \mathtt{get}[w']m : A@w}(\mathtt{get})$$

If m is well-typed for w', then it may be executed using a remote call from another world w. ML$_5$ however restricts the remote invocation to terms of *mobile* types only. The first requirement for a type to be mobile is that all values of that type should be typable at all worlds. This ensures that when the result of m is brought back from w' to w, the value makes sense at w. Base types such as `string` are mobile as their values can be typed at any site. Types representing local resources such as reference cells are not mobile. Function types are also not mobile in ML$_5$ because, by design, computations are never shipped across sites. Instead code meant to be run at a site is compiled and stored at that site. Only the locus of execution shifts from one site to another, as in a remote procedure call.

PCML$_5$ adds a new type $\mathtt{Iam}(A)$ for typing authentication credentials to the types inherited from ML$_5$. The type $\mathtt{Iam}(A)$ is mobile because authentication credentials are typable at all worlds. An existential type $\exists\alpha{::}K.A$ is mobile if the type A of the packed value is mobile, regardless of α. Thus, for instance, the type $\exists\alpha{::}\mathtt{Prin}.\mathtt{Iam}(\alpha)$ is mobile. All constructors are considered mobile in the sense that their kinding is independent of their location. In particular, principal and resource indices, and proofs are typable at all sites.

5.4 Authentication

For each constant principal $\mathbf{a}$ declared in the constructor signature Σ_c, we use a constant $\mathtt{iam}[\mathbf{a}]$ as its authentication token which is typed as $\mathtt{Iam}(\mathbf{a})$.

$$\frac{\mathbf{a}{\Rightarrow}\mathtt{Prin} \in \Sigma_c}{\Delta; \Gamma \vdash_{\Sigma_c;\Sigma_t} \mathtt{iam}[\mathbf{a}] : \mathtt{Iam}(\mathbf{a})@w}(\text{Iam-I})$$

PCML$_5$ provides an abstract operation `authenticate` to authenticate the program on behalf of a principal. In an implementation, this could be achieved through well-established protocols for authentication, e.g. Kerberos (Neuman and Ts'o 1994). The operation returns a principal together with an authentication token that serves as a proof of the principal having been authenticated.

The operation `authenticate` is typed as:

$$\Delta; \Gamma \vdash_{\Sigma_c;\Sigma_t} \mathtt{authenticate} : (\exists\alpha{::}\mathtt{Prin}.\mathtt{Iam}(\alpha))\ \mathtt{option}@w$$

The primitive is typed using an `option` type because it may fail to return any meaningful value. Let us consider the case when the primitive is successful. The result is typed as $\exists\alpha{::}\mathtt{Prin}.\mathtt{Iam}(\alpha)$ in this case. Notice that the authentication credential is typed as $\mathtt{Iam}(\alpha)$ linking it to the abstract component α of the abstraction. The runtime term of type $\mathtt{Iam}(\alpha)$ and the static proxy α can be viewed as dual representations of an authenticated principal. Because of the abstraction, the static component uniquely represents a particular instance of opening up the package, since it is assumed to be different from everything else. Thus a credential of type $\mathtt{Iam}(\alpha)$ represents a particular instance of doing authentication in the program.

5.5 Authorization

An authorization proof is constructed by doing proof search using local security policies at various sites. This is done using a primitive operation $\mathtt{acquire}[A]$. The type system tracks the kinds of proofs starting with $\mathtt{acquire}[A]$ to API calls where the proofs are used.

5.5.1 Distributed proof acquisition

$\mathtt{acquire}[A]$ optionally returns a proof of A. We use an existential type to model this.

$$\frac{\Delta \vdash_{\Sigma_c} A \Leftarrow \mathtt{Prop}}{\Delta; \Gamma \vdash_{\Sigma_c;\Sigma_t} \mathtt{acquire}[A] : (\exists\alpha{::}\mathtt{Prf}(A).\mathtt{unit})\ \mathtt{option}@w}$$

The success case is typed as $\exists\alpha{::}\mathtt{Prf}(A).\mathtt{unit}$. When this existential type is eliminated as $\mathtt{open}\ \{\alpha, x\} = m_1\ \mathtt{in}\ m_2$, an assumption $\alpha{\Rightarrow}\mathtt{Prf}(A)$ is introduced in the scope of m_2 during type-checking. We have mentioned before that all kinds are mobile meaning that constructors are typable at all sites. Thus the variable α stands for a globally-valid proof of proposition A in m_2 even though it is discharged using a proof obtained locally at a site.

5.5.2 Proof checking at API call sites

API constants, $\mathbf{c}$, are introduced using the term signature Σ_t. In order to call an API, its polymorphic arguments have to be fully instantiated. The typing rule statically reflects the proof-verification that happens at the reference monitor during such a call. All the constructor arguments are typed statically according to the type of the API constant.

$$\frac{\begin{array}{c}\Sigma_t(\mathbf{c}) = \forall\langle\alpha_1{:}K_1, \ldots, \alpha_n{:}K_n\rangle(A \to A') \\ \Delta \vdash_{\Sigma_c} A_k \Leftarrow [A_1/\alpha_1]\ldots[A_{k-1}/\alpha_{k-1}]K_k\ (k = 1..n) \\ \Delta; \Gamma \vdash_{\Sigma_c;\Sigma_t} m : [A_1/\alpha_1]\ldots[A_n/\alpha_n]A@w\end{array}}{\Delta; \Gamma \vdash_{\Sigma_c;\Sigma_t} \mathbf{c}[A_1, \ldots, A_n](m) : [A_1/\alpha_1]\ldots[A_n/\alpha_n]A'@w}$$

Notice that for an API call to be well-typed, the constructor arguments should be of the proper kind under the context Δ as per the type specification of the API. At runtime, the hypotheses are discharged using closed constructors. The hypothetical judgment ensures that typing of arguments is preserved upon discharging these hypotheses. This in turn ensures success of runtime verification of these proofs at reference monitors. We elaborate on proof verification at reference monitors when we give the dynamics of API calls in Section 6.6.

An important technical detail here is the dependency among the constructor arguments. Consider the type of an API constant:

$$\forall\langle\alpha_1{:}K_1, \ldots, \alpha_n{:}K_n\rangle(A \to A')$$

The kind of an argument potentially depends on all previous arguments. This is manifest in the second premiss of the typing rule: $\Delta \vdash_{\Sigma_c} A_k \Leftarrow [A_1/\alpha_1]\ldots[A_{k-1}/\alpha_{k-1}]K_k$ that substitutes arguments A_1 through A_{k-1} in K_k while checking the kind of A_j.

6. Runtime semantics

Term evaluation in PCML$_5$ has the following aspects, in addition to distribution:

1. **Principals, databases and APIs:** The transition system is parameterized by a constructor signature Σ_c, and an API signature Σ_t. The signatures remain fixed throughout the evaluation of the program. We assume a fixed set of principal and database indices, introduced through the constructor signature Σ_c. In addition to principals and databases, Σ_c also introduces representation of an authorization logic.
2. **Execution under a security policy:** Each location has a fixed authorization policy associated with it. The policy at site w consists of assertions made by the principal $\bar{w}$ who governs that site. We formulate the local policy at site w as:

   $$\hat{\Phi}_w ::= \alpha_1 \Rightarrow \texttt{Prf}(\langle \bar{w} \rangle A_1), \ldots, \alpha_n \Rightarrow \texttt{Prf}(\langle \bar{w} \rangle A_n)$$

   where $\alpha_1, \ldots, \alpha_n$ are chosen fresh.

   We assume that all local policies are well-formed with respect to the signature Σ_c that defines the authorization logic. As mentioned before, the amalgamated authorization policy $\hat{\Phi}$ is the union of all local policies.
3. **Authorization checks at reference monitors:** An API call $\mathbf{c}[A_1, \ldots, A_n](m)$ results in the reference monitor verifying A_1 through A_n (as shown by Rule (`api-reduce`) later). Some of these A_i's may be proofs constructed by a distributed acquisition of local policy assertions from various sites. As discussed in Section 3, local assertions make sense globally and are implemented in an unforgeable and irrefutable manner by signing the asserted proposition with the principal's signing key. In the formalism, we directly use the global security policy $\hat{\Phi}$ to serve as the context for type-checking the proofs at API call-sites at runtime.
4. **Authentication and active principals:** As evaluation progresses, principals may be authenticated using the primitive `authenticate`. Principals who have been authenticated during a program run are called *active* principals. We keep a record of all active principals as a set of principals $\mathcal{A} \subseteq \{\mathbf{a} \mid \mathbf{a} \Rightarrow \texttt{Prin} \in \Sigma_c\}$. The program starts execution with $\mathcal{A} = \{\}$, i.e. no principal is assumed to be authenticated at the beginning of the program.

6.1 Judgment forms

We describe the dynamic semantics using a transition relation between terms. We use the following judgment forms:

1. $m; \mathcal{A} \ \mapsto_w^{\Sigma_c;\Sigma_t;\hat{\Phi}} \ m'; \mathcal{A}'$: Under the signatures $\Sigma_c; \Sigma_t$ and the amalgamated authorization policy $\hat{\Phi}$, the term m, executed at world w, steps to the term m' in a single transition; the set of active principals changes from $\mathcal{A}$ to $\mathcal{A}'$. Since Σ_c, Σ_t and $\hat{\Phi}$ remain fixed during evaluation, we often omit them while presenting the transition system.
2. $m \ \texttt{val}_{\mathcal{A}}$: means that the term m is a value under the runtime record $\mathcal{A}$, and is not evaluated further.

6.2 The value judgment

The value judgment $m \ \texttt{val}_{\mathcal{A}}$ defines values with respect to the runtime state $\mathcal{A}$. The rules for determining values of the types such as function, product, sum and existential types are straightforward; we adopt eager evaluation for products, sums and existential packages.

The crucial part of this definition is the case for values of type $\texttt{Iam}(A)$. An authentication token $\texttt{iam}[\mathbf{a}]$ is a value only if $\mathbf{a}$ is an authenticated principal, i.e. if $\mathbf{a}$ is part of the active set $\mathcal{A}$. The need for the requirement that $\mathbf{a}$ be in $\mathcal{A}$ is explained in Section 6.6, where the value $\texttt{iam}[\mathbf{a}]$ is used at an API call-site.

$$\frac{\mathbf{a} \in \mathcal{A}}{\texttt{iam}[\mathbf{a}] \ \texttt{val}_{\mathcal{A}}} (\texttt{Iam-V})$$

6.3 Distribution

A remote call $\texttt{get}[w']m$ is evaluated at a world w as follows:

$$\frac{m; \mathcal{A} \ \mapsto_{w'} \ m'; \mathcal{A}'}{\texttt{get}[w']m; \mathcal{A} \ \mapsto_w \ \texttt{get}[w']m'; \mathcal{A}'} (\texttt{get-eval})$$

$$\frac{m \ \texttt{val}_{\mathcal{A}}}{\texttt{get}[w']m; \mathcal{A} \ \mapsto_w \ m; \mathcal{A}} (\texttt{get-reduce})$$

Rule (`get-eval`) expresses the remote execution of m at the world w'. In case m is a value, it is brought to w from w' (Rule (`get-reduce`)). This transfer is safe, i.e., m is well-typed at w because the type system guarantees that m has a mobile type.

6.4 Local policy acquisition

The $\texttt{acquire}[A]$ primitive does a proof search for the proposition A using the local policy $\hat{\Phi}_w$ at the site where it is executed. This proof search may fail. We do not stipulate the procedure used for local theorem-proving. For this reason, we model this primitive using a non-deterministic step: the Rule (`acq-succ`) non-deterministically chooses a proof A' such that A' has the kind $\texttt{Prf}(A)$ under the assumptions $\hat{\Phi}_w$.

$$\frac{\hat{\Phi}_w \vdash_{\Sigma_c} A' \Leftarrow \texttt{Prf}(A)}{\texttt{acquire}[A]; \mathcal{A} \ \mapsto_w \ \texttt{SOME}\,\{\alpha = A'; \langle\rangle : \texttt{unit}\}; \mathcal{A}} (\texttt{acq-succ})$$

The failure case is modeled by a transition to the value `NONE`:

$$\frac{}{\texttt{acquire}[A]; \mathcal{A} \ \mapsto_w \ \texttt{NONE}; \mathcal{A}} (\texttt{acq-fail})$$

6.5 Authentication

Authentication, if successful, results in the production of a token for the authenticated principal. We use a non-deterministic rule which may result in a token for any principal that has been declared in the signature Σ_c.

$$\frac{\mathbf{a}{::}\texttt{Prin} \in \Sigma_c}{\begin{array}{l}\texttt{authenticate}; \mathcal{A} \ \mapsto_w \\ \texttt{SOME}\,\{\alpha = \mathbf{a}; \texttt{iam}[\mathbf{a}] : \texttt{Iam}(\alpha)\}; \mathcal{A} \cup \{\mathbf{a}\}\end{array}} (\texttt{auth-succ})$$

The record $\mathcal{A}$ is augmented with $\mathbf{a}$ to note this authentication.

In case of failure, `authenticate` returns `NONE`, and $\mathcal{A}$ is left unchanged:

$$\frac{}{\texttt{authenticate}; \mathcal{A} \ \mapsto_w \ \texttt{NONE}; \mathcal{A}} (\texttt{auth-fail})$$

An important technical detail to note here is that we require Σ_c to contain all possible principals. This is manifest in the Rule (`auth-succ`) where `authenticate` does not generate a new principal identity but simply picks one up from Σ_c.

6.6 API calls to reference monitors

API constants represent externally implemented functions. We model their behavior using a non-deterministic transition to an arbitrary term of the appropriate type.

$$\frac{m; \mathcal{A} \ \mapsto_w \ m'; \mathcal{A}'}{\mathbf{c}[A_1, \ldots, A_n](m); \mathcal{A} \ \mapsto_w \ \mathbf{c}[A_1, \ldots, A_n](m'); \mathcal{A}'} (\texttt{api-eval})$$

$$\frac{\begin{array}{c}\Sigma_t(\mathbf{c}) = \forall\langle\alpha_1{::}K_1, \ldots, \alpha_n{::}K_n\rangle A \to A'@w \\ \forall i \in [0..n-1] \ \hat{\Phi} \vdash_{\Sigma_c} A_{i+1} \Leftarrow [A_1/\alpha_1] \ldots [A_i/\alpha_i]K_{i+1} \\ m_1 \ \texttt{val}_{\mathcal{A}}\end{array}}{\mathbf{c}[A_1, \ldots, A_n](m_1); \mathcal{A} \ \mapsto_w^{\Sigma_c;\Sigma_t;\Phi} \ m_2; \mathcal{A}} (\texttt{api-reduce})$$

The Rule (`api-reduce`) illustrates the central actions that take place during a call to a reference monitor:

1. **Authorization checks** The central point of PCA is that reference monitors for APIs check proofs. Proofs are passed as constructor arguments to the API. In order to verify them, the reference monitor needs the amalgamated policy $\hat{\Phi}$.

   Consider the Rule (`api-reduce`). Before evaluating the API call, the reference monitor type-checks all the constructor arguments $A_1 \ldots A_n$ under $\hat{\Phi}$. For well-typed programs, the type system (cf. Section 5.5.2) guarantees that all type checks succeed.

2. **Authentication checks** In addition to verifying proofs, the reference monitor checks authentication credentials. APIs for such monitors are polymorphic in the authenticated principal α, and require an extra parameter of type $\texttt{Iam}(\alpha)$. In order to verify an authentication credential $\texttt{iam}[\mathbf{a}]$, the reference monitor simply checks whether $\mathbf{a} \in \mathcal{A}$, where $\mathcal{A}$ is the active set at the time of API call. In the formalism, authentication checks are reflected implicitly in the value judgment. An authentication token $\texttt{iam}[\mathbf{a}]$ is considered a value under an active set $\mathcal{A}$ only if $\mathbf{a} \in \mathcal{A}$ (Rule `Iam-V`). This way, we reduce authentication checking to checking that the argument to the API call (m_1 in Rule (`api-reduce`)) is a value.

6.7 PCA runtime errors

A term may incur a runtime fault at a reference monitor in the following two ways:

1. Either the proofs passed to the monitor API do not type-check,
2. An authentication credential is not valid, in the sense that the purported principal does not appear in the authentication record $\mathcal{A}$.

We formalize these errors explicitly using the judgment $m \uparrow_{\mathcal{A}}$. This judgment formalizes the intuition that m is not a value, and $m; \mathcal{A} \not\mapsto_w$.

$$\frac{\mathbf{a} \notin \mathcal{A}}{\texttt{iam}[\mathbf{a}] \uparrow_{\mathcal{A}}}(\texttt{iam}^{\uparrow})$$

$$\frac{\hat{\Phi} \not\vdash_{\Sigma_c} A_i \Leftarrow [A_1/\alpha_1] \ldots [A_{i-1}/\alpha_{i-1}]K_i}{\mathbf{c}[A_1, \ldots, A_n](m) \uparrow_{\mathcal{A}}}(\mathbf{c}^{\uparrow})$$

In addition, we have rules to propagate errors through evaluation of other terms, of which we present only a sample here:

$$\frac{m_1 \uparrow_{\mathcal{A}}}{\langle m_1, m_2 \rangle \uparrow_{\mathcal{A}}}(\texttt{pair}_1^{\uparrow}) \qquad \frac{m_1\ \texttt{val}_{\mathcal{A}} \qquad m_2 \uparrow_{\mathcal{A}}}{\langle m_1, m_2 \rangle \uparrow_{\mathcal{A}}}(\texttt{pair}_2^{\uparrow})$$

The rules for the error judgment follow the evaluation order. A pair $\langle m_1, m_2 \rangle$ can incur an error at two instances: either when m_1 which is evaluated first incurs an error (as in Rule ($\texttt{pair}_1^{\uparrow}$)), or when m_1 has been evaluated to a value and m_2 incurs an error (as shown in Rule ($\texttt{pair}_2^{\uparrow}$)).

7. Metatheory

We now prove the central result of this paper: a class of well-typed PCML$_5$ programs do not incur runtime faults at reference monitors. We have already summarized a notion of *error* states (Section 6.7) that formalizes exactly the runtime failures we are trying to avoid.

We start by proving progress and preservation for PCML$_5$. This means that well-typed terms do not get stuck. However, static typing alone does not rule out authentication errors, i.e. failures that happen when the reference monitor gets a token $\texttt{iam}[\mathbf{a}]$ and $\mathbf{a}$ is not an active principal. This happens because the type system does not track runtime authentication events, i.e. calls to `authenticate`.

Throughout this section we shall assume that $\Sigma_c; \Sigma_t$ are well-formed signatures and $\hat{\Phi}$ is an amalgamated security policy which is well-formed under Σ_c.

7.1 Type safety

Theorem 7.1 (Progress)**.** *Let $\mathcal{A}$ be a set of active principals from Σ_c. If $\Delta; \cdot \vdash_{\Sigma_c;\Sigma_t} m : A@w$, then*

1. *either $m\ \texttt{val}_{\mathcal{A}}$,*
2. *or $\exists m', \mathcal{A}'. m; \mathcal{A} \mapsto_w^{\Sigma_c;\Sigma_t;\hat{\Phi}} m'; \mathcal{A}'$,*
3. *or $m \uparrow_{\mathcal{A}}$ without using the Rule ($\mathbf{c}^{\uparrow}$).*

Theorem 7.2 (Preservation of typing)**.** *Assume that all API constants declared in Σ_t preserve the typing. If $m; \mathcal{A} \mapsto_w^{\Sigma_c;\Sigma_t;\hat{\Phi}} m'; \mathcal{A}'$ and $\Delta; \cdot \vdash_{\Sigma_c;\Sigma_t} m : A@w$,*
then $\Delta, \hat{\Phi}; \cdot \vdash_{\Sigma_c;\Sigma_t} m' : A@w$.

In order to guarantee that terms do not incur authentication errors in addition to not getting stuck, we define a notion of *authentication safety* for terms with respect to a set of active principals. A well-typed term is regarded as authentication safe if it never evaluates to an unauthenticated token. We prove that an authentication safe term does not evaluate to an error state. In addition, we prove that authentication safety is preserved by evaluation; these two theorems together ensure that authentication safe terms do not incur runtime access violations at reference monitors. Finally we show how authentication safety for the case where the set of active principals is empty can be enforced using the type system.

7.2 Authentication history

The runtime semantic ensures that `authenticate` is the only way in which a new authentication token can be generated. We further wish to enforce that a token $\texttt{iam}[\mathbf{a}]$ appears in a term only when the associated history $\mathcal{A}$ mentions $\mathbf{a}$ as one of the authenticated principals. We use a judgment of the form $\Delta; \Gamma \vdash^{\mathcal{A}}_{\Sigma_c;\Sigma_t} m : A@w$ to denote that, in addition to the term being well-typed, all authentication tokens in m have the corresponding principal recorded in $\mathcal{A}$. This judgment resembles the static typing judgment $\Delta; \Gamma \vdash_{\Sigma_c;\Sigma_t} m : A@w$ except for the case of $\texttt{iam}[\mathbf{a}]$, where we demand that the principal $\mathbf{a}$ be present in $\mathcal{A}$. We present a sample of rules here:

$$\frac{\mathbf{a}{::}\texttt{Prin} \in \Sigma_c \qquad \mathbf{a} \in \mathcal{A}}{\Delta; \Gamma \vdash^{\mathcal{A}}_{\Sigma_c;\Sigma_t} \texttt{iam}[\mathbf{a}] : \texttt{Iam}(\mathbf{a})@w}$$

$$\frac{\Delta; \Gamma, x{:}A@w \vdash^{\mathcal{A}}_{\Sigma_c;\Sigma_t} m : A'@w}{\Delta; \Gamma \vdash^{\mathcal{A}}_{\Sigma_c;\Sigma_t} \lambda x{:}A.m : A \to A'@w}$$

$$\frac{\Delta; \Gamma \vdash^{\mathcal{A}}_{\Sigma_c;\Sigma_t} m_1 : A_1 \to A_2@w \qquad \Delta; \Gamma \vdash^{\mathcal{A}}_{\Sigma_c;\Sigma_t} m_2 : A_1@w}{\Delta; \Gamma \vdash^{\mathcal{A}}_{\Sigma_c;\Sigma_t} (m_1\ m_2) : A_2@w}$$

Theorem 7.3. *If $\Delta; \Gamma \vdash^{\mathcal{A}}_{\Sigma_c;\Sigma_t} m : A@w$ then $\Delta; \Gamma \vdash_{\Sigma_c;\Sigma_t} m : A@w$.*

We assume that API calls do not directly introduce spurious authentication tokens in their results. The following assumption about API calls summarizes this:

Definition 7.4 (Authentication safety for APIs)**.** *Let $\Sigma_t(\mathbf{c}) = \forall\langle\alpha_1{::}K_1, \ldots, \alpha_n{::}K_n\rangle A_1 \to A_2@w$. Let $\mathcal{P}$ be the set of constants $\mathbf{a}$ s.t. $\mathbf{a}{::}\texttt{Prin} \in \Sigma_c$. The API $\mathbf{c}$ is* authentication safe *if the following holds:*

For all constructors $B_1, \ldots, B_n$ such that
$\cdot \vdash_{\Sigma_c} B_{i+1} \Leftarrow [B_1/\alpha_1] \ldots [B_i/\alpha_i]K_{i+1}$. Let v be well-typed,

and be safe for $\mathcal{A}$ as $\Delta;\cdot \vdash^{\mathcal{A}}_{\Sigma_c;\Sigma_t} v : [B_1/\alpha_1]\ldots[B_n/\alpha_n]A_1@w$. *If* $\mathbf{c}[B_1,\ldots,B_n](v); \mathcal{A} \mapsto_w m; \mathcal{A}$, *then* m *is safe for* $\mathcal{A}$, *i.e.* $\Delta;\cdot \vdash^{\mathcal{A}}_{\Sigma_c;\Sigma_t} m : [B_1/\alpha_1]\ldots[B_n/\alpha_n]A_2@w$.

This restriction forces the resulting term to be safe w.r.t. all the possible $\mathcal{A}$'s for which the argument v is safe, thus ensuring that all authentication tokens in the result of the call are inherited from the argument v, i.e. no new $\texttt{iam}[\mathbf{a}]$ are introduced in the result.

If all APIs are authentication safe, then authentication safety is preserved by evaluation:

Theorem 7.5 (Preservation of authentication safety)**.** *Let all API constants declared in* Σ_t *be safe in the sense of Def. 7.4.*
If $m; \mathcal{A} \mapsto^{\Sigma_c;\Sigma_t;\hat{\Phi}}_w m'; \mathcal{A}'$ *and* $\Delta;\cdot \vdash^{\mathcal{A}}_{\Sigma_c;\Sigma_t} m : A@w$,
then $\Delta, \hat{\Phi};\cdot \vdash^{\mathcal{A}'}_{\Sigma_c;\Sigma_t} m' : A@w$.

Furthermore, authentication safety ensures that reference monitors cannot reject calls from well-typed and authentication-safe programs.

Theorem 7.6 (Progress under authentication safety)**.** *Let* $\mathcal{A}$ *be a set of active principals from* Σ_c. *If* $\Delta;\cdot \vdash^{\mathcal{A}}_{\Sigma_c;\Sigma_t} m : A@w$, *then either* m $\texttt{val}_{\mathcal{A}}$, *or* $\exists m', \mathcal{A}'.m; \mathcal{A} \mapsto^{\Sigma_c;\Sigma_t;\hat{\Phi}}_w m'; \mathcal{A}'$.

Notice that in contrast to Theorem 7.1 which has a failure clause ($m \uparrow_{\mathcal{A}}$), Theorem 7.6 excludes the possibility of an authentication-safe term getting stuck and ensures that it is either a value, or that it takes a step to another term. The progress theorem (Theorem 7.6), together with preservation (Theorem 7.5) ensures that an authentication safe, well-typed term never incurs a runtime fault at reference monitors.

7.3 Initializing evaluation

A closed well-typed program m begins evaluation under an empty set of authenticated principals. By virtue of the progress and preservation theorems for authentication safety, in order to ensure that evaluation never incurs a runtime error, it is sufficient to ensure that $\cdot;\cdot \vdash^{\{\}}_{\Sigma_c;\Sigma_t} m : A@w$. This can be enforced easily using the typing rules by simply disallowing all constants $\texttt{iam}[\mathbf{c}]$ as well-typed terms. This is possible because while checking for authentication safety of a term under $\mathcal{A}$, all the sub-terms are checked under the same $\mathcal{A}$.

8. Example

We revisit the example scenario from Figure 1. The program shown in Figure 4 illustrates how a distributed program running at `browser` can access the database `conf` at `acmserver` using a proof constructed using distributed proof-search at `univserver` and `acmserver`.

8.1 External API and other declarations

The program in Figure 4 first declares principal, database and world constants as `extern` declarations. The `extern principal` declaration runs an initialization code to acquire the runtime representation of the principal (using the name supplied) and binds it to the variable in the declaration. The classifiers `bytecode` and `javascript` in world declarations determine the language of the generated code for the respective worlds. Lines 10-12 declare the addresses for the three sites.

Lines 14-16 declare predicates using the `extern prop` declaration. A declaration `extern prop (s1, ..., sn) p` declares `p` to be a predicate with $\texttt{s}_1$ through $\texttt{s}_n$ being the sorts of its arguments.

Next we declare types for the database API at `acmserver`. The constructor `dbhandle` has the kind `Db` $\rightarrow$ `Type`. The API `db.open` takes a database name of type `string` and returns the runtime identity of the database packaged with its static proxy, as discussed in Section 4.3. We use the syntax $\{a{:}K, A\}$ for the existential type $\exists a{::}K.A$. For simplicity, databases in this paper are simply *(key, value)* pairs, where both the *key* and the *value* are strings. Line 21 introduces the proof-carrying API for reading databases: `db.read` is typed polymorphically in the accessing principal p, the database d, and the proof f of `acm` affirming that p is allowed to read d. The argument to `db.read` is a triple consisting of (1) a value of type $\texttt{Iam}(p)$ which forms an evidence of p's authentication, (2) the runtime structures associated with the database d, and (3) the key to be read. Both the database operations `db.open`, and `db.read` are local to `acmserver`.

8.2 Distributed proof construction

We now discuss the steps involved in reading `conf` at `acmserver`. This is done in Lines 28-57 by the function `readpaper` which is typed at `acmserver`.

The first operation (Lines 29-31) is to authenticate the principal. In case authentication fails, the program raises the `Abort` exception and halts. In case of a successful authentication, `authenticate` returns a value of type $\exists\alpha{::}\texttt{Prin.Iam}(\alpha)$, which is bound to `authpkg`.

Let us consider the case when authentication succeeds. In this case `authpkg` gets bound to a package $\{\alpha = k; \texttt{iam}[k] : \texttt{Iam}(\alpha)\}$, where k is the authenticated principal. The side-effect of this successful authentication is that k gets added to the set of active principals.

Assume that the amalgamated policy is given as the following context $\hat{\Phi}$, with the variables $\alpha_1, \alpha_2, \alpha_3$ chosen fresh. The whole program is run under this context. In an implementation, these variables would be bound to digital certificates.

$$\hat{\Phi} = \left\{ \begin{array}{lcl} \alpha_1 & \Rightarrow & \langle\texttt{univ}\rangle\texttt{is\_student}(\texttt{alice}, \texttt{univ}), \\ \alpha_2 & \Rightarrow & \langle\texttt{acm}\rangle\texttt{is\_member}(\texttt{univ}, \texttt{acm}), \\ \alpha_3 & \Rightarrow & \langle\texttt{acm}\rangle\forall x{:}\texttt{prin}.\forall y{:}\texttt{prin}. \\ & & \quad \texttt{is\_member}(x, \texttt{acm}) \wedge \langle x\rangle\texttt{is\_student}(y, x) \\ & & \quad \supset \texttt{mayrd}(\texttt{conf}, y) \\ & \vdots & \end{array} \right\}$$

The task now is to construct a proof of

$$\langle\texttt{acm}\rangle\texttt{mayrd}(\texttt{conf}, k).$$

This task depends on the identity of the authenticated principal. The static index of the authenticated principal is obtained by opening the package `authpkg` (Line 33), binding `me` to k, and `cookie` to $\texttt{iam}[k]$. The rest of the code in the function executes in the scope of this `open`.

We begin by first acquiring the certificate for studentship for k. This is done by moving evaluation to the site `univserver` and searching for a proof of $\langle\texttt{univ}\rangle\texttt{is\_student}(k, \texttt{univ})$ there. The query `acquire` on Line 36 succeeds if k is `alice` because in this case there is a direct proof, α_1, in the policy. In the absence of any other proofs in `univserver`'s local policy, this query fails for other principals k. A successful `acquire` on Line 36 binds `studentcert` to the optional package `SOME` $\{\alpha = \alpha_1; \langle\rangle : \texttt{unit}\}$, and the result is brought back to `acmserver` by the `get`. Notice that the result of an `acquire` is of a mobile type and therefore can be brought over from `univserver` to `acmserver`. The program raises the exception `abort` and terminates in case this `acquire` fails.

Next we need to acquire a proof that k is allowed access to the database `conf`. This proof search depends on the static identity of the database `conf`. This identity is obtained using `db.open` at Line 40, which, if successful binds the proxy to `d`, and the runtime handle to `h` upon opening the package returned by `db.open`. At this point, we have already acquired a proof of studentship for `me`. We now use this fact to guide our proof search. We proceed (Line 44) with a query for the following proposition at `acmserver`:

```
unit
import "std.mlh"

extern principal acm = "acm"
extern principal univ = "univ"
extern bytecode world acmserver
extern bytecode world univserver
extern javascript world browser

extern val acmaddr ~ acmserver addr
extern val univaddr ~ univserver addr
extern val browseraddr ~ browser addr

extern prop (prin, prin) is_member
extern prop (db, prin) mayrd
extern prop (prin, prin) is_student

extern type (d:db) dbhandle
extern val db.open :
   string -> {d:db, dbhandle(d)} option @ acmserver
extern val (p: prin, d: db, f: acm says mayrd(d, p))
           db.read :
           Iam(p) * dbhandle(d) * string -> string
        @ acmserver

exception Abort of string

fun readpaper () =
  let val authpkg = case authenticate of
                      NONE => raise Abort "authentication failed"
                    | SOME p => p
  in
  open authpkg as {me: prin, cookie} in
    let val studcert =
     case from univaddr
          get acquire [univ says is_student(me, univ)] of
       NONE => raise Abort "studentship search failed"
     | SOME c => c
    in
      case db.open "conf" of
          NONE => raise Abort "db.open failed"
        | SOME dbase =>
            open dbase as {d:db, h} in
              case acquire [univ says is_student(me, univ)
                            implies
                            acm says mayrd(d, me)] of
                  NONE => raise Abort "mayrd search failed"
                | SOME prf => open studcert as {studpf, _} in
                              open prf as {pf, _} in
                                db.read[me, d, impE pf studpf]
                                       (cookie, h, "paper.pdf")
                              end
                              end
            end
    end
  end
 end

do from acmaddr get readpaper ()

end
```

Figure 4. PCML$_5$ code for accessing a PCA-enabled resource under a distributed authorization policy. The code corresponds to the setup introduced in Figure 1.

```
univ says is_student(me, univ)
implies
acm says mayrd(d, me)
```

This particular proof search illustrated the idea of combining proofs from various sites. The proof search is done at `acmserver` even though the antecedents of the implication represent non-local facts. The hypothesis is discharged (Line 50) using proofs α_1 acquired separately at `univserver`. This discharge of assumptions using non-local facts to obtain a globally valid proof is possible because constructors are typeable at all locations.

Notice that in order to construct the accessibility proof, it is critical to assemble `univ`'s policy from `univserver`. The policy of `acm` alone does not suffice for the required proof. That is, one cannot hope for the direct query

```
acquire[acm says mayrd (d, me)]
```

to be successful at `acmserver`. The proof query in Lines 44-46 can be viewed as temporarily extending the policy at `acmserver` to include the non-local fact $\langle\texttt{univ}\rangle\texttt{is\_student}(k, \texttt{univ})$, whereupon the accessibility proof becomes derivable at `acmserver`.

8.3 Database access using required proofs

Lines 48-53 show the code that makes an API call to read `conf` using the proof constructed above. First the proof packages returned by `acquire`'s are opened binding the proof constructors to `studpf` and `pf`. The accessibility proof is constructed as `impE pf studpf`. `impE` is a constructor from the embedding of the authorization logic representing $\supset$-elimination; it has the kind

$$\Pi\alpha{::}\texttt{Prop}.\Pi\beta{::}\texttt{Prop}.\texttt{Prf}(\alpha \supset \beta) \rightarrow \texttt{Prf}(\alpha) \rightarrow \texttt{Prf}(\beta)$$

The constructor `pf` is a proof of

$$\langle\texttt{univ}\rangle\texttt{is\_student}(k, \texttt{univ}) \supset \langle\texttt{acm}\rangle\texttt{mayrd}(\texttt{conf}, k)$$

under $\hat{\Phi}_{\texttt{acmserver}}$, and the constructor `studpf` is a proof of

$$\langle\texttt{univ}\rangle\texttt{is\_student}(k, \texttt{univ})$$

under $\hat{\Phi}_{\texttt{univserver}}$. Both proofs are therefore well-typed under $\hat{\Phi}$. Thus the proof `impE pf studpf` proves $\langle\texttt{acm}\rangle\texttt{mayrd}(\texttt{conf}, k)$ under the amalgamated policy $\hat{\Phi}$.

Apart from the authorization check, the monitor checks if the credential `cookie` represents an authenticated principal by consulting the set of active principals. This check also succeeds because `authenticate` done at Line 29 augments the active set with k. Since both checks succeed, the API call is successful. The result, which is of the mobile type `string`, is returned back to `browser` on Line 59.

9. Related work

The problem of specifying and enforcing policies in a scenario with decentralized and spatially distributed policies was first studied in the context of authentication in the Taos operating system (Wobber et al. 1994), and later in the context of trust management systems (Blaze et al. 1996; Clarke et al. 2001; Li et al. 2002). Such policies call for logics that are expressive enough to distinguish assertions made by different principals. A number of logics, beginning with the seminal work by Lampson et al. (Lampson et al. 1992), provide support for such assertions.

Recently, a number of languages for writing programs compliant with authorization policies have been developed. We review those that we consider to be the closest to PCML$_5$, in terms of objectives and/or techniques.

Aura (Jia et al. 2008; Vaughan et al. 2008) is a language for enforcing authorization policies. It is based on DCC (Abadi et al. 1999). Our language differs from Aura in terms of the domain of use because ours is a distributed programming language for distributed authorization policies. Aura is neither a distributed programming language, nor does it handle distributed policies.

As a matter of technique, PCML$_5$ differs from Aura in the way they incorporate the authorization logic. While PCML$_5$ uses a higher-order encoding of an authorization logic as static constructors, with a phase distinction between static and dynamic phases, the proof level in Aura is a Curry-Howard interpretation of an authorization logic, and is based on DCC (Abadi 2006).

Also, Aura does not have a notion of authentication of the principal executing the program. A special principal identifier called `self` is used to refer to the executing principal. In contrast, PCML$_5$ uses authentication tokens as indicators of the fact that a certain principal had been authenticated. This also allows for a program to acquire multiple authenticities during its evaluation.

RCF (Bengtson et al. 2008; Fournet et al. 2007) uses refinement types together with dependent types to express pre- and post-conditions. The proof obligations are represented as preconditions in the API. Thus a function for reading databases may be typed as

$$\texttt{read} : \texttt{file:string}\{\texttt{mayrd}(\texttt{file})\} \rightarrow \texttt{string}$$

where $\texttt{file:string}\{\texttt{mayrd}(\texttt{file})\}$ is the refinement type of strings f for which $\texttt{mayrd}(f)$ holds. Refinement types are introduced using a term of the form $\texttt{assume}\ C$, which is typed as $\_{:}\texttt{unit}\{C\}$. The typing context can be thought of as defining a theory which is the set of all the formulae appearing in it. The most important difference between PCML$_5$ and RCF is in the respective problems being addressed by the two languages. While PCML$_5$ is a language for programming with PCA-enabled resources that demand proofs of accessibility along with each access request, the motivation of RCF is how to verify that a program conforms to a security policy. As a matter of technique, PCML$_5$ uses explicit proof terms unlike RCF. In absence of proof terms, the type-checking algorithm of RCF uses an SMT solver to verify if the typing context proves a particular logic formula. Another difference is that RCF does not have a phase distinction since runtime values can appear inside types, because formulas need to mention runtime entities.

PCAL (Chaudhuri and Garg 2009) is another language that relies on external SMT solvers during compile-time to construct PCA proofs. Users annotate program points with propositions that they expect to hold there. The compiler first checks that the annotation at a point is sufficient to guarantee access to the command executed at that point. Then it attempts to construct a proof statically as per the annotation. In case it cannot construct a proof statically, the compiler produces code to dynamically construct the required proof. Using a combination of both methods, the compiler ensures compliance with the PCA interface.

Fable(Swamy et al. 2008) is another language that provides statically enforced compliance with security policies. The idea is to have an abstract type of tagged program values that can only be manipulated using trusted policy functions. A program is divided into two fragments: the policy fragment that provides the abstraction, and the application fragment that functions as a client for the abstraction, treating tagged values abstractly. Different kinds of security properties can be expressed by having different interpretations for the tags. Thus instead of designing a language around a particular form of policies, such as is PCML$_5$, Fable attempts to provide a general framework in which different kinds of policies can be expressed. This however comes at a cost: the language itself does not guarantee any security property itself (other than type safety); the relevant properties need to be proved separately for every policy fragment.

The idea of statically checking the permission for accessibility has been used in a completely different setting as compared to ours by Krishnaswami *et al.* (Krishnaswami and Aldrich 2005). They

use the notion of *domains* with inter-domain accessibility permissions to statically enforce that code from a domain may access an element of another domain only if there is a chain of access permissions from the former to the latter domain. Analogous to the proof-carrying APIs in PCML_5, their proposal allows specification of access permissions associated with a domain. They use domains to encapsulate stateful parts of modules for which it is desirable to restrict access from outside domains. The type system enforces compliance of the module and its interface to a high-level policy of accessibility, i.e., protected parts of the module are not leaked out by the interface.

10. Conclusion and future work

We have presented a language-based approach for enforcing distributed authorization policies. We are currently working on a prototype implementation of PCML_5, building upon the implementation of ML_5 (Murphy 2008). We plan to implement distributed applications using PCML_5. Currently the programmer constructs all the proofs by himself. In future, we would like to provide support for automated proof construction. We also plan to mechanize the metatheory of PCML_5. In this paper, we have assumed a very simple model of spatial distribution of policies: the local policy at a site w contains only assertions made by the principal who governs w. In practice, however, assertions made by one principal may be cached at other sites, as is done in trust management systems (Blaze et al. 1996). In future work, we plan to support such richer forms of distribution of authorization policies and distributed proof search algorithms.

Acknowledgments

We would like to thank Deepak Garg, Limin Jia and Dan Licata for many helpful comments and suggestions.

References

Martín Abadi. Access control in a core calculus of dependency. In *ICFP*, pages 263–273, 2006.

Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. A core calculus of dependency. In *Proceedings 26th ACM Symp. on Principles of Programming Languages (POPL)*, pages 147–160, 1999.

Andrew W. Appel and Edward W. Felten. Proof-carrying authentication. In *CCS '99: Proceedings of the 6th ACM conference on Computer and communications security*, pages 52–62, 1999.

Kumar Avijit, Anupam Datta, and Robert Harper. Distributed programming with distributed authorization, 2009. Available at http://www.cs.cmu.edu/~kavijit/papers/pcml5-full.pdf.

Lujo Bauer, Michael A. Schneider, and Edward W. Felten. A proof-carrying authorization system. Technical Report TR-638-01, Princeton University, April 2001. URL `http://www.ece.cmu.edu/~lbauer/papers/pcaprototr.pdf`.

Lujo Bauer, Scott Garriss, Jonathan M. Mccune, Michael K. Reiter, Jason Rouse, and Peter Rutenbar. Device-enabled authorization in the grey system. In *Proceedings of the 8th Information Security Conference (ISC05*, pages 431–445, 2005.

Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. Refinement types for secure implementations. In *Proceedings of the 2008 21st IEEE Computer Security Foundations Symposium*, pages 17–32, 2008.

M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 164–173, May 1996.

Avik Chaudhuri and Deepak Garg. PCAL: Language support for proof-carrying authorization systems. In *Proceedings of the European Symposium on Research in Computer Security*, pages 184–199, 2009.

Dwaine E. Clarke, Jean-Emile Elien, Carl M. Ellison, Matt Fredette, Alexander Morcos, and Ronald L. Rivest. Certificate chain discovery in SPKI/SDSI. *Journal of Computer Security*, 9(4):285–322, 2001.

Cedric Fournet, Andrew Gordon, and Sergio Maffeis. A type discipline for authorization in distributed systems. In *CSF '07: Proceedings of the 20th IEEE Computer Security Foundations Symposium*, pages 31–48, 2007.

Deepak Garg. *Proof Theory for Authorization Logic and its Application to a Practical File System*. PhD thesis, School of Computer Science, Carnegie Mellon University, 2009. Available as Technical Report CMU-CS-09-123.

Deepak Garg and Frank Pfenning. Non-interference in constructive authorization logic. In *Proceedings of the 19th IEEE Computer Security Foundations Workshop (CSFW 19)*, pages 283–296, 2006.

Robert Harper and Daniel R. Licata. Mechanizing metatheory in a logical framework. *Journal of Functional Programming*, 17(4–5):613–673, July 2007.

Robert Harper, John C. Mitchell, and Eugenio Moggi. Higher-order modules and the phase distinction. In *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 341–354, 1990.

Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993.

Limin Jia, Jeffrey A. Vaughan, Karl Mazurak, Jianzhou Zhao, Luke Zarko, Joseph Schorr, and Steve Zdancewic. Aura: a programming language for authorization and audit. In *ICFP '08: Proceeding of the 13th ACM SIGPLAN International Conference on Functional programming*, pages 27–38, 2008.

Neel Krishnaswami and Jonathan Aldrich. Permission-based ownership: encapsulating state in higher-order typed languages. *SIGPLAN Notices*, 40(6):96–106, 2005.

Butler W. Lampson, Martín Abadi, Michael Burrows, and Edward Wobber. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems*, 10(4):265–310, 1992.

Ninghui Li, John C. Mitchell, and William H. Winsborough. Design of a role-based trust-management framework. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pages 114–130, 2002.

Tom Murphy, VII. *Modal Types for Mobile Code*. PhD thesis, Carnegie Mellon, January 2008. Available as technical report CMU-CS-08-126.

Tom Murphy, VII, Karl Crary, Robert Harper, and Frank Pfenning. A symmetric modal lambda calculus for distributed computing. In *Proceedings of the 19th IEEE Symposium on Logic in Computer Science (LICS 2004)*, pages 286–295,, 2004.

Tom Murphy, VII, Karl Crary, and Robert Harper. Type-safe distributed programming with ML5. In *Trustworthy Global Computing 2007*, November 2007.

B. C. Neuman and T. Ts'o. Kerberos: An authentication service for computer networks. *Communications Magazine, IEEE*, 32(9):33–38, 1994.

Nikhil Swamy, Brian J. Corcoran, and Michael Hicks. Fable: A language for enforcing user-defined security policies. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, May 2008.

Jeffrey A. Vaughan, Limin Jia, Karl Mazurak, and Steve Zdancewic. Evidence-based audit. In *CSF '08: Proceedings of the 2008 21st IEEE Computer Security Foundations Symposium*, pages 177–191, 2008.

Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework I: Judgments and properties. Technical Report CMU-CS-02-101, Carnegie Mellon University, School of Computer Science, 2002.

Edward Wobber, Martín Abadi, Michael Burrows, and Butler Lampson. Authentication in the taos operating system. *ACM Transactions on Computer Systems*, 12:256–269, 1994.