xDomain: Cross-border Proofs of Access[*]

Lujo Bauer[§]
lbauer@cmu.edu

Limin Jia[†]
liminjia@seas.upenn.edu

Michael K. Reiter[‡]
reiter@cs.unc.edu

David Swasey[§]
swasey@cmu.edu

[§] Carnegie Mellon University
[†] University of Pennsylvania
[‡] University of North Carolina at Chapel Hill

ABSTRACT

A number of research systems have demonstrated the benefits of accompanying each request with a machine-checkable proof that the request complies with access-control policy — a technique called *proof-carrying authorization*. Numerous authorization logics have been proposed as vehicles by which these proofs can be expressed and checked. A challenge in building such systems is how to allow delegation between institutions that use different authorization logics. Instead of trying to develop *the* authorization logic that all institutions should use, we propose a framework for interfacing different, mutually incompatible authorization logics. Our framework provides a very small set of primitives that defines an interface for communication between different logics without imposing any fundamental constraints on their design or nature. We illustrate by example that a variety of different logics can communicate over this interface, and show formally that supporting the interface does not impinge on the integrity of each individual logic. We also describe an architecture for constructing authorization proofs that contain components from different logics and report on the performance of a prototype proof checker.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Access controls; H.2.0 [**Information Systems**]: Security, integrity, and protection; K.6.5 [**Security and Protection**]: Authentication

General Terms

Security, Verification

Keywords

Logic-based access control, trust management, distributed authorization

1. INTRODUCTION

Formal logics are often used to model access-control systems to achieve high assurance of the systems' correctness. An increasingly popular and practical method of using these logics is instead to *implement* access control, e.g., as in proof-carrying authorization (PCA) [3]. In this approach, the credentials that define a policy are specified in an access-control logic, and the request to access a resource is accompanied by a logical proof that the request satisfies access-control policy and should therefore be granted. This has several benefits, including (1) shrinking the reference monitor's trusted computing base (and hence increasing its trustworthiness); and (2) creating irrefutable evidence, which can be saved in an audit log, that explains why an access was granted.

In recent years, a variety of authorization logics has been proposed (e.g., [22, 17, 25, 15, 12]). Although many of them have similarities, e.g., similar notions of principal or a *says* operator to describe the beliefs of principals, in general these logics differ along many axes. For example, they may differ in expressive power, the set of axioms they consider appropriate for delegation, as well as more fundamental ways such as whether they are classical or constructive, linear or nonlinear, and what formal properties can be proved of them.

Each such logic has its advantages and disadvantages and is well suited to describe some set of access-control scenarios. Hence, different systems are modeled (or built) using different logics, and it is unlikely that a single logic will arise and displace all these individual logics. A particular challenge in building access-control systems is to allow systems based on different logics to interoperate, for example, for the purpose of delegating from one domain or system to another.

Several approaches have been proposed that define a substrate for defining logics with the aim of making these different logics interoperable [3, 23]. However, the substrate itself typically imposes constraints that can severely restrict the design of logics that are defined in it.

In this paper, we propose a general framework that allows two access-control logics (and therefore the systems that use them) to interoperate while restricting their design far less than previous approaches. We accomplish this by defining the interface for communication between two logics in a way that is independent of the choices made in designing the logics themselves.

1.1 A Motivating Scenario

We motivate our approach with the following practical example. The ACM Digital Library partners with a number of academic institutions to provide members of those institu-

tions full-text access to documents held by the library. Part of the agreements between the ACM and its partner institutions is that determining who is a member of a particular institution is the responsibility of that institution. For example, CMU chooses to count as a member anyone who connects from an IP address that belongs to the range of IP addresses provided by the campus network. On the other hand, UNC counts as a member anyone who has authenticated via UNC's proprietary single-sign-on mechanism. In each case, the ACM trusts the institution to use its own judgment in deciding on the details of how membership is determined.[1]

To model this scenario in an access-control logic, one would typically represent each of the delegations as a logical statement. The chain of delegations giving Alice, a UNC student, access to doc1 could be represented as

ACM says ((UNC says open(doc1)) → open(doc1))

to indicate that ACM has delegated to UNC the right to decide when and by whom doc1 may be accessed;

UNC says ((UNC.members says open(doc1)) → open(doc1))

to indicate that UNC gives access to anybody who is part of UNC.members, a name space controlled by UNC; and

UNC says ((Alice says F) → (UNC.members says F))

to indicate that Alice is a member of UNC's community.

To gain access to doc1, Alice would then combine these statements into a proof of ACM says open(doc1). Problems arise, however, if ACM and UNC use different access-control logics (e.g., instead of the same says operator they use a $\mathsf{says_{ACM}}$ and $\mathsf{says_{UNC}}$ that have different, incompatible definitions). In this case, ACM may not understand UNC's delegation to Alice and hence the proof that explains why Alice is authorized to access doc1.

If ACM and UNC agreed on the format of digitally signed credentials, a trivial solution to this problem might be for UNC to certify each attempt by Alice to access doc1 by issuing a digitally signed credential (UNC signed open(doc1)). Such a solution, however, has significant disadvantages. For one, UNC must run an on-line certification service to field requests from Alice, which may be inefficient or infeasible. More importantly, the proofs submitted to ACM are much less informative than before: they no longer contain information, which may necessary in case of a breach or audit, about why UNC allowed Alice access.

1.2 Overview of Our Approach

A framework that presents a comprehensive solution to interoperability among different logics should address the following challenges. First, considering the variety of authorization logics that has been proposed and the trend to develop even more, the framework must be general and flexible so that it can accommodate a large variety of different logics and not become obsolete when new logics are developed. Second, the framework should guarantee that delegation respects proper boundaries between different domains. In particular, it must ensure that a delegation expressed in one logic, to a domain that uses a different logic, does not affect the integrity of the logic in which the delegation is issued, e.g., by rendering it logically inconsistent. Third,

the framework must cleanly separate different logics, so that rules from one logic cannot be erroneously used within a proof component written in another logic. Fourth, the interface exposed by the framework should be sufficiently narrow that only minimal burden is placed on each of the logics that participate in cross-domain delegation. Finally, in situations when two logics with different expressiveness delegate to one another, e.g., when one logic can reason about time and the other cannot, the framework must provide a way to bridge this gap in expressiveness.

The framework we describe in this paper is an attempt to address the above concerns. The key idea behind our framework is to use a universal wrapper to package a proof component with a definition of its logic. More precisely, we introduce a new logical primitive, $\mathsf{seal}(K_{\mathsf{domain}}, H, F)$, that is used to refer to a formula F specified in the logic defined by a principal whose public key is K_{domain}. To identify the logic unambiguously, seal includes a cryptographic hash H of the logic's definition. This primitive, which is part of the interface we require all logics to implement, enables ACM to delegate to UNC via a statement like

ACM $\mathsf{says_{ACM}}$ (seal(K_{UNC}, h, UNC says∗ open(doc1))
    → open(doc1))

without knowing the precise definition of says∗ or how it can be proved in UNC's logic that UNC says∗ open(doc1). More precisely, ACM is stating that as long as it is possible to prove, in UNC's logic, that UNC believes that access should be allowed, then ACM will be willing to allow the access. The proof of ACM $\mathsf{says_{ACM}}$ open(doc1) now contains a subproof of seal(K_{UNC}, h, UNC says∗ open(doc1)) that is specified in a logic completely independent of the one used by ACM. This subproof of seal contains the definition of UNC's logic so that ACM's reference monitor can verify that the certificates issued by UNC and Alice in fact support the conclusion UNC says∗ open(doc1). Moreover, ACM can record this proof in an audit log. If something has gone wrong, ACM will be able to inspect the proof and apportion blame (and penalties) accordingly.

Issuing the above delegation requires ACM to trust UNC not just to redelegate access correctly, but also to design a logic with no bugs that could lead to access being inadvertently granted. Hence, ACM would presumably delegate in this way (1) after manually inspecting UNC's logic, (2) after being formally convinced of the validity of UNC's logic (e.g., via mechanically verifiable proofs), or (3) if it trusts UNC to do the right thing. The less well-founded reasons for trust (1, 3) are consistent with what ACM is currently forced to do in reality; we show that informing trust via mechanized proofs is also feasible in our framework.

1.3 Contributions and Roadmap

This paper makes the following contributions.

- We describe in detail our framework for delegating between logics using the seal primitive. We specify the interface that logics need to implement in order to use this primitive (Section 2.1), and illustrate how several fundamentally different logics can coordinate in this way (Section 2.2).

- We show a more flexible version of our framework that allows several logics to agree on a richer set of formulas for communicating between each other, while at the

---

[1]The ACM's policy in this example is for illustration and may not correspond exactly to the policy intended or implemented by the ACM.

same time further minimizing the interface exported by our framework (Section 3).

- For several of the logics we consider, we prove formally that implementing the interface for cross-domain delegation does not interfere with desirable properties, such as consistency, that those logics may have in isolation (Section 5). We extend this argument to explain why any logic can safely be made compatible with our framework.

- We describe designs for a checker that can verify proofs that contain components from multiple logics and a prover that allows such proofs to be assembled in a distributed manner (Section 4). We have developed a prototype implementation of the checker, and we report on its performance (Section 4.1).

1.4 Related Work

The study of logics for access-control gained prominence with the work on the Taos operating system [2]. Since then, significant effort has been put into formulating formal languages and logics (e.g., [2, 4, 10, 25, 1, 20, 21, 22]) that can be used to describe a wide range of practical scenarios. The usefulness of mechanically generated proofs led to efforts to balance the decidability and expressiveness of access-control logics. These efforts resulted in various first-order classical logics, each of which describes a comprehensive but not exhaustive set of useful access-control scenarios [4, 18, 24, 25], and more powerful higher-order logics that served as a tool for defining simpler, application-specific ones [3, 5]. Researchers have recently started to examine constructive authorization logics [15], about which they have proved metaproperties such as soundness and non-interference, as well as logics that reason about linearity and time [14, 13]. That there exist so many compelling alternatives among authorization logics makes our exploration of cross-domain delegation even more relevant in improving the likelihood of adoption of logic-based access-control systems.

Most related to our work are proof-carrying authorization (PCA) [3] and Alpaca [23]. Appel and Felten's PCA was an attempt to develop a framework in which different logics could be encoded and could perhaps interoperate [3]. PCA used higher-order logic (HOL) as a universal substrate in which other logics were defined; by virtue of being encoded in HOL, these application-specific logics inherited some desirable properties, such as consistency, of the underlying logic. However, this implicitly assumed that the application-specific logics would share the substrate's notion of judgment and fundamental axioms, hence severely restricting the ways in which these logics could differ (specifically, they would have to be non-linear, classical logics). Although encoding logics with incompatible axioms was possible, such encodings would not derive any desirable properties from the substrate or be amenable to interaction with other application-specific logics.

PCA's more enduring innovation was that it brought the proof-carrying paradigm [27, 28] to logic-based access control. Since Appel and Felten proposed PCA, many systems have adopted the proof-carrying approach (e.g., [6, 23]). However, these typically use only one authorization logic for the entire system and do not consider interoperation between two or more different authorization logics.

Alpaca is an authentication framework based on PCA that uses logic to specify widely varying, but interoperating, public-key infrastructures [23]. Even though Alpaca allows different principals to have different logical axioms for reasoning, it still requires a fixed notion of logical judgments and derivation rules for the *says* connective. As with PCA, this severely restricts the ways in which specific logics encoded in the system can differ from each other.

2. A FRAMEWORK FOR CROSS-DOMAIN DELEGATION

In this section, we describe in detail each component of our framework for cross-domain delegation. First, we enumerate and discuss the elements of the interface that all logics that participate in the framework are required to implement (Section 2.1). Second, we show by example how several fundamentally different logics can implement the abstract portions of the interface, e.g., how they translate their local reasoning into proofs of $\mathsf{seal}(K, H, F)$ and how they can import and make use of a proof of $\mathsf{seal}(K, H, F)$ (Section 2.2). Finally, we show how our framework can be used for the example of Alice attempting to access $\mathsf{doc1}$ (Section 2.3).

In order for any amount of interoperation between systems to be possible, the systems have to agree on a common syntax in which this interoperation is going to take place. In our case, we start by defining all the logics and our interface in LF^2, a common language for defining logics [19]. The choice of LF is arbitrary; any of several such languages (e.g., Coq [31] or HOL [16]) would have done as well. It is important to note that this places no restriction on the logics themselves with respect to expressiveness, the axioms they support, or any other property.
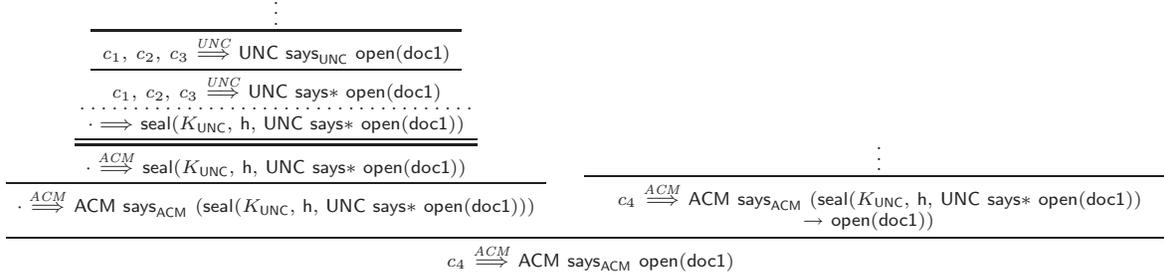
2.1 Core Definitions

The key primitive our framework provides is $\mathsf{seal}(K, H, F)$. Its type, given in LF, is

$$\mathsf{seal} : \mathsf{str} \to \mathsf{str} \to \mathsf{form} \to \mathsf{form}.$$

to indicate that the key and hash arguments are strings, F is a formula, and the type of seal itself is also a formula.

Our intended use of seal is to allow proofs expressed in one logic to be packaged together with the definition of the logic so that they can be verified and used within proofs of another logic. Continuing the example from Section 1: to access $\mathsf{doc1}$, Alice will have to submit to ACM a proof, in ACM's logic, of $\mathsf{ACM\ says_{ACM}\ open(doc1)}$. Part of this proof (namely, that UNC allows the access) will be specified in UNC's logic, while the main proof will be specified in ACM's logic. Figure 1 sketches a proof that Alice might submit to ACM. Since ACM doesn't understand UNC's logic and UNC doesn't understand ACM's, $\mathsf{seal}(K_{\mathsf{UNC}}, \mathsf{h}, \mathsf{UNC\ says*\ open(doc1)})$, which is understood by both logics, will act as the interface between the two. A sketch of logics cooperating using our framework is shown in Figure 2. Here, a thin interface shared by all domains defines the syntax of packaged proofs like $\mathsf{seal}(K_{\mathsf{UNC}}, \mathsf{h}, \mathsf{UNC\ says*\ open(doc1)})$. We include in our framework, in addition to the definition of seal, a mechanism that will allow a proof in UNC's logic to be exported as a proof of $\mathsf{seal}(K_{\mathsf{UNC}}, \mathsf{h}, \mathsf{UNC\ says*\ open(doc1)})$. Similarly, we include a mechanism for ACM to import this

---

[2] We use an enhanced version of LF that supports strings in the style of Twelf [30].

$$\vdots$$

$$\frac{c_1,\ c_2,\ c_3 \xLongrightarrow{UNC} \textsf{UNC says}_{\textsf{UNC}}\ \textsf{open(doc1)}}{\begin{array}{c} c_1,\ c_2,\ c_3 \xLongrightarrow{UNC} \textsf{UNC says}*\ \textsf{open(doc1)} \\ \hdashline \cdot \Longrightarrow \textsf{seal}(K_{\textsf{UNC}},\ \textsf{h},\ \textsf{UNC says}*\ \textsf{open(doc1)}) \end{array}}$$

$$\frac{\cdot \xLongrightarrow{ACM} \textsf{seal}(K_{\textsf{UNC}},\ \textsf{h},\ \textsf{UNC says}*\ \textsf{open(doc1)})}{\cdot \xLongrightarrow{ACM} \textsf{ACM says}_{\textsf{ACM}}\ (\textsf{seal}(K_{\textsf{UNC}},\ \textsf{h},\ \textsf{UNC says}*\ \textsf{open(doc1)}))}$$

$$\vdots$$

$$\frac{}{c_4 \xLongrightarrow{ACM} \textsf{ACM says}_{\textsf{ACM}}\ (\textsf{seal}(K_{\textsf{UNC}},\ \textsf{h},\ \textsf{UNC says}*\ \textsf{open(doc1)}) \rightarrow \textsf{open(doc1)})}$$

$$c_4 \xLongrightarrow{ACM} \textsf{ACM says}_{\textsf{ACM}}\ \textsf{open(doc1)}$$

where
c_1 = $\textsf{UNC signed}_{\textsf{UNC}}\ ((\textsf{Alice says}_{\textsf{UNC}}\ \textsf{open(doc1)}) \rightarrow (\textsf{UNC.members says}_{\textsf{UNC}}\ \textsf{open(doc1)}))$
c_2 = $\textsf{Alice signed}_{\textsf{UNC}}\ (\textsf{open(doc1)})$,
c_3 = $\textsf{UNC signed}_{\textsf{UNC}}\ (\textsf{UNC.members says}_{\textsf{UNC}}\ \textsf{open(doc1)} \rightarrow \textsf{open(doc1)})$,
c_4 = $\textsf{ACM signed}_{\textsf{ACM}}\ (\forall \textsf{h}.(\textsf{seal}(K_{\textsf{UNC}},\ \textsf{h},\ \textsf{UNC says}*\ \textsf{open(doc1)}) \rightarrow \textsf{open(doc1)}))$

Figure 1: Sketch of Alice's proof of access to doc1. The judgment $c_1,\ldots,c_n \xLongrightarrow{ACM} F$ means that using credentials $c_1,\ldots,c_n$ it is possible, in ACM's logic, to derive a proof of F. The dotted line denotes an application of export and the double line denotes an application of import.

proof into its own logic so that it can be used toward the larger proof, as well as to verify that this proof of seal is valid given the set of digitally signed certificates that accompanies it. This verification will implicitly include verification that the proof of UNC says∗ open(doc1) is valid in UNC's logic.

It follows, then, that although ACM and UNC may not agree on the details of whether or how something can be proved, and even on whether a particular formula is a valid statement in the logic, they have to agree that there is a notion of *proving* a formula, and that such proofs are based on digitally signed certificates that represent components of a security policy. To allow each logic to decide on the content of digital certificates (i.e., what formulas they may carry), we define certificates as a 3-tuple of strings: the signer's public key, the hash of the logic in which the content of the certificate is specified, and the (uninterpreted) content.

```
cert : type.
cert-signed : str → str → str → cert.
cert-list : type.
cert-nil : cert-list.
cert-cons : cert → cert-list → cert-list.
```

These definitions establish that certificates have the form cert-signed(key,hash,content), that certificates can be marshaled (using cert-cons) into a list (cert-list) and that the list may be empty (cert-nil). Modeling the content of the certificate as a string makes it possible to have a shared notion of a certificate without placing any constraints on the specific formulas each logic may choose to allow certificates to carry. In general, we will leave such decisions to the individual logics and we will not require that the different logics agree in their implementations.

The next set of definitions codifies the notion of proving that a formula is true.

Figure 2: An overview of the xDomain framework.

```
ctx : type.
ctx-nil : ctx.
form : type.
prove : ctx → form → type.
```

These definitions establish that everyone must agree that a formula (of type form) is proved based on the assumptions in some context (of type ctx). In other words, an object of type prove(c,f) is a proof that the assumptions c imply that the formula f is true. As with the content of certificates, we do not specify valid shapes for formulas or contexts, except that a context may be empty (ctx-nil).

Conceptually, we can divide the procedure of producing a proof of seal($K_{\textsf{UNC}}$, h, UNC says∗ open(doc1)) into two stages: the first is to construct a proof of UNC says∗ open(doc1); the second is to package this proof with references to the proper logic definition and the certificates that support the proof.

```
check : cert-list → str → {c : ctx}{f : form}prove c f → type.
local : cert-list → str → form → type.

export : {g : cert-list}{h : str}{f : form}{c : ctx}{e : prove c f}
        check g h c f e →
        local g h f.
```

The first definition, check, specifies the type for a predicate that relates digitally signed certificates (cert-list) to the logical context c used to prove a formula f.[3] Abstracting the relationship between certificates and the logical context allows our framework to work with logics that use different definitions of formulas that certificates may carry. We will show examples of such in Section 2.2. check takes five arguments: the list of certificates, the hash of the logic, the logical context, the formula f to be proved, and the proof of f under context c. By defining a predicate of this type, UNC will provide a link between the digitally signed certificates that will be passed to ACM and the logical context that was used to prove UNC says∗ open(doc1).

Once a proof has been packed into a self-sufficient package with its certificates, it has the type local(g,h,f), where g is the list of certificates, h is the hash of the definition of the local logic, and f is the formula whose proof local represents.

---

[3]The notation $\{x{:}T_1\}T_2$ describes a function that takes an argument of type T_1 and produces a type T_2. When x is not free in T_2, $T_1 \rightarrow T_2$ is the shorthand for $\{x : T_1\}T_2$.
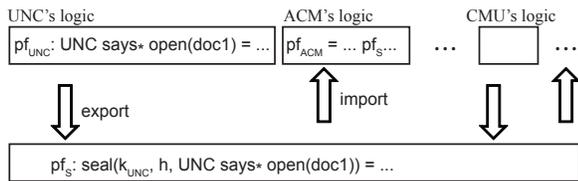
This encapsulated proof is produced by the export function, whose type is shown above.

Next we define import, which is used to import an encapsulated proof into a different logic so that it can be used to make inferences in that logic.

```
import : {k : str}{u : str}{h : str}{g : cert-list}{f : form}
         local g h f →
         prove ctx-nil (seal k h f).
```

import takes as arguments the public key k of the logic's author, e.g., K_{UNC}; the URL u from which the logic definition can be downloaded; the hash h of the logic defined at this URL; the list of certificates g that were used in constructing the proof; the proved formula f; and the package produced by exporting a proof of f. It produces the proof of seal(k, h, f).

Finally, we define the minimal set of formulas f that can appear in seal(k,h,f). Later, in Section 3, we will describe how to generalize our framework to make it unnecessary to globally define this set of formulas, and to allow sets of individual logics to communicate with sets of formulas capturing other notions (e.g., delegation).

```
says* : str → form → form.
open : str → str → form.
```

These definitions specify that says∗ takes as arguments a string that represents a principal's public key and a formula that represents that principal's belief, and is itself a formula. The only other formula is open, which takes as parameters two strings: one to identify the resource to access, the other (which we will typically omit for readability) a nonce to ensure that proofs of access cannot be replayed. With these definitions in place, the formula seal(K_{UNC}, h, K_{UNC} says∗ open(doc1)) is finally well defined.

The full interface that we have described piecemeal in this section is shown in a companion technical report [9].

2.2 Defining Individual Logics

In addition to including the shared components described in Section 2.1, the definition of each logic needs to provide local instantiations of constructs such as prove and check. We will illustrate how these constructs can be implemented in several different logics. The first example is a constructive authorization logic (Section 2.2.1); the second is a constructive, linear authorization logic (Section 2.2.2); the last, detailed in our companion technical report [9], is a constructive, timed authorization logic. Each of the three examples is representative of a category of access-control logics (although there are other categories, as well). The distinct formal feature of the logic in each example is that each uses a different form of logical judgment. For all other logics that use the same form of judgment, the method of implementing the global interface will be similar to what we show here.

These examples also help demonstrate the generality of our framework and show that providing local instantiations of the abstract parts of the interface is fundamentally similar across different logics. The linear and timed authorization logics are more expressive than the constructive authorization logic with which they interoperate, and so these examples also illustrate how our interface helps bridge the different levels of expressiveness.

2.2.1 A Constructive Authorization Logic

Figure 3 shows the LF encoding of a relatively standard authorization logic, which we presume ACM uses in our run-

```
 1  %% logical connectives
 2  acm/imp : form → form → form.
 3  ...
 4  %% context
 5  acm/form-list : type.
 6  acm/form-nil : acm/form-list.
 7  acm/form-cons : form → acm/form-list → acm/form-list.
 8  acm/context : acm/form-list → ctx.
 9
10  %% check certs
11  acm/check_certs : cert-list → str → ctx → type.
12  acm/check_base : check G H C F E
13                    ← acm/check_certs G H C.
14
15  %% parsing functions from string
16  %% in the certificates to abstract syntax
17  acm/parse : string → form → type.
18  ...
19  %% check_certs is simply a map from a list of certificates
20  %% to a list of formulas
21  acm/check_certs_base :
22      acm/check_certs cert-nil H (acm/context acm/form-nil).
23  acm/check_certs_init :
24      acm/check_certs
25        (cert-cons (cert-signed K H (const S)) Certs) H
26        (acm/context (acm/form-cons (acm/signed K F) Ctx))
27      ← acm/check_certs Certs H (acm/context Ctx)
28      ← acm/parse S F.
29
30  acm/ctx-nil-weak :
31      prove (ctx-nil) (seal K H F) →
32      prove (acm/context acm/form-nil) (seal K H F).
33
34  %% inference rules
35  ...
```

Figure 3: Definition of ACM's logic.

ning example. We prefix the definitions local to ACM's logic with acm/. Starting with Line 2, the connectives of ACM's logic are defined. Here we only show one connective, implication: imp A B is a formula if both A and B are formulas. Other connectives are defined similarly. The type for formulas used here is the type form declared in the shared interface. In other words, ACM's logic provides local instantiations of the abstract concept formulas.

Next, starting from Line 4, we define ACM's logical context. In this particular implementation, the logical context is implemented as a list of formulas. acm/form-list is the type for a list of formulas. acm/nil is the empty list and acm/cons concatenates one element with a list to produce a new list. Line 8 connects a list of formulas to the logical context defined in the interface. acm/context is a construct that wraps up a list of formulas and produces a logical context understood by the interface. Line 11 defines the type for ACM's check predicate, which maps the list of certificates to a logical context. In this particular implementation, there is a straightforward mapping between the list of formulas in the logical context and the list of certificates. acm/check_certs takes as arguments the list of certificates, the logical context, and the hash of ACM's logic. acm/check_base on Line 12 has a similar purpose as acm/context on 8: it is used to link ACM's acm/check_certs with the check function defined in the interface. On Line 17, we define a parse function that converts a string to a valid formula in ACM's logic. Parsing is used for checking the list of certificates. On Lines 21–29, we instantiate the acm/check_certs predicate. The first case

handled by the function is when the lists are empty. The second case is the recursive case where we check that the heads of the lists correspond to each other, after which we recursively check the tails of the lists.

The above-described additions to ACM's local logic allow ACM's logic to smoothly interface with import and export. In some sense, the local details are wrapped up using constructs such as acm/context and acm/check_base to produce an object of the type specified in the interface. Lines 30–32 define a rule to inform ACM how ACM's local logic can make use of the sealed formula. The reason for having this rule is that the proof of seal is an abstract object provided through the interface that needs to be rewritten in ACM's local terms before it can interact with other proofs in ACM's logic. The rest of the logic definition is composed of definitions of derivation rules, which are standard.

In this definition, ACM uses the notion of judgment (prove) from the interface to formulate its own logical rules. This is legitimate; however, ACM could also maintain its own formulation of the logical rules and add wrappers to translate a proof in ACM's logic to an object of type prove C F. We show an example of such wrappers for an authorization logic with explicit time in a companion technical report [9].

2.2.2 A Linear Authorization Logic

Let us assume that CMU uses a linear authorization logic, which enforces the property that some credentials can be used in a proof only once. A fragment of CMU's logic definition is shown in Figure 4.

The structure of the logic definition is very similar to that of ACM's non-linear logic. The main difference is that the logical context is no longer a list of formulas, but a pair of lists of formulas. One element of the pair represents the linear logical context where contraction and weakening are not allowed and the other is an unrestricted context where contraction and weakening are permitted.

Relating certificates to the logical context becomes more complicated, too. Some certificates correspond to assumptions in the linear context and others correspond to assumptions in the unrestricted context. Following the approach of Bowers et al. for implementing a linear authorization logic [11], a linear assumption cmu/signed K F corresponds to two certificates: cert-signed K H linear(F, K′) and cert-signed K′ H valid(K, F, hash_e). The second certificate is created by a ratifying agency attesting that the "linear" credential (cmu/signed K F) that is used linearly in a proof e (with hash hash_e) had not been used in previous proofs. ACM and CMU can have their own interpretations of logical contexts and checks; the details of how this is done are hidden below the interface.

Finally, the cmu/trans-says∗ rule allows CMU to prove says∗ K F if CMU proves cmu/says K F. Though we did not show it in Figure 3, ACM's logic contains a similar rule for elevating acm/says to says∗. cmu/ctx-nil-weak is the logical rule stating that if seal is proved under an empty context, then, in CMU's logic, seal L H F is proved. This is similar to the rule in ACM's logic except now the context is different.

2.3 A Cross-domain Access Control Example

In this section, we explain how to use our framework to put together a proof that takes advantage of cross-domain delegation. We continue with the ACM Digital Library example, this time considering how a user, Bob, in the CMU

```
1   %% list of formula defs and operations
2   cmu/form-list : type.
3   cmu/form-nil : cmu/form-list.
4   cmu/form-cons : form → cmu/form-list → cmu/form-list.
5
6   %% logical context for linear logic,
7   %% the context is a pair of formula lists
8   cmu/context : cmu/form-list → cmu/form-list → ctx.
9
10  %% logical connectives
11  cmu/imp : form → form → form.
12  ...
13  %% parse certificates
14  cmu/parse : string → form → type.
15  ...
16  %% check certificates
17  cmu/check_certs : cert-list → str → ctx → str → type.
18
19  cmu/check_base : check G H C F E
20                ← cmu/check_certs G H C (cmu/hash E).
21
22  %% linear formula (cmu/signed K goal) needs two credentials
23  %% 1. cert-signed K H linear(goal, K′) and
24  %% 2. cert-signed K′ H valid(K, goal, hash_e)
25
26  cmu/check_certs_lin :
27    cmu/check_certs
28      (cert-cons (cert-signed K HD (const S))
29        (cert-cons (cert-signed K′ HD (const S′))
30          Certs))
31      HD
32      (cmu/context U (cmu/form-cons (cmu/signed K F) Ctx))
33      H
34      ← cmu/check_certs Certs HD (cmu/context Ctx L) H
35      ← cmu/parse/lin S K′ F
36      ← cmu/parse/valid S′ (cmu/signed K F) H.
37  ...
38  cmu/trans-says∗ : prove C (cmu/says K F)
39      → prove C (says∗ K F)
40  cmu/ctx-nil-weak : prove (ctx-nil) (seal K H F)
41      → prove (cmu/context cmu/form-nil cmu/form-nil)
42          (seal K H F).
43
44  %% Logic inference rules
45  ...
```

Figure 4: Definition of CMU's logic.

domain can access a document. In the example scenario, ACM's policy states that ACM will grant access to a document if one can show that CMU will grant access to the document. ACM uses a constructive authorization logic, and CMU uses a constructive linear logic (defined in Sections 2.2.1 and 2.2.2).

Before going into the details of the proof, we introduce two logical formulas in ACM's and CMU's logics that allow delegations to be made more succinctly. These new formulas are just syntactic sugar: they make it more convenient to write proofs, but are implemented merely as definitions on top of the respective authorization logics.

The first new formula we will use is acm/delegate_seal. It takes as arguments the recipient of a delegation and the name of the document to which access is being delegated, and is defined in terms of more basic formulas as follows.

```
acm/delegate_seal delegatee doc =
  forall h, (seal delegatee h (says∗ delegatee (open doc)))
          → (open doc)
```

```
1   pf : prove ctx-nil
2       (seal (const "k_acm")
3             (const "acm-logic-hash")
4             (says∗ (const "k_acm") (open (const "doc2"))))
5   = import (const "k_acm")(const "acmurl")
6             (const "acm-logic-hash")
7       %% certificates
8       (cert-cons (cert-signed
9                   (const "k_acm")
10                  (const "acm-logic-hash")
11                  (const "(acm/delegate_seal cmu (doc2))"))
12              cert-nil)
13      %% formula to prove
14      (says∗ (const "k_acm") (open (const "doc2")))
15      (export (cert-cons ...) %% certificates, same as above
16      %% formula to prove
17      (says∗ (const "k_acm") (open (const "doc2")))
18      (const "acm-logic-hash")
19      %% logical context
20      (acm/context
21        (acm/form-cons
22        (acm/signed (const "k_acm")
23            (acm/delegate_seal (const "k_cmu") (const "doc2")))
24        acm/form-nil))
25      (acm/says (const "k_acm") (open (const "doc2")))
26      %% proofs
27      (acm/trans-says∗
28        ...
29        (acm/ctx-nil-weak
30          %% a subproof of seal (cmu, h, cmu says∗ open doc2)
31          (import (const "k_cmu") (const "cmuurl")
32            (const "hash of cmu logic")
33            (cert-cons %% certificate
34                ... )
35            (says∗ (const "k_cmu") (open (const "doc2")))
36            (export
37              (cert-cons ...) %% same certificates as above
38              (const "hash of cmu logic")
39              (says∗ (const "k_cmu") (open (const "doc2")))
40            %% logical context
41            (cmu/context
42                ... )
43            (says∗ (const "k_cmu") (open (const "doc2")))
44            %% proofs
45            (cmu/trans-says∗ ...)
46            %% cmu check
47            (cmu/check_base ...)
48            )))))
49        )
50      %% check that the context match the certificates
51      (acm/check_base (...))).
```

Figure 5: A proof with cross-domain delegation.

Second, we define cmu/delegate similarly.

cmu/delegate delegatee doc =
   (cmu/says delegatee (open doc)) → (open doc)

We now proceed to describe the construction of the proof that will allow Bob access to an ACM document. ACM creates the following credential to express its policy that access is being delegated to CMU.

acm/signed (const ''k_acm'')
              (acm/delegate_seal (const ''k_cmu'')
                              (const ''doc2''))

CMU's policy is to delegate one-time access to a user Bob.

cmu/signed (const ''k_cmu'')
              (cmu/delegate (const ''k_Bob'') (const ''doc2''))

Bob wants to open the document doc2 and hence issues a credential to indicate his desire to do so.

cmu/signed (const ''k_Bob'') (open (const ''doc2''))

Instead of creating just a proof p (as we showed informally in Figure 1) of

acm/says (const ''k_acm'') (open (const ''doc2''))

Bob must use p to produce a proof pf of

seal (const ''k_acm'') (''acm-logic-hash'')
        (says∗ (const ''k_acm'') (open (const ''doc2'')))

The advantage of pf over p is that part of pf is an explanation of how the digitally signed certificates supplied as part of pf match the logical context used to prove p. This makes it possible for pf to be interpreted by any reference monitor; conversely, to interpret p, a reference monitor would need to understand the mapping of certificates to context that is specific to ACM's logic.

We show the key parts of Bob's proof in Figure 5. The formula we are trying prove is on Lines 2–4. The proof of this formula starts at Line 5. Since we are trying construct a proof of seal (...) the proof uses import (Line 5) and export (Line 15). Lines 8–12 are the logical descriptions of the certificate that states ACM's policy. The logical context of the proof in ACM's logic contains exactly the formulas representing ACM's policy. Lines 27–49 are a proof of

says∗ (const ''k_acm'') (open (const ''doc2''))

specified in ACM's logic. The last argument to import (Line 5) is the check on Line 51 that maps ACM's certificates to ACM's logical context.

Now let us examine the structure of the proof between lines 27 and 49. The outermost rule, acm/trans-says∗, takes a proof of (acm/says (const ''k_acm'') (open (const ''doc2''))) and produces a proof of (says∗ (const ''k_acm'') (open (const ''doc2''))). This illustrates how to translate a proof of a formula that is specific to ACM's logic to a proof of a formula that is known to the common interface.

On line 29, the acm/ctx-nil-weak rule takes a subproof of

prove (ctr-nil)
      seal (const ''k_cmu'') (const ''hash of cmu logic'')
          (says∗ (const ''k_cmu'') (open (const ''doc2''))))

known as an abstract object from the interface and interprets it as a logical statement that is known to ACM's logic:

prove (acm/context acm/form-nil)
      seal (const ''k_cmu'') (const ''hash of cmu logic'')
          (says∗ (const ''k_cmu'') (open (const ''doc2''))))

The subproof between Lines 30 and 48 is again constructed using import and export, but this time using CMU's logic. The proof structure is very similar to the proof between Lines 5 and 51. This time, the mapping between the certificates and the logical context is more complicated because the context is linear.

3. PAIR-WISE SHARING

In the framework we proposed in the previous section, we assumed that all participants implement the same interface. While most of the interface is sufficiently generic that it is reasonable for it to be fixed for all participants, it is less reasonable to predefine the set of formulas, such as says∗ and

open, that are used for delegation. For example, suppose that IEEE decides to use our framework to allow institutions access to its journals in electronic form. IEEE could have a similar policy as ACM, stating that if CMU allows access to a document to a principal, then IEEE will also allow that access. Suppose, further, that IEEE distinguishes between granting regular access to journals (via open) and granting administrative access (admin); the latter allows posting of comments in online forums. Now, IEEE's policy for granting administrative access is stated as follows, provided that admin is shared between IEEE and CMU.

```
ieee/sign (const ''k_ieee'')
  (forall doc,
     seal (const ''k_cmu'') H
          (says* (const ''k_cmu'') (admin (const ''doc'')))
     → admin (const ''doc''))
```

Notice that admin was not one of the shared formulas CMU previously knew about, since it was not specified in our framework's interface. This suggests that there is a need for our framework to support different interfaces for different groups of domains, so that individual domains can delegate with sufficient expressiveness to fulfill their needs (e.g., distinguishing between access via admin and access via open). Fortunately, the only part of the framework's interface that needs to be customized is the part that defines the formulas that can be used to delegate between logics.

In this section, we generalize our framework to deal with cross-domain delegation between groups of principals; we call this pair-wise cross-domain delegation.

In pair-wise delegation, there is a distinction between globally shared and pair-wise shared components. The interface (import/export) is still globally shared, but the common set of formulas for delegating (open, says*, admin) is pair-wise shared. That is, the set of formulas used for delegating between ACM and CMU is different from the set used between IEEE and CMU. Moreover, this requires each individual logic to implement one stub per set of pair-wise delegation formulas. For instance, suppose CMU and ACM agree to use says*, while CMU and IEEE agree to use says+. As a result, CMU needs to implement

```
cmu/trans-says* : prove C (cmu/says K F) →
                    prove C (says* K F)
```

to interface with ACM; and

```
cmu/trans-says+ : prove C (cmu/says K F) →
                    prove C (says+ K F)
```

to interface with IEEE. However, the two above rules are never both required at the same time; which rule is needed depends on whether CMU is communicating with ACM or IEEE. More concretely, CMU's packaged proof, created using import and export, needs to specify which subset of CMU's logic definition is needed to interpret the proof.

This suggests that we need to enrich the import interface to specify not just the definition of the individual logic used to create the proof that is being imported, but also the pair-wise shared interface agreed upon by the importer and exporter. Hence, we revise import to specify, by the hash of its contents, a specific pair-wise interface.

```
import : {k : str}{u : str}{h : str}{hpw : str}
         {g : cert-list}{f : form}
         local g h f →
         prove ctx-nil (seal k h f).
```

Once both the individual logic (e.g., CMU's) and the pair-wise interface (e.g., the one shared by IEEE and CMU) have been identified, the pair uniquely identifies the set of additional definitions that need to be loaded (e.g., CMU's definitions for implementing the pair-wise interface shared by IEEE and CMU). To make these components easy to locate when importing a proof, we decide on the following convention: given an import statement like import k u h hpw ..., the definition of k's logic can be found at u/main, the pair-wise interface at u/pairwise/hpw, and the implementation of the pair-wise interface at u/stubs/hpw.

4. CHECKING AND BUILDING PROOFS

In addition to developing an interface that allows different logics to interoperate, our goal in designing our framework was to make it possible for reference monitors to use a single, logic-independent proof checker to verify the validity of proofs that are submitted to it. We describe such a checker in Section 4.1.

A secondary, but important, concern is how proofs of access are generated. For our framework to be practical, it must support the ability to generate proofs of access in an automated way. We discuss considerations relevant to automated proof generation and a preliminary design of a proving framework in Section 4.2.

4.1 Checker

For simplicity, we will describe a checker that implements the simplified version of our framework as defined in Section 2. The enhancements to this checker required to check proofs specified in the full version of our framework (as described in Section 3) are straightforward.

The interesting proofs in our framework are composed of nested subproofs that are specified in different authorization logics. Following the example of Section 2.3, a proof specified in ACM's logic may contain a subproof in CMU's logic. The border between the two parts is composed of subsequent applications of the export and import rules. The application of the export rule and the subproofs to which it is applied are all in CMU's logic; the application of the import rule and the remainder of the enclosing proof to which it contributes are specified in ACM's logic. Since this separation is, by design, clean, the main challenge in designing and building a proof checker is to provide a mechanism that will ensure that each component of a proof that is specified in a particular logic will be verified with respect to the definition of that logic.

As an occurrence of the import rule in a proof is the indicator that a subproof will be specified in a different logic, we enhance our standard LF checker to specially interpret such occurrences. More specifically, when our checker encounters a proof component like import key url hash cert-list fmla pf, it will perform the following steps.

1. Download from url the definition of the logic used in key's domain.
2. Verify that the downloaded logic definition is signed with the private key that corresponds to the (public) key and that the hash of the logic definition matches hash.
3. Verify that each certificate specified in cert-list corresponds to a valid binary certificate packaged with the proof.
4. Create a new instance of a checker, and initialize it with the downloaded logic definition.
5. Use the newly created instance of the checker to check pf.

Table 1: Checker costs in example of Section 2.3 in ms (averaged over 50 runs on an Intel 2.83GHz Core 2 Quad processor with 4GB RAM; fetches occur over 100Mb/s LAN).

Loading ACM logic	**10**
Fetching	3
Parsing	6
Hashing (SHA1)	1
Loading CMU logic	**6**
Fetching	1
Parsing	4
Hashing (SHA1)	1
Proof loading	**37**
Certificate parsing	7
Certificate verification	**3**
Typechecking (beyond parsing logics)	**21**
Total	**77**

Failure to complete any step will result in the overall proof being rejected.

Since we use LF as the language in which we define our framework and the logics that make use of it, the bulk of the checker for verifying logical proofs in our framework is a standard LF checker. In fact, to verify proofs that are specified in a single logic and do not use the import rule, an off-the shelf LF checker is sufficient.

For our prototype checker, example costs for these steps in checking the proof of the example in Section 2.3 are shown in Table 1. Since downloading a logic definition, verifying it, and using it to initialize a fresh checker is time-consuming, we can as an optimization maintain instantiated checkers for later use in checking proofs specified in the same logics.

4.2 Prover

In this section, we explore the design space for a prover that can automatically generate proofs using the cross-domain delegation framework. Again, we use ACM's Digital Library as example. In our framework, if Alice, a member of CMU's domain, wants to access ACM's document, she needs to create a proof that (ACM says$_{\text{ACM}}$ open(doc2)). As discussed, this proof contains a part specified in ACM's logic and a subproof in CMU's logic. Since this prover has to search for proofs in different domains, it inherits many challenges known for distributed proving (e.g., [26, 7, 8]), including where to locate useful credentials that belong to each domain, how to usefully interact with the user, and what needs to be communicated between each domain-specific prover. While we can use many of the ideas from existing distributed provers, there are also problems that are specific to our framework due to the heterogeneity of the domains. For example, ACM and CMU could use wildly different proof search strategies: they could use forward reasoning, or backward chaining, or even some distributed proving algorithm. The challenging question for developing a prover for this cross-domain framework is to decide on the minimal set of restrictions for each domain's native theorem prover that will make cross-domain proof search feasible.

In our proposed prover design, we assume that each domain that participates in cross-domain delegation maintains a clearinghouse from which any member of that domain can download (1) provers for foreign domains that have delegated to the local domain and (2) the certificates by which the foreign domains delegate to the local domain. For our example, this means that CMU will have a downloadable copy of ACM's prover and the credential by which ACM delegated to CMU. The goal of making provers portable in this manner is to permit Alice to engage in the equivalent of one of the previously studied approaches to distributed proving [26, 7, 8], except that now multiple provers will all be co-located on Alice's computer.

In our scenario, the proof goal is passed from one prover to another when a prover determines that the goal is specified in a different logic from that implemented by the first prover. For example, Alice will embark on generating a proof of (ACM says$_{\text{ACM}}$ open(doc2)) by locally running a copy of ACM's prover and asking it to prove this goal. ACM's prover will eventually attempt to prove the subgoal

seal(K, H, CMU says* open(doc2)),

at which point it will detect that this subgoal is specified in CMU's logic and will invoke CMU's prover.

In general, whenever a prover encounters a subgoal of seal(K,H,F), it needs to send the proof obligation of F to K's prover. We can view F as a communication channel between two logics. If F is allowed to contain unification variables, K's prover is obliged to produce a unification of those variables. This gives rise to a problem: since unification variables are considered part of a prover's state, and are now being implicitly shared across provers, we have widened the interface on which logics (and their provers) have to agree. Hence, we restrict F to be closed (i.e., not contain unification variables). Although this is in general a significant restriction, for proofs of the nature we consider here, this restriction is not a problem, since CMU is only asked to prove statements like CMU says* open(doc2), which is closed.

5. FORMAL PROPERTIES

Authorization logics are useful only if they are correct. To ensure correctness, theorems are often proved to demonstrate that logics have certain desirable properties. The most common such property is consistency, meaning that the logic cannot prove false under no assumptions.

It is important that the interface defined by our framework respects the consistency of the individual authorization logics that implement the interface. We explain in this section why the interface we describe in Section 2 indeed preserves the consistency of the individual authorization logics. Moreover, we have formally proved theorems stating that the interface maintains the consistency property for the three authorization logics we discussed in Section 2.

Interfacing with the framework introduces new logical rules such as ctx-nil-weak (see Figure 3) into the local authorization logic of each participant in cross-domain delegation. At a high-level, the reason that adding new rules in a local logic preserves the consistency of the logic is because the sealed formulas are used only atomically. The local logic does not reason about seal(L, H, F) in the same way that it does about formulas such as (A and B), where if (A and B) is true, then both formula A and formula B can be assumed to be true. A logic might have inference rules that give meaning to the subcomponents of the formula A and B. When a proof of formula A is needed, a proof of A and B will do. On the other hand, seal(L, H, F) is treated as an atomic formula, just like an atomic predicate such as open. No logical rules give additional meaning to these atomic constructs. A proof of seal(L, H, F) can only be used in the places where a proof

of seal(L, H, F) is needed; it cannot be used to help create a proof of F or other unrelated formulas. Adding ctx-nil-weak does not add additional proving power to the logic; therefore, if false could not be proved without ctx-nil-weak, it cannot be proved with it, either.

One way to prove the consistency of a logic defined in the style of sequent calculus is to prove the cut-elimination theorem. We have proved cut-elimination theorems for the logics formalized in Section 2.2 before they were augmented to implement the cross-domain interface. We need to prove that the cut-elimination theorems still hold after the addition of the new rules required by the cross-domain interface.

We can indeed prove that, because in any logic that implements our interface there will be only one rule that can be used to infer seal(L, H, F), and no rule that takes seal(L, H, F) as a premise. If a proof happens to use seal(L, H, F) as an intermediate step (via the cut rule), then we can eliminate this use of the cut rule by pushing the use of cut further up in the proof tree. If we repeat this step, we will in the end eliminate the use of seal(L, H, F) as an intermediate step altogether. In more technical terms, this means that in a proof of cut-elimination, the cases involving seal(L, H, F) can only be commutative or init cases [29]. Consequently, the cut-elimination proof for a local logic before it is augmented with rules to implement our interface would already have covered the proof cases that are relevant for seal(L, H, F).

6. REFERENCES

[1] M. Abadi, M. Burrows, B. Lampson, and G. D. Plotkin. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(4):706–734, Sept. 1993.

[2] M. Abadi, E. Wobber, M. Burrows, and B. Lampson. Authentication in the Taos Operating System. In *Proceedings of the 14th ACM Symposium on Operating System Principles*, pages 256–269, Dec. 1993.

[3] A. W. Appel and E. W. Felten. Proof-carrying authentication. In *Proceedings of the 6th ACM Conference on Computer and Communications Security*, 1999.

[4] D. Balfanz, D. Dean, and M. Spreitzer. A security infrastructure for distributed Java applications. In *Proceedings of the 2000 IEEE Symposium on Security & Privacy*, 2000.

[5] L. Bauer. *Access Control for the Web via Proof-carrying Authorization*. PhD thesis, Princeton University, Nov. 2003.

[6] L. Bauer, S. Garriss, J. M. McCune, M. K. Reiter, J. Rouse, and P. Rutenbar. Device-enabled authorization in the Grey system. In *Information Security: 8th International Conference, ISC 2005*, volume 3650 of *Lecture Notes in Computer Science*, pages 431–445, Sept. 2005.

[7] L. Bauer, S. Garriss, and M. K. Reiter. Distributed proving in acess-control systems. In *Proceedings of the 2005 IEEE Symposium on Security & Privacy*, 2005.

[8] L. Bauer, S. Garriss, and M. K. Reiter. Efficient proving for practical distributed access-control systems. In *Computer Security—ESORICS 2007: 12th European Symposium on Research in Computer Security*, volume 4734 of *Lecture Notes in Computer Science*, pages 19–37, Sept. 2007.

[9] L. Bauer, L. Jia, M. K. Reiter, and D. Swasey. xDomain: Cross-border proofs of access. Technical Report CMU-CyLab-09-005, CyLab, Carnegie Mellon University, Mar. 2009.

[10] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. D. Keromytis. *The KeyNote trust-management system, version 2*, 1999. IETF RFC 2704.

[11] K. D. Bowers, L. Bauer, D. Garg, F. Pfenning, and M. K. Reiter. Consumable credentials in logic-based access-control systems. In *Proceedings of the 2007 Network & Distributed System Security Symposium*, pages 143–157, Feb. 2007.

[12] J. DeTreville. Binder, a logic-based security language. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, 2002.

[13] H. DeYoung, D. Garg, and F. Pfenning. An authorization logic with explicit time. In *Proceedings of the 21st IEEE Symposium on Computer Security Foundations (CSF-21)*, 2008.

[14] D. Garg, L. Bauer, K. D. Bowers, F. Pfenning, and M. K. Reiter. A linear logic of authorization and knowledge. In *Computer Security—ESORICS 2006: 11th European Symposium on Research in Computer Security*, volume 4189 of *Lecture Notes in Computer Science*, pages 297–312, Sept. 2006.

[15] D. Garg and F. Pfenning. Non-interference in constructive authorization logic. In *Proceedings of the 19th Computer Security Foundations Workshop (CSFW'06)*, 2006.

[16] M. J. C. Gordon and T. F. Melham. *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[17] J. Halpern and R. van der Meyden. A logic for SDSI's linked local name spaces. *Journal of Computer Security*, 9(1,2):47–74, 2001.

[18] J. Y. Halpern and V. Weissman. Using first-order logic to reason about policies. In *Proceedings of the 16th IEEE Computer Security Foundations Workshop*, pages 187–201, June 2003.

[19] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, 1993.

[20] J. Howell. *Naming and sharing resources across administrative boundaries*. PhD thesis, Dartmouth College, May 2000.

[21] J. Howell and D. Kotz. A formal semantics for SPKI. In *Proceedings of the 6th European Symposium on Research in Computer Security*, pages 140–158, 2000.

[22] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems*, 10(4):265–310, 1992.

[23] C. Lesniewski-Laas, B. Ford, J. Strauss, R. Morris, and M. F. Kaashoek. Alpaca, a proof-carrying authentication framework for cryptographic primitives and protocols. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, 2007.

[24] N. Li, B. N. Grosof, and J. Feigenbaum. Delegation logic: a logic-based approach to distributed authorization. *ACM Transactions on Information and Systems Security*, 6(1):128–171, Feb. 2003.

[25] N. Li, J. C. Mitchell, and W. H. Winsborough. Design of a role-based trust management framework. In *Proceedings of the 2002 IEEE Symposium on Security & Privacy*, 2002.

[26] K. Minami and D. Kotz. Secure context-sensitive authorization. *Journal of Pervasive and Mobile Computing*, 1(1), 2005.

[27] G. C. Necula. Proof-carrying code. In N. D. Jones, editor, *Proceedings of the Symposium on Principles of Programming Languages*, pages 106–119, Jan. 1997.

[28] G. C. Necula. *Compiling with Proofs*. PhD thesis, Carnegie Mellon University, Oct. 1998.

[29] F. Pfenning. Structural cut elimination I. intuitionistic and classical logic. *Information and Computation*, 157(1/2):84–141, Mar. 2000.

[30] F. Pfenning and C. Schürmann. System description: Twelf—a meta-logical framework for deductive systems. In *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, 1999.

[31] The Coq Development Team. *The Coq Proof Assistant Reference Manual*. LogiCal Project, 2006.