Constraining Credential Usage in Logic-Based Access Control

Lujo Bauer
Carnegie Mellon University
Pittsburgh, PA, USA
Email: lbauer@cmu.edu

Limin Jia
Carnegie Mellon University
Pittsburgh, PA, USA
Email: liminjia@cmu.edu

Divya Sharma
Carnegie Mellon University
Pittsburgh, PA, USA
Email: divyasharma@cmu.edu

***Abstract*—Authorization logics allow concise specification of flexible access-control policies, and are the basis for logic-based access-control systems. In such systems, resource owners issue credentials to specify policies, and the consequences of these policies are derived using logical inference rules. Proofs in authorization logics can serve as capabilities for gaining access to resources.**

Because a proof is derived from a set of credentials possibly issued by different parties, the issuer of a specific credential may not be aware of all the proofs that her credential may make possible. From this credential issuer's standpoint, the policy expressed in her credential may thus have unexpected consequences. To solve this general problem, we propose a system in which credentials can specify constraints on how they are to be used. We show how to modularly extend well-studied authorization logics to support the specification and enforcement of such constraints. A novelty of our design is that we allow the constraints to be arbitrary well-behaved functions over authorization proofs. Since all the information about an access is contained in the proofs, this makes it possible to express many interesting constraints. We study the formal properties of such a system, and give examples of constraints.

***Keywords*-Access control, Logic, Formal languages, Computer security**

I. Introduction

Today's computing environments are characterized by an ongoing dramatic increase in connectivity and data sharing. The number of computing devices and kinds of devices that can communicate with each other is increasing sharply, as is the amount of data and kinds of data that they store, generate, and exchange. These trends are ubiquitous: they are present in business computing environments, which are being transformed by cloud computing; in basic infrastructure, where the Smart Grid is transforming the national power grid; and in home environments, in which the household is increasingly inhabited by dozens of interconnected digital devices ranging from portable media players and digital picture frames to smart refrigerators and Internet-connected security systems. The new functionality arising through this transformation is enabling increases in productivity, efficiency, and safety and giving rise to fundamentally new applications in almost all aspects of our lives.

A critical concern in this setting is access control—the ability to easily and quickly allow access to authorized users or devices, while preventing misuse, unauthorized access, and violations of privacy. Flexible and secure access-control mechanisms are part of what makes the new functionality and applications possible and sustainable.

In the past decade and a half, significant progress has been made in increasing the assurance and expressiveness offered by access-control systems, in large part through the use of formal logics. When used to model a system, such logics can confirm that the system exhibits various desirable correctness properties, e.g., that its decision procedure will never erroneously permit an access if such an access is not consistent with access-control policy. These guarantees can be even stronger when the gap between model and system is bridged by making the logical infrastructure part of the system. An increasingly popular and practical method of accomplishing this is using the proof-carrying authorization (PCA) [4] paradigm. 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.

Because a proof is derived from a set of credentials possibly issued by different parties, the issuer of a specific credential may not be aware of all the proofs that her credential may make possible. From this credential issuer's standpoint, the policy expressed in her credential may thus have unexpected consequences, potentially leading to security breaches, insider attacks, and, generally, policies that are more permissive than is desired.

Our contribution: In this paper, we propose a robust mechanism for allowing credential issuers to specify constraints that restrict the circumstances under which their credentials and the authority they express may be used. Building on the observation that a proof of access describes in full detail the manner in which a credential is used, we allow credential issuers to specify constraints as functions over the proofs in which their credentials are used. This powerful mechanism allows our framework to capture a wide variety of constraints of practical interest, including the following: limiting (re-)delegation depth when delegating authority; enforcing strict revocation policies on all credentials used in a proof; preventing a credential from being used to access resources outside a particular set, which can be specified explicitly or by indirection; and constraining the size of the proof. To make it easier to understand and implement constraints, we classify them in two dimensions: in one, according to their intended use; and, in another, according to the information they need in order to be enforced.

In addition to supporting a wider variety of constraints than other approaches, our framework for enforcing constraints exhibits desirable formal properties. We designed our framework to modularly build on existing authorization logics with robust proof theories. This makes it possible to apply our approach to different authorization logics, and at the same time makes it easy to prove meta-properties about the resulting system, thus providing a high assurance of correctness. We demonstrate this by applying our framework to a specific authorization logic.

After examining our approach in a formal setting, we discuss several issues that needed to be resolved to implement our approach in a practical setting, and propose a detailed implementation strategy for deploying our approach in a proof-carrying authorization framework. We illustrate both the formal treatment of our framework and the discussion of implementation concerns with a range of example constraints, several of which we develop in full detail and show in the context of sample proofs of access.

Roadmap: The rest of this paper is organized as follows. First, in Section II, we review a simple authorization logic that will form the basis for our exploration. In Section III, we further discuss the need for constraining the use of credentials, describe our method for specifying constraints, and illustrate it with examples. We explain how to enforce these constraints, and examine the formal properties of our approach, in Section IV. In Section V we describe in detail how to implement our approach in the context of a proof-carrying authorization system. We discuss the applicability of our framework to other authorization logics in Section VI, and conclude in Section VII with a discussion of related work.

II. Review of a Simple Authorization Logic

We build our constraints on top of a simple core logic for access control, which is based on simply-typed CDD [1]. The syntax of the logic is given below.

$$\begin{array}{ll} \textit{formulas}\ F ::= P \mid F_1 \to F_2 \mid A \ \mathsf{sign}\ F \mid A \ \mathsf{says}\ F \\ \textit{contexts}\ \Gamma ::= \cdot \mid \Gamma, F \end{array}$$

We use P to denote basic predicates, and $F_1 \to F_2$ to denote implication. Formulas A sign F and A says F both state that principal A affirms that F is true. The difference is that A sign F corresponds directly to a credential that A has signed using his private key. The formula A says F denotes that it can be deduced that A believes F from credentials that A and other principals have signed. The main difference between our logic and simply-typed CDD is our treatment of credentials. CDD does not distinguish between directly asserted facts and derived facts, whereas we do.

$$\frac{}{\Gamma, F \Longrightarrow F}\ \textit{init} \qquad \frac{\Gamma, F_1 \Longrightarrow A \ \mathsf{says}\ F_2}{\Gamma, A \ \mathsf{sign}\ F_1 \Longrightarrow A \ \mathsf{says}\ F_2}\ \mathsf{sign}L$$

$$\frac{\Gamma \Longrightarrow F}{\Gamma \Longrightarrow A \ \mathsf{says}\ F}\ \mathsf{says}R \qquad \frac{\Gamma, F_1 \Longrightarrow A \ \mathsf{says}\ F_2}{\Gamma, A \ \mathsf{says}\ F_1 \Longrightarrow A \ \mathsf{says}\ F_2}\ \mathsf{says}L$$

$$\frac{\Gamma, F_1 \Longrightarrow F_2}{\Gamma \Longrightarrow F_1 \to F_2}\ {\to}R \qquad \frac{\Gamma \Longrightarrow F_1 \quad \Gamma, F_2 \Longrightarrow F_3}{\Gamma, F_1 \to F_2 \Longrightarrow F_3}\ {\to}L$$

Figure 1. Sequent calculus for a simple authorization logic

A logical context Γ contains a list of formulas.

The sequent calculus rules for this simple logic are shown in Figure 1. The sign$L$ and says$L$ rules allow reasoning about the consequences of a principal's beliefs. In both rules, the assumptions and the formulas that are derived from these assumptions involve only a single principal; this ensures that one principal's beliefs cannot influence another's except through the use of explicit delegation between principals. The says$R$ rule states that a principal affirms any tautology. Notice that there is no sign$R$ rule because A sign F is considered an atomic formula that cannot be derived. Delegation can be encoded as implication. For instance, the formula A says $(B \ \mathsf{says}\ P \to P)$ denotes that principal A has delegated the authority P to principal B.

Proofs play a crucial role in our work. We formally define them as follows.

$$\textit{Proofs} \quad p \ ::= \ \cdot \mid \langle \textit{rname}, \Gamma, F, \{p_1, \ldots, p_n\} \rangle$$

A proof has four components: the name of the last inference rule applied (*rname*), the context containing the assumptions from which the proof is constructed (Γ), the goal being proved (F), and the list of premises ($p_1, \ldots, p_n$) from which the inference rule derived the goal. For example, the following proof tree, labeled $\mathcal{E}$, is equivalent to the proof immediately below it.

$$\mathcal{E} = \quad \frac{\overline{P \Longrightarrow P}\ \textit{init}}{\cdot \Longrightarrow P \to P}\ {\to}R$$

$$\mathcal{E} = \langle {\to}R, \cdot, P \to P, \{\langle \textit{init}, P, P, \{\}\rangle\}\rangle$$

The proof ends with rule ${\to}R$, the context of the conclusion is empty $(\cdot)$, the final goal formula is $P \to P$, and the premise is another proof. This previous proof ends with an application of the *init* rule, both the context and the final goal are P, and there are no premises.

III. Constraints in Policy Specification

We can use an authorization logic such as the one shown in Section II to express flexible distributed access-control polices. A defining characteristic of distributed access-control systems is that multiple principals can contribute credentials that together define the access-control policy protecting some resource.

An inherent shortcoming of such systems is that it is difficult for a principal to foresee all the consequences that her credentials may have. This is for two reasons. First, the credentials created by different parties may not be all available in a single place in order to be analyzed. Hence, the global policy, which is the set of inferences that can be made from the sum of all the credentials in the system, is likely to be unknown. Second, as principals issue new credentials over time, these new credentials may be combined with previously created ones in ways that the authors of the previously created credentials were not able to foresee.

There is an inherent tension in distributed access control between the desires for flexible policy specification and for clear understanding of the global policy. To make accurate and sound credential-creation decisions, a credential issuer needs to know how her credentials could be used in the global system. We propose a method for allowing credential issuers to accomplish this by embedding in their credentials explicitly specified constraints. These constraints describe the permitted uses of the credentials in which they are embedded, and are enforced using proof rules.

In Section III-A we review the design space of methods for limiting undesired inferences in logic-based access-control systems and describe our approach. Then, in Section III-B, we show that this approach supports two intuitively distinct categories of constraints. In Section III-C, we give concrete examples of a range of different, practically interesting constraints. We defer discussion of how to enforce the constraints we describe here to Sections IV and V.

A. An Approach to Limiting Undesired Inferences

In logic-based access-control systems, there are several general ways to allow a credential issuer to control how her credentials may be used. One possible approach is to implement in a manner completely external to the authorization logic a method for allowing one principal to prohibit other principals from issuing credentials that may lead to undesired inferences. For example, all credential creation could be mediated by a central authority that would prevent the creation of credentials that are inconsistent with extra-logically specified policies. A second option is to redesign the rules of the authorization logic to directly enforce the commonly desired constraints by prohibiting logical inferences that would violate these constraints, i.e., to enhance authorization logics by making them more restrictive in the inferences they allow. Finally, one can allow credentials to specify the kinds of authorization proofs within which the use of those credentials is valid, leaving the specification and enforcement of authorization policies and constraints modularly connected.

The first of these approaches is arguably the least useful, since it is antithetical to the goals of distributed access control: distributed access-control systems seek to decentralize authority, and so it would be unreasonable in such a system to give one principal control over what other principals may assert or believe.

The second option is more reasonable. However, since in this case constraints would be enforced directly via the inference rules of an authorization logic, any one logic is likely to support only a set of the most commonly used, predefined constraints. Further, incorporating constraint enforcement directly in the rules of the logic in this way makes it harder to reason about the properties of the logic, and hence harder to maintain guarantees of the logic's correctness. Two examples of systems that fall roughly into this category are SecPAL [11], [10] and DL [24]. We discuss them, and this approach, in more detail in Section VII.

The third option is the approach that we develop in this paper. We observe that a proof of access fully describes all the credentials used during an access and the details of how these were combined. A proof records every deduction step from the assumptions, which in the case of authorization logic correspond to credentials, to the final conclusion. Hence, a powerful way to define constraints is as functions over proofs. More specifically, a constraint $\mathcal{C}$ takes an authorization proof p as an argument and returns either true, when the constraint is satisfied, or false, otherwise.

$$\textit{Constraint } \mathcal{C} \;:\; \textit{proof} \rightarrow \textit{bool}$$

Since we can easily extract from a proof all the assumptions that are used in it, this allows us to define constraints based on properties of these assumptions. An example of such a constraint is one that limits the number of distinct credentials used in a proof. Another example is a limit on the number of distinct principals that may contribute credentials to a particular proof.

The formula that represents the proof goal (i.e., the statement that must be proved in order for access to be allowed) can also be used in the specification of constraints. For example, we can define a constraint that returns true only when the goal of the proof belongs to a predefined set. This allows a credential issuer to make the credential that contains this constraint valid for for proving only a specific set of goals.

Finally, more complex constraints can be defined inductively over the structure of the proof. For instance, we can define a constraint on the depth of a proof.

At this point, we do not impose any restrictions on constraints: they can be arbitrary functions over authorization-logic proofs. However, this is too general to be practical. For instance, if the constraint is a diverging function, then the access-control system may suffer from nontermination. We discuss this further in Section V.

B. Constraints on Credential Usage

Having decided on the general approach for specifying constraints as functions over authorization proofs, we now describe two different ways to use constraints that can be enforced in this manner: constraints on the proofs that a credential can participate in and constraints on (re-)delegation.

1) Final-usage constraints: Consider the following scenario. If Admin signed a credential stating that Alice is a student, and some other principal in the system signed another credential stating that all students are allowed to access the wireless network, then Alice will be granted access to the wireless network. By examining only his own credentials (which is all he may be able to do given the distributed nature of the system), Admin may have no way of knowing all the resources that Alice can access by combining Admin's credential with credentials from other principals. When something goes wrong—that is, when a principal gains unintended access to some resource using a proof that includes Admin's credential—Admin might be held responsible for signing such a credential and so having unwittingly participated in making the access possible.

Therefore, the first kind of constraint that we propose is one that allows a credential issuer to restrict which proofs can be constructed by using a particular credential as one of their assumptions. We call such a constraint a *final-usage constraint*. We write A sign $F \blacktriangleleft \mathcal{C}$ to denote a credential signed by principal A to assert that A believes F is true under the condition that this credential is used as an assumption only in proofs that satisfy constraint $\mathcal{C}$. The intuition behind this kind of constraint is that the credential issuer can directly prevent her credential from being used to derive facts that the issuer has not thought of as being reasonable or possible. For example, Admin can specify as a policy that Alice is a student *only* for the purpose of Alice being granted access to the wireless network. We define this policy below.

$$\mathsf{Admin\ sign}\ \mathit{Student}(\mathsf{Alice}) \blacktriangleleft \mathcal{C}_1 \quad \text{where}$$

$$\mathcal{C}_1(p) = \mathit{GoalOf}(p) \in \{\mathsf{Admin\ says}\ \mathit{MayAccessWifi}(\mathsf{Alice})\}$$

Constraint $\mathcal{C}_1$ takes a proof p as an argument and returns true, i.e., allows the credential to be valid, only when this credential is used in a proof whose final goal is Admin says *MayAccessWifi*(Alice).

Since they can be arbitrary functions, final-usage constraints need not individually enumerate the proofs within which credentials are to be considered valid, as was done in the above example: they can describe proofs at the level of abstraction that the credential issuer finds most appropriate.

2) Delegation constraints: The second kind of constraint regulates (re-)delegation of authority. Recall that formulas of the form A says (B says $P \to Q$) specify the delegation of authority from A to B. Specifically, in this example A is delegating to B any privileges that A has over Q. To exercise this privilege, B exhibits a proof of B says P, perhaps by directly asserting P, making it possible to derive a proof of A says Q. Notice that P and Q might not be the same formula. For instance, the following formula states that the library will allow Alice to borrow books as long as the registrar affirms that Alice is a student.

$$\mathsf{Library\ sign}\ \big((\mathsf{Registrar\ says}\ \mathit{Student}\ \mathsf{Alice}) \\ \to \mathit{MayBorrow}\ \mathsf{Alice}\big)$$

The principal A may not want his delegation credential to be used with an arbitrary proof of B says P. In the library example, the library might decide that it will only accept a proof that Alice is a student when Registrar actually signed the credential, and would refuse proofs derived from credentials that are signed by principals other than Registrar (even if Registrar signed some of them).

We write A sign $((B \text{ says } P \lhd \mathcal{C}) \to P)$ to denote that principal A delegates the authority P to principal B under the condition that the proof of B says P satisfies constraint $\mathcal{C}$.

Constraint $\mathcal{C}$ could be, for example, a constraint on the number of distinct principals that contribute to the credentials used in the proof of B says P. In that case, using $\mathcal{C}$, A would be enforcing a strong bound on the delegation depth (i.e., on the number of times P could be redelegated starting from B) because each re-delegation would require a different principal to sign a credential.

C. Example Constraints

We illustrate the range of constraints that can be supported by our approach by exhibiting a set of example constraints that can be useful in access-control scenarios and describing how to implement them using our approach. We group these examples into three categories based on the information they need in order to be enforced: constraints based on the logical context, constraints based on the final goal of the proof, and constraints based on the entire proof structure.

Some of the constraints are straightforward; we will define in detail only the more complicated ones. To increase our assurance that we did not overlook any critical details, we have encoded all the examples using Twelf [27], an implementation of LF; we discuss details of these low-level encodings in Section V.

1) Context-based: Access-control decisions are normally based on some set of policies, which are represented as assumptions in the context of a logical judgment. In practice, these assumptions are typically implemented as digital certificates signed using principals' private keys. Part of the process of verifying the validity of a proof involves confirming that each logical assumption is backed by a corresponding, valid digital certificate. The context of an authorization proof hence provides us with rich information as to which credentials are involved in granting access and which principals are involved in issuing these policies. The first kind of constraint for which we show examples is based solely on the context of a proof.

Before showing examples, we define an auxiliary function, which we will use frequently, to extract the logical context from a proof.

$$\mathit{ctxOf}(\langle \mathit{rname}, \Gamma, F, \{...\}\rangle) = \Gamma$$

Number of unique credential issuers. This constraint specifies that the number of credential issuers involved in the proof should not exceed a particular limit. Given a proof, one can easily calculate how many principals have contributed credentials to this proof by examining the context. Such a constraint on proofs is particularly useful when a credential issuer wishes to constrain *delegation depth*. For instance, suppose a manager delegates certain authority to a subordinate and permits this subordinate to re-delegate the authority to another, but does not want the delegation chain to extend any further. We can conservatively limit the delegation depth by limiting the number of unique credential issuers contributing credentials to the proof. To do so, we first define a function *uniqCredOf* that calculates the number of unique credentials issuers in the context; its definition is straightforward. Then, we can define the constraint that the number of unique credential issuers is at most two as follows.

$$\mathcal{C}_2(p) = \mathit{uniqCredOf}(\mathit{ctxOf}(p)) \leq 2$$

The following credential then limits the depth of re-delegation starting from the subordinate to at most one.

$$\mathsf{manager\ sign\ (subordinate\ says\ } P \lhd \mathcal{C}_2 \rightarrow P)$$

To make use of this delegation, one would have to first prove subordinate says P. Constraint $\mathcal{C}_2$ ensures that this proof contains credentials from only two issuers, i.e., that the delegation depth cannot be more than two. Usefully, this constraint does not prevent the subordinate from organizing his policies via roles and groups, since the credentials expressing such organization would be issued by the subordinate himself and so would not contribute to the issuer-uniqueness count.

Attributes required of credential issuers. In many cases, a principal may wish to require that all credential issuers who contribute credentials to a proof possess certain attributes. For instance, an admin in the Computer Science department may want to specify that his credential can be used only in those proofs in which all credentials have been issued by principals affiliated with the Computer Science department. As another example, Alice may want to give Bob access to her pictures, but only under the condition that if this access is redelegated, it must have been redelegated to someone who is both Alice's and Bob's friend.

One way of ensuring this is for Alice to define a constraint that will make her delegation to Bob valid only when all credentials in a proof are issued by Alice, Bob, or one of their joint friends. This constraint can define a function f to recurse over the structure of the context Γ and make sure that each credential in Γ is signed by a principal who is both Alice's and Bob's friend. This function will use several auxiliary constructs. The predicate $\mathit{friend}(p)$ asserts that p is a friend of the principal making such an assertion. E.g., Alice says friend(Bob) indicates that Alice believes Bob to be her friend. A helper function *findProof* takes a formula as an argument and finds a proof for the formula, if a proof exists. There are several ways to implement such a function; we discuss this in more detail in Section V. We write && to denote the binary operator *and* on booleans.

$$\begin{aligned} f(\cdot) &= \mathsf{true} \\ f(A \ \mathsf{sign}\ F \blacktriangleleft C, \Gamma) &= \mathit{findProof}(\mathsf{Alice\ says}\ \mathit{friend}(A)) \\ &\quad \&\&\ \mathit{findProof}(\mathsf{Bob\ says}\ \mathit{friend}(A)) \\ &\quad \&\&\ f(\Gamma) \end{aligned}$$

Now we can define the constraint Alice desires.

$$\mathcal{C}(p) = f(\mathit{ctxOf}(p))$$

To delegate to Bob under the desired constraint, Alice issues a credential of the following form.

$$\mathsf{Alice\ sign\ (Bob\ says\ } \mathit{access} \lhd \mathcal{C} \rightarrow \mathit{access})$$

Flexible revocation. The enforcement of credential revocation in logic-based access-control systems is often implemented in one of the following ways. (1) The authority conveyed by a credential is made explicitly contingent on the agreement of a revocation authority, e.g., instead of A sign F we would have something like A sign (CA says $\mathit{notRevoked} \ \rightarrow F$). (2) The reference monitor verifies that each credential in a proof is still valid according to some revocation authority.

Neither of these approaches is very flexible, however, and resource owners might have different ideas of which revocation servers they prefer and how fresh they would like revocation lists to be. Using our constraints, we can allow any credential issuer to easily specify and enforce her revocation policies on *every* credential in a proof to which she has contributed. We are able to achieve flexible, credential-issuer-controlled revocation checking because we allow credential issuers to specify constraints based on the proof, which includes all the credentials which make the logical derivation of the proof possible.

The definition of this constraint is similar to the previous example. For each credential in the context, the constraint requires a proof from a revocation server asserting that the credential is not in its revocation list, with the choice of the revocation server and revocation interval at the discretion of the credential issuer specifying the constraint.

2) Goal-formula-based: To control the consequences of one's credential, one can specify that her credential can only be used in a proof with a specific final goal. We showed an example of such a constraint in Section III-B. Similar, but more expressive constraints might use indirection to determine whether the final goal of a proof is acceptable. For example, one could use *findProof*, defined earlier in this section, to test whether a particular final goal is in scope. This would allow the scope of permitted final goals to be widened *after* issuing credentials constrained in such a manner.

3) Proof-structure-based: Previous examples use only small pieces of the proof, namely, the logical context or the final goal formula. The full power of our constraints is best demonstrated in cases where the entire proof is examined.

Proof size. First, we show how to define a constraint that limits the size of a proof. This constraint could be useful either as a redelegation constraint or as a final-usage constraint. For example, Alice may be willing to delegate her authority only when any redelegation is "simple," i.e., the subproof by which the recipient of the delegation exercises the delegation is small. Such a constraint could be used in parallel with the previously described constraint for limiting the number of unique credential issuers to enforce an even tighter bound on redelegation by limiting redelegation not just between individuals but also between their roles or groups.

Another use for such a constraint, this time as a final-usage constraint, is in scenarios in which Alice distrusts, and hence does not want her credentials to be associated with, any suspiciously large proofs, perhaps because she cannot envision why those would be necessary.

To implement the proof-size constraint, we first define the size of a proof in terms of its derivation depth. The following function is recursively defined over the structure of a proof and returns the depth of the proof.

$$\begin{aligned} &\mathit{depth}(\cdot) &&= 0 \\ &\mathit{depth}(\langle rn, \Gamma, F, \{p_1, \cdots, p_k\}\rangle) &&= 1 + \mathit{max}(\mathit{depth}(p_1), \cdots, \\ & && \qquad \mathit{depth}(p_k)) \end{aligned}$$

The constraint that the depth of the proof is no greater than n is then defined straightforwardly as follows.

$$\mathcal{C}(p) = \mathit{depth}(p) \leq n$$

Usage. By examining the structure of the proof, one can find out if a specific credential is used as part of the derivation or was just spuriously added to the logical context. For example, suppose we were given a derivation of the following judgment.

$$A \ \mathsf{sign}\ F, B \ \mathsf{sign}\ P \Longrightarrow B \ \mathsf{says}\ P$$

Here, the formula B says P can be proved solely from B's credential (B sign P), making A's credential (A sign F) spurious, as it does not (and cannot) contribute to the proof.

A principal A may want to prevent her credential from appearing in such proofs in which it is not needed. For example, she may want to avoid the unnecessary hassle of explaining to an auditor why her credential showed up in a suspicious proof (e.g., a proof that was possible only because some credential issuer delegated more authority than he should have). Of course, a manual examination of the proof would show that A's credential didn't participate in the proof in a meaningful way and so A cannot have been responsible for the improper delegation. However, A would prefer for it to be impossible to construct a valid proof if that proof used her credentials spuriously.

The key idea for defining such a constraint is to iteratively examine the proof structure from bottom to top. A formula A says F is held to contribute to constructing the proof if its subformula F contributes. The base case is at the leaves of the proof tree where the *init* rule is used; any formula directly used in the *init* rule is a contributor to the proof. Due to space constraints, we omit a more detailed definition.

Intermediate step. In distributed access-control systems, there are often natural boundaries between institutions for decision making. Because of these boundaries, it is useful to be able to specify which inferences based on the credentials issued in one domain ought to be used across domain boundaries.

For instance, suppose that the Computer Science department (CS) is in charge of saying who are its employees, while the Human Resources department (HR) is in charge of allocating benefits packages. CS will issue credentials, perhaps describing Alice's qualifications when she was hired, that will allow a proof that CS considers Alice a full-time staff member. HR will use CS's proof to provide Alice with a benefits package. In an effort to prevent HR and other departments from misinterpreting their internal assessment of Alice, credential issuers in CS may wish to state that their credentials can be used to prove that Alice is an employee of CS, and that only as part of that proof in its entirety can be used in the context of other proofs. In this way, HR can make the assignment of the benefits package explicitly contingent only on the proof of the fact that CS believes Alice to be an employee, and not independently on any of the credentials that contributed to that proof.

The key part of defining this constraint is to traverse the proof from the goal backward and check if the specified intermediate goal is reached before the use of any credential with this constraint. For these purposes, a credential is "used" when the sign$L$ rule is applied to it. We write F_{cred} to denote the credential being constrained and F_{inter} to denote the intermediate goal in which F_{cred} must first be used in order for its use in a larger proof to be valid. We define a helper function $\mathit{unused} : \mathit{form} \rightarrow \mathit{proof} \rightarrow \mathit{bool}$ that takes as arguments a formula and a proof and returns true if the formula is not used in the proof. This function can be inductively defined over the structure of the proof;

we omit the details here. We define two other functions, $=_f$ and $\neq_f$, to test the equality of two formulas. We further assume that the constraint language supports if-statements and pattern matching. We can then specify the constraint with the following pseudocode.

```
$\mathcal{C}(p) =$
    if $unused(F_{cred}, p)$ then true
    elseif $(GoalOf(p) =_f F_{inter})$ then true
    else match $p$ with
        | $\langle \to L, G, F, \{p_1, p_2\}\rangle$ => $\mathcal{C}(p_1)$ && $\mathcal{C}(p_2)$
        | $\langle \mathsf{sign}L, (G, F_1), F, p'\rangle$ => $(F_1 \neq_f F_{cred})$ && $\mathcal{C}(p')$
        ...
        | _ => false
```

In more detail: We first check whether F_{cred} is used in the proof; if it is not, then the constraint is trivially satisfied. Otherwise, we check if the goal of the proof is F_{inter}, in which case the constraint is also satisfied. If neither of the above is true, we pattern match on the proof. In most cases, the constraint simply recursively checks for the same conditions on the subproofs. For instance, if the current proof ends with the $\to L$ rule, then we check if the condition holds for the proofs of each premise of the $\to L$ rule. The interesting case is when the proof ends with the $\mathsf{sign}L$ rule. In that case, we only recurse on the proof of the premise if the credential being decomposed by $\mathsf{sign}L$ is not F_{cred}. If F_{cred} is the credential being decomposed, the fall-through case will return false. This is because at the time when F_{cred} is used, the desired intermediate goal F_{inter} has not yet been reached.

Choice of participating in delegation. Our final example constraint allows a principal to specify the conditions under which she is willing to act as a delegatee. Suppose, for example, that Admin creates a credential that requires Alice, Bob, and Charlie to cooperate in order to exercise privilege P.

$$\begin{aligned}\mathsf{Admin\ sign}\ ((&\mathsf{Alice\ says}\ P)\\ &\to (\mathsf{Bob\ says}\ P)\\ &\to (\mathsf{Charlie\ says}\ P) \to P)\end{aligned}$$

Alice decides that she is willing to participate in using the delegated authority only as long as both Bob and Charlie make their own decisions without delegating the responsibility to others. In other words, Alice will participate only if she can be guaranteed that all the credentials used in the proof of Bob says P are issued by Bob, and, similarly, all credentials used in the proof of Charlie says P are issued by Charlie.

The high-level idea of the definition of the constraint that will give Alice this guarantee is as follows. First, we induct over the structure of the proof and find the subproof that ends with the $\mathsf{sign}L$ rule that decomposes Admin's delegation credential. Then, we follow the implication inside Admin's credential to identify the subproofs of Bob says P and Charlie says P. Finally, we check that the only credentials in the assumptions of each of the subproofs are issued by Bob or Charlie, as appropriate. We described how to implement these component functions in previous examples.

IV. Enforcement and Formal Properties

In this section, we describe how to enforce the constraints introduced in Section III, which we do by incorporating them in the simple authorization logic described in Section II. We take a layered approach in which we maintain the structure of the original proof rules of the authorization logic, and add constraint checking as a layer on top of the authorization logic rules.

A. Syntax

We first summarize the syntactic constructs we need to express the two uses of our constraints described in Section III. We use $\mathcal{C}$ to denote constraints. We write $\mathtt{tt}$ to denote a constant function that always returns true, which is the most permissive constraint. We also write $\mathcal{C}_1 \wedge \mathcal{C}_2$ to denote the composition of two constraints which returns true if both of them return true and returns false otherwise. Formal definitions for constraints are given in Section IV-C.

To allow credential issuers to add constraints to their credentials, we define constrained formulas, denoted by F_c, and credentials, denoted by F_k, as follows.

regular formulas	$F ::= P \mid F_1 \to F_2 \mid A\ \mathsf{says}\ F$
constrained formulas	$F_c ::= F \mid (F \lhd \mathcal{C} \to F_c)$
credentials	$F_k ::= A\ \mathsf{sign}\ F_c \blacktriangleleft \mathcal{C}$
contexts	$\Gamma ::= \Gamma, F \mid \Gamma, F_c \mid \Gamma, F_k$

A constrained formula (F_c) can be either a regular formula (F) or a special implication ($F \lhd \mathcal{C} \to F_c$). The assumption of this special implication is a formula guarded by a constraint ($F \lhd \mathcal{C}$). The conclusion of the implication is another constrained formula. Hence, a constrained formula can be of the form $F_1 \lhd \mathcal{C}_1 \to (F_2 \lhd \mathcal{C}_2 \to (\cdots \to F))$.

Credentials have the form $A\ \mathsf{sign}\ F_c \blacktriangleleft \mathcal{C}$, which allows a credential issuer A to specify that this credential is valid only in proofs that satisfy constraint $\mathcal{C}$. In addition, A can use F_c to specify delegation constraints. For instance, $A\ \mathsf{sign}\ (B\ \mathsf{says}\ P \lhd \mathcal{C} \to P) \blacktriangleleft \mathcal{C}'$ is a delegation credential issued by principal A, and states that A will delegate P to B under the condition that the proof of $B\ \mathsf{says}\ P$ satisfies constraint $\mathcal{C}$. When the constraint $\mathcal{C}$ is the constant $\mathtt{tt}$ constraint, we often omit them and write $A\ \mathsf{sign}\ (B\ \mathsf{says}\ P \to P)$.

We also extend the logical context Γ to include constrained formulas and credentials.

B. Sequent Calculus Rules

To enforce the constraints we specified, we augment the basic sequent calculus rules shown in Figure 1 with rules for enforcing constraints. At a high level, we modify or

$$\frac{}{\Gamma, F \Longrightarrow F\backslash \mathtt{tt}}\ \mathit{init} \qquad \frac{\Gamma, F_c \Longrightarrow A \ \mathsf{says}\ F\backslash \mathcal{C}'}{\Gamma, A\ \mathsf{sign}\ F_c \blacktriangleleft \mathcal{C} \Longrightarrow A\ \mathsf{says}\ F\backslash(\mathcal{C} \wedge \mathcal{C}')}\ \mathsf{sign}L$$

$$\frac{\Gamma \Longrightarrow F\backslash \mathcal{C}}{\Gamma \Longrightarrow A\ \mathsf{says}\ F\backslash \mathcal{C}}\ \mathsf{says}R \qquad \frac{\Gamma, F_1 \Longrightarrow A\ \mathsf{says}\ F_2\backslash \mathcal{C}}{\Gamma, A\ \mathsf{says}\ F_1 \Longrightarrow A\ \mathsf{says}\ F_2\backslash \mathcal{C}}\ \mathsf{says}L$$

$$\frac{\Gamma, F_1 \Longrightarrow F_2\backslash \mathcal{C}}{\Gamma, \Longrightarrow F_1 \rightarrow F_2\backslash \mathcal{C}} \rightarrow R \qquad \frac{\Gamma \Longrightarrow F_1\backslash \mathcal{C}_1 \quad \Gamma, F_2 \Longrightarrow F_3\backslash \mathcal{C}_2}{\Gamma, F_1 \rightarrow F_2 \Longrightarrow F_3\backslash \mathcal{C}_1 \wedge \mathcal{C}_2} \rightarrow L$$

$$\frac{\mathcal{E} :: \Gamma \Longrightarrow F\backslash \mathcal{C}_1 \quad \mathbf{V}(\mathcal{C}, \mathcal{E}) \quad \Gamma, F_c \Longrightarrow F'\backslash \mathcal{C}_2}{\Gamma, F \lhd \mathcal{C} \rightarrow F_c \Longrightarrow F'\backslash \mathcal{C}_1 \wedge \mathcal{C}_2} \rightarrow L' \qquad \frac{\mathcal{E} :: \Gamma \Longrightarrow F\backslash \mathcal{C} \quad \mathbf{V}(\mathcal{C}, \mathcal{E})}{\Gamma \overset{v}{\Longrightarrow} F}\ \mathit{verify}$$

Figure 2. Sequent calculus for a sample logic with constraints

augment the basic sequent calculus rules in the following ways. First, constrained formulas (F_c) are treated specially when being decomposed, because the verification of redelegation constraints occurs at these decomposition points. Second, the final step in proof construction now needs to enforce verification of final usage constraints. Finally, rules that do not interact with constraints in an interesting way still need to propagate the constraints from premises to their conclusions. We show the augmented sequent calculus rules in Figure 2, and now discuss them in more detail.

There are two main judgments. Judgment $\Gamma \overset{v}{\Longrightarrow} F$ states that, under the assumptions in Γ, we can prove that F is true and that all the constraints present in the assumptions in Γ are satisfied. Judgment $\Gamma \Longrightarrow F\backslash\mathcal{C}$ states that under the assumptions in Γ we can prove F is true, and the constraints present in Γ will be satisfied if the constraint $\mathcal{C}$ evaluates to true. Intuitively, this $\mathcal{C}$ is the conjunction of all constraints that the credentials in Γ require the final proof to satisfy, and it will hence be checked in the last derivation step.

We use the following notation. We write $\mathcal{E} :: J$ to mean that $\mathcal{E}$ is the proof of judgment J. We write $\mathbf{V}(\mathcal{C}, \mathcal{E})$ to denote the process that checks that proof $\mathcal{E}$ satisfies constraint $\mathcal{C}$; in other words, that $\mathcal{C}(\mathcal{E}) = \mathtt{true}$. The logical inference rule does not specify how the constraint is to be verified, just that it should be. We discuss different ways of implementing this verification, and detail our preferred version, in Section V.

The *init* rule states that we can prove F if F is assumed to be true and that the constraint to be satisfied is a trivial constraint that is always true. The next four rules ($\mathsf{says}R$, $\mathsf{says}L$, $\rightarrow L$, and $\rightarrow R$) are the same as the ones in the basic sequent calculus in Figure 1 except that the constraint from the premise is carried over to the conclusion. The constraint in the conclusion of the $\rightarrow L$ rule is the conjunct of the two constraints from the premises.

Rules $\mathsf{sign}L$, $\rightarrow L'$, and *verify* are the most interesting ones, because they operate on constraints in nontrivial ways. Rule $\mathsf{sign}L$ states that to use a constrained credential, one needs to check that the constraint $\mathcal{C}$ associated with that credential is true. Rule $\rightarrow L'$ decomposes a constrained implication. The first premise states that there exists a proof $\mathcal{E}$ for deriving F from Γ. The second premise checks that $\mathcal{E}$ satisfies constraint $\mathcal{C}$. If the constraint is satisfied, we can proceed to prove F' using the assumption F_c. The last rule (*verify*) transitions from an unverified proof to a verified proof. The only remaining thing we need to do is to check that the proof $\mathcal{E}$ satisfies constraint $\mathcal{C}$. The rules in Figure 2 explicitly specify where the constraints are checked, and we believe they match the high-level intuition about which part of the proof is subject to constraints. Note that the delegation constraints are checked eagerly, while the final-goal constraints are propagated top-down and checked in the very last step of the proof.

C. Formal Definition of Constraints

Proofs constructed using the sequent calculus rules in Figure 2 include constraints propagated at each application of those rules. The definitions of proofs in Section II need to be updated to include these constraints. The new definitions of proofs include the constraint in the conclusion.

$$\mathit{Proofs} \quad p \quad ::= \quad \cdot \ \mid\ \langle \mathit{rname}, \Gamma, F, \mathcal{C}, \{p_1, \ldots, p_n\}\rangle$$

With this definition of proofs, constraints associated with credentials can refer to constraints in the proofs as well. For instance, Alice can specify that she will only delegate to Bob, if Bob can come up with a proof with no constraints other than the constant constraint $\mathtt{tt}$. This kind of restriction is quite useful in preventing Bob from hiding in his constraints policies that would normally be expressed through credentials. We discuss this in more detail in Section VI.

Recall that constraints are functions on proofs. Now that our definition of proofs include constraints, we have created a cycle in our definitions. In order for definitions to be well-founded, we need to break this cycle. We use a standard trick and index atomic constraints by their name strings. We assume there is a constraint map Ψ that maps each name string to the definition of the constraint. The proofs mention constraints by their names, allowing the definitions

$$\frac{\Gamma, F_c \overset{a}{\Longrightarrow} A \text{ says } F}{\Gamma, A \text{ sign } F_c \blacktriangleleft \mathcal{C} \overset{a}{\Longrightarrow} A \text{ says } F}\ \mathsf{sign}L$$

$$\frac{\mathcal{E} :: \Gamma \overset{a}{\Longrightarrow} F \quad \mathbf{V}(\mathcal{C}, \mathcal{E}) \quad \Gamma, F_c \overset{a}{\Longrightarrow} F'}{\Gamma, F \lhd \mathcal{C} \to F_c \overset{a}{\Longrightarrow} F'}\ {\to}L'$$

$$\frac{\mathcal{E} :: \Gamma \overset{a}{\Longrightarrow} F \quad \mathcal{C}_{of}(\Gamma) = \mathcal{C} \quad \mathbf{V}(\mathcal{C}, \mathcal{E})}{\Gamma \overset{v}{\Longrightarrow} F}\ \textit{verify}$$

Figure 3. Selected alternative sequent calculus rules for a sample logic with constraints

of constraints to mention proofs directly without creating a cycle. The formal definitions are as follows.

$$\begin{array}{llll} \textit{Constraints} & \mathcal{C} & ::= & \textit{nameOf } s \mid \mathcal{C}_1 \wedge \mathcal{C}_2 \\ \textit{Constraint map} & \Psi & : & \textit{string} \to \textit{proof} \to \textit{bool} \end{array}$$

D. Alternative Formulation

The formulation discussed thus far propagates constraints via the rules of the logic and checks them at the end. Because our constraints are layered on top of ordinary authorization-logic constructs, it is fairly easy to separate the constraint checking from the logic rules themselves. To demonstrate this point, we show an alternative formulation of the system, which has a clearer distinction between the constraints and the logical rules.

We first define a function $\mathcal{C}_{of}(\Gamma)$ to extract all the final-usage constraints from the assumptions in Γ. It is inductively defined as follows.

$$\begin{array}{lcl} \mathcal{C}_{of}(\cdot) & = & \mathsf{tt} \\ \mathcal{C}_{of}(\Gamma, F) & = & \left\{ \begin{array}{ll} \mathcal{C}_{of}(\Gamma) \wedge \mathcal{C} & \text{if } F = A \text{ sign } F_c \blacktriangleleft \mathcal{C} \\ \mathcal{C}_{of}(\Gamma) & \text{otherwise} \end{array} \right. \end{array}$$

In this formulation, specifying the sequent calculus that enforces constraints requires modifying or adding only three rules of the original set shown in Figure 1. The modified or new rules are shown in Figure 3. We ignore the final-usage constraints in rule sign$L$. Instead, in rule *verify*, we collect all the constraints present in the assumptions in Γ, and verify that they are satisfied with respect to the final proof.

We can prove that the new system is sound with regard to the system shown in Figure 2. To state the soundness theorem, we first define $\mathcal{S}(\mathcal{C})$ to be the set of atomic constraints contained in $\mathcal{C}$.

$$\begin{array}{lcl} \mathcal{S}(\mathsf{tt}) & = & \emptyset \\ \mathcal{S}(\mathcal{C}) & = & \{\mathcal{C}\} \text{ when } \mathcal{C} \text{ is an atomic constraint} \\ \mathcal{S}(\mathcal{C}_1 \wedge \mathcal{C}_2) & = & \mathcal{S}(\mathcal{C}_1) \bigcup (\mathcal{C}_2) \end{array}$$

The following theorem then states that a verfied proof in the new system implies the existence of a verified proof in the old one.

Theorem 1: If $\Gamma \Longrightarrow F \backslash \mathcal{C}_1$, and $\mathcal{C}_{of}(\Gamma) = \mathcal{C}_2$ then $\Gamma \overset{a}{\Longrightarrow} F$ and $\mathcal{S}(\mathcal{C}_1) \subseteq \mathcal{S}(\mathcal{C}_2)$

Proof (sketch): By induction on the derivation of $\Gamma \Longrightarrow F \backslash \mathcal{C}_1$. □

This theorem tells us that the new system imposes more constraints than the old one, and therefore admits fewer proofs. The difference between these two systems lies in whether to enforce constraints specified by credentials that are not *used* in the proof. By used, we mean the credential is decomposed by the sign$L$ rule in the proof. In the new system, all constraints specified in credentials and checked. On the other hand, in the old system, only constraints specified in credentials that actually contributed to the proof are checked. Consequently, we do not have a general completeness result for the new system. However, if we further assume that each credential in Γ is used in the proof, then these two systems are equivalent.

Each of these formulations has its advantages and disadvantages. The second formulation is closer to the original authorization logic rules, which makes extending an existing authorization logic very easy. We chose to use the first formulation because it encodes the extraction of constraints in the rules, and therefore offers a declarative view of how constraints are enforced on a per-rule basis.

E. Revisiting the Need for Delegation Constraints

Notice that final-usage constraints are very expressive, since they take as input strictly more information than delegation constraints. Hence, we could dispense with enforcing delegation constraints in our inference rules and instead encode any delegation constraints as final-usage constraints.

Given a delegation constraint $\mathcal{C}$, we can define another constraint $\mathcal{C}'$ such that A sign $(F_1 \to F_2) \blacktriangleleft \mathcal{C}'$ is equivalent to A sign $(F_1 \lhd \mathcal{C} \to F_2)$. In other words,

$$\begin{array}{l} \Gamma, A \text{ sign } (F_1 \to F_2) \blacktriangleleft \mathcal{C}' \overset{v}{\Longrightarrow} F \quad \text{iff} \\ \Gamma, A \text{ sign } (F_1 \lhd \mathcal{C} \to F_2) \overset{v}{\Longrightarrow} F \end{array}$$

However, if rewritten in this manner, the definitions of constraints could become very complicated, because any delegation constraint rewritten as a final-usage constraint would first need to recurse into the proof to reach the proof component that the original constraint cared about. Having two separate constraint forms makes the intention of the constraint much clearer and the definitions of constraints more concise.

F. Formal Properties

One of the most important properties required of a logic is consistency, i.e., that from an empty context one cannot prove arbitrary formulas. One of the main benefits of our layered approach is that a logic that has been extended to support constraints inherits many of the good properties from the underlying authorization logic, including consistency.

We are able to prove the following lemma which states that each proof in our system can be mapped to one in the basic authorization logic shown in Figure 1. We define Γ^-

to be the context consisting of all formulas in Γ with the constraints of each formula erased.

Lemma 1: If $\Gamma \Longrightarrow F \backslash \mathcal{C}$ then $\Gamma^{-} \Longrightarrow F$.

Proof (sketch): By induction over the structure of the derivation of $\Gamma \Longrightarrow F \backslash \mathcal{C}$. □

Our logic is consistent because rules in Figure 2 admit strictly fewer proofs than rules in Figure 1, and we have proved the consistency of the simple authorization logic described in Figure 1. We first proved the standard cut-elimination theorem for this logic; the consistency proof is derived directly from the cut-elimination theorem.

V. A Proof-Carrying-Authorization Implementation

In previous sections we described how to define constraints and extend the inference rules of an existing authorization logic to enforce these constraints. We left unexplored, however, several important issues that have to be resolved before our constraint framework can be used in a practical, deployed system. We discuss these issues here, and describe how to implement our constraint framework in a proof-carrying authorization system. In the appendix, we substantiate this description with a detailed use case, described at a level of detail sufficient for direct implementation.

In a proof-carrying authorization system, a reference monitor permits access to a resource once it has been presented with a valid proof that confirms that the policy guarding the resource has been satisfied (e.g., a proof of Admin says *MayAccess*(*resource*)). To determine whether a proof submitted with a request is valid, the reference monitor checks (1) that the proof is well formed with respect to the agreed-upon definition of an authorization logic and (2) that each of the assumptions in the proof corresponds to a (cryptographically) valid digitally signed certificate (i.e., credential).

In our scheme for enforcing constraints via logical inference rules, we make use of a verification predicate $\mathbf{V}(\mathcal{C}, \mathcal{E})$, which holds whenever all the constraints $\mathcal{C}$ hold for the proof $\mathcal{E}$. To implement our constraint-enforcement scheme in a practical system we have to decide how a reference monitor should determine whether various $\mathbf{V}(\mathcal{C}, \mathcal{E})$ encountered throughout the proof of access hold. There are two main ways in which this could be accomplished: the reference monitor could evaluate the constraints $\mathcal{C}$ with respect to $\mathcal{E}$, or the reference monitor could be supplied with evidence that $\mathbf{V}(\mathcal{C}, \mathcal{E})$ holds.

In the first case, constraints could be represented as programs in an agreed-upon programming language. When the reference monitor encounters a $\mathbf{V}(\mathcal{C}, \mathcal{E})$ in a proof, it would execute the code comprising $\mathcal{C}$ and proceed with verification only if the program returned true. This approach to confirming whether $\mathbf{V}(\mathcal{C}, \mathcal{E})$ holds has the advantage of making it straightforward to represent arbitrarily complex constraints, but also significant disadvantages, most particularly that the reference monitor would have to either trust or ensure that the constraint programs are not malicious and terminate in a reasonable amount of time.

The second case, in which the reference monitor is furnished with easily verifiable evidence that $\mathbf{V}(\mathcal{C}, \mathcal{E})$ holds, is closer to the spirit of proof-carrying authorization. In this case, the constraints still need to be specified in an agreed-upon language, but now the authorization proof needs to include (as subproofs) proofs that each constraint $\mathcal{C}$ evaluates to true when applied to the appropriate proof $\mathcal{E}$. Since verifying constraints is now done by proof checking, from the reference monitor's standpoint, this approach is safer—there is no need to trust credential issuers to produce only benign constraints or to include in the TCB mechanisms to ensure safe execution of foreign code. In the design of our prototype implementation, we pursue this option.

One tradeoff made by using proofs of constraint compliance is that proofs of access may become large—they include not only the authorization proof and the definitions of constraints, but also proofs that the constraints have been met. In principle, such proofs could become arbitrarily large, since their size could be proportional to the execution time of arbitrary functions that express constraints. We have not found this to be a concern in practice, however: all the constraints that we discuss here have proofs of compliance that are linear in the size of their input, i.e., the authorization proof.

An important distinction between regular credentials and constraints is that regular credentials are expressed via agreed-upon predicates of an authorization logic (e.g., says), which restricts the form that credentials can take. Constraints, on the other hand, we would like to keep as unrestricted as possible so as to not limit their expressiveness. Hence, we would like to allow constraints to be specified in a general language that permits them to define functions or relations useful for describing the properties that they want to ensure hold.

In light of this, there is a significant difference between verifying the validity of authorization proofs and proofs of constraint compliance. Authorization proofs are verified with respect to an agreed-upon definition of an authorization logic, whereas constraint compliance proofs are verified with respect to the functions or relations that comprise the constraint. When a complete proof, which contains both an authorization proof and one or more constraint compliance proofs, is being verified by a reference monitor, the reference monitor will have to ensure that the definitions of the constraints do not interfere with the definition of the authorization logic, e.g., by including in the definition of the constraint new inference rules that modify the semantics of the authorization logic. This kind of separation between the authorization logic and the constraint definitions can be accomplished in several ways, including alpha-renaming each constraint definition and its associated proofs or processing the authorization proof and each constraint definition and proof in independent proof-checking environments.

We pursue the latter strategy. We encode our authorization logic and proofs, and constraints and constraint-compliance proofs in LF [19]. To verify the validity of a proof of access, we use a modified LF type checker which maintains independent environments for processing authorization proofs and proofs of constraint compliance. Our previous investigation of using such modular LF type checkers has found that they are straightforward to design and introduce only minimal overhead into the proof-checking process [9].

VI. Discussion and Future Work

In this section we discuss several issues raised by or relevant to our framework for specifying constraints.

Tracing hypothesis in the contexts: In many of our examples, the constraints trace the decomposition of hypotheses in the context. For instance, in the usage-constraint example, we would like to determine if Alice sign F has contributed to the final proof. If rule sign$L$ is used on this particular credential, and F is useful in proving the final goal of the subproof, then Alice sign F is useful. However, there could be multiple instances of F in the context, and ideally we would like to track the usage of the specific one that comes from Alice sign F. To do so, we need to attach unique labels to formulas in the context. We give F a label, and check whether the F with that label is used in the subproofs. In our prototype LF encoding, we omit the labeling of hypotheses. As a consequence, our constraints are not as precise as one would like. We plan to improve our encoding in future work.

Constraints vs more precise authorization policies: Our approach to specifying constraints is sufficiently expressive that parts of policies that would normally be encoded as formulas in the authorization logic could instead be encoded as constraints. For example, suppose Alice wishes to delegate to Bob the authority to access her computer, but only if Charlie consents to this. The condition requiring Charlie's consent could be expressed as a constraint $\mathcal{C}$,

$$\begin{aligned}&\mathsf{Alice}\ \mathsf{sign}\ (\mathsf{Bob}\ \mathsf{says}\ \mathit{MayAccess}(X) \lhd \mathcal{C}\\ &\qquad\qquad \to \mathit{MayAccess}(X))\end{aligned}$$

or expressed directly in the authorization logic

$$\begin{aligned}&\mathsf{Alice}\ \mathsf{sign}\ (\mathsf{Bob}\ \mathsf{says}\ \mathit{MayAccess}(X)\\ &\qquad\qquad \to \mathsf{Charlie}\ \mathsf{says}\ \mathit{Consent}(\mathsf{Bob}, \mathit{MayAccess}(X))\\ &\qquad\qquad \to \mathit{MayAccess}(X))\end{aligned}$$

One subtle point is that Bob could use constraints to circumvent Alice's constraints on delegation. For instance, if Alice does not allow Bob to re-delegate by requiring that Bob is the only contributor of credentials in the proof of Bob says $\mathit{MayAccess}(\mathit{door})$, Bob can hide his re-delegation to Charlie in the constraint $\mathcal{C}$. This is not a desirable property. Luckily, since our constraints are functions on proofs, which include the constraints accumulated in the proofs, Alice can refine her requirement not to allow the proof of Bob says $\mathit{MayAccess}(\mathit{door})$ to depend on any constraints other than the constant `tt` constraint.

When given the choice of encoding a policy restriction as a formula or as a constraint, it is preferable to encode it as a formula, since this helps to make the meaning of policies clearer. Many constraints, however, cannot be encoded in typical authorization logics, most particularly those constraints that depend on parts of the proof that are not directly related to the credential in question.

Constraints vs more expressive authorization logics: The expressive power of the constraints comes from their ability to examine the proof structure. This raises the question of whether we should design authorization logics with richer syntax and more complex inference rules so as to natively be able to restrict inference in the way that we do here with constraints. While in some cases this may be a feasible alternative to using constraints, it is not likely to suffice in the general case. For one, revising a logic in this way is likely to make it more difficult to prove formal properties about the logic. For another, such revision will result in a logic that implements only a fixed set of constraints, rather than leaving it up to credential issuers to create whichever constraints they choose.

Generalization: We have demonstrated how to augment a simple authorization logic with a mechanism for specifying and checking constraints. Our approach requires small and relatively straightforward changes to the inference rules of a logic, and we believe would be applicable to a variety of logics other than the one we showed here. Recently, various new authorization logics have been developed [16], [13], [15]. We plan to investigate general principles and methods for extending the syntax and proof rules of different logics with constraints in a manner similar to what we have shown in this paper. We would also like to prove meta-theorems that these more general methods do not interfere with or still allow properties such as consistency to be proved.

VII. Related Work

The study of logics for access-control gained prominence with the work on the Taos operating system [3]. Since then, significant effort has been put into formulating formal languages and logics (e.g., [3], [5], [12], [25], [2], [20], [21], [22], [14]) that can be used to describe a wide range of practical scenarios.

The proof-carrying approach that we adopt as an enforcement paradigm in this work was pioneered by Appel and Felten [4] and further explored in Alpaca [23] and other projects [7], [8], [17], [26].

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 [5], [18], [24], [25], and more powerful higher-order logics that served as a tool for defining simpler, application-specific ones [4], [6], [23]. Researchers have recently started to examine constructive authorization logics [16], about which they have proved meta-properties such as soundness and non-interference, as well as logics that reason about linearity and time [15], [13].

Some of the goals of our framework, in particular, the high-level notion of constraining the use of credentials and constraining authority based on context have been explored to a degree in projects such as DL [24] and SecPAL [11], [10]. Both of these systems, for example, support constraints designed to limit the depth of re-delegation of authority. Both DL and SecPAL can be categorized as approaches in which the rules of an authorization logic are made more restrictive so as to make it possible for credentials to more precisely specify what inferences they ought to enable. DL itself does not have a proof theory, and the semantics of DL is given in terms of a translation to a Prolog program. This makes it difficult to prove potentially important meta-properties about the language, and, indeed, in DL it is possible to circumvent delegation-depth constraints by delegating without using the explicit delegation predicate. SecPAL has a more developed proof theory, although meta-properties of the logic remain somewhat difficult to prove.

Our proposed framework is notably different from these related existing approaches in several ways. First, the constraints that we describe in this paper are more general than those that appear in related work in that we allow constraints to be arbitrary predicates on authorization proofs. Since proofs contain crucial information about why access should be granted, this enables our framework to support more scenarios and finer-grained constraints than existing approaches. Second, we build our framework as an extension of well-behaved constructive authorization logics. The enforcement of the constraints in our framework is based on sequent calculus, which is highly declarative and provides a good basis for proving useful properties about the resulting logics and systems. Related projects typically provide a language for specifying policies, but are not founded on formal foundations that lend themselves as well to proofs of meta-properties. Furthermore, unlike much work on modeling access-control systems, our approach lends itself naturally to use not just as a model, but as the actual enforcement mechanism, via proof-carrying authorization. Even with respect to related work that is used for enforcement and not just modeling, the proof-carrying approach has a number of benefits, most critically in terms of the assurances of correctness that it can provide.

VIII. Conclusion

In this paper we propose a general framework for allowing credential issuers to have fine-grained control over the circumstances under which their credentials can be used. Our framework is built on top of authorization logics with robust proof theory and hence exhibits desirable formal properties, such as consistency.

We demonstrate the flexibility of our framework by showing a wide range of example constraints relevant to access-control scenarios and describing how these constraints can be expressed using our framework.

We also describe how our framework can be used in practice, and show a detailed encoding that demonstrates the integration of our framework into a traditional proof-carrying authorization system.

Acknowledgement

This work was supported in part by Carnegie Mellon CyLab under Army Research Office grant DAAD-19-02-1-0389, by ONR grant N000141010343, and by NSF grant CCF-0917047.

References

[1] M. Abadi, “Access control in a core calculus of dependency,” *Computation, Meaning, and Logic: Articles dedicated to Gordon Plotkin ENTCS*, vol. 172, pp. 5–31, April 2007.

[2] M. Abadi, M. Burrows, B. Lampson, and G. Plotkin, “A calculus for access control in distributed systems,” *ACM Transactions on Programming Languages and Systems*, vol. 15, pp. 706–734, 1993.

[3] 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*, Dec. 1993, pp. 256–269.

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

[5] 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.

[6] L. Bauer, “Access control for the Web via proof-carrying authorization,” Ph.D. dissertation, Princeton University, Nov. 2003. [Online]. Available: http://www.ece.cmu.edu/~lbauer/papers/thesis.pdf

[7] 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*, ser. Lecture Notes in Computer Science, vol. 3650, Sep. 2005, pp. 431–445.

[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*, ser. Lecture Notes in Computer Science, vol. 4734, Sep. 2007, pp. 19–37.

[9] L. Bauer, L. Jia, M. K. Reiter, and D. Swasey, "xDomain: Cross-border proofs of access," in *Proceedings of the 14th ACM Symposium on Access Control Models and Technologies*, Jun. 2009, pp. 43–52.

[10] M. Y. Becker, "SecPAL formalization and extensions," Microsoft Research, Tech. Rep. MSR-TR-2009-127, 2009.

[11] M. Y. Becker, C. Fournet, and A. D. Gordon, "Design and semantics of a decentralized authorization language," *Journal of Computer Security*, 2007.

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

[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 and M. Abadi, "A modal deconstruction of access control logics," in *Foundations of Software Science and Computation Structures (FOSSACS)*, 2008, pp. 216–230.

[15] D. Garg, L. Bauer, K. Bowers, F. Pfenning, and M. Reiter, "A linear logic of affirmation and knowledge," in *Proceedings of the 11th European Symposium on Research in Computer Security (ESORICS '06)*, 2006, pp. 297–312.

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

[17] ——, "A proof-carrying file system," in *Proceedings of the 2010 IEEE Symposium on Security & Privacy*, 2010.

[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*, Jun. 2003, pp. 187–201.

[19] R. Harper, F. Honsell, and G. Plotkin, "A framework for defining logics," *Journal of the Association for Computing Machinery*, vol. 40, no. 1, pp. 143–184, 1993.

[20] J. Howell, "Naming and sharing resources across administrative boundaries," Ph.D. dissertation, 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*, 2000, pp. 140–158.

[22] B. Lampson, M. Abadi, M. Burrows, and E. Wobber, "Authentication in distributed systems: Theory and practice," *ACM Transactions on Computer Systems*, vol. 10, no. 4, pp. 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 System Security*, vol. 6, no. 1, pp. 128–171, 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] S. Maffeis, M. Abadi, C. Fournet, and A. D. Gordon, "Code-carrying authorization," in *Computer Security—ESORICS 2008: 13th European Symposium on Research in Computer Security*, ser. Lecture Notes in Computer Science, vol. 5283, Sep. 2008, pp. 563–579.

[27] 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)*, 1999, pp. 202–206.

Appendix

We now revisit an example from Section III-C, for which we will show an LF encoding that integrates an authorization-logic proof with a proof that constraints have been verified. The specific example constraint we use is the attribute constraint. We proceed as follows. We first provide a set of basic definitions (e.g., of concepts like "formula"), and define the trivial tt constraint. We then show an encoding of the authorization logic from Figure 2 and of the attribute constraint. Finally, we show the authorization proof and constraint-verification proof and demonstrate how they integrate. The LF code of this and other examples can be downloaded from http://www.ece.cmu.edu/~lbauer/constraints/code/.

Basic definitions: Figure 4 contains the type definitions for our framework. Lines 1 and 2 define a string type str and a wrapper (strconst) for creating str's from Twelf's built-in

```
 1 str  : type.
 2 strconst  : string → str.
 3
 4 form  : type.
 5 cform :  type.
 6 cred :  type.
 7 ctx :  type.
 8 constraint :  type.
 9 fromDef :  {def : str}constraint.
10 ...
11 prove :  ctx → form → constraint → type.
12 vprove :  ctx → form → type.
13 verify :  prove G F C → constraint → type.
14
15 %% Conjunction for constraints
16 andc :  constraint → constraint → constraint.
17 %% Construct for constructing a proof of a pair of constraints
18 verifyPair :  verify P (andc C1 C2)
19                    ← verify P C1
20                    ← verify P C2.
21
22 %% Create tt constraint
23 ttc :  constraint = fromDef "...".
24 %% trivial proof for tt constraint
25 vtt = tt/check2v tt/checktt.
```

Figure 4. Basic definitions

```
1  tt/Def :  str = strconst "a string of the next three lines".
2
3  tt/check :  prove G F C → type.
4  tt/check2v :  tt/check PF → verify PF (fromDef tt/Def).
5
6  tt/checktt :  tt/check PF.
```

Figure 5. Basic definitions for constraints

strings. Next, we declare types for formulas (form), constraint formulas (cform), credentials (cred), the logical context (ctx), and constraints (constraint). fromDef constructs a constraint object from the string that represents the definition of the constraint.

An authorization logic proof has type prove G F C (line 11), where G is the context, F is the goal formula, and C is the constraint to be verified. This type represents the judgment $\Gamma \implies F\backslash\mathcal{C}$. A verified proof has type vprove G F (corresponding to the judgment $\Gamma \overset{v}{\implies} F$). On line 13, verify is the verification predicate $\mathbf{V}(\mathcal{E},\mathcal{C})$. An LF term of type verify pf c is evidence that proof pf satisfies constraint c.

We define andc (line 16) to be the conjunction operator for constraints. An object of type verifyPair v1 v2 is evidence for the constraint andc c1 c2 if v1 attests to c1 and v2 attests to c2. Finally, we declare the trivially true constraint ttc. Its definition would normally be encoded in the string argument to fromDef, but for clarity we show it in Figure 5.

The first line of Figure 5 is a placeholder for a self-referential pointer that can be used within the constraint definition to refer to the constraint. Here we declare it manually, although in practice it would be automatically generated by the system processing the definition of the constraint while checking the proofs. Next, tt/check PF is a predicate stating that proof PF satisfies the constraint being defined. Line 4 defines the glue for converting a proof (of type tt/check) specified in the constraint's language into a globally intelligible proof (of type verify ...) that the constraint has been verified. Finally, line 6 defines the concrete construct for constructing evidence for the tt/check predicate. Because the constraint we are defining is the trivially true constraint, there are no requirements on the proof itself, and evidence that the constraint has been satisfied is constructed trivially by the tt/checktt construct.

The definition of the authorization logic (which corresponds to the sequent calculus in Figure 2) is shown in Figure 6. We first define context to be a list of formulas and constraint formulas (lines 2–5), and then define operations on the context, e.g., for looking up a formula (lines 7–8). Next, we define logical connectives (lines 10–14). Finally, we define the set of sequent rules. Most of the definitions are straightforward. The interesting one is the cimpL rule, where LF's dependent types help us to link the evidence that constraints are satisfied to a specific proof. Line 24 represents the premise for proving the left-hand side of the implication. There is a constraint associated with this

```
1  %%% Context operations
2  nil :  ctx.
3  consf :  form → ctx → ctx.
4  consc :  cform → ctx → ctx.
5  consk :  cred → ctx → ctx.
6  ...
7  lookup :  ctx → form → type.
8  lookup/hit :  lookup (consf F G) F.
9  ...
10 %% Logic connectives
11 says  : str → form → form.
12 imp  : form → form → form.
13 cimp :  form → constraint → cform → cform.
14 signed  : str → cform → constraint → cred.
15
16 %% Deduction rules
17 init :  prove G A ttc  ← lookup G A.
18
19 signedL :  prove (consk (signed K Fc C) G) (says K F)
20              (andc C C′)
21               ← prove (consc Fc G) (says K F) C′.
22 ...
23 cimpL :  prove (consc Fc G) F C2
24             → {ee :  prove G F1 C1} verify ee C
25             → prove (consc (cimp F1 C Fc) G) F (andc C1 C2).
26
27 final :  {ee : prove G F C} verify ee C → vprove G F.
```

Figure 6. Sequent calculus

proof, so we also need to verify that this proof satisfies the constraint. In the LF encoding, {ee : prove G F1 C1} verify ee C means that ee is the name for the argument representing a proof of prove G F1 C, and LF's dependent types allow ee to directly show up in the argument immediately after it, which is evidence for the verification of constraint C on ee.

Encoding an attribute constraint: In Figure 7 we show an encoding of the constraint, previously described in Section III-C, that requires each credential in a proof to have been created by Alice, Bob, or a principal who is friends with both. Notice that the first few lines are similar to the

```
1  attrib/check :  prove G F C → type.
2  attrib/check2v :  attrib/check E → verify E (fromDef attrib/Def).
3
4  attrib/alice  : str = strconst "Alice".
5  attrib/bob  : str = strconst "Bob".
6  attrib/friend :  str → form.
7
8  attrib/ceq :  constraint → constraint → type.
9  attrib/ceq/ttc :  attrib/ceq ttc ttc.
10 attrib/ceq/and
11       : attrib/ceq (andc C1 C2) ttc  ← attrib/ceq C1 ttc  ← attrib/ceq C2 ttc.
12
13 attrib/checkG :  ctx → type.
14 attrib/checkG/nil :  attrib/checkG nil.
15 attrib/checkG/alice :  attrib/checkG (consk (signed attrib/alice F C) G)
16                           ← attrib/checkG G.
17 attrib/checkG/bob :  attrib/checkG (consk (signed attrib/bob F C) G)
18                           ← attrib/checkG G.
19 attrib/checkG/consk :  attrib/checkG (consk (signed A F C) G)
20                           ← attrib/checkG G
21                           ← prove G′ (says attrib/alice (attrib/friend A)) C1
22                           ← prove G′′ (says attrib/bob (attrib/friend A)) C2
23                           ← attrib/ceq C1 ttc
24                           ← attrib/ceq C2 ttc.
25
26 attrib/checkP :  {P :  prove G F C}attrib/checkG G → attrib/check P.
```

Figure 7. Encoding of an attribute constraint

definitions in Figure 5. These are the boilerplate for defining constraints: a predicate for checking the constraint locally, and a wrapper that makes it possible to integrate a proof of constraint compliance (specified in the constraint language) with the authorization proof. The definitions for constructing proofs of constraint compliance are customized.

Lines 8–11 define a predicate stating when two constraints are equivalent. We will use these definitions to state that the conjunction of two trivially true constraints (ttc) is equivalent to ttc. Line 13 defines the predicate attrib/checkG G, which holds when all the credentials in G satisfy the constraint. Lines 14–24 define constructs for constructing evidence that attrib/checkG G holds. The base case is on line 14, which states that an empty context is always OK. There are three inductive cases. The first two deal with situations in which the first credential in the context is signed by Alice or Bob. Lines 19–24 describe the case where we check if the credential issuer A is a friend of both Alice and Bob (lines 21–22). Lines 23–24 check that the constraints in those proofs are nothing more than the trivially true constraint.

We can use these definitions to build a term attesting that the context of a proof satisfies the constraint, and this term can be further incorporated into the authorization proof to be sent to the reference monitor at the time of an access request. In the example proof in Figure 8, line 68 is the final proof of the fact that Alice authorizes access from credentials ka, kb and kc. ka (line 18) is Alice's credential delegating access to Bob with the attribute constraint; kb (line 21) is Bob's credential delegating access to Carl; and kc (line 24) is Carl's credential stating that access is allowed. Furthermore, Alice and Bob have both asserted that Carl is a friend (kaf on line 12 and kbf on line 15).

The only construct we can use to produce a verified proof is verifyR, which takes two arguments: the first one (p4) is a proof that Alice allows access with certain constraints; and the second one (v3) is a proof that p4 satisfies those constraints. Lines 55–57 give the type and definition of p4. By looking at the type of p4 we can see that the constraints associated with p4 are not that interesting: they are conjunctions of tt. Lines 59–61 are the type and definition of v3. We use the pairing construct defined in Figure 4 to construct a proof for the constraints associated with p4.

The interesting part of the proof is where delegation constraints are checked. When constructing p4, we first need to prove that Bob allows access, so that we can use Alice's delegation to Bob (ka). On line 57, p3 is a proof that Bob allows access. To use Alice's delegation credential, we need to check that p3 satisfies the attribute constraint. This check is witnessed by v2 (lines 43–52). v2 is constructed by using constructs defined as part of the attribute constraint shown in Figure 7.

```
 1 %% encoding of a proof that makes use of the attribute constraint
 2 alice = strconst ”Alice”.
 3 bob = strconst ”Bob”.
 4 carl = strconst ”Carl”.
 5 access :  form.
 6 access′ = form2cform access.
 7
 8 %% define the constraint from its content
 9 cattrib = fromDef (strconst ”the definition of the constraint”).
10
11 %% credential alice signed (friend carl)
12 kaf = signed alice (form2cform (attrib/friend carl)) ttc.
13
14 %% credential bob signed (friend carl)
15 kbf = signed bob (form2cform (attrib/friend carl)) ttc.
16
17 %% credential alice signed (bob says access <| C → access) <|| tt
18 ka = signed alice (cimp (says bob access) cattrib access′) ttc.
19
20 %% credential bob signed (carl says access <| tt → access) <|| tt
21 kb = signed bob (cimp (says carl access) ttc access′) ttc.
22
23 %% credential carl signed access <|| tt
24 kc = signed carl access′ ttc.
25
26 %% some small proofs for building up the real one
27 pinit :  prove (consc (form2cform F) G) F ttc = init lookup/hitc.
28
29 paf :  prove (consk kaf nil) (says alice (attrib/friend carl)) (andc ttc ttc)
30 = signedL( saysR pinit).
31 pbf :  prove (consk kbf nil) (says bob (attrib/friend carl)) (andc ttc ttc)
32 = signedL( saysR pinit).
33
34 p2 :  prove (consk kc nil) (says carl access) (andc ttc ttc)
35    = signedL (saysR pinit).
36
37 p3 :  prove (consk kb (consk kc nil)) (says bob access)
38        (andc ttc (andc (andc ttc ttc) ttc)) =
39 signedL (saysR (cimpL pinit p2 vtt)).
40
41 %% v2 attesting that the context (kb, kc) only contains credentials
42 %% issued by alice, bob, or alice and bob′s friend
43 v2 :  verify p3 cattrib
44 = attrib/check2v
45     (attrib/checkP p3
46         (attrib/checkG/bob
47             (attrib/checkG/consk
48                 (attrib/ceq/and attrib/ceq/ttc attrib/ceq/ttc)
49                 (attrib/ceq/and attrib/ceq/ttc attrib/ceq/ttc)
50                 pbf
51                 paf
52                 attrib/checkG/nil))).
53
54 %% notice that v2 is a witness for (verify p3 cattrib)
55 p4 :  prove (consk ka (consk kb (consk kc nil))) (says alice access)
56         (andc ttc (andc (andc ttc (andc (andc ttc ttc) ttc)) ttc))
57 = signedL (saysR (cimpL pinit p3 v2)).
58
59 v3 = verifyPair vtt
60         (verifyPair (verifyPair vtt
61                         (verifyPair (verifyPair vtt vtt) vtt)) vtt).
62
63 %% the entire proof
64 %% p4 is a proof of ka, kb, kc ==> alice says access \ C
65 %% where C is (some true constraints)
66 %% v3 is the witness of C
67 proof :  vprove (consk ka (consk kb (consk kc nil))) (says alice access)
68 = verifyR p4 v3.
```

Figure 8. Encoding of an example proof