Distributed Proving in Access-Control Systems*

Lujo Bauer† Scott Garriss‡ Michael K. Reiter†‡§

Abstract

We present a distributed algorithm for assembling a proof that a request satisfies an access-control policy expressed in a formal logic, in the tradition of Lampson et al. [16]. We show analytically that our distributed proof-generation algorithm succeeds in assembling a proof whenever a centralized prover utilizing remote certificate retrieval would do so. In addition, we show empirically that our algorithm outperforms centralized approaches in various measures of performance and usability, notably the number of remote requests and the number of user interruptions. We show that when combined with additional optimizations including caching and automatic tactic generation, *which we introduce here, our algorithm retains its advantage, while achieving practical performance. Finally, we briefly describe the utilization of these algorithms as the basis for an access-control framework being deployed for use at our institution.*

1. Introduction

In order to permit a requested operation, a reference monitor must verify evidence that the request should be granted. In classical approaches to access control, this evidence may be the presence of an authenticated identity on an access-control list, or the verification of a capability presented with the request. Several more recent proposals encode access-control policy and supporting credentials in a formal logic (e.g., [16]). Of particular interest here are those in which the evidence supporting a request is a proof in this logic that the request satisfies the access-control policy (e.g., [3]). That is, credentials (i.e., certificates) are encoded as formulas in the logic (e.g., "K_{Alice} **signed** (K_{Bob} **speaksfor** Bob)", using the notation of [3]; see Section 3 for a summary) and used as premises, from which the policy is proved using inference rules of the logic.

In this paper, we introduce a distributed strategy by which this proof can be generated and show that the strategy outperforms prior approaches in many contexts. All prior works of which we are aware employ what we call an *eager* strategy, in which the party assigned to submit the proof[1] (the reference monitor or requesting client) generates it singlehandedly, retrieving only certificates from others when necessary. Instead, here we advocate a *lazy* strategy, in which a party enlists the help of others to prove particular subgoals in the larger proof—versus merely retrieving certificates from them—yielding a proof that is assembled in a more distributed fashion.

There are compelling reasons to depart from the eager strategy employed in previous works. Fundamentally, eager strategies place a burden on the prover to request certificates without knowledge of what certificates are available or will be signed. As such, in systems where delegations occur dynamically and at user discretion, an eager strategy may request a certificate from a user that the user will be unwilling to sign because it conveys too much authority, or that conveys too little authority and so dooms the user to be interrupted again later. For example, an access-control policy requiring Alice **says** $\mathbf{action}(X)$ in order to perform X (e.g., open a door) can be satisfied by a request Bob **says** $\mathbf{action}(X)$ if Alice signs Bob **speaksfor** Alice. However, as this conveys far more authority to Bob than merely the authority to perform X—namely, the ability to perform *any* action on behalf of Alice—Alice may refuse to sign it. Similarly, asking Alice for a weak certificate, e.g., K_{Alice} **signed** (Bob **says** $\mathbf{action}(X) \supset$ Alice **says** $\mathbf{action}(X)$), precludes Alice from making more general statements that will save her from being interrupted later to approve another action Y for Bob. For example, Alice might instead add Bob to a group (e.g., K_{Alice} **signed** (Bob **speaksfor** Alice.Students)) to which she has already delegated the right to perform X (e.g., Alice **says** (Alice.Students **says** $\mathbf{action}(X) \supset$

†CyLab, Carnegie Mellon University
‡Electrical & Computer Engineering Department, Carnegie Mellon University
§Computer Science Department, Carnegie Mellon University

*This research was supported in part by National Science Foundation grant no. CNS-0433540, U.S. Navy grant no. N00014-04-1-0724, and U.S. Army Research Office contract no. DAAD19-02-1-0389.

[1] In contrast to our goals here, most systems do not submit a formal proof, but rather informal (but sound) evidence that a request should be granted. Except where appropriate in Section 2, in the rest of this paper we will nevertheless refer to this evidence as a "proof".

Alice $\mathbf{says}\ \mathbf{action}(X)$)) as well as other actions. From this, Alice's device can then assemble a *proof* of Alice $\mathbf{says}$ (Bob $\mathbf{says}\ \mathbf{action}(X) \supset$ Alice $\mathbf{says}\ \mathbf{action}(X)$), which is exactly what was needed. More importantly, Alice need not be contacted the next time Bob needs to prove access to a resource to which Alice.Students are authorized.

As such, we advocate a distributed ("lazy") proving strategy, whereby (continuing our example) Bob asks Alice to prove the subgoal (Alice $\mathbf{says}$ (Bob $\mathbf{says}\ \mathbf{action}(X) \supset$ Alice $\mathbf{says}\ \mathbf{action}(X)$)). In addition to permitting Alice more flexibility in choosing how to prove this (if she chooses to at all), we show empirically that this approach can have significant performance and usability benefits in a system that uses a tactical theorem prover to assemble this proof. In particular, we demonstrate using an access-control policy for physical access at our institution that the lazy approach we advocate achieves significantly better performance and usability in natural measures, including the number of messages sent and the number of interruptions to users. We also describe extensions to lazy proving that further improve these measures, even when compared to the same improvements applied to an eager strategy, and reduce overheads to practical levels. While some of these extensions, notably caching, have been explored elsewhere, we demonstrate that caching must be used in unintuitive ways to achieve its potential, and we further introduce a novel and more effective optimization called *automatic tactic generation*. These empirical improvements are achieved despite the fact—which we prove here—that our lazy strategy will always succeed in completing a proof when the eager approach would.

Our motivation for pursuing this work is a system that we are presently implementing at our institution to build a robust and secure authorization device from a standard converged mobile device ("smartphone"). In the context of this paper, each phone is equipped with a tactical theorem prover for generating proofs of authorization to access resources, which in our testbed include computer accounts and physical rooms. At the time of this writing, we are equipping a new building on campus to control access to over 25,000 square feet of space, including over 60 doors, as well computer accounts and other virtual resources for persons occupying this space. The algorithms described here are central to this testbed.

The remainder of this paper is structured as follows. We discuss related work in Section 2. We cover background in access-control logics and tactical theorem proving in Section 3. We detail our approach to distributed proof generation in Section 4. We evaluate our approach empirically and introduce optimizations including caching and automatic tactic generation in Section 5. We conclude in Section 6.

2. Related Work

Distributed authorization has received considerable attention from the research community. Much of the related research, however, revolves around formalizing and analyzing the expressive power of authorization systems (c.f., [1, 3, 12, 17]), and only a fraction of it addresses the practical details and strategies for distributing and collecting certificates.

Taos The Taos operating system made two main contributions to distributed access control [23]: its access-control mechanism was inspired by a formal logic [2, 16]; and its access-control mechanism was built in at the OS, rather than application, level. The former quality inspired a greater degree of trust in the well-foundedness, and therefore correctness, of the implementation. The latter allowed the notion of identity to be embedded at a lower level, making it easier, for example, to reason about the security of communication channels within the OS.

In Taos, authority is initially derived from login credentials, and then partially or fully delegated via secure channels to other processes. A credential manager builds, checks, and stores the credentials as they are passed around. An authentication agent determines whether a requesting process has the right to execute a particular action by querying the credential manager and referring to access-control lists (ACLs). A trusted certification authority (CA) maintains the mappings between cryptographic keys and the names used in ACLs. Reasoning about credentials is performed locally by the credential manager, and there are no provisions for identifying and locating missing credentials.

PolicyMaker and KeyNote PolicyMaker [7] is a trust-management framework which blurs the distinction between policies and credentials by expressing them both as (possibly signed) programs. Determining whether a policy is satisfied involves executing the policy and the supplied credentials. Execution is local to the entity that is trying to verify whether a request is valid.

In the general case, allowing credentials to include arbitrary programs causes the evaluation of these credentials to become potentially intractable. However, by imposing constraints on credentials (in particular, by requiring each to be executable in polynomial time, monotonic, and authentic) it is possible to specify a polynomial-time algorithm for determining whether a set of credentials satisfies a policy [8]. These and other constraints led to the creation of KeyNote [6], which refines the ideas of PolicyMaker into a more practical system.

Although credentials contain code to be executed and can be authored by different entities, the credentials are all collected by and executed in the local environment of the entity that is evaluating a policy. Hence, at evaluation time a credential cannot take advantage of any specialized knowledge present in the environment of the node on which the credential originated. No provision is built into PolicyMaker to automatically collect credentials as they are needed. In fact, generalizing credentials in the style of PolicyMaker may as a side effect make it more difficult to determine how to go about locating a missing credential.

SD3 and QCM SD3 [15] is a trust-management system that further develops the idea of automatically distributing and fetching certificates that was introduced in QCM [14]. SD3 is implemented as middleware, shielding users from the details of using cryptographic primitives and certificate distribution. Unlike most other distributed authorization systems, but similarly to our approach, it produces easily verifiable proofs of access—this makes it possible for a potentially complex credential-collection algorithm to reside outside of the system's TCB. An SD3 query evaluator automatically fetches remote certificates needed to fulfill access requests. In addition, it allows certificates to be requested using wildcards and caches remote certificates after they have been fetched. In this paper we investigate more powerful methods for fetching the needed certificates while allowing the authors of the certificates more control over which certificates are used.

Placeless Documents Balfanz et al. have developed a distributed access-control infrastructure for Java applications [4], one of the first implemented systems to be built around a sound formal core. Requests to access resources are accompanied by certificates that can be used to verify the validity of the request. The system does not specify, however, how certificates are collected or how a requester determines which certificates should be attached to a particular request; this is a focus of the present paper. Once a certificate is transmitted, it is cached by the recipient.

Proof-Carrying Authorization Appel and Felten [3] proposed a distributed authorization framework that uses a higher-order logic as a language for defining arbitrary application-specific access-control logics. The underlying higher-order logic allows the application-specific logics to be remarkably expressive. At the same time, proofs of access constructed in any such application-specific logic can easily be verified by a simple, general checker. Bauer et al. [5] used this framework to develop an access-control system for regulating access to web pages. Their system also included a mechanism for automatically fetching and caching certificates needed to construct proofs of access. Like SD3, this system implements only a simple certificate-retrieval strategy, upon which we improve here.

SPKI/SDSI SPKI 2.0 [13], a merger of the SPKI [12] and SDSI [21] efforts, is a digital-certificate scheme that inherits the binding of privileges to keys proposed in SPKI and the local names of SDSI. SPKI certificates are represented as tuples, and can bind names to keys, names to privileges, and privileges to keys. The authorization process for SPKI involves verifying the validity of certificates, translating the uses of names to a canonical form, and computing the intersection of the privileges described in authorization tuples.

SPKI has recently been implemented as an access-control mechanism for web pages [9, 19]. In the implemented system, the web server presents a web browser with the ACL protecting a requested page. It is the browser's responsibility to provide the server with a set of certificates which can be used to verify the browser's authority. Efficient algorithms for selecting such a set of certificates from a local cache have been proposed [10, 11] and extended to retrieve certificates from a distributed credential store [18]; however, in each case the algorithm for selecting this set is executed locally by the browser.

3. Background

To be able to precisely discuss the constructions of proofs of access, we first need to define a logic that will allow us to describe our access-control scenarios. The access-control logic we will use is straightforward and developed in the style of Lampson et al. [16]. However, we emphasize that our techniques are not specific to this logic.

3.1. Access-Control Logic

Our access-control logic is inhabited by terms and formulas. The terms denote principals and strings, which are the base types of our logic.

The **key** constructor elevates strings representing public keys to the status of principals. For example, if `pubkey` is a particular public key, then **key**(`pubkey`) is the principal that corresponds to that key.

Principals may want to refer to other principals or to create local name spaces—this gives rise to the notion of compound principals. We will write Alice.`secretary` to denote the principal whom Alice calls "secretary."

More formally, the terms of our logic can be described as follows:

$$\begin{array}{rcl} t & ::= & s \mid p \\ p & ::= & \mathbf{key}(s) \mid p.s \end{array}$$

where s ranges over strings and p principals.

The formulas of our logic describe principals' beliefs. If Alice believes that the formula F is true, we write

Alice **says** F. To indicate that she believes a formula F is true, a principal signs it with her private key—the resulting sequence of bits will be represented with the formula `pubkey` **signed** F.

To describe a resource that a client wants to access, we introduce the **action** constructor. The first parameter to this constructor is a string that describes the resource. To allow for unique resource requests, the second parameter of the **action** constructor is a nonce. A principal believes the formula $\mathbf{action}(resource, nonce)$ if she thinks that it is OK to access *resource* during the session identified by *nonce*. We will usually omit the nonce in informal discussion and simply say $\mathbf{action}(resource)$.

Delegation is described with the **speaksfor** and **delegate** predicates. The formula Alice **speaksfor** Bob indicates that Bob has delegated to Alice his authority to make access-control decisions about any resource. $\mathbf{delegate}(\mathsf{Bob}, \mathsf{Alice}, resource)$ transfers to Alice only the authority to access the particular resource called *resource*.

The formulas of our logic are described by the following syntax:

$$\begin{array}{rcl} \phi & ::= & s\ \mathbf{signed}\ \phi' \mid p\ \mathbf{says}\ \phi' \\ \phi' & ::= & \mathbf{action}\ (s,\ s) \mid p\ \mathbf{speaksfor}\ p\ \mid \\ & & \mathbf{delegate}(p, p, s) \end{array}$$

where s ranges over strings and p principals.

Note that the **says** and **signed** predicates are the only formulas that can occur at top level.

The inference rules for manipulating formulas are also straightforward (see Appendix A). For the purposes of illustration, we present the SPEAKSFOR-E rule, which allows principals to exercise delegated authority.

$$\frac{A\ \mathbf{says}\ (B\ \mathbf{speaksfor}\ A) \quad B\ \mathbf{says}\ F}{A\ \mathbf{says}\ F}\ (\text{SPEAKSFOR-E})$$

3.2. Tactical Theorem Provers

To gain access to a resource controlled by Bob, Alice must produce a proof of the formula Bob $\mathbf{says}\ \mathbf{action}(resource)$. To generate such proofs automatically, we use a theorem prover.

One common strategy used by automated theorem provers, and the one we adopt here, is to recursively decompose a goal (in this case, the formula Bob $\mathbf{says}\ \mathbf{action}(resource)$) into subgoals until each of the subgoals can be proved. Goals can be decomposed by applying inference rules. For example, the SPEAKSFOR-E rule allows us to prove Bob $\mathbf{says}\ \mathbf{action}(resource)$ if we can derive proofs of the subgoals Bob **says** (Alice **speaksfor** Bob) and Alice $\mathbf{says}\ \mathbf{action}(resource)$.

Attempting to prove a goal simply by applying inference rules to it often leads to inefficiency or even nontermination. Instead of blindly applying inference rules, *tactical theorem provers* use a set of tactics to guide their search. Roughly speaking, each tactic corresponds either to an inference rule or to a series of inference rules. Each tactic is a tuple (P, q), where P is a list of subgoals and q the goal that can be derived from them. Each successful application of a tactic yields a list of subgoals that remain to be proved and a substitution that instantiates the free variables of the original goal. Suppose, for example, that the SPEAKSFOR-E inference rule was a tactic which we applied to Bob $\mathbf{says}\ \mathbf{action}(resource)$. In this tactic the names of principals are free variables (i.e., A and B rather than Bob and Alice), so the produced substitution list would include the substitution of Bob for the free variable A (Bob/A). A certificate is represented as a tactic with no subgoals; we commonly refer to such a tactic as a fact. In practice, facts would only be added to the set of tactics after verifying the corresponding digital certificate.

4. Distributed Proof Generation

4.1. Proving Strategies

In traditional approaches to distributed authorization, credentials are distributed across multiple users. A single user (either the requester of a resource or its owner, depending on the model) is responsible for proving that access should be allowed, and in the course of proving the user may fetch credentials from other users. All users except for the one proving access are passive; their only responsibility is to make their credentials available for download.

We propose a different model: each user is both a repository of credentials and an active participant in the proof-generation process. In this model, a user who is generating a proof is now able to ask other users not only for their certificates, but also to prove for him subgoals that are part of his proof. Each user has a tactical theorem prover that he uses to prove both his own and other users' goals. In such a system there are multiple strategies for creating proofs.

Eager The traditional approach, described above, we recast in our environment as the *eager* strategy for generating proofs: a user eagerly keeps working on a proof until the only parts that are missing are credentials that she can download. More specifically to our logic, to prove that she is allowed access to a resource controlled by Bob, Alice must generate a proof of the formula Bob **says** **action**$(resource)$. The eager approach is for Alice to keep applying tactics until the only subgoals left are of the form A **signed** F and then query the user A for the certificate A **signed** F. In Alice's case, her prover might suggest that a simple way of generating the desired proof is by demonstrating Bob **signed** **action**$(resource)$, in which case Alice will ask Bob for the matching certificate. For non-trivial policies, Alice's prover might not know of a particular certificate that would satisfy the proof, but would instead try to find any certificate that matches a particular form. For example, if Bob is unwilling to provide Alice with the certificate she initially requested, Alice might ask him for any certificates that match Bob **signed** (A **speaksfor** Bob), indicating that Bob delegated his authority to someone else. If Bob provided a certificate Bob **signed** (Charlie **speaksfor** Bob), Alice's prover would attempt to determine how a certificate from Charlie would let her finish the proof.

Lazy An inherent characteristic of the eager strategy is that Alice's prover must *guess* which certificates other users might be willing to contribute. The guesses can be confirmed only by attempting to download each certificate. In any non-trivial security logic (that is, almost any logic that allows delegation), there might be many different combinations of certificates that Bob and others could contribute to Alice that would allow her to complete the proof. Asking for each of the certificates individually is very inefficient. Asking for them in aggregate is impractical—for example, not only might a principal such as a certification authority have an overwhelming number of certificates, but it's unlikely that a principal would always be willing to release *all* of his certificates to anyone who asks for them.

With this in mind, we propose the *lazy* strategy for generating proofs. Recall that credentials (A **signed** F) imply beliefs (A **says** F). The typical reason for Alice to ask Bob for a credential Bob **signed** F is so that she could use that credential to demonstrate that Bob has a belief that can lead to Alice being authorized to perform a particular action. Alice is merely guessing, however, that this particular credential exists, and that it will contribute to a successful proof.

The lazy strategy is, instead of asking for Bob **signed** F, to ask Bob to *prove* Bob **says** F. From Alice's standpoint this is a very efficient approach: unlike in the eager strategy, she won't have to keep guessing how (or even whether) Bob is willing to prove Bob **says** F; instead she will get the subproof (or a negative answer) with exactly one request. From Bob's standpoint the lazy approach also has clear advantages: Bob knows what certificates he has signed, so there is no need to guess; he simply assembles the relevant certificates into a proof. Additionally, Bob is able to select certificates in a manner that conveys to Alice exactly the amount of authority that he wishes. This is particularly beneficial in an interactive system, in which Bob the person (as opposed to Bob the network node) can be asked to generate certificates on the fly.

In the lazy strategy, then, as soon as Alice's theorem prover produces a subgoal of the form A **says** F, Alice asks the node A (in the above example, Bob) to prove the goal for her. In other words, Alice is lazy, and asks for assistance as soon as she finds a subgoal that might be more easily solved by someone else. In Section 5 we demonstrate empirically the advantages of the lazy strategy.

Our prover assumes a cooperative environment in which a malicious node may easily prevent a proof from being found or cause a false proof to be generated. Our system adopts the approach of prior work (e.g., [3, 15]), in which the reference monitor verifies the proof before allowing access, which means that these attacks will merely result in access being denied.

4.2. A General Tactical Theorem Prover

We introduce a proving algorithm that, with minor modifications, can produce proofs in either a centralized (all certificates available locally) or distributed manner (each node knows all of the certificates it has signed). The distributed approach can implement either the eager or the lazy strategy. We will use this algorithm to show that both distributed proving strategies will successfully produce a proof in all cases in which a centralized prover can produce a proof.

Our proving algorithm, which is derived from a standard backchaining algorithm (e.g., [22, p.288]), is shown in Figure 1. The proving algorithm, bc-ask, takes as input a list of goals, and returns either failure, if all the goals could not be satisfied, or a substitution for any free variables in the goals that allows all goals to be satisfied simultaneously. The algorithm finds a solution for the first goal and recursively determines if that solution can be used to produce a global solution. bc-ask proves a goal in one of two fashions: locally, by applying tactics from its knowledge base (Figure 1, lines 15–20); or remotely, by iteratively asking for help (lines 10–14).

The helper function subst takes as parameters a substitution and a formula, returning the formula after replacing its free variables as described by the substitution. compose takes as input two substitutions, θ_1 and θ_2, and returns a substitution θ' such that subst(θ',F) = subst(θ_2, subst(θ_1, F)). rpc$_l$ takes as input a function name and parameters and returns the result of invoking that function on the machine with address l. We assume that the network does not modify or delete data, and that all messages arrive in a finite amount of time. unify takes as input two formulas, F_1 and F_2, and determines if a substitution

```
0   global set $KB$                                                     /* knowledge base */

1   substitution bc-ask(                                                /* returns a substitution */
        list $goals$,                                                   /* list of conjuncts forming a query */
        substitution $\theta$,                                          /* current substitution, initially empty */
        set $failures$)                                                 /* set of substitutions that are known
                                                                           not to produce a complete solution */
2       local substitution $answer$                                     /* a substitution that solves all $goals$ */
3       local set $failures'$                                           /* local copy of $failures$ */
4       local formula $q'$                                              /* result of applying $\theta$ to first goal */

5       if $(goals = [\,] \ \wedge \ \theta \in failures)$ then return $\perp$    /* $\theta$ known not to produce global solution */
6       if $(goals = [\,])$ then return $\theta$                        /* base case, solution has been found */
7       $q' \leftarrow \mathsf{subst}(\theta, \mathsf{first}(goals))$

8       $l \leftarrow \mathsf{determine\text{-}location}(q')$          /* prove first goal locally or remotely? */
9       $failures' \leftarrow failures$

10      if $(l \neq localmachine)$
11          while $((\alpha \leftarrow \mathsf{rpc}_l(\mathsf{bc\text{-}ask}(\mathsf{first}(goals), \theta, failures'))) \neq \perp)$   /* make remote request */
12              $failures' \leftarrow \alpha \cup failures'$            /* prevent $\alpha$ from being returned again */
13              $answer \leftarrow \mathsf{bc\text{-}ask}(\mathsf{rest}(goals), \alpha, failures)$   /* prove remainder of goals */
14              if $(answer \neq \perp)$ then return $answer$          /* if answer found, return it */

15      else foreach $(P, q) \in KB$                                    /* investigate each tactic */
16              if $((\theta' \leftarrow \mathsf{unify}(q, q')) \neq \perp)$   /* determine if tactic matches first goal */
17                  while $((\beta \leftarrow \mathsf{bc\text{-}ask}(P, \mathsf{compose}(\theta', \theta), failures')) \neq \perp)$   /* prove subgoals */
18                      $failures' \leftarrow \beta \cup failures'$     /* prevent $\beta$ from being returned again */
19                      $answer \leftarrow \mathsf{bc\text{-}ask}(\mathsf{rest}(goals), \beta, failures)$   /* prove remainder of goals */
20                      if $(answer \neq \perp)$ then return $answer$  /* if answer found, return it */
21      return $\perp$                                                  /* if no proof found, return failure */
```

Figure 1. bc-ask, our proving algorithm

θ exists such that $\mathsf{subst}(\theta, F_1) = \mathsf{subst}(\theta, F_2)$, i.e., it determines if F_1 and F_2 can be made equivalent through free-variable substitution. If such a substitution exists, unify returns it. A knowledge base, KB, consists of a list of tactics as described in Section 3.2. determine-location decides whether a formula F should be proved locally or remotely and, if remotely, by whom. Figure 2 shows an implementation of determine-location for the lazy strategy; an implementation for the eager strategy can be obtained by removing line 1 and removing the if-then clause from line 2. When bc-ask is operating as a centralized prover, determine-location always returns $localmachine$.

When proving a formula F locally, bc-ask will iterate through each tactic in the knowledge base. If a tactic matches the formula being proved (line 16), bc-ask will attempt to prove all the subgoals of that tactic (line 17). If the attempt is successful, bc-ask will use the resulting substitution to recursively prove the rest of the goals (line 19). If the rest of the goals cannot be proved with the substitution, bc-ask will attempt to find another solution for F and then repeat the process.

The algorithm terminates when invoked with an empty goal list. If the current solution has been marked as a failure, bc-ask returns failure ($\perp$) (line 5). Otherwise, bc-ask will return the current solution (line 6).

Note that this algorithm does not explicitly generate a proof. However, it is straightforward to design the goal and tactics so that upon successful completion a free variable in the goal has been unified with the proof [5].

We proceed to show that all of the strategies proposed thus far are equivalent in their ability to generate a proof.

Theorem 1 *For any goal G, a distributed prover using tactic set $\mathcal{T}$ will find a proof of G if and only if a centralized prover using $\mathcal{T}$ will find a proof of G.*

For the full proof, please see Appendix B. Informally: By close examination of the algorithm, we show by induction that bc-ask explores the same proof search

```
0 address determine-location($q$)                          /* returns machine that should prove $q$ */
1     $\theta \leftarrow$ unify($q$, "$A$ **says** $F$")                 /* unify with constant formula "$A$ **says** $F$" ... */
2     if ($\theta = \bot$) then $\theta \leftarrow$ unify($q$, "$A$ **signed** $F$")   /* ... or with "$A$ **signed** $F$" */
3     if ($\theta = \bot \vee$ is-local(subst($\theta$, "$A$"))) then return $localmachine$
4     else return name-to-addr(subst($\theta$, "$A$"))            /* instantiate $A$ to a principal, then return
                                                            * the corresponding address */
```

Figure 2. Algorithm for determining the target of a request

space whether operating as a centralized prover or as a distributed prover. In particular, the centralized and distributed prover behave identically except when the distributed prover asks other nodes for help. In this case, we show that the distributed prover iteratively asks other nodes for help (lines 10–14) in exactly the manner that a centralized prover would consult its own tactics (lines 15–20).

Corollary 1 *For any goal G, a lazy prover using tactic set $\mathcal{T}$ will find a proof of G if an eager prover using tactic set $\mathcal{T}$ will find a proof of G.*

Proof Sketch Lazy and eager are both strategies for distributed proving. By Theorem 1, if a lazy prover finds a proof of goal G, then the centralized prover will also find a proof of G, and if a centralized prover can find a proof of G then an eager prover will also. □

4.3. Distributed Proving with Multiple Tactic Sets

So far we have only considered systems in which the tactic sets used by all principals are identical. This is only realistic when all resources are in a single administrative domain. It is possible, and indeed likely, that different domains may use a different sets of tactics to improve performance under different policies. It is also likely that different domains will use different security logics, which would also necessitate different sets of tactics.

In this more heterogenous scenario, it is more difficult to show that a distributed prover will terminate. Since each prover is allowed to use an arbitrary set of tactics, asking a prover for help could easily lead to unproductive cycles of expanding and reducing a goal without ever generating a proof. Consider the following example: Alice has a tactic that will prove Alice **says** (Bob **says** F) if Alice has a proof of Bob **says** F. However, Bob has the opposite tactic: Bob will say F if Bob has a proof of Alice **says** (Bob **says** F). If Bob attempts to prove Bob **says** F by asking Alice for help, a cycle will develop in which Bob asks Alice to prove Alice **says** (Bob **says** F), prompting Alice to ask Bob to prove the original goal, Bob **says** F.

In order to force the system to always terminate, we must impose an additional constraint—a request-depth limiter that increments a counter before each remote request, and decrements it after the request terminates. The counter value is passed along with the request, so that the remote prover can use the value during subsequent requests. When the counter exceeds a preset value, the prover will return false, thus breaking any possible cycles. While it is possible that this modification will prevent the prover from discovering a proof, in practice the depth of a proof is related to the depth of the policy, which is bounded. Even in this environment, we would like to show that distributed proof generation is beneficial. As a step towards this, we introduce the following lemma:

Lemma 1 *A locally terminating distributed prover operating in an environment where provers use different tactic sets, in conjunction with a request-depth limiter, will terminate on any input.*

Proof Sketch We construct a prover bc-ask$'$ that will operate in a scenario with multiple tactic sets by removing the else statement from Line 15 of bc-ask, causing Lines 16–20 to be executed regardless of the outcome of Line 10. If the request depth is greater than the maximum, Line 11 will immediately return failure. If the request depth is less than the maximum, we use induction over the recursion depth of bc-ask$'$ to show that Lines 11 and 17 terminate, which means that bc-ask$'$ terminates. □

Although it is necessary that a distributed prover terminate when operating under multiple tactic sets, our goal is to show that such a prover can prove a larger set of goals than any node operating on its own. This is accomplished by forcing the distributed prover to attempt to locally prove any goals for which a remote request failed.

Theorem 2 *A locally terminating distributed prover operating in an environment where provers use different tactic sets, in conjunction with a request-depth limiter, will prove at least as many goals as it could prove without making any requests.*

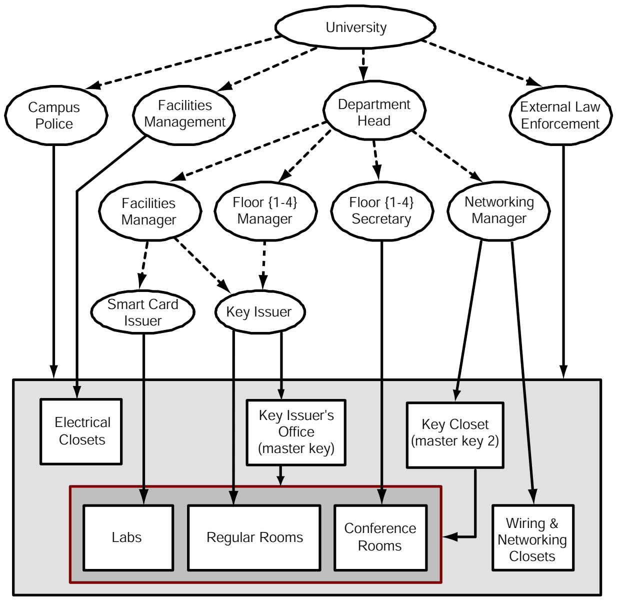


Figure 3. The authorization scheme for physical space in Hamerschlag Hall, home of the Carnegie Mellon Electrical & Computer Engineering Department

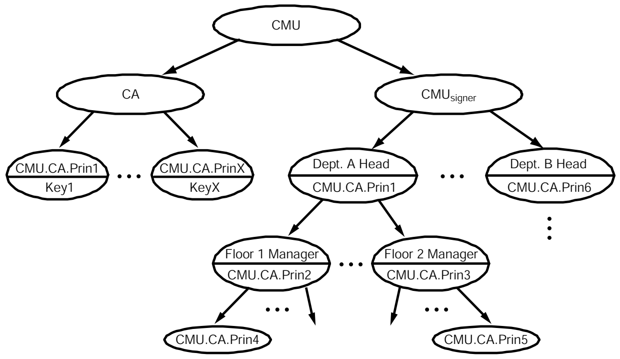


Figure 4. An expanded version of the authorization scheme for Hamerschlag Hall, modified for use in an digital access-control system

Proof Sketch We define a localized prover *LP* to be a prover that does not interact with other principals, and *DP* to be a distributed prover as described above. We want to show that if *LP* can find a proof of a goal G, then *DP* can find a proof as well. Both *LP* and *DP* use bc-ask′ which we construct from bc-ask by removing the else statement from Line 15, causing Lines 16–20 to be executed regardless of the outcome of Line 10. Indirectly from Lemma 1, the call on line 11 will always terminate, which means that lines 10–14 will terminate. If lines 10–14 produce a solution, we are done. If lines 10–14 do not produce a solution, *DP* will try to find a solution in the same manner as *LP*. We use induction to show that the results of further recursive calls will be identical between the scenarios, which means that *DP* will produce a solution if *LP* does. □

5. Empirical Evaluation

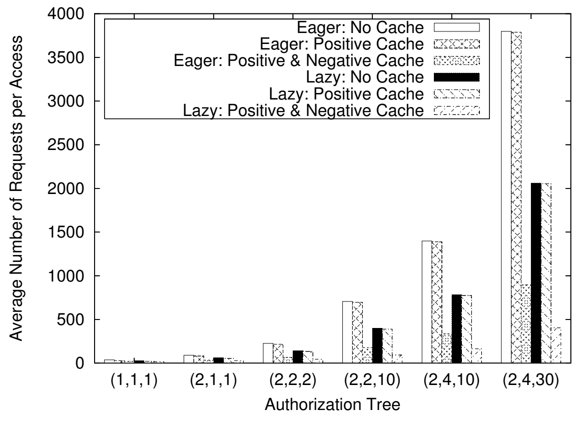
To fully understand the performance of lazy proving, we have undertaken a sizeable empirical study; we present the results here.

We implemented our proving algorithm in Prolog, taking advantage of Prolog's built in backchaining. We augmented the prover to maintain the current network location, and extended the definition of certificates such that the prover may only use certificates known to its current location. A request is recorded whenever the location of the prover changes. We note that our techniques are specific neither to prolog nor our choice of tactics and could be implemented in other automated theorem proving environments (e.g., [20]).

5.1. Constructing a Policy

One of the difficulties in evaluating distributed authorization systems is the lack of well-defined policies with which they can be tested. In the absence of such policies, it is often hard to conjecture how the performance of a system on simple example policies would relate to the performance of the same system if used in practice.

To remedy this problem, we first undertook to map the physical access-control policy for rooms in our department's building (Figure 3). Such policies are often not explicitly recorded, however the policy reflects the hierarchical structure of authorization in our department, which leads us to believe that it is representative of most organizations. A close examination of this policy reveals that it contains elements that would be superfluous in a digital access-control system. For example, delegation of authority is conveyed either through physical tokens (the key issuer gives a user a key) or through the organizational hierarchy (the head of the department delegates to the floor manager the responsibility of managing access to all the rooms on a floor, but doesn't provide him with a physical token). In a digital access-control policy, delegation of authority is always explicitly represented; furthermore, in the digital domain it is unnecessary to have a policy include elements, such as the Key Issuer and Smart Card Issuer, whose sole purpose is the distribution of physical tokens. At the same time, a practical digital policy requires the mapping of keys to names. Universities typically have a registrar's office that performs similar bookkeeping; we add to the registrar the duties of a local certification authority. Another characteristic of physical access-control



Eager		No Cache		Positive & Negative Cache	
Tree	Principals	Requests	STDEV	Requests	STDEV
(1,1,1)	6	37	0	20	0
(2,1,1)	9	90	53	34.5	14.5
(2,2,2)	17	226	132.9	65.5	29.8
(2,2,10)	49	706	409.5	177.5	94.3
(2,4,10)	93	1398	810.5	334.5	184.5
(2,4,30)	253	3798	2196.1	894.5	507.8

Lazy		No Cache		Positive & Negative Cache	
Tree	Principals	Requests	STDEV	Requests	STDEV
(1,1,1)	6	28	0	16	0
(2,1,1)	9	61	33	27.5	11.5
(2,2,2)	17	141	80.1	44.5	20.4
(2,2,10)	49	397	227.4	92.5	48.0
(2,4,10)	93	781	450.1	164	88.2
(2,4,30)	253	2061	1189.1	404	226.7

Figure 5. Performance of initial access with different caching strategies

policies used in practice is the difficulty in maintaining the separation between users and the roles they inhabit (for example, the role of department head and the person who has that position). In a digital system, where delegation of authority is always explicit, this separation is easier to manage. Due to the importance of the university's key, we split it into a master key and a signing key. Figure 4 roughly illustrates our derived policy.

Ideally, we would like to simulate the deployment of our system on a university-wide scale. However, helped by the hierarchical organization of the university's access-control policy (and access-control policies in general), the search for proofs is limited to a small subset of the overall population; consequently, we restrict our simulation to several such subsets without significantly impacting the accuracy of our results.

We chose to structure the authorization tree from the university to individual users as a complete tree. We describe a policy with a (j, k, l) tree to indicate that there are j department heads, k floor managers under each department head, and l users under each floor manager. We test our algorithms with several different (j, k, l) trees. We chose to use complete trees for simplicity only; when simulating unbalanced trees constructed by randomly removing a fixed number of nodes from a complete tree, our results differ by less than 4% [2].

Each of the policies protecting a room requires that the university approve access to it (e.g., CMU **says action**(`room15`)). The proof that a user may access the room is based on a chain of certificates leading from CMU to the user himself. The proof also shows which inference rules (of the logic described in Section 3.1) need to be applied to the certificates and in what order to demonstrate that the certificates imply that access should be granted (e.g., CMU **says action**(`room15`)). Appendix C shows how a particular set of certificates is formalized in our logic and provides a proof of access representative of those generated by our prover; it also explains how we populate our simulations with certificates.

[2] We constructed 20 unbalanced trees with 253 principals each by randomly removing 216 nodes from a complete (3,5,30) tree. The performance of the initial access with both forms of caching enabled decreased by up to 4%, with an average decrease of 2%.

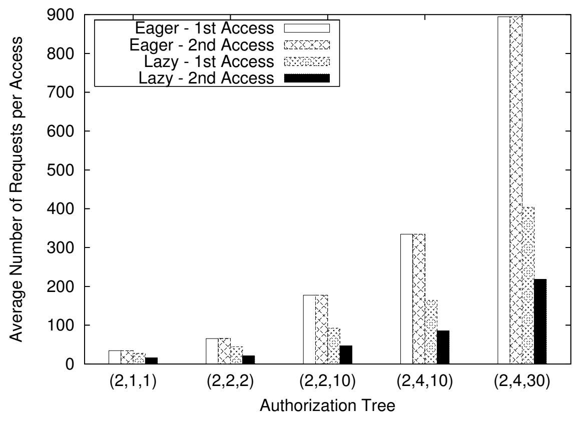
5.2. Evaluation Criteria

The primary criteria we use to evaluate the performance of the two proving strategies detailed in Section 4 is the number of requests made while attempting to construct a proof. Since requests in our system may ultimately cause an actual user to be queried to approve the creation of a certificate, the number of requests roughly approximates the required level of user interaction. Additionally, since much of the communication may be between poorly connected devices (such as cell phones connected via GPRS), the number of requests involved in generating a proof will be one of the dominant factors in determining the time necessary to generate a proof.

When running the simulations, the only principals who access resources are those located in the lowest level in the hierarchy. The resources they try to access are rooms on their floor to which they are allowed access. Unless otherwise specified, the performance results reflect the average over all allowed combinations of users and resources.

5.3. First Access

Figure 5 shows the average number of requests made by each proving strategy when first attempting to prove access to a resource. On average, lazy outperforms eager by between 25% and 45%, with the performance difference growing wider on larger authorization trees. However, the number of requests made is far too large for ei-



Eager		First Access		Second Access	
Tree	Principals	Requests	STDEV	Requests	STDEV
(2,1,1)	9	34.5	14.5	34.5	14.5
(2,2,2)	17	65.5	29.8	65.5	32.0
(2,2,10)	43	177.5	94.3	177.5	96.5
(2,4,10)	93	334.5	184.5	334.5	186.6
(2,4,30)	253	894.5	507.8	894.5	509.9

Lazy		First Access		Second Access	
Tree	Principals	Requests	STDEV	Requests	STDEV
(2,1,1)	9	27.5	11.5	15.5	10.5
(2,2,2)	17	44.5	20.4	21.1	16.3
(2,2,10)	43	92.5	48.0	47.1	37.7
(2,4,10)	93	164	88.2	85.1	69.4
(2,4,30)	253	404	226.7	218.3	179.3

Figure 6. Performance of subsequent access to a different resource by a different principal

ther strategy to be used in a practical setting. Upon further investigation, we discovered that more than half of all requests are redundant (that is, they are repetitions of previous requests), indicating that caching would offer a significant performance benefit.

Our initial intuition was to cache proofs of all successful subgoals found by the prover. However, as Figure 5 indicates, caching the results of successful proof requests offers surprisingly little performance benefit. We discovered that most of the redundant requests will, correctly, result in failure; that is, most of the redundant requests explore avenues that cannot and should not lead to a successful access. We modified the caching mechanism to cache failed results as well as positive results (also shown in Figure 5). This reduced the number of queries by up to 75% for both strategies.

5.4. Effects of Caching on a Second Access

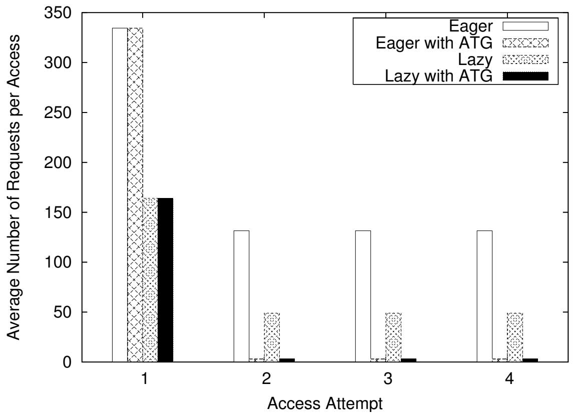
Since all of the results discovered by the eager strategy are cached only by the principal who accessed the resource, the cache is of no benefit when another principal attempts to access a resource. The lazy scheme distributes work among multiple nodes, each of which can cache the subproofs it computes. In the lazy scheme, access of the same or a similar resource by a second, different principal will likely involve nodes that have cached the results of previous accesses. This enables the lazy strategy to take advantage of caching in a way that the eager strategy cannot, resulting in significant performance gains. To compute the average performance, we ran the simulation for every possible combination of principals making the first and second access. Figure 6 shows that the average case eager performance in the second access is identical to its performance in the first attempted, as expected. The figure also shows that caching on interior nodes in the lazy strategy decreases the number of requests made by the second access by approximately a factor of 2. The result is that lazy completes the second access with approximately one-fourth the number of requests of eager.

5.5. Automatic Tactic Generation

Caching subgoals and certificates is clearly helpful when subsequent requests are identical to those that have already been proved. Often the second and subsequent accesses will have different proof goals, in which case caching will be of limited use even if there is great similarity between the two proofs. To take advantage of the similar *shape* of different proofs, we introduce *automatic tactic generation* (ATG).

Automatic tactic generation aims to remember the shape of previously computed proofs while abstracting away from the particular certificates from which the proofs are built. In order to leverage the knowledge of the proof shape gained during the first access, the prover must cache a proof that is not fully instantiated. The proof is stripped of references to particular resources and nonces; these are replaced by unbound variables. The certificates that were part of the proof, similarly abstracted, become the subgoals of a new tactic. The stripped proof is the algorithm for assembling the now abstracted certificates into a similarly abstracted goal. This allows any future access attempt to to directly search for certificates pertaining to that resource without generating intermediate subgoals.

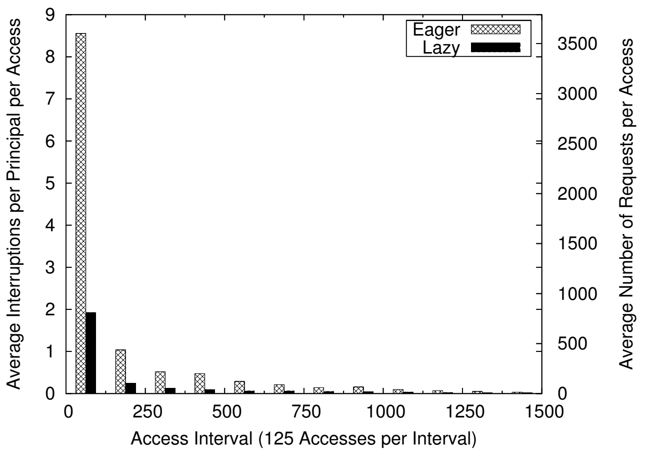
A common scenario in which automatic tactic generation is very useful is when attempting to access several rooms on the same floor. The policies protecting each of the rooms are likely to be very similar, since they belong to the same organizational unit and share the same administrator. Pure caching is not likely to help much because the rooms are all named differently, but automatic tactic generation allows proofs to be computed very efficiently, as shown in Figure 7. ATG is an optimization allows both the eager and the lazy strategy to complete subsequent proofs with a minimal number of requests.



Eager	No ATG		With ATG	
Access	Requests	STDEV	Requests	STDEV
1	334.5	186.6	334.5	186.6
2	131.5	74.7	3	0
3	131.5	74.7	3	0
4	131.5	74.7	3	0

Lazy	No ATG		With ATG	
Access	Requests	STDEV	Requests	STDEV
1	164	88.2	164	88.2
2	49	26.2	3	0
3	49	26.2	3	0
4	49	26.2	3	0

Figure 7. Sequential access of four resources by same principal in a (2,4,10) tree



	Requests per Principal	
Interval	Eager	Lazy
1-125	8.55	1.93
126-250	1.04	0.25
251-375	0.52	0.12
376-500	0.48	0.10
501-625	0.29	0.06
626-750	0.21	0.06
751-875	0.14	0.05
876-1000	0.16	0.04
1001-1125	0.10	0.04
1126-1250	0.07	0.03
1251-1375	0.05	0.02
1376-1500	0.03	0.02

Figure 8. Average of 10 simulations with 1500 random accesses in a (4,4,25) tree

5.6. Simulating a User's Experience in a Deployed System

The results thus far clearly demonstrate the benefits of the lazy strategy in simple, controlled scenarios. A more practical scenario, which we explore here, may involve many users accessing different resources in somewhat arbitrary order and frequency.

In this scenario, we have chosen to use a (4,4,25) tree. This means that there are four department heads, each with four floor managers. Each floor has 25 residents, for a total of 400 users who will be accessing resources. The system controls access to the main door to the building, security doors on each of the sixteen floors, and 400 offices: one for each user. Each of these principals has access to his office, the floor on which his office resides, and the building's main door. We show the performance for the first 1500 accesses that occur in this system. Each access is made by a randomly chosen principal to one of the three resources which he can access (again chosen at random). This scenario was too large to be simulated exhaustively, so instead we show the average of ten runs.

Figure 8 shows the performance of the system with all optimizations enabled, measured both as the average number of requests each principal has to answer per access attempt, and the total number of requests per access attempt. In this more realistic scenario, the lazy strategy continues to do well. During the first interval of 125 accesses, the lazy strategy is at least three times more efficient in the number of requests made. Note also that the number of requests quickly drops to a level that could be practical for a deployed system.

In practice, the number of times a user receives a request will be somewhat lower because a sizeable percentage of requests are made to the CA and the root node of the authorization tree. It is likely that the CA and the root node will either generate all certificates prior to bringing the system online, or will have an automated system for signing certificates, thus alleviating the burden on the user. Furthermore, we do not restrict whom a principal may ask for help, which would be necessary in practice.

6. Conclusion and Future Work

Previous work on distributed authorization systems largely did not focus on practical strategies for collecting the certificates used to show that a request satisfies an access-control policy. However, attention to these strategies is necessary for the deployment of rich certificate-based access control, particularly in cases where credentials are created dynamically with user involvement.

In this paper we introduced a new distributed approach to assembling access-control proofs. The strength of our approach is that it places the burden of proving a statement on the party who is most likely to have (or be willing to create) credentials relevant to proving it. In contrast, prior approaches asked the prover to guess credentials that might be available, thereby inducing greater numbers of attempted retrievals and user interruptions. In addition to these advantages, we showed empirically that this approach responds very well to caching and to a new optimization, *automatic tactic generation*. We achieve these advances with no loss in proving power: our distributed approach completes a proof whenever a centralized approach that uses certificate retrieval would do so.

Our algorithms are a cornerstone of a testbed we are developing that leverages smartphones to create and enforce an access-control policy for both physical rooms and virtual resources. Once complete, this testbed will regulate access for a population of roughly 150 people to over 60 doors, in addition to computer logins and other virtual resources. Each person's smartphone will hold cryptographic keys for creating credentials, as well as a tactical theorem prover for generating proofs of authority. If in the course of generating a proof of authority, the tactical theorem prover on a phone encounters a subgoal that, according to the distributed proving algorithm of Section 4, should be sent to another for proof, then the subgoal will be conveyed in real time over cellular data services (SMS/MMS over GPRS) to that party. The tactical theorem prover on that phone, in turn, will attempt to prove the subgoal with credentials it already has stored, other subgoals others prove for it (recursively), and various possible credentials it could create with its user's permission. For the last of these, the smartphone prompts the user to determine which of these credentials, if any, it should create. Upon receiving user instruction, the credential is created, and the subgoal proof is generated and returned to the requesting smartphone. We expect such interruptions to be infrequent; for most requests, caching and automatic tactic generation should yield proofs silently.

References

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

[2] 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.

[3] A. W. Appel and E. W. Felten. Proof-carrying authentication. In *Proceedings of the 6th ACM Conference on Computer and Communications Security*, Singapore, Nov. 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 and Privacy*, Oakland, CA, May 2000.

[5] L. Bauer, M. A. Schneider, and E. W. Felten. A general and flexible access-control system for the Web. In *Proceedings of the 11th USENIX Security Symposium*, San Francisco, CA, Aug. 2002.

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

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

[8] M. Blaze, J. Feigenbaum, and M. Strauss. Compliance checking in the PolicyMaker trust-management system. In *Proceedings of the 2nd Financial Crypto Conference*, volume 1465 of *Lecture Notes in Computer Science*, Berlin, 1998. Springer.

[9] D. E. Clarke. SPKI/SDSI HTTP server / certificate chain discovery in SPKI/SDSI. Master's thesis, Massachusetts Institute of Technology, Sept. 2001.

[10] D. E. Clarke, J.-E. Elien, C. M. Ellison, M. Fredette, A. Morcos, and R. L. Rivest. Certificate chain discovery in SPKI/SDSI. *Journal of Computer Security*, 9(4):285–322, 2001.

[11] J.-E. Elien. Certificate discovery using SPKI/SDSI 2.0 certificates. Master's thesis, Massachusetts Institute of Technology, May 1998.

[12] C. M. Ellison, B. Frantz, B. Lampson, R. L. Rivest, B. M. Thomas, and T. Ylonen. Simple public key certificate. Internet Engineering Task Force Draft IETF, July 1997.

[13] C. M. Ellison, B. Frantz, B. Lampson, R. L. Rivest, B. M. Thomas, and T. Ylonen. *SPKI Certificate Theory*, Sept. 1999. RFC2693.

[14] C. A. Gunter and T. Jim. Policy-directed certificate retrieval. *Software—Practice and Experience*, 30(15):1609–1640, Dec. 2000.

[15] T. Jim. SD3: A trust management system with certified evaluation. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, pages 106–115, Los Alamitos, CA, May 14–16 2001.

[16] 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, Nov. 1992.

[17] 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 and Privacy*, pages 114–130, Oakland, CA, May 2002.

[18] N. Li, W. H. Winsborough, and J. C. Mitchell. Distributed

credential chain discovery in trust management. *Journal of Computer Security*, 11(1):35–86, Feb. 2003.

[19] A. J. Maywah. An implementation of a secure Web client using SPKI/SDSI certificates. Master's thesis, Massachusetts Institute of Technology, May 2000.

[20] L. Paulson. Isabelle: A generic theorem prover. *Lecture Notes in Computer Science*, 828, 1994.

[21] R. L. Rivest and B. Lampson. SDSI—A simple distributed security infrastructure. Presented at CRYPTO'96 Rumpsession, Apr. 1996.

[22] S. Russel and P. Norvig. *Artificial Intelligence, A Modern Approach*. Prentice Hall, second edition, 2003.

[23] E. Wobber, M. Abadi, M. Burrows, and B. Lampson. Authentication in the Taos operating system. *ACM Transactions on Computer Systems*, 12(1):3–32, Feb. 1994.

A. Inference Rules of Our Logic

$$\frac{\mathit{pubkey}\ \textbf{signed}\ F}{\textbf{key}(\mathit{pubkey})\ \textbf{says}\ F}\qquad(\textsc{says-i})$$

$$\frac{A\ \textbf{says}\ (A.S\ \textbf{says}\ F)}{A.S\ \textbf{says}\ F}\qquad(\textsc{says-ln})$$

$$\frac{A\ \textbf{says}\ (B\ \textbf{speaksfor}\ A)\quad B\ \textbf{says}\ F}{A\ \textbf{says}\ F}\qquad(\textsc{speaksfor-e})$$

$$\frac{A\ \textbf{says}\ (B\ \textbf{speaksfor}\ A.S)\quad B\ \textbf{says}\ F}{A.S\ \textbf{says}\ F}\qquad(\textsc{speaksfor-e2})$$

$$\frac{A\ \textbf{says}\ (\textbf{delegate}(A,B,U))\quad B\ \textbf{says}\ (\textbf{action}(U,N))}{A\ \textbf{says}\ (\textbf{action}(U,N))}\qquad(\textsc{delegate-e})$$

B. Proof of Termination for a Distributed Prover

Notation Let *CP* refer to a centralized prover with tactics $\mathcal{T}$ and facts $\mathcal{F}$. Let *DP* refer to a distributed prover consisting of i cooperating nodes, each using tactics $\mathcal{T}$ and facts f_i such that $\bigcup_i f_i = \mathcal{F}$.

When comparing *CP* to *DP*, we will refer to line N as [Nc] or [Nd] if being run by *CP* or *DP* respectively. To refer to variable A on this line, we state [Nc].A or [Nd].A. When B is a function parameter, we shorten the notation to [c].B or [d].B. We introduce a special constant *localmachine* that represents the principal associated with the machine on which the prover is being run. Let [c].*result* represent the substitution returned by bc-ask in the centralized scenario, and [d].*result* represent the substitution returned in the distributed scenario. We make the assumption that all invocations of rpc are transparent to bc-ask.

B.1. Lemma 2

Lemma 2 *Consider two invocations of* bc-ask *made by CP and DP made under the following assumptions:*

1. bc-ask *is invoked with identical parameters in both scenarios*
2. *goals* $\neq$ [ ]
3. first(*goals*) *is such that [8d].l* $\neq$ *localmachine*
4. *Any recursive call to* bc-ask *will produce the same answer if invoked with the same parameters in both scenarios.*

Let $\alpha_1, \ldots, \alpha_k, \alpha_{k+1}$ *denote the sequence of return results from the* $(k+1)$ bc-ask *invocations on line 11 by DP, and let* $\beta_1, \ldots, \beta_{k'}$ *denote the sequence of return results of the* k' bc-ask *invocations on line 17 by CP that do not return* $\perp$. *Then,* $k = k'$ *and for each* $1 \leq i \leq k$, $\alpha_i = \beta_i$.

Proof We prove Lemma 2 by induction over i. Our induction hypothesis is that [11d].$\mathit{failures}'_i$ = [17c].$\mathit{failures}'_i$. Note that $\alpha_{k+1} = \perp$.

Base Case We must show that [11d].α_1 = [17c].β_1 and that [11d].$\mathit{failures}'_2$ = [17c].$\mathit{failures}'_2$. Since [11d].$\mathit{failures}'_1$ = [d].*failures* and [17c].$\mathit{failures}'_1$ = [c].*failures*, we can use Assumption 1 to conclude that [11d].$\mathit{failures}'_1$ = [17c].$\mathit{failures}'_1$. Assumption 1 tells us that [d].θ = [c].θ, from which we can conclude that [7d].q' = [7c].q'.

DP will call bc-ask (line 11) on machine l. Let [Nr] represent the execution of line N within this remote call.

5r–6r [r].*goals*=[d].first(*goals*), which cannot be empty, by Assumption 2, so the body of these if statements will never be executed.

7r first([r].*goals*) = first(first([d].*goals*)) = first([d].*goals*). Additionally, [r].θ = [d].θ. Since we know that [7d].q' = [7c].q', we can conclude that [7r].q' = [7c].q'.

8r Since *DP* made the RPC to [8d].l, [8r].l is *localmachine*.

9r [9r].$\mathit{failures}'_1$ = [r].*failures* = [11d].$\mathit{failures}'_1$.

10r Since [8r].l = *localmachine*, the body of this if statement ([11r]–[14r]) will never be executed.

15r Since [8r].l = *localmachine*, the body of this else statement will always be executed.

16r We let [c].$R \subseteq$ [c].KB represent the set of tactics with which [16c].q' can unify and [r].$R \subseteq$ [r].KB represent the set of tactics with which [16r].q' can unify. Knowing that [16r].q' = [16c].q', we now show that [r].R = [c].R. If [c].R_t represents the subset of [c].R that is tactics with subgoals and if [c].R_f represents the subset of [c].R that is facts of the form A **signed** F, [c].$R_t\cup$ [c].R_f = [c].R. By definition of our scenario, all machines in *DP* know all tactics with subgoals, so [r].R_t = [c].R_t. Furthermore, our scenario states that machine A knows all facts of the form A **signed** F. Since [8r].l = *localmachine*, [r].R_f = [c].R_f with respect to the formula q'. Having shown [r].R_t = [c].R_t and [r].R_f = [c].R_f, we can conclude that [r].R = [c].R.

Since [r].R = [c].R, if unify succeeds in one scenario, it will succeed in both. As a result, [16r].(P, q) = [16c].(P, q), which means that [16r].θ' = [16c].θ'.

17r [17r].$\mathit{failures}'$ = [11d].$\mathit{failures}'_1$, which we have shown to be equal to [17c].$\mathit{failures}'_1$. Assumption 4 tells us that any recursive call to bc-ask made by *DP* will produce the same answer as a call made by *CP* with the same parameters. Having shown the equality of all parameters to bc-ask, we can conclude that [17r].β = [17c].β. If $\beta = \bot$, both [c] and [r] will go to line 15 and repeat lines 16–17 using the next tactic. If no such tactic exists, they will both fall through to line 21 and return $\bot$. If $\beta \neq \bot$, then we have found [17c].β_1, and that [17r].β = [17c].β_1.

19–20r Since [r].$goals$ = first([d].$goals$), rest([r].$goals$) must be the empty set. Therefore, [19r].$answer$= [17r].β, which is equal to [17c].β_1.

Since [11d].α_1 = [r].$result$ and [r].$result$ = [17c].β_1, we can conclude [11d].α_1 = [17c].β_1 as desired. Since [11d].$failures'_1$ = [17c].$failures'_1$ and [11d].α_1 = [17c].β_1, the execution of [12d] and [18c] will produce [12d].$failures'_2$ = [18c].$failures'_2$ as desired.

Induction When the recursive call on [11d] is made for the ith time, [11d].$failures'_i$ = [d].$failures\cup$ [11d].$\alpha_1 \cup \ldots \cup$ [11d].α_{i-1} and [17c].$failures'_i$ = [c].$failures\cup$ [17c].$\beta_1 \cup \ldots \cup$ [17c].β_{i-1}.

5r–8r These lines will behave identically to the base case.

9r [9r].$failures'$ = [11d].$failures'_i$. Using our induction hypothesis, we can conclude that [9r].$failures'$ = [17c].$failures'_i$.

10r, 15r–16r These lines will behave identically to the base case.

17r Having shown the equality of all parameters to bc-ask, we can use Assumption 4 to conclude that [17r].β = [17c].β. As in the base case, if $\beta = \bot$, both [c] and [r] will go to line 15 and repeat lines 16–17 using the next tactic. If no such tactic exists, they will both fall through to line 21 and return $\bot$. If $\beta \neq \bot$, then we have found [17c].β_i, and that [17r].β = [17c].β_i.

19r–20r As in the base case, [r].$result$ = [17r].β.

[11d].α_i = [r].$result$, which is equal to [17c].β_i as desired. Since [11d].$failures'_i$ = [17c].$failures'_i$ and [11d].α_i = [17c].β_i, the execution of [12d] and [18c] will produce [12d].$failures'_{i+1}$ = [18c].$failures'_{i+1}$ as desired. Finally, we have shown that there is a one-to-one correspondence between α_i and β_i, and so $k = k'$. □

B.2. Lemma 3

Using Lemma 2, we now prove a stronger result. For the purposes of the following lemma, we define the recursion depth to be the number of times bc-ask directly invokes itself (i.e., invocations wrapped in RPC calls do not increase the recursion depth, but all others do).

Lemma 3 *If both CP and DP invoke* bc-ask *with parameters goals, θ, and failures, then [c].result = [d].result.*

Proof We prove Lemma 3 via induction on the recursion depth of bc-ask. Our induction hypothesis is that at a particular recursion depth, subsequent calls to bc-ask with identical parameters will return the same answer in *DP* as in *CP*.

Base Case The deepest point of recursion is when $goals$ is the empty list. Since [d].$failures$ = [c].$failures$ and [d].θ = [c].θ, lines 5–6 will execute identically in *DP* and *CP* returning either θ or $\bot$.

Induction In this case, $goals \neq$ [ ].

5d–6d Since [c].$goals$ = [d].$goals \neq$ [ ], both *DP* and *CP* proceed to line 7.

7d Because [c].$goals$ = [d].$goals$ and [c].θ = [d].θ, [7d].q' = [7c].q'.

8d–9d By definition of determine-location, [8c].l = $localmachine$. Depending on [7d].q', [8d].l may or may not be $localmachine$. We proceed to show that in either situation, [c].$result$ = [d].$result$.

In both cases, [c].$failures$ = [d].$failures$, and so [9c].$failures'$ = [9d].$failures'$.

Case A of 8d–9d: [8d].$l \neq$ *localmachine* We show that each assumption of Lemma 2 holds.

1 is an assumption of the current lemma as well.

2 is fulfilled by the definition of the inductive case we are trying to prove.

3 is true by the definition of Case A.

4 is true by our induction hypothesis.

Therefore, by Lemma 2, the sequence $\alpha_1, \ldots, \alpha_k, \alpha_{k+1}$ of return results from the $(k+1)$ bc-ask invocations on line 11 by *DP*, and the sequence $\beta_1, \ldots, \beta_{k'}$ of return results of the k' bc-ask invocations on line 17 by *CP* that do not return $\bot$ satisfy $k = k'$ and for each $1 \leq i \leq k$, $\alpha_i = \beta_i$. As a result, applying the induction hypothesis at [13d] and [19c] yields [13d].$answer$ = [19c].$answer$ in each iteration, and [c].$result$ = [d].$result$.

Case B of 8d–9d: [8c].l = [8d].l=*localmachine*

Analogously to the argument in the base case of Lemma 2 (line [16r]), [d].R = [c].R, where [c].R is set of tactics with which [16c].q' can unify, and [d].R is the set of tactics with which [16d].q' can unify. As a result, applying the induction hypothesis at [19d] and [19c] yields [19d].$answer$ = [19c].$answer$ in each iteration, and [c].$result$ = [d].$result$.

□

B.3. Theorem 1

Theorem 1 *For any goal G, a distributed prover using tactic set $\mathcal{T}$ will find a proof of G if and only if a centralized prover using $\mathcal{T}$ will find a proof of G.*

Proof Both *CP* and *DP* will attempt to prove G by invoking bc-ask with $goals = G$, θ equal to the empty substitution, and $failures$ = [ ]. Lemma 3 states that in this situation, the result returned by *CP* and *DP* is identical. From this, we can conclude that *DP* will find a solution to G if and only if *CP* finds a solution. □

$\mathcal{P}_1 = K_{CMU}\ \mathbf{signed}\ (\mathbf{key}(K_{CMU_S})\ \mathbf{speaksfor}\ \mathbf{key}(K_{CMU}))$
$\mathcal{P}_2 = K_{CMU}\ \mathbf{signed}\ (\mathbf{key}(K_{CMU_{CA}})\ \mathbf{speaksfor}\ \mathbf{key}(K_{CMU}).CA)$

$\mathcal{P}_3 = K_{CMU_{CA}}\ \mathbf{signed}\ (\mathbf{key}(K_{\mathsf{UserA}})\ \mathbf{speaksfor}\ \mathbf{key}(K_{CMU}).CA.\mathsf{UserA})$
$\mathcal{P}_4 = K_{CMU_{CA}}\ \mathbf{signed}\ (\mathbf{key}(K_{\mathsf{UserB}})\ \mathbf{speaksfor}\ \mathbf{key}(K_{CMU}).CA.\mathsf{UserB})$
$\mathcal{P}_5 = K_{CMU_{CA}}\ \mathbf{signed}\ (\mathbf{key}(K_{\mathsf{UserC}})\ \mathbf{speaksfor}\ \mathbf{key}(K_{CMU}).CA.\mathsf{UserC})$

$\mathcal{P}_6 = K_{CMU_S}\ \mathbf{signed}\ (\mathbf{delegate}(\mathbf{key}(K_{CMU}),\ \mathbf{key}(K_{CMU}).DH_1,\ \mathit{resource}))$
$\mathcal{P}_7 = K_{CMU_S}\ \mathbf{signed}\ (\mathbf{key}(K_{CMU}).CA.\mathsf{UserA}\ \mathbf{speaksfor}\ \mathbf{key}(K_{CMU}).DH_1)$

$\mathcal{P}_8 = K_{\mathsf{UserA}}\ \mathbf{signed}\ (\mathbf{delegate}(\mathbf{key}(K_{CMU}).DH_1,\ \mathbf{key}(K_{CMU}).DH_1.FM_1,\ \mathit{resource}))$
$\mathcal{P}_9 = K_{\mathsf{UserA}}\ \mathbf{signed}\ (\mathbf{key}(K_{CMU}).CA.\mathsf{UserB}\ \mathbf{speaksfor}\ \mathbf{key}(K_{CMU}).DH_1.FM_1)$

$\mathcal{P}_{10} = K_{\mathsf{UserB}}\ \mathbf{signed}\ (\mathbf{delegate}(\mathbf{key}(K_{CMU}).DH_1.FM_1,\ \mathbf{key}(K_{CMU}).CA.\mathsf{UserC},\ \mathit{resource}))$

$\mathcal{P}_{11} = K_{\mathsf{UserC}}\ \mathbf{signed}\ (\mathbf{action}(\mathit{resource}, \mathit{nonce}))$

0	$\mathbf{key}(K_{CMU})\ \mathbf{says}\ (\mathbf{key}(K_{CMU_S})\ \mathbf{speaksfor}\ \mathbf{key}(K_{CMU}))$	SAYS-I($\mathcal{P}_1$)
1	$\mathbf{key}(K_{CMU})\ \mathbf{says}\ (\mathbf{key}(K_{CMU_{CA}})\ \mathbf{speaksfor}\ \mathbf{key}(K_{CMU}).CA)$	SAYS-I($\mathcal{P}_2$)
2	$\mathbf{key}(K_{CMU_{CA}})\ \mathbf{says}\ (\mathbf{key}(K_{\mathsf{UserA}})\ \mathbf{speaksfor}\ \mathbf{key}(K_{CMU}).CA.\mathsf{UserA})$	SAYS-I($\mathcal{P}_3$)
3	$\mathbf{key}(K_{CMU_{CA}})\ \mathbf{says}\ (\mathbf{key}(K_{\mathsf{UserB}})\ \mathbf{speaksfor}\ \mathbf{key}(K_{CMU}).CA.\mathsf{UserB})$	SAYS-I($\mathcal{P}_4$)
4	$\mathbf{key}(K_{CMU_{CA}})\ \mathbf{says}\ (\mathbf{key}(K_{\mathsf{UserC}})\ \mathbf{speaksfor}\ \mathbf{key}(K_{CMU}).CA.\mathsf{UserC})$	SAYS-I($\mathcal{P}_5$)
5	$\mathbf{key}(K_{CMU}).CA\ \mathbf{says}\ (\mathbf{key}(K_{\mathsf{UserA}})\ \mathbf{speaksfor}\ \mathbf{key}(K_{CMU}).CA.\mathsf{UserA})$	SPEAKSFOR-E2(1, 2)
6	$\mathbf{key}(K_{CMU}).CA\ \mathbf{says}\ (\mathbf{key}(K_{\mathsf{UserB}})\ \mathbf{speaksfor}\ \mathbf{key}(K_{CMU}).CA.\mathsf{UserB})$	SPEAKSFOR-E2(1, 3)
7	$\mathbf{key}(K_{CMU}).CA\ \mathbf{says}\ (\mathbf{key}(K_{\mathsf{UserC}})\ \mathbf{speaksfor}\ \mathbf{key}(K_{CMU}).CA.\mathsf{UserC})$	SPEAKSFOR-E2(1, 4)
8	$\mathbf{key}(K_{CMU_S})\ \mathbf{says}\ (\mathbf{key}(K_{CMU}).CA.\mathsf{UserA}\ \mathbf{speaksfor}\ \mathbf{key}(K_{CMU}).DH_1)$	SAYS-I($\mathcal{P}_7$)
9	$\mathbf{key}(K_{CMU})\ \mathbf{says}\ (\mathbf{key}(K_{CMU}).CA.\mathsf{UserA}\ \mathbf{speaksfor}\ \mathbf{key}(K_{CMU}).DH_1)$	SPEAKSFOR-E(0, 8)
10	$\mathbf{key}(K_{\mathsf{UserA}})\ \mathbf{says}\ (\mathbf{key}(K_{CMU}).CA.\mathsf{UserB}\ \mathbf{speaksfor}\ \mathbf{key}(K_{CMU}).DH_1.FM_1)$	SAYS-I($\mathcal{P}_9$)
11	$\mathbf{key}(K_{CMU}).CA.\mathsf{UserA}\ \mathbf{says}\ (\mathbf{key}(K_{CMU}).CA.\mathsf{UserB}\ \mathbf{speaksfor}\ \mathbf{key}(K_{CMU}).DH_1.FM_1)$	SPEAKSFOR-E2(5, 10)
12	$\mathbf{key}(K_{CMU}).DH_1\ \mathbf{says}\ (\mathbf{key}(K_{CMU}).CA.\mathsf{UserB}\ \mathbf{speaksfor}\ \mathbf{key}(K_{CMU}).DH_1.FM_1)$	SPEAKSFOR-E2(9, 11)
13	$\mathbf{key}(K_{CMU_S})\ \mathbf{says}\ \mathbf{delegate}(\mathbf{key}(K_{CMU}),\ \mathbf{key}(K_{CMU}).DH_1,\ \mathit{resource})$	SAYS-I($\mathcal{P}_6$)
14	$\mathbf{key}(K_{CMU})\ \mathbf{says}\ \mathbf{delegate}(\mathbf{key}(K_{CMU}),\ \mathbf{key}(K_{CMU}).DH_1,\ \mathit{resource})$	SPEAKSFOR-E(0, 13)
15	$\mathbf{key}(K_{\mathsf{UserA}})\ \mathbf{says}\ \mathbf{delegate}(\mathbf{key}(K_{CMU}).DH_1,\ \mathbf{key}(K_{CMU}).DH_1.FM_1,\ \mathit{resource})$	SAYS-I($\mathcal{P}_8$)
16	$\mathbf{key}(K_{CMU}).CA.\mathsf{UserA}\ \mathbf{says}\ \mathbf{delegate}(\mathbf{key}(K_{CMU}).DH_1,\ \mathbf{key}(K_{CMU}).DH_1.FM_1,\ \mathit{resource})$	SPEAKSFOR-E2(5, 15)
17	$\mathbf{key}(K_{CMU}).DH_1\ \mathbf{says}\ \mathbf{delegate}(\mathbf{key}(K_{CMU}).DH_1,\ \mathbf{key}(K_{CMU}).DH_1.FM_1,\ \mathit{resource})$	SPEAKSFOR-E2(9, 16)
18	$\mathbf{key}(K_{\mathsf{UserB}})\ \mathbf{says}\ \mathbf{delegate}(\mathbf{key}(K_{CMU}).DH_1.FM_1,\ \mathbf{key}(K_{CMU}).CA.\mathsf{UserC},\ \mathit{resource})$	SAYS-I($\mathcal{P}_{10}$)
19	$\mathbf{key}(K_{CMU}).CA.\mathsf{UserB}\ \mathbf{says}\ \mathbf{delegate}(\mathbf{key}(K_{CMU}).DH_1.FM_1,\ \mathbf{key}(K_{CMU}).CA.\mathsf{UserC},\ \mathit{resource})$	SPEAKSFOR-E2(6, 18)
20	$\mathbf{key}(K_{CMU}).DH_1.FM_1\ \mathbf{says}\ \mathbf{delegate}(\mathbf{key}(K_{CMU}).DH_1.FM_1,\ \mathbf{key}(K_{CMU}).CA.\mathsf{UserC},\ \mathit{resource})$	SPEAKSFOR-E2(12, 19)
21	$\mathbf{key}(K_{\mathsf{UserC}})\ \mathbf{says}\ \mathbf{action}(\mathit{resource}, \mathit{nonce})$	SAYS-I($\mathcal{P}_{11}$)
22	$\mathbf{key}(K_{CMU}).CA.\mathsf{UserC}\ \mathbf{says}\ \mathbf{action}(\mathit{resource},\ \mathit{nonce})$	SPEAKSFOR-E2(7, 21)
23	$\mathbf{key}(K_{CMU}).DH_1.FM_1\ \mathbf{says}\ \mathbf{action}(\mathit{resource},\ \mathit{nonce})$	DELEGATE-E(20, 22)
24	$\mathbf{key}(K_{CMU}).DH_1\ \mathbf{says}\ \mathbf{action}(\mathit{resource},\ \mathit{nonce})$	DELEGATE-E(17, 23)
25	$\mathbf{key}(K_{CMU})\ \mathbf{says}\ \mathbf{action}(\mathit{resource},\ \mathit{nonce})$	DELEGATE-E(14, 24)

Figure 9. Proof of $\mathbf{key}(K_{CMU})\ \mathbf{says}\ \mathbf{action}(\mathit{resource},\ \mathit{nonce})$

C. Sample Proof of Access

Figure 9 shows a proof that allows UserC to access *resource*, a resource controlled by K_{CMU} using the policy described in Section 5.1. The goal that must be proved is $\mathbf{key}(K_{CMU})\ \mathbf{says}\ \mathbf{action}(\mathit{resource}, \mathit{nonce})$. $\mathcal{P}_1$–$\mathcal{P}_{11}$ represent the necessary certificates, and below them is the proof.

The inference rules used by this proof are those of Appendix A. This proof is representative of those generated by our prover during the simulations of Section 5.

In our simulations, a certificate like $\mathcal{P}_3$–$\mathcal{P}_5$ is generated for each principal. Each department head is given authority over each resource in the corresponding department via certificates like $\mathcal{P}_6$, and the job of department head is assigned to a particular user via a certificate like $\mathcal{P}_7$; each floor manager position is similarly created and populated by certificates such as $\mathcal{P}_8$–$\mathcal{P}_9$; and each user authorized to use *resource* receives a certificate similar to $\mathcal{P}_{10}$. Finally, every user attempting to access a resource creates a certificate like $\mathcal{P}_{11}$.