Efficient Proving for Practical Distributed Access-Control Systems

Lujo Bauer Scott Garriss Michael K. Reiter

August 3, 2007
CMU-CyLab-06-015R

Revision of:
September 29, 2006
CMU-CyLab-06-015

CyLab
Carnegie Mellon University
Pittsburgh, PA 15213

Efficient Proving for Practical Distributed Access-Control Systems*

Lujo Bauer† Scott Garriss† Michael K. Reiter‡

Abstract

We present a new technique for generating a formal proof that an access request satisfies access-control policy, for use in logic-based access-control frameworks. Our approach is tailored to settings where credentials needed to complete a proof might need to be obtained from, or reactively created by, distant components in a distributed system. In such contexts, our approach substantially improves upon previous proposals in both computation and communication costs, and better guides users to create the most appropriate credentials in those cases where needed credentials do not yet exist. At the same time, our strategy offers strictly superior proving ability, in the sense that it finds a proof in every case that previous approaches would (and more). We detail our method and evaluate an implementation of it using both policies in active use in an access-control testbed at our institution and larger policies indicative of a widespread deployment.

1 Introduction

Much work has given credence to the notion that formal reasoning can be used to buttress the assurance one has in an access-control system. While early work in this vein *modeled* access-control systems using formal logics (e.g., [11, 25]), recent work has imported logic into the system as a means to *implement* access control (e.g., [5]). In these systems, the resource monitor evaluating an access request requires a proof, in formal logic, that the access satisfies access-control policy. In such a proof, digitally signed credentials are used to instantiate formulas of the logic (e.g., "K_{Alice} **signed delegate**$(\mathsf{Alice}, \mathsf{Bob}, \mathsf{resource})$" or "$K_{\mathsf{CA}}$ **signed** K_{Alice} **speaksfor** $K_{\mathsf{CA}}.\mathsf{Alice}$"), and then inference rules are used to derive a proof that a required policy is satisfied (e.g., "$\mathsf{Manager}$ **says open**$(\mathsf{resource})$"). The resource monitor, then, need only validate that each request is accompanied by a valid proof of the required policy.

Because the resource monitor accepts *any* valid proof of the required policy, this framework offers potentially a high degree of flexibility in how proofs are constructed. This flexibility, however, is not without its costs. First, it is essential that the logic is sound and free from unintended consequences, giving rise to a rich literature in designing appropriate authorization logics (e.g., [11, 27, 19, 16]). Second, and of primary concern in this paper, it must be possible to efficiently find proofs for accesses that should be allowed. Rather than devising a proving strategy customized to each application, we would prefer to develop a general proof-building strategy that is driven by the logic itself and that is effective in a wide range of applications.

In this paper we focus on systems where needed credentials are distributed among different components, if they exist at all, and may be created at distant components reactively and with human intervention. Such systems give rise to new requirements for credential-creation and proof-construction algorithms. To

*This work was supported in part by NSF grant 0433540, grant DAAD19-02-1-0389 from the Army Research Office, and the AFRL/IF Pollux project.

†Carnegie Mellon University

‡University of North Carolina at Chapel Hill

address these requirements, we combine a number of new and existing techniques into a proof-generation strategy that is qualitatively different from those proposed by previous works. In comparison to these works (notably [4]), we show that our strategy offers dramatic improvements in the efficiency of proof construction in practice, consequently making such systems significantly more useable. Moreover, our strategy will find proofs whenever previous algorithms would (and sometimes even when they would not). Our method builds from three key principles. First, our method strategically delays pursuing "expensive" subgoals until, through further progress in the proving process, it is clear that these subgoals would be helpful to prove. Second, our method precomputes delegation chains between principles in a way that can significantly optimize the proving process on the critical path of an access. Third, our method eliminates the need to hand-craft *tactics*, a fragile and time-intensive process, to efficiently guide the proof search. Instead, it utilizes a new, systematic approach to generating tactics from the inference rules of the logic.

The technique we report here is motivated by an ongoing deployment at our institution of a testbed environment where proof-based access control is used to control access to both physical resources (e.g., door access) and information resources (e.g., computer logins). The system has been deployed for over a year, guards access to about 35 resources spanning two floors of our office building, and is used daily by over 35 users. In this deployment, smartphones are used as the vehicle for constructing proofs and soliciting consent from users for the creation of new credentials, and the cellular network is the means by which these smartphones communicate to retrieve needed proofs of subgoals. In such an environment, both computation and communication have high latency, and so limiting use of these resources is essential to offering reasonable response times to users. And, for the sake of usability, it is essential that we involve users in the proof generation process (i.e., to create new credentials) infrequently and with as much guidance as possible. We have developed the technique we report here with these goals in mind, and our deployment suggests that it offers acceptable performance for the policies with which we have experimented and is a drastic improvement over previous approaches. All of the examples used in this paper are actual policies drawn from the deployment. We evaluate the scalability of our algorithm on larger, synthetically generated policies in Section 7.2 and show that the quantity of precomputed state remains reasonable and the performance advantage of our approach remains or increases as the policy grows. Our approach has applications beyond the particular setting in which we describe it; we briefly discuss one such application in Section 8.

The contributions of this paper are to: (1) identify the requirements of a proving algorithm in a distributed access-control system with dynamic credential creation (Section 2); (2) propose mechanisms for precomputing delegation chains (Section 5) and systematically generating tactics (Section 6.2); (3) describe a technique for utilizing these pre-computed results to find proofs in dramatically less time than previous approaches (Section 6); and (4) evaluate our technique on a collection of policies representative of those used in practice (Section 7.1) and those indicative of a larger deployment (Section 7.2). In Section 8, we discuss the use of our techniques in the context of additional logics, systems, and applications.

2 Goals and Contributions

As discussed in Section 1, we will describe new techniques for generating proofs in an authorization logic that an access request is consistent with access-control policy. It will be far easier to discuss our approach in the context of a concrete authorization logic, and for this purpose we utilize the same sample logic as we used in previous work [4], which is reproduced in Figure 1. However, our techniques are not specific to this logic, or even necessarily to a logic-based system; rather, they can be adapted to a wide range of authorization systems provided that they build upon a similar notion of delegation, as discussed in Section 8.

If pubkey is a particular public key, then $\mathbf{key}(\texttt{pubkey})$ is the principal that corresponds to that key. If Alice is a principal, we write Alice.$\texttt{secretary}$ to denote the principal whom Alice calls "secretary." The formulas of our logic describe principals' beliefs. If Alice believes that the formula F is true, we write Alice $\mathbf{says}$ 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 by the formula pubkey $\mathbf{signed}$ F, which can be transformed into a belief ($\mathbf{key}(\texttt{pubkey})$ $\mathbf{says}$ F) using the SAYS-I inference rule. To describe a resource that a client wants to access, we use the $\mathbf{open}$ constructor. A principal believes the formula $\mathbf{open}(\mathit{resource})$ if she thinks that it is OK to access $\mathit{resource}$.[1] Delegation is described with the $\mathbf{speaksfor}$ and $\mathbf{delegate}$ predicates. The formula Alice $\mathbf{speaksfor}$ Bob indicates that Bob has delegated to Alice his authority to make access-control decisions about any resource. $\mathbf{delegate}(\mathsf{Bob}, \mathsf{Alice}, \mathit{resource})$ transfers to Alice only the authority to access the resource called $\mathit{resource}$.

$$\phi \quad ::= \quad s\ \textbf{signed}\ \phi' \mid p\ \textbf{says}\ \phi'$$
$$\phi' \quad ::= \quad \textbf{open}\ (s) \mid p\ \textbf{speaksfor}\ p \mid \textbf{delegate}(p, p, s)$$

(s ranges over strings and p over principals)

$$\frac{\mathit{pubkey}\ \textbf{signed}\ F}{\textbf{key}(\mathit{pubkey})\ \textbf{says}\ F} \quad (\text{SAYS-I})$$

$$\frac{A\ \textbf{says}\ (A.S\ \textbf{says}\ F)}{A.S\ \textbf{says}\ F} \quad (\text{SAYS-LN})$$

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

$$\frac{A\ \textbf{says}\ (B\ \textbf{speaksfor}\ A.S) \quad B\ \textbf{says}\ F}{A.S\ \textbf{says}\ F} \quad (\text{SPEAKSFOR-E2})$$

$$\frac{A\ \textbf{says}\ (\textbf{delegate}(A, B, U)) \quad B\ \textbf{says}\ (\textbf{open}(U, N))}{A\ \textbf{says}\ (\textbf{open}(U, N))} \quad (\text{DELEGATE-E})$$

Figure 1: A sample access-control logic [4]

2.1 Requirements

To motivate our requirements, we use as an example a simple policy in use on a daily basis in our system. This policy is chosen for illustrative purposes; the performance advantage of our technique actually widens as the policy becomes more complicated (see Section 7.2). All the resources in our example are owned by our academic department, and so to access a resource (resource) one must prove that the department has authorized the access (Dept $\mathbf{says}$ $\mathbf{open}(\mathsf{resource})$).

Alice is the manager in charge of a machine room with three entrances: door1, door2, and door3. To place her in charge, the department has created credentials giving Alice access to each door, e.g., K_{Dept} $\mathbf{signed}$ $\mathbf{delegate}(\mathsf{Dept}, \mathsf{Alice}, \mathsf{door1})$. Alice's responsibilities include deciding who else may access the machine room. Instead of individually delegating access to each door, Alice has organized her security policy by (1) creating a group Alice.machine-room; (2) giving all members of that group access to each door (e.g., K_{Alice} $\mathbf{signed}$ $\mathbf{delegate}(\mathsf{Alice}, \mathsf{Alice.machine\text{-}room}, \mathsf{door1})$); and, finally, (3) making individuals like Bob members of the group (K_{Alice} $\mathbf{signed}$ (Bob $\mathbf{speaksfor}$ Alice.machine-room)).

Suppose that Charlie, who currently does not have access to the machine room, wishes to open one of the machine-room doors. When his smartphone contacts the door, it is told to prove Dept $\mathbf{says}$ $\mathbf{open}(\mathsf{door1})$. The proof is likely to require credentials created by the department, by Alice, and perhaps also by Bob, who may be willing to redelegate the authority he received from Alice.

Previous approaches to distributed proof generation (notably [4] and [31]) did not attempt to address three requirements that are crucial in practice. Each requirement may appear to be a trivial extension of some previously studied proof-generation algorithm. However, straightforward implementation attempts suffer from problems that lead to greater inefficiency than can be tolerated in practice, as will be detailed below.

[1] $\mathbf{open}$ takes a nonce as a second parameter, which we omit here for simplicity.

Credential creation Charlie will not be able to access door1 unless Alice, Bob, or the department creates a credential to make that possible. The proof-generation algorithm should intelligently guide users to create the "right" credential, e.g., K_{Alice} **signed** (Charlie **speaksfor** Alice.machine-room), based on other credentials that already exist. This increases the computation required, as the prover must additionally investigate branches of reasoning that involve credentials that have not yet been created.

Exposing choice points When it is possible to make progress on a proof in a number of ways (i.e., by creating different credentials or by asking different principals for help), the choice points should be exposed to the user instead of being followed automatically. Exposing the choice points to the user makes it possible both to generate proofs more efficiently by taking advantage of the user's knowledge (e.g., Charlie might know that Bob is likely to help but Alice isn't) and to avoid undesired proving paths (e.g., bothering Alice at 3AM with a request to create credentials, when she has requested she not be). This increase in overall efficiency comes at a cost of increased local computation, as the prover must investigate all possible choice points prior to asking the user.

Local proving Previous work showed that proof generation in distributed environments was feasible under the assumption that each principal attempted to prove only the formulas pertaining to her own beliefs (e.g., Charlie would attempt to prove formulas like Charlie **says** F, but would immediately ask Bob for help if he had to prove Bob **says** G) [4]. In our example, if Charlie asks Alice for help, Alice is able to create sufficient credentials to prove Dept **says** **open**(door1), even though this proof involves reasoning about the department head's beliefs. Avoiding a request to the department head in this case improves the overall efficiency of proof generation, but in general requires Alice to try to prove all goals for which she would normally ask for help, again increasing the amount of local computation.

The increase in computation imposed by each requirement may seem reasonable, but when implemented as a straightforward extension of previous work, Alice's prover running on a Nokia N70 smartphone will take over 5 *minutes* to determine the set of possible ways in which she can help Charlie gain access. Using the technique described in this paper, Alice is able to find the most common options (see Section 6.2) in 2 seconds, and is able to find a provably complete set of options in well less than a minute.

2.2 Insights

We address the requirements outlined in Section 2.1 with a new distributed proving strategy that is both efficient in practice and that sacrifices no proving ability relative to prior approaches. The insights embodied in our new strategy are threefold and we describe them here with the help of the example from Section 2.1.

Minimizing expensive proof steps In an effort to prove Dept **says** **open**(door1), suppose Charlie's prover directs a request for help to Alice. Alice's prover might decompose the goal Dept **says** **open**(door1) in various ways, some that would require the consent of the user Alice to create a new credential (e.g., Alice **says** Charlie **speaksfor** Alice.machine-room) and others that would involve making a remote query (e.g., to Dept, since this is Dept's belief). We have found that naively pursuing such options inline, i.e., when the prover first encounters them, is not reasonable in a practical implementation, as the former requires too much user interaction and the latter induces too much network communication and remote proving.

We employ a *delayed* proof procedure that vastly improves on these alternatives for the policies we have experimented with in practice. Roughly speaking, this procedure strategically bypasses formulas that are the most expensive to pursue, i.e., requiring either a remote query or the local user consenting to signing the formula directly. Each such formula is revisited only if subsequent steps in the proving process show that proving it would, in fact, be useful to completing the overall proof. In this way, the most expensive steps in the proof process are skipped until only those that would actually be useful are determined. These useful steps may be collected and presented to the user to aid in the decision-making process. This is done by our delayed distributed proving algorithm, which is described in Section 6.1.

Precomputing delegation chains A second insight is to locally precompute and cache delegation chains using two approaches: the well-studied *forward chaining* algorithm [33] and *path compression*, which we introduce here. Unlike backward chaining, which recursively decomposes goals into subgoals, these techniques work forward from a prover's available credentials (its *knowledge base*) to derive both facts and metalogical implications of the form "if we prove $\mathsf{Charlie}\ \mathbf{says}\ F$, then we can prove $\mathsf{David}\ \mathbf{says}\ F$". By computing these implications off the critical path, numerous lengthy branches can be avoided during backward chaining. While these algorithms can theoretically produce a knowledge base whose size is exponential in the number of credentials known, our evaluation indicates that in practice most credentials do not combine, and that the size of the knowledge base increases roughly linearly with the number of credentials (see Section 7.2). As we discuss in Section 6.2, the chief challenge in using precomputed results is to effectively integrate them in an exhaustive time-of-access proof search that involves hypothetical credentials.

If any credential should expire or be revoked, any knowledge derived from that credential will be removed from the knowledge base. Each element in the knowledge base is accompanied by an explicit derivation (i.e., a proof) of the element from credentials. Our implementation searches the knowledge base for any elements that are derived from expired or revoked credentials and removes them. Our technique is agnostic to the underlying revocation mechanism.

Systematic tactic generation Another set of difficulties in constructing proofs is related to constructing the tactics that guide a backward-chaining prover in how it decomposes a goal into subgoals. One approach to constructing tactics is simply to use the inference rules of the logic as tactics. With a depth-limiter to ensure termination, this approach ensures that all possible proofs up to a certain size will be found, but is typically too inefficient for use on the critical path of an access because it may enumerate all possible proof shapes. A more efficient construction is to hand-craft a set of tactics by using multiple inference rules per tactic to create a more specific set of tactics [15]. The tactics tend to be designed to look for certain types of proofs at the expense of completeness. Additionally, the tactics are tedious to construct, and do not lend themselves to formal analysis. While faster than inference rules, the hand-crafted tactics can still be inefficient, and, more importantly, often suffer loss of proving ability when the policy grows larger or deviates from the ones that inspired the tactics.

A third insight of the approach we describe here is a new, *systematic* approach for generating tactics from inference rules. This contribution is enabled by the forward chaining and path compression algorithms mentioned above. In particular, since our prover can rely on the fact that all delegation chains have been precomputed, its tactics need not attempt to derive the delegation chains directly from credentials when generating a proof of access. This reduces the difficulty of designing tactics. In our approach, an inference rule having to do with delegation gives rise to two tactics: one whose chief purpose is to look up previously computed delegation chains, and another that identifies the manner in which previously computed delegation chains may be extended by the creation of further credentials. All other inference rules are used directly as tactics.

3 Proposed Approach

The prover operates over a *knowledge base* that consists of tactics, locally known credentials, and facts that can be derived from these credentials. The proving strategy we propose consists of three parts. First, we use the existing technique of forward chaining to extend the local knowledge base with all facts that it can derive from existing knowledge (Section 4). Second, a path-compression algorithm (which we introduce in Section 5) computes delegation chains that can be derived from the local knowledge base but that cannot be derived through forward chaining. Third, a backward-chaining prover uses our systematically generated tactics to take advantage of the knowledge generated by the first two steps to efficiently compute proofs of a particular goal (e.g., Dept $\mathbf{says}\ \mathbf{open}(\mathsf{door1})$) (Section 6).

The splitting of the proving process into distinct pieces is motivated by the observation that if Charlie is trying to access door1, he is interested in minimizing the time between the moment he indicates his intention to access door1 and the time he is able to enter. Any part of the proving process that takes place *before* Charlie attempts to access door1 is effectively invisible to him. By completely precomputing certain types of knowledge, the backward-chaining prover can avoid some costly branches of investigation, thus reducing the time the user spends waiting.

4 Forward Chaining

Forward chaining (FC) is a well-studied proof-search technique in which all known ground facts (true formulas that do not contain free variables) are exhaustively combined using inference rules until either a proof of the formula contained in the query is found, or the algorithm reaches a fixed point from which no further inferences can be made. We use a variant of the algorithm known as incremental forward chaining [33] in which state is preserved across queries, allowing the incremental addition of a single fact to the knowledge base. The property we desire from FC is *completeness*—that it finds a proof of every formula for which a proof can be found from the credentials in the knowledge base (KB). More formally:

Theorem 1 *After each credential $f \in KB$ has been incrementally added via* FC*, for any $p_1 \ldots p_n \in KB$, if $(p_1 \wedge \ldots \wedge p_n) \supset q$ then $q \in KB$.*

Forward chaining has been shown to be complete for Datalog knowledge bases, which consist of definite clauses without function symbols [33]. Many access-control logics are a subset of, or can be translated into, Datalog (e.g., [21, 27]). Our sample security logic is not a subset of Datalog because it contains function symbols, but we proceed to show that in certain cases, such as ours, FC is complete for knowledge bases that contain function symbols. The initial knowledge base may be divided into definite clauses that have premises, and those that do not. We refer to these as rules and credentials, respectively. A *fact* is any formula for which a proof exists. A *term* is either a constant, a variable, or a function applied to terms. A *ground* term is a term that does not contain a variable. A *function symbol*, such as the successor function $s()$ that iterates natural numbers, maps terms to terms. Function symbols present a problem in that they may be used to create infinitely many ground terms (e.g., as with the successor function), which in turn may cause FC to construct an infinite number of ground facts. However, some knowledge bases may contain function symbols yet lack the proper rules to construct an infinite number of terms.

Our sample security logic is one such knowledge base. The logic makes use of two functions: $\mathbf{key}$ and dot $(.)$. The single inference rule that applies $\mathbf{key}$ is only able to match its premise against a credential. As a result, $\mathbf{key}$ may be applied only once per credential. Dot is used to specify nested names, for which the depth of nesting is not constrained. However, no inference rule has a conclusion with a nested name of depth greater than the depth of any name mentioned in one of its premises. Therefore, the deepest nested name possible for a given KB is the deepest name mentioned explicitly in a credential.

Completeness of forward chaining can be proven by first showing that there are a constant number of ground terms, which implies that the algorithm will terminate. By analyzing the structure of the algorithm, it can be shown that when the algorithm terminates, forward chaining has inferred all possible facts. In the class of knowledge bases described above, there is a finite number of constants to which a finite number of functions can be applied a finite number of times. This implies that the number of possible ground terms is also finite. From this point, the proof of completeness is analogous to the one presented by Russell and Norvig [33].

5 Path Compression

A *path* is a delegation chain between two principals A and B such that a proof of B **says** F implies that a proof of A **says** F can be found. Some paths are represented directly in the logic (e.g., B **speaksfor** A). Other paths, such as the path between A and C that results from the credentials K_A **signed** (B **speaksfor** A) and K_B **signed** (C **speaksfor** B), cannot be expressed directly—they are metalogical constructs, and cannot be computed by FC. More formally, we define a path as follows:

Definition 1 *A path* (A **says** F, B **says** F) *is a set of credentials* $c_1, \ldots, c_n$ *and a proof* P *of* ($c_1, \ldots, c_n$, A **says** F) $\supset B$ **says** F.

For example, the credential K_{Alice} **signed** Bob **speaksfor** Alice will produce the path (Bob **says** F, Alice **says** F), where F is an unbound variable. Now, for any concrete formula g, if Bob **says** g is true, we can conclude Alice **says** g. If Bob issues the credential K_{Bob} **signed delegate**(Bob, Charlie, resource), then we can construct the path (Charlie **says open**(resource), Bob **says open**(resource)). Since the conclusion of the second path unifies with the premise of the first, we can combine them to construct the path (Charlie **says open**(resource), Alice **says open**(resource)). Unlike the two credentials above, some delegation credentials represent a meaningful path only if another path already exists. For example, Alice could delegate authority to Bob on behalf of Charlie (e.g., K_{Alice} **signed delegate**(Charlie, Bob, resource)). This credential by itself is meaningless because Alice lacks the authority to speak on Charlie's behalf. We say that this credential *depends on* the existence of a path from Alice to Charlie, because this path would give Alice the authority to speak on Charlie's behalf. Consequently, we call such credentials *dependent*, and others *independent*.

Algorithm Our path compression algorithm, shown in Figure 2, is divided into two subroutines: PC and add-path. The objective of PC is to determine if a given credential represents a meaningful path, and, if so, add it to the set of known paths by invoking add-path. add-path is responsible for constructing all other possible paths using this new path, and for adding all new paths to the knowledge base. The subroutine subst performs a free-variable substitution and unify returns the most general substitution (if one exists) that, when applied to both parameters, produces equivalent formulas.

PC ignores any credential that does not contain a delegation statement (Line 3 of Figure 2). If a new credential does not depend on another path, or depends on a path that exists, it will be passed to add-path (Line 9). If the credential depends on a path that does not exist, the credential is instead stored in $incompletePaths$ for later use (Lines 5–7). Whenever a new path is added, PC must check if any of the credentials in $incompletePaths$ are now meaningful (Lines 10–12), and, if so, covert them to paths and add the result to the knowledge base (Lines 13–14).

After adding the new path to the global set of paths (Line 17), add-path finds the already-computed paths that can be appended to the new path, appends them, and adds the resulting paths to the global set (Lines 19–23). Next, add-path finds the existing paths that can be prepended to the paths created in the first step, prepends them, and saves the resulting paths (Lines 24–28). To prevent cyclic paths from being saved,

```
global set paths                                        /* All known delegation chains */
global set incompletePaths                              /* All known incomplete chains */

PC(credential f)
    if (credToPath(f) = ⊥), return                      /* If not a delegation, do nothing. */
    (x, y) ← depends-on(f)                              /* If input is a third-person
    if (((x, y) ≠ ⊥) ∧ ¬((x, y) ∈ paths))                  delegation, add it to incompletePaths.*/
        incompletePaths ← incompletePaths ∪ (f, (x, y))
        return

    (p, q) ← credToPath(f)                              /* Convert input credential into a path. */
    add-path((p, q))

    foreach (f′, (x′, y′)) ∈ incompletePaths            /* Check if new paths make any previously
        foreach (p″, q″) ∈ paths                           encountered third-person credentials
            if((θ ← unify((x′, y′), (p″, q″))) ≠ ⊥)        useful. */
                (p′, q′) ← credToPath(f′)
                add-path((subst(θ, p′), subst(θ, q′)))

add-path(path (p, q))
    local set newPaths = {}
    paths ← union((p, q), paths)                        /* Add the new path to set of paths. */
    newPaths ← union((p, q), newPaths)

    foreach (p′, q′) ∈ paths
        if((θ ← unify(q, p′)) ≠ ⊥)                      /* Try to prepend new path to
            c ← (subst(θ, p), subst(θ, q′))                all previous paths. */
            paths ← union(c, paths)
            newPaths ← union(c, paths)

    foreach (p′, q′) ∈ paths
        foreach (p″, q″) ∈ newPaths                     /* Try to append all new paths
            if((θ ← unify(q′, p″)) ≠ ⊥)                    to all previous paths. */
                c ← (subst(θ, p′), subst(θ, q″))
                paths ← union(c, paths)
```

Figure 2: PC, an incremental path-compression algorithm

the union subroutine adds a path only if the path does not represent a cycle. That is, $\mathsf{union}((p,q),S)$ returns S if $\mathsf{unify}(p,q) \neq \bot$, and $S \cup \{(p,q)\}$ otherwise.

5.1 Completeness of PC

The property we desire of PC is that it constructs all possible paths that are derivable from the credentials it has been given as input. We state this formally below; the proof is in Appendix A.

Theorem 2 *If* PC *has completed on* KB*, then for any* A, B *such that* $A \neq B$*, if for some* F $(B\ \mathbf{says}\ F \supset A\ \mathbf{says}\ F)$ *in the context of* KB*, then* $(B\ \mathbf{says}\ F, A\ \mathbf{says}\ F) \in KB$.

Informally: We first show that add-path will combine all paths that can be combined—that is, for any paths (p,q) and (p',q') if q unifies with p' then the path (p,q') will be added. We then show that for all credentials that represent a path, add-path is immediately invoked for independent credentials (Line 9), and all credentials that depend on the existence of another path are passed to add-path whenever that path becomes known (Lines 10–14).

6 Backward Chaining

Backward-chaining provers are composed of tactics that describe how formulas might be proved and a backward-chaining engine that uses tactics to prove a particular formula. The backward-chaining part of our technique must perform two novel tasks. First, the backward-chaining engine needs to expose choice points to the user. At each such point the user can select, e.g., which of several local credentials to create, or which of several principals to ask for help. Second, we want to craft the tactics to take advantage of facts precomputed through forward chaining and path compression to achieve greater efficiency and better coverage of the proof space than previous approaches.

6.1 Delayed Backward Chaining

While trying to generate a proof, the prover may investigate subgoals for which user interaction is necessary, e.g., to create a new credential or to determine the appropriate remote party to ask for help. We call these subgoals *choice subgoals*, since they will not be investigated unless the user explicitly chooses to do so. The distributed theorem-proving approach of our previous work [4] attempted to pursue each choice subgoal as it was discovered, thus restricting user interaction to a series of yes or no questions. Our insight here is to pursue a choice subgoal only after all other choice subgoals have been identified, thus *delaying* the proving of all choice subgoals until input can be solicited from the user. This affords the user the opportunity to guide the prover by selecting the choice subgoal that is most appropriate to pursue first.

Converting the algorithm from previous work to the delayed strategy is straightforward. Briefly, the delayed algorithm operates by creating a placeholder proof whenever it encounters a choice subgoal. The algorithm then backtracks and attempts to find alternate solutions, returning if it discovers a proof that does not involve any choice subgoals. If no such proof is found, the algorithm will present the list of placeholder proofs to the user, who can decide which one is most appropriate to pursue first. As an optimization, heuristics may be employed to sort or prune this list. As another optimization, the prover could determine whether a choice subgoal is worth pursing by attempting to complete the remainder of the proof before interacting with the user. This algorithm will identify a choice subgoal for every remote request made by previous approaches, and will additionally identify a choice subgoal for every locally creatable credential such that the creation of the credential would allow the completion of the proof from local knowledge.

We present our delayed distributed backward chaining algorithm ($\mathsf{bc\text{-}ask}_D$, shown in Figure 3) as a modification to the distributed backward chaining algorithm of our previous work [4]. For ease of presentation, $\mathsf{bc\text{-}ask}_D$ does not show additional parameters necessary to construct a proof term. In practice, the proof is constructed in parallel with the substitution that is returned by $\mathsf{bc\text{-}ask}_D$.

When we identify a choice subgoal, we construct a *marker* to store the parameters necessary to make the remote request. A marker differs from a choice subgoal in that a choice subgoal is a formula, while a marker has the same type as a proof.[2] This allows a marker to be included as a subproof in a larger proof. For ease of formal comparison with previous work, the algorithm we present here is only capable of creating markers for remote subgoals; we describe a trivial modification to allow it to handle locally creatable credentials.

[2] In Figure 3, a marker is typed as a substitution—this is because $\mathsf{bc\text{-}ask}_D$, as shown, does not construct proof terms. We will refer to a marker as having the same type as a proof to foster an intuition that is consistent with the implementation.

0	global set KB	/* knowledge base */
1	$\langle$substitution, credential[ ]$\rangle$ $\mathsf{bc\text{-}ask}_D($	
	list $goals$,	/* list of conjuncts forming a query */
	substitution θ,	/* current substitution, initially empty */
	set $failures)$	/* set of substitutions that are known not to produce a complete solution */
2	local set $failures'$	/* local copy of $failures$ */
3	if $(goals = [\,] \wedge \theta \in failures)$ then return $\perp$	/* θ known not to produce global solution */
4	if $(goals = [\,])$ then return $\langle \theta, [\,] \rangle$	/* base case, solution has been found */
5	$q' \leftarrow \mathsf{subst}(\theta, \mathsf{first}(goals))$	
6	$l \leftarrow \mathsf{determine\text{-}location}(q')$	
7	$failures' \leftarrow failures$	
8	if $(l \neq localmachine)$	/* if q' is the belief of a remote principal */
9	$m \leftarrow \mathsf{constructMarker}(\mathsf{first}(goals), \theta, failures')$	
10	if $\neg(m \in failures')$, return $\langle m, [\,] \rangle$	
11	foreach $(P, q) \in KB$	/* investigate each fact, tactic */
12	if $((\theta' \leftarrow \mathsf{unify}(q, q')) \neq \perp)$	/* determine if tactic matches first goal */
13	$\phi \leftarrow \mathsf{compose}(\theta', \theta)$	
14	if $(P = [\,])$	/* if (P, q) represents a credential */
15	if $\neg(\phi \in failures')$	
16	$failures' \leftarrow \phi \cup failures'$	
17	$\langle answer, creds \rangle \leftarrow \mathsf{bc\text{-}ask}_D(\mathsf{rest}(goals), \phi, failures)$	/* prove remainder of goals */
18	if$(answer \neq \perp)$ return $\langle answer, [creds \mid q] \rangle$	/* append q to $creds$ and return */
19	else while $((\langle \beta, creds \rangle \leftarrow \mathsf{bc\text{-}ask}_D(P, \phi, failures')) \neq \perp)$	/* prove subgoals */
20	$failures' \leftarrow \beta \cup failures'$	/* prevent β from being returned again */
21	if $(\mathsf{isRemoteMarker}(\beta))$, return $\langle \beta, [\,] \rangle$	
22	$\langle answer, creds \rangle \leftarrow \mathsf{bc\text{-}ask}_D(\mathsf{rest}(goals), \beta, failures)$	/* prove remainder of goals */
23	if $(answer \neq \perp)$ then return $\langle answer, creds \rangle$	/* if answer found, return it */
24	return $\langle \perp, [\,] \rangle$	/* if no proof found, return failure */

Figure 3: $\mathsf{bc\text{-}ask}_D$, a delayed version of bc-ask

$\mathsf{bc\text{-}ask}_D$ operates by recursively decomposing the original goal into subgoals. The base case occurs when $\mathsf{bc\text{-}ask}_D$ is invoked with an empty goal, in which case $\mathsf{bc\text{-}ask}_D$ will determine if the current solution has been previously marked as a failure and return appropriately (Lines 3–4) of Figure 3. For non-empty goals, $\mathsf{bc\text{-}ask}_D$ will determine if the formula represents the beliefs of a remote principal using the determine-location subroutine (Line 6), which returns either the constant $localmachine$ or the address of the remote principal. If the formula is remote, rather than making a remote procedure call, we create a marker and return (Lines 8–10).

If the formula is local or the choice subgoal represented by the remote marker was previously investigated (indicated by the marker's presence in $failures$), $\mathsf{bc\text{-}ask}_D$ will attempt to prove the goal either directly from a credential, or via the application of a tactic to the goal (Lines 11–12). $\mathsf{bc\text{-}ask}_D$ handles the case where the goal is directly provable from a credential (Lines 14–18) separately from the case where it is not (Lines 19–23) only to allow the credential to be appended to the return tuple (Line 18). When the goal is directly provable from a credential, $\mathsf{bc\text{-}ask}_D$ first performs the same function as the base case without recursing, then attempts to prove the remainder of the goals. If $\mathsf{bc\text{-}ask}_D$ applied a tactic, it attempts to recursively prove the premises of that tactic (Line 19). If this results in a remote marker, $\mathsf{bc\text{-}ask}_D$ suspends proving and returns (Line 21). Otherwise, $\mathsf{bc\text{-}ask}_D$ attempts to recursively prove the remainder of the goals (Lines 22–23).

Locally creatable credentials We can extend $\mathsf{bc\text{-}ask}_D$ to support the creation of local credentials by the addition of a section of code prior to Line 11 that, similarly to Lines 8–10, creates a marker when the first goal is a locally creatable credential. Line 21 suspends proving only for remote markers, so $\mathsf{bc\text{-}ask}_D$ will attempt to complete the proof to determine if creating the credential will lead to a complete proof.

Suspending proving for remote markers For theoretical completeness, $\mathsf{bc\text{-}ask}_D$ must suspend proving after the addition of a remote marker beause the response to a remote request may add credentials to the local knowledge base, which in turn may increase the ability of $\mathsf{bc\text{-}ask}_D$ to prove of the remainder of the proof. Rather than enumerating all possible ways in which the remainder of the proof might be proved, we simply suspend the investigation of this branch of the proof once a remote marker has been created. If the user decides to make a remote request then, upon receipt of the response to this request, we will add any new credentials to the knowledge base and re-run the original query. In this manner, multiple *rounds* of proving can be used to find proofs that involve more than one choice subgoal.

In practice, we expect that proofs requiring multiple rounds will be encountered infrequently—in fact, they have not arisen in our deployment. Based on the assumption that multiple rounds are seldom necessary, we introduce an optimization in which the prover, after adding a marker, attempts to complete the remainder of the proof from local knowledge. In this manner, we can bias the user's decision towards choices that produce a complete proof in a single round and away from choices that may never lead to a complete solution.

6.1.1 Completeness of delayed backward chaining

A delayed backward chaining prover offers strictly greater proving ability than an inline backward chaining prover. This is stated more formally below; the proof is in Appendix B.

Theorem 3 *For any goal G, a delayed distributed prover with local knowledge base KB will find a proof of G if an inline distributed prover using KB will find a proof of G.*

Informally: We first show that in the absence of any markers, the delayed prover will find a proof if the inline prover finds a proof. We then extend this result to allow remote markers, but under the assumption that any remote request made by the delayed prover will produce the same answer as an identical remote request made by the inline prover. Finally, we relax this assumption to show that the delayed prover has strictly greater proving ability than the inline prover.

6.2 Tactics

In constructing a set of tactics to be used by our backward-chaining engine, we have two goals: the tactics should make use of facts precomputed by FC and PC, and they should be generated systematically from the inference rules of the logic.

If a formula F can be proved from local credentials, and all locally known credentials have been incrementally added via FC, then, by Theorem 1, a proof of F already exists in the knowledge base. In this case, the backward-chaining component of our prover need only look in the knowledge base to find the proof.

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

left tactic

prove(A **says** F) :-
    pathLookup(B **says** F, A **says** F),
    prove(B **says** F).

right tactic

prove(A **says** F) :-
    proveWithChoiceSubgoal(A **says** (B **speaksfor** A)),
    factLookup(B **says** F).

Figure 4: Example construction of LR tactics from an inference rule

Tactics are thus used only when F is not provable from local knowledge, and in that case their role is to identify choice subgoals to present to the user.

Since the inference rules that describe delegation are the ones that indirectly give rise to the paths precomputed by PC, we need to treat those specially when generating tactics; all other inference rules are imported as tactics directly. We discuss here only delegation rules with two premises; for further discussion see Section 8.

Inference rules about delegation typically have two premises: one that describes a delegation, and another that allows the delegated permission to be exercised. Since tactics are applied only when the goal is not provable from local knowledge, one of the premises must contain a choice subgoal. For each delegation rule, we construct two tactics: (1) a *left* tactic for the case when the choice subgoal is in the delegation premise, and (2) a *right* tactic for the case when the choice subgoal is in the other premise.[3] We call tactics generated in this manner LR tactics.

The insight behind the left tactic is that instead of looking for complete proofs of the delegation premise in the set of facts in the knowledge base, it looks for proofs among the paths precomputed by PC, thus following an arbitrarily long delegation chain in one step. The premise exercising the delegation is then proved normally, by recursively applying tactics to find any remaining choice subgoals. Conversely, the right tactic assumes that the delegation premise can be proved only with the use of a choice subgoal, and restricts the search to only those proofs. The right tactic may then look in the knowledge base for a proof of the right premise in an effort to determine if the choice subgoal is useful to pursue.

Figure 4 shows an inference rule and the two tactics we construct from that rule. All tactics are constructed as *prove* predicates, and so a recursive call to *prove* may apply tactics other than the two shown. The *factLookup* and *pathLookup* predicates inspect the knowledge base for facts produced by FC and paths produced by PC. The *proveWithChoiceSubgoal* acts like a standard *prove* predicate, but restricts the search to discard any proofs that do not involve a choice subgoal. We employ rudimentary cycle detection to prevent repeated application of the same right rule. In practice, each of these restrictions can be accomplished through additional parameters to bc-ask_D that restrict what types of proof may be returned.

Optimizations to LR The dominant computational cost of running a query using LR tactics is repeated applications of right tactics. Since a right tactic handles the case in which the choice subgoal represents a delegation, identifying the choice subgoal involves determining who is allowed to create delegations, and then determining on whose behalf that person wishes to delegate. This involves exhaustively searching through all paths twice. However, practical experience with our deployed system indicates that people rarely delegate on behalf of anyone other than themselves. This allows us to remove the second path application

[3]For completeness, if there are choice subgoals in both premises, one will be resolved and then the prover will be rerun. In practice, we have yet to encounter a circumstance where a single round of proving was not sufficient.

and trade completeness for speed in finding the most common proofs. If completeness is desired, the optimized set of tactics could be run first, and the complete version could be run afterwards. We refer to the optimized tactics as LR'. This type of optimization is made dramatically easier because of the systematic approach used to construct the LR tactics.

Alternative approaches to caching Naive constructions of tactics perform a large amount of redundant computation both within a query and across queries. An apparent solution to this problem is to cache intermediate results as they are discovered to avoid future recomputation. As it turns out, this type of caching does not improve performance, and even worsens it in some situations. If attempting to prove a formula with an unbound variable, an exhaustive search requires that all bindings for that variable be investigated. Cached proofs will be used first, but as the cache is not necessarily all-inclusive, tactics must be applied as well. These tactics in turn will re-derive the proofs that are in cache. Another approach is to make caching part of the proving engine (e.g., Prolog) itself. Tabling algorithms [12] provide this and other useful properties, and have well-established implementations (e.g., `http://xsb.sourceforge.net/`). However, this approach precludes adding to cache proofs that are discovered via different proving techniques (e.g., FC, PC, or a remote prover using a different set of tactics).

6.2.1 Completeness of LR

Despite greater efficiency, LR tactics have strictly greater proving ability than the depth-limited inference rules. We state this formally below; the proof is in Appendix C.

Theorem 4 *Given one prover whose tactics are depth-limited inference rules (*IR*), and a second prover that uses* LR *tactics along with* FC *and* PC*, if the prover using* IR *tactics finds a proof of goal F, the prover using* LR *tactics will also find a proof of F.*

Informally: We first show show that provers using LR and IR are locally equivalent—that is, if IR finds a complete proof from local knowledge then LR will do so as well and if IR identifies a choice subgoal then LR will identify the same choice subgoal. We show this by first noting that if IR finds a complete proof from local knowledge, then a prover using LR will have precomputed that same proof using FC. We show that LR and IR find the same choice subgoals by induction over the size of the proof explored by IR and noting that left tactics handle the case where the proof of the right premise of an inference rule contains a choice subgoal and that right tactics handle the case where the the left premise contains a choice subgoal. Having shown local equivalence, we can apply induction over the number of remote requests made to conclude that a prover using LR will find a proof of F if a prover using IR finds a proof of F.

7 Empirical Evaluation

Since the usability of the distributed access-control system as a whole depends on the timeliness with which it can generate a proof of access, the most important evaluation metric is the amount of time it takes either to construct a complete proof, or, if no complete proof can be found, to generate a list of choices to give to the user. We also consider the number of subgoals investigated by the prover and the size of the knowledge base produced by FC and PC. The number of subgoals investigated represents a coarse measure of efficiency that is independent of any particular Prolog implementation.

We compare the performance of five proving strategies: three that represent previous work and two (the combination of FC and PC with either LR or LR') that represent the strategies introduced here. The strategies that represent previous work are backward chaining with depth-limited inference rules (IR), inference rules with basic cycle detection (IR-NC), and hand-crafted tactics (HC). HC evolved from IR during our early deployment as an effort to improve the efficiency of the proof-generation process. As such, HC represents our best effort to optimize a prover that uses only backward chaining to the policies used in our deployment, but at the cost of theoretical completeness.

We analyze two scenarios: the first represents the running example presented previously (which is drawn from our deployment), and the second represents the policy described by our previous work [4], which is indicative of a larger deployment. As explained in Section 7.2, these large policies are the most challenging for our strategy.

Our system is built using Java Mobile Edition (J2ME), and the prover is written in Prolog. We perform simulations on two devices: a Nokia N70 smartphone, which is the device used in our deployment, and a dual 2.8 Ghz Xeon workstation with 1 GB of memory. Our Prolog interpreter for the N70 is JIProlog (`http://www.ugosweb.com/jiprolog/`) due to its compatibility with J2ME. Simulations run on the workstation use SWI-Prolog (`http://www.swi-prolog.org/`).

7.1 Running Example

Scenario As per our running example, Alice controls access to a machine room. We simulate a scenario in which Charlie wishes to enter the machine room for the first time. To do so, his prover will be asked to generate a proof of Dept **says** **open**(door1). His prover will immediately realize that Dept should be asked for help, but will continue to reason about this formula using local knowledge in the hope of finding a proof without making a request. Lacking sufficient authority, this local reasoning will fail, and Charlie will be presented with the option to ask Dept for help. Preferring not to bother the department head, Charlie will decide to ask his manager, Alice, directly.

Creating a complete proof in this scenario requires three steps: (1) Charlie's prover attempts to construct a proof, realizes that help is necessary, and asks Alice, (2) Alice's phone constructs a proof containing a delegation to Charlie, and (3) Charlie assembles Alice's response into a final proof. As Alice's phone holds the most complicated policy, step 2 dominates the total time required to find a proof.

0 K_{Dept} **signed** (**delegate**(Dept,Alice,door1))
1 K_{Dept} **signed** (**delegate**(Dept,Alice,door2))
2 K_{Dept} **signed** (**delegate**(Dept,Alice,door3))
3 K_{Alice} **signed** (**delegate**(Alice, Alice.machine-room, door1))
4 K_{Alice} **signed** (**delegate**(Alice, Alice.machine-room, door2))
5 K_{Alice} **signed** (**delegate**(Alice, Alice.machine-room, door3))
6 K_{Alice} **signed** (Bob**speaksfor**Alice.machine-room)
7 K_{Alice} **signed** (David**speaksfor**Alice.machine-room)
8 K_{Alice} **signed** (Elizabeth**speaksfor**Alice.machine-room)

9 K_{Dept} **signed** (**delegate**(Dept,Alice,office))
10 K_{Dept} **signed** (**delegate**(Dept,Dept.residents,lab-door))
11 K_{Dept} **signed** (Alice**speaksfor**Dept.residents)

12 K_{Charlie} **signed** **open**(door1)

Figure 5: Credentials on Alice's phone

13 K_{Dept} **signed** (**delegate**(Dept,Dept.residents,lab-door))
14 K_{Dept} **signed** (Charlie**speaksfor**Dept.residents)
15 K_{Charlie} **signed** **open**(door1)

Figure 6: Credentials on Charlie's phone

Policy The policy for this scenario is expressed in the credentials known to Alice and Charlie, shown in Figures 5 and 6. The first six credentials of Figure 5 represent the delegation of access to the machine-room doors from the department to Alice, and her redelegation of these resources to the group Alice.machine-room. Credentials 6–8 indicate that the group Alice.machine-room already includes Bob, David, and Elizabeth. Notably, Alice has not yet created a credential that would give Charlie access to the machine room. We will analyze the policy as is, and with the addition of a credential that adds Charlie to the machine-room group. Credentials 9–11 deal with other resources that Alice can access. The final credential is given to Alice when Charlie asks her for help: it indicates Charlie's desire to open door1.

Charlie's policy (Figure 6) is much simpler. He has access to a shared lab space through his membership in the group Dept.residents, to which the department has delegated access. He has no credentials pertaining to the machine room.

The only credential in Figures 5 and 6 that was created at the time of access is the one indicating Charlie's desire to access door1. This means that FC and PC have already been run on all other credentials.

Performance Figure 7 describes the proving performance experienced by Alice when she attempts to help Charlie. Alice wishes to delegate authority to Charlie by making him a member of the Alice.machine-room group. We show performance for the case where this credential does not yet exist, and the case where it does. In both cases, Alice's phone is unable to complete a proof with either IR or IR-NC as both crash due to lack of memory after a significant amount of computation. To demonstrate the relative performance of IR and IR-NC, Figure 7 includes (on a separate y-axis) results collected on a workstation. IR, IR-NC, and HC were run with a depth-limit of 7, chosen high enough to find all solutions on this policy.

In the scenario where Alice has not yet delegated authority to Charlie, HC is over six times slower than LR, and more than two orders of magnitude slower than LR′. If Alice has already added Charlie to the group, the difference in performance widens. Since FC finds all complete proofs, it finds the proof while processing the credentials supplied by Charlie, so the subsequent search by LR and LR′ is a cache lookup. The result is that a proof is found by LR and LR′ almost 60 times faster than HC. When run on the workstation, IR and IR-NC are substantially slower than even HC.

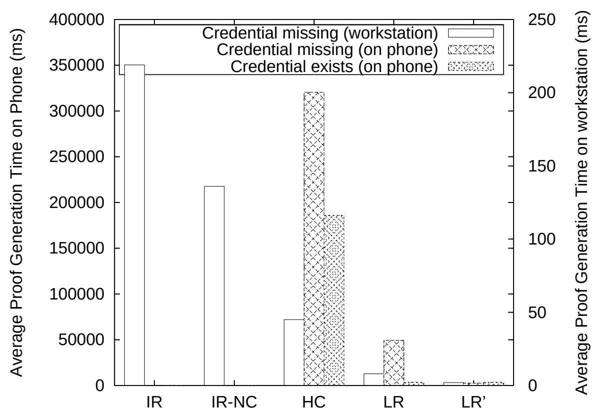

Figure 7: Alice's prover generates complete proof or list of credentials that Alice can create

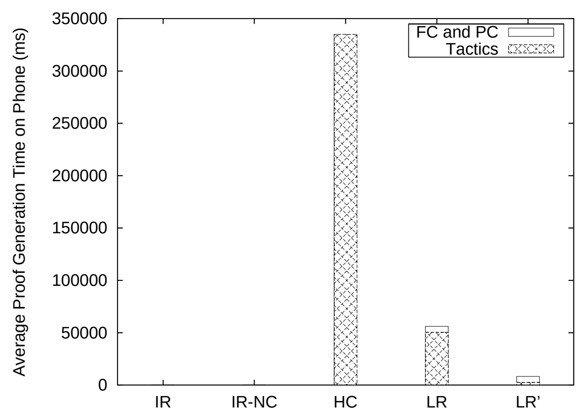

Figure 8: Aggregate proving time: Charlie's before help request + Alice's + Charlie's after help request

Figure 8 shows the total time required to generate a proof of access in the scenario where Alice must reactively create the delegation credential (IR and IR-NC are omitted as they crash). This consists of Charlie's initial attempt to generate a proof, Alice's proof generation that leads to the creation of a new credential, and Charlie assembling Alice's reply into a final proof. The graph also shows the division of computation between the incremental algorithms FC and PC and the backward search using tactics. In overall computation, HC is six times slower than LR and 60 times slower than LR′. This does not include the transit time between phones, or the time spent waiting for users to choose between different options.

Since computation time is dependent on the Prolog implementation, as a more general metric of efficiency we also measure the number of formulas investigated by each strategy. Figure 9 shows the total number of formulas investigated (including redundant computation) and the number of unique formulas investigated (note that each is measured on a separate y-axis). LR and LR′ not only investigate fewer unique formulas than previous approaches, but drastically reduce the amount of redundant computation.

7.2 Large Policies

Although our policy is a real one used in practice, in a widespread deployment it is likely that policies will become more complicated, with users having credentials for dozens of resources spanning multiple organizations. Our primary metric of evaluation is proof-generation time. Since backward chaining only considers branches, and hence credentials, that are relevant to the proof at hand, it will be least efficient when all credentials must be considered, e.g., when they are generated by members of same organization. As a secondary metric, we evaluate the size of the knowledge base, as this directly affects the memory requirements of the application as well as the speed of unification. Since credentials from the same organization are more likely to be combined to produce a new fact or path, the largest knowledge base will occur when all credentials pertain to the same organization. In this section, we evaluate a policy where all credentials pertain to the same organization as it represents the worst case for both metrics.

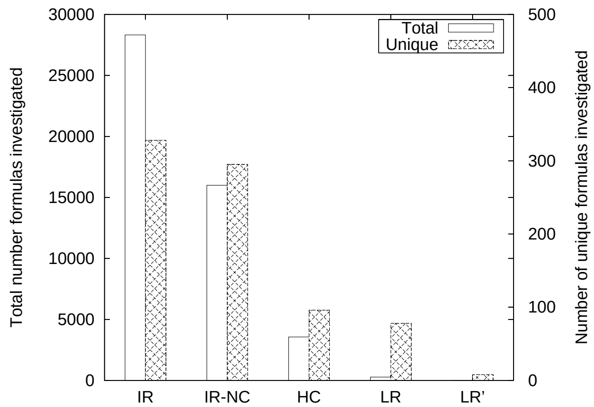

Figure 9: Number of formulas investigated by Alice

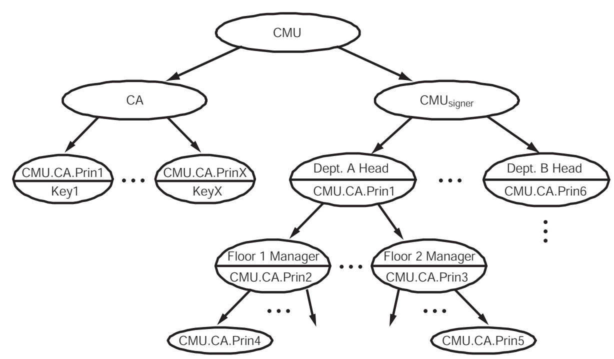

Figure 10: An access-control policy presented in our previous work [4]

Policy We evaluate our work with respect to the policy presented in our previous work [4]. This policy represents a university-wide deployment. In addition to its larger size, this policy has a more complex structure than the policy described in Section 7.1. For example, the university maintains a certification authority (CA) that binds names to public keys, thus allowing authority to be delegated to a principal's name. Furthermore, many delegations are made to roles (e.g., Dept.Manager1), to which principals are assigned using additional credentials.

We simulate the performance of our approach on this policy from the standpoint of a principal who has access to a resource via a chain of three delegations (assembled from 10 credentials), and wants to extend this authority to a subordinate.

Performance Figure 11 shows the proof-generation time of the different strategies for different numbers of subordinates on the workstation. For these policies, the depth limit used by IR, IR-NC, and HC must be 10 or greater. However, IR crashed at any depth limit higher than 7, and is therefore not included in these simulations. Simulations on this policy used a depth-limit of 10. IR-NC displays the worst performance on the first three policy sizes, and exhausts available memory and crashes for the two largest policies. HC appears to outperform LR, but, as the legend indicates, was unable to find 11 out of the 14 possible solutions, including several likely completions, the most notable of which is the desired completion Alice **says** (Charlie **speaksfor** Alice. machine-room). This completion is included in the subset of common solutions that LR′ is looking for. This subset constitutes 43% of the total solution space, and LR′ finds all solutions in this subset several orders of magnitude faster than any other strategy.

The size of the knowledge base for each policy is shown in Figure 12. The knowledge base consists of certificates and, under LR and LR′, facts and paths precomputed by FC and PC. We observe that many credentials from the same policy cannot be combined with each other, yielding a knowledge base whose size is approximately linear with respect to the number of credentials.

In summary, the two previous, theoretically complete approaches (IR and IR-NC) are unable to scale to the larger policies. HC, tailored to run on a particular policy, is unable to find a significant number of solutions when used on larger policies. LR is able to scale to larger policies while offering theoretical completeness guarantees. LR′, which is restricted to finding a common subset of solutions, finds all of those solutions dramatically faster than any other approach.

8 Generality of Our Approach

Although we described and evaluated our technique with respect to a particular access-control logic and system, it can be applied to others, as well. There are three aspects of generality to consider: supporting the logical constructs used by other logics, performing efficiently in the context of different systems, and enabling other applications.

Other logics When applying our approach to other logics, we must consider individually the applicability of each component of our approach: FC, PC, and the generation of LR tactics. We consider our technique with respect to only monotonic authorization logics, i.e., logics where a formula remains provable when given more credentials. This constraint is commonly used in practical systems (cf., [10]).

As discussed previously, to ensure that the forward-chaining component of our prover terminates, the logic on which it is operating should be a subset of Datalog, or, if function symbols are allowed, their use must be constrained (as described in Section 4). This is sufficient to express most access-control logics, e.g., the logics of SD3 [21], Cassandra [6], and Binder [13], but is not sufficient to express higher-order

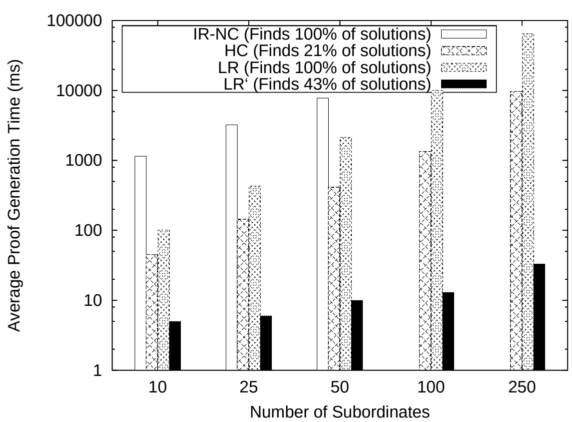

Figure 11: Proof generation in larger policies with missing credential

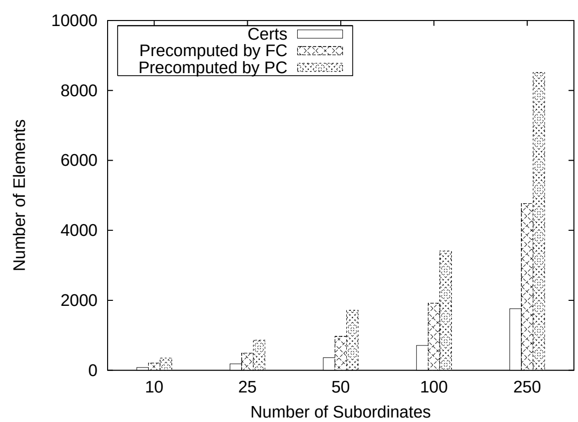

Figure 12: Size of knowledge base in larger policies with missing credential

logic, and, as such, we cannot fully express the access-control logic presented by Appel and Felten [2]. The general notion of delegation introduced in Definition 1 is conceptually very similar to that of the various logics that encode SPKI [1, 27, 19], the RT family of logics [28], Binder [13], Placeless Documents [3], and the domain-name service logic of SD3 [21], and so our technique should apply to these logics as well.

Our path-compression algorithm and our method for generating LR tactics assume that any delegation rule has exactly two premises. Several of the logics mentioned above (e.g., [21, 13, 3]) have rules involving three premises; however, initial investigation suggests that any multi-premise rule may be rewritten as a collection of two-premise rules.

Path compression requires a decidable algorithm for computing the intersection of two permissions. That is, when combining the paths $(\mathsf{Alice}\ \mathbf{says}\ F, \mathsf{Bob}\ \mathbf{says}\ F)$ and $(\mathsf{Bob}\ \mathbf{says}\ \mathbf{open}(\mathsf{door1}), \mathsf{Charlie}\ \mathbf{says}\ \mathbf{open}(\mathsf{door1}))$, we need to determine the intersection of F and $\mathbf{open}(\mathsf{door1})$ for the resulting path. For our logic, computing the permission is trivial, since in the most complicated case we unify an uninstantiated formula F with a fully instantiated formula, e.g., $\mathbf{open}(\mathsf{door1})$. In some cases, a different algorithm may be appropriate: for SPKI, for example, the algorithm is a type of string intersection [14].

Other systems Our strategies should be of most benefit in systems where (a) credentials can be created dynamically, (b) credentials are distributed among many parties, (c) long delegation chains exist, and (d) credentials are frequently reused. Delayed backward chaining pursues fewer expensive subgoals, thus improving performance in systems with properties (a) and (b). Long delegation chains (c) can be effectively compressed using either FC (if the result of the compression can be expressed directly in the logic) or PC (when the result cannot be expressed in the logic). FC and PC extend the knowledge base with the results of their computation, thus allowing efficient reuse of the results (d).

These four properties are not unique to our system, and so we expect our technique, or the insights it embodies, will be useful elsewhere. For example, Greenpass [17] allows users to dynamically create credentials. Properties (b) and (c) have been the focus of considerable previous work, notably SPKI [1, 27, 19], the DNS logic of SD3 [21], RT [28], and Cassandra [6]. Finally, we feel that (d) is common to the vast majority of access-control systems, as a statement of delegation is typically intended to be reused.

Other applications There are situations beyond our smartphone-oriented setting when it is necessary to efficiently compute similar proofs and where the efficiency offered by our approach is welcome or necessary. For example, user studies conducted at our institution indicated that, independently of the technology used to implement an access-control system, users strongly desired an auditing and credential-creation tool that would allow them to better understand the indirect effects on policy of creating new credentials by giving them real-time feedback as they experimented with hypothetical credentials. If Alice wants to create a new credential $K_{\mathsf{Alice}}\ \mathbf{signed}\ \mathbf{delegate}(\mathsf{Alice}, \mathsf{Alice.machine\text{-}room}, \mathsf{door4})$, running this hypothetical credential through the path-compression algorithm could inform Alice that an effect of the new credential is that Bob now has access to door4 (i.e., that a path for door4 was created from Bob to Alice). Accomplishing an equivalent objective using IR or IR-NC would involve assuming that everyone is willing to access every resource, and attempting to prove access to every resource in the system—a very inefficient process.

9 Related Work

Many distributed access-control systems model access-control decisions in a formal logic. Section 8 briefly discusses the expressiveness of various logics; here we focus on the mechanisms for making access-control decisions in a distributed system. These mechanisms fall, roughly speaking, into two categories: remote credential retrieval and distributed reasoning. We briefly describe previous work in this context; however, we are aware of no previous algorithm that meets all of our requirements detailed in Section 2.1, and no works that analyze the performance of the distributed proving alternatives we consider.

Remote credential retrieval Several existing distributed access-control systems support distributed knowledge bases by retrieving credentials from remote parties when needed; we highlight a few here. Binder [13], PolicyMaker [9], and KeyNote [10, 8], provide general languages for defining access-control policy in a distributed scenario. Signed credentials allow the policy to be transported between nodes, but the mechanisms for accomplishing this are left unspecified. Placeless Documents defines a logic for enforcing access-control for distributed Java applications [3], but does not specify how remote credentials are to be retrieved. SPKI is a syntax for digital certificates dealing with authentication [14] that has been modeled in formal logic [1, 27, 19] and used to implement access control for web pages [30] and wireless networks [17] in a manner that includes provisions for collecting remote credentials for these particular scenarios. RT is a language for defining trust where the provided search algorithm for evaluating access-control decisions is capable of retrieving remote credentials [29]. SD3 is a distributed trust-management framework that extends the evaluation techniques of QCM [18, 22] and utilizes both a push and a pull model for migrating credentials between parties [21]. Bauer et al. use the proof-carrying authorization framework [2] to guard access to web content; the client's theorem prover will retrieve remote credentials in the course of proof-generation as needed [5]. Cassandra provides a general framework for specifying distributed multi-domain policies and supports remote credential retrieval from locations specified in the credential [6]. The Strongman architecture specifies policy as KeyNote programs; credentials are stored on designated repositories [24]. Zhang et al. propose a usage-control authorization system for collaborative applications [35]. In their system, access-control decisions are made continuously based on credentials from distant sources. They investigate various mechanisms for efficiently retrieving needed credentials, but the process that reasons about these credentials runs on a single node. Our work differs from these in that, instead of collecting individual credentials from remote parties, our approach involves these parties in the reasoning process itself.

Distributed reasoning The second category of systems supports distributed reasoning about access-control policies. This approach allows the remote retrieval of subproofs rather than credentials, which previous work showed substantially reduces the number of remote requests necessary to complete a proof [4]. Minami et al. use this approach for context-sensitive authorization queries [31], and extend it to provide distributed cache consistency in the face of certificate revocation [32]. Alpaca supports the assembly of a proof from a collection of subproofs constructed by distant parties, but does not specify how these proofs are to be collected [26]. We improve upon these works by providing new techniques that improve the efficiency of the proof-generation process in light of the requirements of Section 2.1.

Other related work Several other systems incorporate ideas relevant to our work. PeerAccess provides a framework for describing policies that govern interactions between principals [34]. PeerAccess uses proof hints encoded in the knowledge base to guide whom a principal asks when local knowledge is insufficient to construct a proof. In Cassandra, credentials may explicitly encode their location. In contrast to PeerAccess and Cassandra, our approach seeks to leverage the user's intuition when directing remote queries. Know is a system for providing user feedback when access is denied; meta-policies are used to govern what information about policies is revealed to whom [23]. Know is complementary to our work, as its primary concern is providing the user with information *after* an access-control decision has been made.

Greenpass engages users to gain consent for granting delegations [17], but a user may create only a more limited form of delegation than we consider here. This eliminates the need to explore the many hypothetical possibilities that our method explores. Jajodia et al. propose a general framework for describing partially specified hierarchical access-control policies [20]. To allow quick access-control decisions, a graph of permissions is computed a priori. However, their framework operates on a centralized policy managed by a single entity, and does not allow new permissions to be created in response to a query. Bertino et al. propose Trust-X to negotiate trust between peers [7]. As negotiation is expensive, they seek to bypass it using two optimizations: trust tickets and sequence prediction. Trust tickets are new credentials that contain the result of previous negotiations. Sequence prediction uses past negotiations to suggest credentials that will establish trust without going through the negotiation process. Our precomputation techniques are complementary as they are aimed at improving the efficiency of the expensive step in cases where it is unavoidable.

10 Conclusion

In this paper we presented a new approach to generating proofs that accesses comply with access-control policy. Our strategy is targeted for environments in which credentials must be collected from distributed components, perhaps only after users of those components consent to their creation, and our design is informed by such a testbed we have deployed and actively use at our institution. Our technique embodies three contributions, namely: novel approaches for minimizing proof steps that involve remote queries or user interaction; methods for inferring delegation chains off the critical path of accesses that significantly optimize proving at the time of access; and a systematic approach to generating tactics that yield efficient backward chaining. We demonstrated analytically that the proving ability of this technique is strictly superior to previous work, and demonstrated empirically that it is efficient on policies drawn from our deployment and will scale effectively to larger policies. Our method will generalize to other security logics that exhibit the common properties detailed in Section 8.

References

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

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

[3] 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.

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

[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*, 2002.

[6] M. Becker and P. Sewell. Cassandra: Flexible trust management, applied to electronic health records. In *Proceedings of the 17th IEEE Computer Security Foundations Workshop*, 2004.

[7] E. Bertino, E. Ferrari, and A. C. Squicciarini. Trust-X: a Peer to Peer Framework for Trust Establishment. *IEEE Transactions on Knowledge and Data Engineering*, 16(7):827–842, 2004.

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

[9] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *Proceedings of the 1996 IEEE Symposium on Security & Privacy*, 1996.

[10] 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*, 1998.

[11] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *ACM Transactions on Computer Systems*, 8(1):18–36, 1990.

[12] W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. *Journal of the ACM*, 43(1):20–74, 1996.

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

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

[15] A. Felty. Implementing tactics and tacticals in a higher-order logic programming language. *Journal of Automated Reasoning*, 11(1):43–81, 1993.

[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] N. C. Goffee, S. H. Kim, S. Smith, P. Taylor, M. Zhao, and J. Marchesini. Greenpass: Decentralized, PKI-based authorization for wireless LANs. In *Proceedings of the 3rd Annual PKI Research and Development Workshop*, 2004.

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

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

[20] S. Jajodia, P. Samarati, M. Sapino, and V. S. Subrahmanian. Flexible Support for Multiple Access Control Policies. *ACM Transactions on Database Systems*, 26(2):214–260, 2001.

[21] T. Jim. SD3: A trust management system with certified evaluation. In *Proceedings of the 2001 IEEE Symposium on Security & Privacy*, 2001.

[22] T. Jim and D. Suciu. Dynamically distributed query evaluation. In ACM, editor, *Proceedings of the Twentieth ACM Symposium on Principles of Database Systems: PODS 2001: Santa Barbara, California, May 2001*, pages 28–39, 2001.

[23] A. Kapadia, G. Sampemane, and R. H. Campbell. Know why your access was denied: Regulated feedback for usable security. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, 2004.

[24] A. D. Keromytis, S. Ioannidis, M. B. Greenwald, and J. M. Smith. The STRONGMAN architecture. In *Third DARPA Information Survivability Conference and Exposition*, 2003.

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

[26] C. Lesniewski-Laas, M. F. Kaashoek, and R. Morris. Alpaca, a proof-carrying authentication framework for cryptographic primitives and protocols. Manuscript, October 2006.

[27] N. Li and J. C. Mitchell. Understanding SPKI/SDSI using first-order logic. *International Journal of Information Security*, 2004.

[28] 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.

[29] N. Li, W. H. Winsborough, and J. C. Mitchell. Distributed credential chain discovery in trust management. *Journal of Computer Security*, 11(1):35–86, 2003.

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

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

[32] K. Minami and D. Kotz. Scalability in a secure distributed proof system. In *Proceedings of the Fourth International Conference on Pervasive Computing*, 2006.

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

[34] M. Winslett, C. C. Zhang, and P. A. Bonatti. PeerAccess: A logic for distributed authorization. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, 2005.

[35] X. Zhang, M. Nakae, M. J. Covington, and R. Sandhu. A Usage-Based Authorization Framework for Collaborative Computing Systems. In *Proceedings of the 11th ACM Symposium on Access Control Models and Technologies (SACMAT)*.

A Completeness of PC

At a high level, we first show that add-path will combine all paths that can be combined—that is, for any paths (p, q) and (p', q') if q unifies with p' then the path (p, q') will have been added. As mentioned in Section 5, union is defined to prevent cyclic paths (i.e., (p, p)) from ever being added. We then proceed to show that for all credentials that represent a path, all independent credentials are added, and all credentials that depend on the existence of another path are added whenever that path becomes known.

For the purpose of clarity, within the scope of this Appendix, if $\mathsf{unify}(a, b) \neq \perp$ then we will write $a = b$ and use a and b interchangeably. We let $[\mathrm{k}p]$ represent line k in PC, and $[\mathrm{k}a]$ represent line k in add-path.

Lemma 1 *If $paths$ and $incompletePaths$ are initially empty, and* add-path *is invoked repeatedly with a series of inputs, then after any invocation of* add-path *completes, the following holds:* $\forall(x, y) \in paths, \forall(x', y') \in paths, (\mathsf{unify}(y', x) \neq \perp) \supset (x', y) \in paths.$

Proof We prove Lemma 1 by induction over the number of calls i that have been made to add-path, where a call corresponds to the addition of a single credential. Our induction hypothesis states that condition $\mathcal{C}$ holds after $i - 1$ calls, where $\mathcal{C}$ is $(\forall(x, y) \in paths, \forall(x', y') \in paths, (\mathsf{unify}(y', x) \neq \perp) \supset (x', y) \in paths)$.
Base case: The base case occurs when $i = 1$. Prior to this call, $paths$ is empty. Since $[17a]$ will add (p, q) to $paths$ and $\mathsf{unify}(p, q) = \perp$ (or else credToPath would have returned $\perp$ on $[3p]$) the if statements on $[20a]$ and $[26a]$ will fail and the function will exit with $paths$ containing the single element (p, q). Since $paths$ will contain a single element, the induction hypothesis holds.
Inductive case: For the inductive case, we must show that $\mathcal{C}$ holds after the ith call to add-path. We prove the inductive case by first making the following observations:
Observation 1: By the time $[24a]$ is reached, if (p, q) is the parameter to add-path ($[15a]$), then $paths$ and $newPaths$ will contain every path (p, q') such that $\mathsf{unify}(q, p') \neq \perp$ and $(p', q') \in paths$.
Observation 2: When add-path terminates, $paths$ will contain every (p', q'') such that $(p', q') \in paths$, $(p'', q'') \in newPaths$, and $\mathsf{unify}(q', p'') \neq \perp$.
Observation 3: If an invocation of add-path, when given input parameter (p, q), adds the path (x, y), then, ignoring the mechanics of add-path, (x, y) must be characterized by one of the following:

1. $(x, y) = (p, q)$
2. $x = p$ and $(q, y) \in paths$ prior to invocation i
3. $(x, p) \in paths$ prior to invocation i and $y = q$
4. $(x, p) \in paths$ prior to invocation i and $(q, y) \in paths$ prior to invocation i

In situation 2, (x, y) represents the addition of an existing path to one end of (p, q). Situation 3 is simply the opposite of 2. In situation 4, (x, y) represents the addition of existing path to both ends of (p, q).

We must show that $\mathcal{C}$ holds in the following cases, which describe all possible combinations of (x, y) and (x', y') prior to invocation i:

1. $(x, y) \in paths, (x', y') \in paths$;
2. $(x', y') \in paths$ but $(x, y) \notin paths$ (or, conversely, that $(x, y) \in paths$ but $(x', y') \notin paths$);
3. $(x, y) \notin paths$ and $(x', y') \notin paths$.

Case 1: Lemma 1 assumes that $y' = x$. From this and the definition of Case 1, our induction hypothesis tells us that $(x', y) \in paths$ prior to invocation i. Since $\mathsf{add\text{-}path}$ does not remove elements from $paths$, $(x', y) \in paths$ after invocation i, and so $\mathcal{C}$ holds.

Case 2 From the definition of Case 2, we know that (x, y) is added during the ith invocation of $\mathsf{add\text{-}path}$. This implies that (x, y) must have been added at one of the following locations (with the situation that led to the addition in parenthesis):

a. $[17a]$ ((1) of Observation 3)

b. $[22a]$ ((2) of Observation 3)

c. $[28a]$ ((3) **or** (4) of Observation 3)

We note that the most complex scenario is the second possibility for subcase c (4). We prove only the second possibility for subcase c, and note that subcases a, b, and the first possibility of subcase c can be proven analogously.

Step 2.1: We first observe that prior to invocation i, $(x', y') \in paths$ (by the assumptions of Case 2) and $(x, p) \in paths$ (by the assumptions of the second possibility of Case 2c). If $(x', y') \in paths$ and $(x, p) \in paths$ prior to invocation i, and $y' = x$ (by assumption of Lemma 1), then our induction hypothesis tells us that $(x', p) \in paths$.

Step 2.2: Since $(q, y) \in paths$ (by the assumptions of the second possibility of Case 2c) we can apply Observation 1 to conclude that, by the time $[24a]$ is reached, $(p, y) \in newPaths$.

From Step 2.1, we know that prior to invocation i, $(x', p) \in paths$ and from Step 2.2 we know that by the time $[24a]$ is reached, $(p, y) \in newPaths$. From this, we can apply Observation 2 to conclude that (x', y) will be added to $paths$. Thus $\mathcal{C}$ holds.

Case 3 We note that both (x, y) and (x', y') can be added to $paths$ in any of the three locations mentioned in case 2. Again, we prove the most complex case (the second possibility of subcase c), where both (x, y) and (x', y') are added on $[28a]$.

Since Case 3 assumes that both (x, y) and (x', y') are added during the ith invocation of $\mathsf{add\text{-}path}$, we can apply Observation 3 to both (x, y) and (x', y') to conclude that $(x, p), (x', p), (q, y)$, and (q, y') are all elements of $paths$ prior to invocation i. Since $(q, y) \in paths$, Observation 1 tells us that by the time $[24a]$ is reached, $(p, y) \in newPaths$. Since $(x', p) \in paths$ and $(p, y) \in newPaths$, Observation 2 tells us that when $\mathsf{add\text{-}path}$ terminates, $(x', y) \in paths$ fulfilling $\mathcal{C}$.

Since each of the three subcases of the inductive case allows us to conclude $\mathcal{C}$, the induction hypothesis is true after invocation i.

Having shown that $\forall (x, y) \in paths$, $\forall (x', y') \in paths$, $(\mathsf{unify}(y', x) \neq \bot) \supset (x', y) \in paths$ holds for the base case and the inductive case, we can conclude that Lemma 1 holds. $\Box$

Lemma 2 *From an initially empty knowledge base, If $c_1, \ldots, c_n$ are the credentials given to* PC *as input for invocations $1, \ldots, n$, then after the nth invocation of* PC*, the following must hold for each $c_j, j \leq n$:*

1. *If* $((p, q) \leftarrow \mathsf{credToPath}(c_j)) \neq \bot$ *and* $\mathsf{depends\text{-}on}(c_j) = \bot$, $\mathsf{add\text{-}path}((p, q))$ *has been invoked and* $(p, q) \in paths$.

2. *If* $((p, q) \leftarrow \mathsf{credToPath}(c_j)) \neq \bot$, $\pi \leftarrow \mathsf{depends\text{-}on}(c_j)$ *and* $\pi \in paths$, $\mathsf{add\text{-}path}((p, q))$ *has been invoked and* $(p, q) \in paths$.

Proof We note that $incompletePaths$ is a list of tuples that contain a credential and the path it depends on. The path is derivable directly from the credential, and is included only for ease of indexing. For ease of presentation, we will refer to the elements of $incompletePaths$ as credentials. We prove Lemma 2 by induction over the number of invocations i of PC. Our induction hypothesis is that conditions 1-2 of Lemma 2 (which we label $\mathcal{C}$) hold after invocation $i - 1$.

Base case: The base case occurs when $i = 1$. It is straightforward to see that for any credential c_1 such that $\mathsf{credToPath}(c_1) \neq \bot$ and $\mathsf{depends\text{-}on}(c_1) = \bot$, c_1 will be converted to a path and given to add-path on $[9p]$. Since $paths$ is initially empty, it is not possible for a path to depend on a $\pi \in paths$ as is required by the second condition of Lemma 2. In this case, if $\mathsf{credToPath}(c_1) \neq \bot$ and $\pi \leftarrow \mathsf{depends\text{-}on}(c_1)$, c_1 must be added to $incompletePaths$ on $[6p]$.

Inductive case: For the inductive case, if $\mathsf{credToPath}(c_i) = \bot$, PC immediately exits ($[3p]$). If $\mathsf{credToPath}(c_1) \neq \bot$, $\pi \leftarrow \mathsf{depends\text{-}on}(c_1)$, and $\pi \notin paths$, c_i is added to $incompletePaths$, and PC will exit without adding any new paths. In both cases, $\mathcal{C}$ is trivially true by the induction hypothesis.

In all other cases, c_i will be converted to a path (p, q) and given to add-path ($[9p]$), which adds (p, q) to $paths$ ($[17a]$). However, if c_i was given to add-path ($[9p]$), it is possible that the invocation of add-path added to $paths$ a path π that a previous credential c_j (where $0 < j < i$) depends on. If such a path was added, then $c_j \in incompletePaths$ (by $[6p]$ of the jth invocation of PC). To compensate for this, after invoking add-path for c_i, PC iterates through $incompletePaths$ ($[10p]$) and invokes add-path for any credential that depends on a path $\pi \in paths$.

We have shown that after the ith invocation of PC completes, add-path has been invoked for c_i. We have also shown that for any credential $c_j \in incompletePaths$ that depends on a path created during the ith invocation of PC, add-path has been invoked for c_j as well. From this and our induction hypothesis, we can conclude that when PC exits, add-path has been invoked with each credential that depends on a path $\pi \in paths_i$, which satisfies the conditions of $\mathcal{C}$. $\square$

Theorem 2 *If* PC *has completed on* KB*, then for any* A, B *such that* $A \neq B$*, if for some* F $(B\ \mathbf{says}\ F \supset A\ \mathbf{says}\ F)$ *in the context of* KB*, then* $(B\ \mathbf{says}\ F, A\ \mathbf{says}\ F) \in KB$.

Proof If $(B\ \mathbf{says}\ F \supset A\ \mathbf{says}\ F)$ is true, then there must exist a set of delegation credentials from which we can conclude $(B\ \mathbf{says}\ F \supset A\ \mathbf{says}\ F)$. Since all credentials are given as input to PC, from Lemma 2 we can conclude that add-path has been invoked for all independent credentials and for all dependent credentials that depend on a path that exists. We then apply Lemma 1 to show that, from the set of credentials given to add-path, all possible paths have been constructed, thus proving Theorem 2. $\square$

B Completeness of Delayed Backward Chaining

Our objective is to demonstrate that the proving ability of a prover that uses delayed backward chaining is strictly greater than the proving ability of a prover that uses the inline backward chaining algorithm we presented previously [4] (reproduced in Figure 13). For the purpose of formal comparison, we assume that all caching optimizations described in our previous work are disabled. We also assume that all participants contributing to the construction of a distributed proof use the same set of tactics.

We refer to the inline backward chaining prover as $\mathsf{bc\text{-}ask}_I$. $\mathsf{bc\text{-}ask}_I$ outputs either a complete proof or $\bot$, while $\mathsf{bc\text{-}ask}_D$ (shown in Figure 3) may additionally output a marker indicating a choice subgoal that

0	global set KB	/* knowledge base */
1	substitution $\mathsf{bc\text{-}ask}_I($	/* returns a substitution */
	list $goals$,	/* list of conjuncts forming a query */
	substitution θ,	/* 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 θ to first goal */
5	if $(goals = [\,] \wedge \theta \in failures)$ then return $\perp$	/* θ known not to produce global solution */
6	if $(goals = [\,])$ then return θ	/* 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 $((\alpha \leftarrow \mathsf{rpc}_l(\mathsf{bc\text{-}ask}_I(\mathsf{first}(goals), \theta, failures'))) \neq \perp)$	/* make remote request */
12	$failures' \leftarrow \alpha \cup failures'$	/* prevent α from being returned again */
13	$answer \leftarrow \mathsf{bc\text{-}ask}_I(\mathsf{rest}(goals), \alpha, failures)$	/* 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 $((\beta \leftarrow \mathsf{bc\text{-}ask}_I(P, \mathsf{compose}(\theta', \theta), failures')) \neq \perp)$	/* prove subgoals */
18	$failures' \leftarrow \beta \cup failures'$	/* prevent β from being returned again */
19	$answer \leftarrow \mathsf{bc\text{-}ask}_I(\mathsf{rest}(goals), \beta, failures)$	/* 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 13: $\mathsf{bc\text{-}ask}_I$, an inline backward chaining algorithm [4]

needs to be proved. As such, a wrapper mechanism must be used to repeatedly invoke $\mathsf{bc\text{-}ask}_D$, aggregate markers, and chose which marker to purse, e.g., by asking the user. To accomplish this, we introduce the abstraction of a *distributed prover*, of which $\mathsf{bc\text{-}ask}_I$ is an example. To construct a distributed prover using $\mathsf{bc\text{-}ask}_D$, we define a wrapper, bc_D (shown in Figure 14) that accomplishes the above objectives. bc_D is designed explicitly for formal comparison; as such, it lacks mechanisms (e.g., for local credential creation, user interaction) that are necessary in practice, but not present in $\mathsf{bc\text{-}ask}_I$. The addition of these mechanisms allows the delayed distributed prover to find proofs in situations where an inline distributed prover is unable to do so.

Our task is now to show that a delayed distributed prover will find a proof of a goal if an inline distributed prover finds a proof. We let $[kd]$ represent line k in $\mathsf{bc\text{-}ask}_D$, $[kbcd]$ represent line k in bc_D, and $[ki]$ represent line k in $\mathsf{bc\text{-}ask}_I$. We will make use of the term *recursion height*, defined below. Note that because all the functions we consider here are deterministic, the recursion height is well-defined.

$\langle$substitution, credential[ ]$\rangle$ bc_D(/* returns a substitution */
    list $goals$, /* list of conjuncts forming a query */
    substitution θ, /* current substitution, initially empty */
    set $failures$) /* set of substitutions that are known not to produce a complete solution */

    local set $markers, failures'$
    $failures' \leftarrow failures$
    while $((\langle\beta, creds\rangle \leftarrow \mathsf{bc\text{-}ask}_D(goals, \theta, failures')) \neq \perp)$ /* find all solutions */
        if $\mathsf{notMarker}(\beta)$, return $\langle\beta, creds\rangle$ /* if complete proof found, return*/
        $markers \leftarrow markers \cup \{\beta\}$
        $failures \leftarrow failures \cup \{\beta\}$

    for each $m \in markers$
        $\langle f, \theta, failures'\rangle \leftarrow m$
        while$((\langle\alpha, creds\rangle \leftarrow \mathsf{rpc}_l(\mathsf{bc}_D(f, \theta, failures'))) \neq \perp)$
            $failures' \leftarrow \alpha \cup failures'$
            $\mathsf{addToKB}(creds)$
            $\langle\beta, creds\rangle \leftarrow \mathsf{bc}_D(goals, \theta, failures)$
            if $(\beta \neq \perp)$, return $\langle\beta, creds\rangle$
    return $\langle\perp, null\rangle$

Figure 14: bc_D, a wrapper to process partial proofs returned by $\mathsf{bc\text{-}ask}_D$

Definition 2 *We define the* environment *of a function invocation to be the values of all globals when the function is called and the parameter values passed to the function. The* recursion height *of a function in a given environment is the depth of recursive calls reached by an invocation of that function with that environment.*

Terminating tactics We note that when the inline prover finds a proof by making a remote request, it may not fully explore the search space on the local node. Since a delayed prover investigates all branches locally before making a request, should a later branch not terminate, no solution will be found. We assume here that all sets of tactics terminate on any input, which is a requirement in practice to handle the case in which no proof exists. In the case where a depth limiter is necessary to guarantee termination, the same limit will be used for both delayed and inline provers.

Lemma 3 *Consider two knowledge bases KB and KB' such that $KB \subset KB'$. Assume that when trying to prove goal G using KB,* $\mathsf{bc\text{-}ask}_D$ *finds ρ, which is either a complete proof or a proof containing a marker. If* $\mathsf{bc\text{-}ask}_D$ *is invoked repeatedly with goal G and knowledge base KB' and each previous proof is added to $failures$, then an invocation of* $\mathsf{bc\text{-}ask}_D$ *will find ρ.*

Proof sketch As discussed in Section 8, we assume that the logic is monotonic—that is, if a proof of G exists from KB, it also exists from KB'. Line $[11d]$ iterates through all elements of the knowledge base. The only places that exit this loop prematurely are $[18d]$, $[21d]$, and $[23d]$. Through induction over the recursion height of $\mathsf{bc\text{-}ask}_D$, we can show that if the proof returned by one of these lines is added to $failures$ on a subsequent call to $\mathsf{bc\text{-}ask}_D$($[17d]$, $[19d]$, or $[22d]$), then that proof will be disregarded and the next element of the knowledge base will be considered ($[11d]$). If this is repeated sufficiently many times, $\mathsf{bc\text{-}ask}_D$ using KB' will find the same proof ρ produced by $\mathsf{bc\text{-}ask}_D$ using KB. □

Lemma 4 *For any goal G and knowledge base KB, $\textsf{bc-ask}_D$ using tactic set $\mathcal{T}$ will find a proof of G without making any remote requests if $\textsf{bc-ask}_I$ using $\mathcal{T}$ will find a proof of G without making any remote requests.*

Proof sketch If $\textsf{bc-ask}_I$ finds a proof without making a request, then the proof must be completable from the local knowledge base and the search for the proof must not involve investigating any formulas of the form A **says** F such that $\textsf{determine-location}(A) \neq \textit{localmachine}$. Our induction hypothesis states that if both $\textsf{bc-ask}_I$ and $\textsf{bc-ask}_D$ make a recursive call with identical environments that will have recursion height h, then the recursive $\textsf{bc-ask}_D$ call will return the same result as the recursive $\textsf{bc-ask}_I$ call.
Base case: The base case is when the recursion height $= 0$, which occurs when $goals = [\,]$. Since an assumption of this case is that all input parameters to $\textsf{bc-ask}_D$ are the same as $\textsf{bc-ask}_I$, by inspection of the algorithms ($[3d]$-$[4d]$, $[5i]$-$[6i]$), $\textsf{bc-ask}_D$ and $\textsf{bc-ask}_I$ will both return $\perp$ if $\theta \in failures$, or θ otherwise.
Inductive case: For the inductive case, we assume that, at recursion height $h+1$, $\textsf{bc-ask}_D$ was invoked with the same parameters as $\textsf{bc-ask}_I$. Since, by the assumptions of this lemma, $\textsf{bc-ask}_I$ does not make any remote requests, l must resolve to the local machine on $[6d]$ and $[8i]$, thus bypassing $[9d]$-$[10d]$ and $[11i]$-$[14i]$. This means that both strategies will behave identically until after q' is unified against an element in the KB ($[12d]$, $[16i]$). At this point, there are two cases to consider: (1) (P, q) is a tactic or (2) (P, q) represents a credential or a fact. In either case, $\textsf{bc-ask}_I$ will continue to $[17i]$.

Case 1: If (P, q) is a tactic, then P will be non-empty, causing $\textsf{bc-ask}_D$ to continue to $[19d]$. At this point, the parameters to the recursive call on $[19d]$ are identical to those of $[17i]$, and we can apply our induction hypothesis to conclude that $[19d].\beta = [17i].\beta$. β is then added to $\textit{failures}'$, ensuring that the parameters to $[19d]$ will remain identical to $[17i]$ on subsequent iterations. Since, by assumption, no remote requests are necessary, $[21d]$ will never be executed. Since all parameters to the recursive call on $[22d]$ are identical to those of $[19i]$, we can apply our induction hypothesis to conclude that $[22d].answer = [19i].answer$. If $answer \neq \perp$, it will be returned in both scenarios, otherwise $\textsf{bc-ask}_D$ and $\textsf{bc-ask}_I$ will continue to the next iteration of $[19d]$ and $[17i]$. With $[22d].answer = [19i].answer$ for each iteration, if $\textsf{bc-ask}_I$ returns a solution, $\textsf{bc-ask}_D$ will also. Otherwise, $\textsf{bc-ask}_D$ and $\textsf{bc-ask}_I$ will return $\perp$.

Case 2: The second case occurs when (P, q) represents a credential or a fact. This implies that P is an empty list. Then, $[17i]$ will return with $\beta = \perp$ if $\textsf{compose}(\theta', \theta) \in \textit{failures}'$ ($[5i]$), and $\beta = \textsf{compose}(\theta', \theta)$ ($[6i]$) otherwise. Note that β is added to $\textit{failures}'$ on $[18i]$, so the recursive call inside the while loop on $[17i]$ will succeed only once.

Because $P = [\,]$, the condition of the if statement on $[14d]$ will be true. If $\phi \in \textit{failures}'$ (where $\phi = \textsf{compose}(\theta', \theta)$ from $[13d]$) then $[16d]$-$[18d]$ will not be executed. $\textsf{bc-ask}_D$ will then proceed to try the next element in the knowledge base ($[11d]$), which is the same as the behavior of $\textsf{bc-ask}_I$ when $[17i].\beta = \perp$. If $[15d].\phi$ is not in $\textit{failures}'$, then $[16d]$-$[18d]$ will be executed. Since ϕ is not modified between $[13d]$ and $[17d]$, $[17d].\phi = \textsf{compose}(\theta', \theta) = [19i].\beta$. At this point, we know that all parameters to the recursive call on $[17d]$ equal those of $[17i]$. At this point, we can apply our induction hypothesis to show that $[17d].answer = [19i].answer$. From this, we can conclude that if $\textsf{bc-ask}_I$ finds a proof, $\textsf{bc-ask}_D$ will find a proof as well. □

Lemma 5 *For any distributed proving node attempting to prove goal G with knowledge base KB, $\textsf{bc}_D$ will find a proof of G if $\textsf{bc-ask}_I$ would find a proof of G, under the assumption that all remote requests, if given identical parameters, will return the same answer in both strategies.*

Proof We prove Lemma 5 via induction over the number of remote requests r that a local prover using $\textsf{bc-ask}_I$ makes to complete the proof. Our induction hypothesis states that for all queries such that $\textsf{bc-ask}_I$ finds a proof with $r-1$ requests, $\textsf{bc}_D$ will find a proof as well.
Base Case: The base case occurs when $r=0$ and can be shown by direct application of Lemma 4.
Inductive Case: We prove the inductive case by (1) showing that $\textsf{bc}_D$ will eventually make a remote request that is identical to the initial remote request made by $\textsf{bc-ask}_I$, (2) showing that when $\textsf{bc}_D$ re-runs the entire query after the remote request finishes ($[12bcd]$), this query will be able to find a proof of the formula proved remotely in (1) using only local knowledge, and (3) showing that, after deriving the proof described in (2), $\textsf{bc-ask}_D$ will recurse with the same parameters that $\textsf{bc-ask}_I$ recurses with after $\textsf{bc-ask}_I$ finishes its initial remote request.

Step 1: By an argument analagous to that of Lemma 4, we assert that $\textsf{bc-ask}_D$ will behave identically to $\textsf{bc-ask}_I$ until the point at which $\textsf{bc-ask}_I$ encounters a goal for which it needs to make a remote request ($[11i]$). At this point, $\textsf{bc-ask}_D$ will construct a marker ($[9d]$) containing the same parameters as $\textsf{bc-ask}_I$ would use to make the remote request.

Since $\textsf{bc}_D$ exhaustively investigates all markers ($[7bcd]$), it will investigate the marker described above. Thus, $\textsf{bc}_D$ will make a remote request with identical parameters to the request made by $\textsf{bc-ask}_I$, which, by the assumption of this lemma, will return the same result under both strategies.

Step 2: By inspection of $\textsf{bc-ask}_D$ (in particular, $[18d]$ and $[23d]$), we can see that all credentials used in a proof of a goal are returned when the query terminates. Thus, when a remote request for a goal f ($[9bcd]$) returns with a complete proof, the response will include all of the credentials necessary to re-derive that proof. These credentials are added to the knowledge base on $[11bcd]$, so from Lemma 3 we can conclude that the local prover can now generate a complete proof of f.

We refer to the invocation of $\textsf{bc-ask}_D$ that constructed the marker in Step 1 as M. When $\textsf{bc}_D$ re-runs the entire query ($[12bcd]$), $\textsf{bc-ask}_D$ will retrace its steps to M. Since new credentials have been added to the knowledge base, $\textsf{bc-ask}_D$ may first explore additional branches, but because it is exhaustive and deterministic, it will eventually explore the same branch as the first query. Upon reaching M, $\textsf{bc-ask}_D$ will first construct a remote marker identical to the one produced in the first round ($[9d]$), but, since $\textsf{bc}_D$ repeatedly invokes $\textsf{bc-ask}_D$ until a either complete proof has been found or no more markers exist, $\textsf{bc-ask}_D$ will be invoked again with that marker in $failures$ ($[12bcd]$). This time, $\textsf{bc-ask}_D$ will attempt to prove the $\textsf{first}(goals)$ ($[11d]$), and having sufficient credentials, will generate the same proof ($[14d]$ or $[19d]$) as was returned by the remote request. Note that it is possible to generate alternative proofs first, but the combination of $\textsf{bc}_D$ repeatedly invoking $\textsf{bc-ask}_D$ ($[12bcd]$) with previous solutions included in $failures$ and the loops on $[11d]$ and $[19d]$ ensures that the solution identical to the result of the remote request is eventually found.

Step 3: From Step 2, we know that $\textsf{bc-ask}_D$ finds the same proof of $\textsf{first}(goals)$ as $\textsf{bc-ask}_I$ does. In the case where $\textsf{first}(goals)$ is provable directly from a credential, this means that $[17d].\phi = [13i].\alpha$ In the case where $\textsf{first}(goals)$ is not provable directly from a credential, $[22d].\beta = [13i].\alpha$. In either case, $\textsf{rest}(goals)$ and $failures$ are identical to those of invocation M, which, in turn, is effectively identical to the invocation of $\textsf{bc-ask}_I$ that made the remote request for which M constructed a marker. At this point, we have shown that all parameters to the recursive $\textsf{bc-ask}_D$ call (either $[17d]$ or $[22d]$) are identical to those of $[13i]$.

The knowledge base KB' used by $\textsf{bc-ask}_D$ was initially identical to the knowledge base KB used by $\textsf{bc-ask}_I$. However, $\textsf{bc}_D$ added the credentials returned by the remote request to KB' ($[11bcd]$), resulting in a KB' such that $KB \subset KB'$. By Lemma 3, we can conclude that bc-ask$_D$ will eventually find the same proof using KB' as it finds using KB. Thus, if we can prove Lemma 5 when bc-ask$_D$ uses KB, the result will hold when bc-ask$_D$ uses KB'. From the previous paragraph, we know that all parameters to the recursive bc-ask$_D$ call (either $[17d]$ or $[22d]$) are identical to those of $[13i]$, and from this paragraph, we can conclude that the knowledge base in use by bc-ask$_D$ is identical to that of bc-ask$_I$.

The proof created in the inline strategy on $[13i]$ must be completable with $r-1$ remote requests, as one remote request has already been made. At this point, we can apply our induction hypothesis to show that either $[17d].answer = [13i].answer$ or $[22d].answer = [13i].answer$. From this and inspection of $[18d]$, $[23d]$, and $[14i]$, it is clear that bc-ask$_D$ will find a proof if bc-ask$_I$ is able to find a proof. □

Theorem 3 *For any goal G, a delayed distributed prover with global knowledge base KB will find a proof of G if an inline distributed prover using KB will find a proof of G.*

Proof We first define the remote request height h of a proof to be the recursion height of the algorithm with respect to remote requests. For example, if A asks B and C for help, and C asks D for help, the remote request height of A's query is 2.

We are trying to show that bc$_D$ (which invokes bc-ask$_D$) will produce a complete proof if bc-ask$_I$ produces a complete proof. We prove Theorem 3 by induction over the remote request height of the proof. Our induction hypothesis states that if bc-ask$_I$ and bc$_D$ are invoked with parameters P (which include goal G), and bc-ask$_I$ finds a proof of G with remote request height at most h, bc$_D$ will find a proof of G as well.
Base Case: The base case occurs when $h = 0$. Since this corresponds to the case where bc-ask$_I$ does not make any remote requests, we can apply Lemma 4 to conclude that bc$_D$ will produce a proof if bc-ask$_I$ produces a proof.
Inductive Case: For the inductive case, we note that any remote requests made by bc-ask$_I$ operating at request height $h+1$ must have height at most h. Lemma 5 proves that bc$_D$ will find a proof of G if bc-ask$_I$ finds a proof of G under the assumption that any remote request made by bc$_D$ with parameters P will return the same result as a remote request made by bc-ask$_I$ with parameters P. Since any remote requests must have height at most h, we can apply our induction hypothesis to discharge the assumption of Lemma 5 which allows us to conclude that bc$_D$ will find a proof if bc-ask$_I$ finds a proof with request height $h+1$. □

C Completeness of LR Tactics

IR and LR are both tactic sets that are used in a common distributed proving framework, which we will refer to as *DP*. This framework, formed by the combination of bc-ask$_D$ (Figure 3) and bc$_D$ (Figure 14), is responsible for identifying choice subgoals, determining if the formula under consideration can be proved either directly from cache or by recursively applying tactics. We assume that all tactics and inference rules are encoded such that their premises are proved from left to right. In any situation where IR must use a depth limit to ensure termination (rather than for efficiency), we assume that LR uses the same depth limit.[4]

For simplicity, when referencing the version of the distributed proving framework that uses IR tactics, we will simply write IR. We write LR to refer to a version of the distributed proving framework that uses LR tactics in conjunction with FC and PC.

[4]In practice, we have not encountered a situation in which a depth limit was necessary for LR.

Lemma 6 *Consider the case in which* IR *is given the query* A **says** F *and each of the first series of recursive* bc-ask$_D$ *calls made by* IR *is an application of a delegation rule for which the left premise is provable, and the next recursive* bc-ask$_D$ *call by* IR *is to prove* B **says** F. *If* LR *is also given query* A **says** F, LR *will eventually attempt to prove* B **says** F.

Proof sketch If the first r recursive bc-ask$_D$ calls made by IR are applications of a delegation rule for which the first premise is provable, and the $(r+1)$'th recursive bc-ask$_D$ call ($[17d]$, $[22d]$) by IR is to prove B **says** F, then it must be true that B **says** $F \supset A$ **says** F. From Theorem 2, we know that the path $(B$ **says** F, A **says** $F)$ is in the knowledge base. Since LR has a left tactic whose conclusion is either equal to, or more general than, the conclusion of the delegation rule that was applied by IR in the rth recursive call, LR will use this tactic to exhaustively try all paths whose conclusion unifies with A **says** F and attempt to prove the premise of each such path. Eventually, LR will try the path $(B$ **says** F, A **says** $F)$, and attempt to prove B **says** F. □

Lemma 7 *If* IR *finds a proof of* F *with marker* m *using knowledge base* KB, *a version of* IR *with cycle prevention will also find a proof of* F *with marker* m *using* KB.

Proof sketch As defined in Section 7, we refer to the version of IR with cycle prevention as IR-NC. We define a cycle to exist if a prover attempts to prove formula F as part of the recursive proof of F. In this case, when IR attempts to prove F, it will apply a sequence of inference rules that lead it to attempt to prove F again. As repeated applications of bc-ask$_D$ may only decrease the generality of a substitution θ ($[13d]$), the subsequent attempt to prove F will be with a θ that is more specific than the initial attempt. Additionally, the substitutions present in $failures$ accumulate as bc-ask$_D$ recurses ($[16d]$, $[20d]$) . From this, we know that the subsequent attempt to prove F will do so in an environment that is strictly more restrictive (more specific θ, more substitutions in $failures$). Thus, if IR finds a proof of F on the subsequent attempt, we can conclude that a proof of F can be found on the initial attempt. Since the only difference between IR-NC and IR is that IR-NC eliminates cycles, IR-NC will be restricted to finding a proof of F on the initial attempt. Since we have shown that IR is capable of finding a proof of F on the initial attempt if it is able to find a proof on the subsequent attempt, we can conclude that IR-NC will find a proof of F on the initial attempt as well. □

Lemma 8 *If both* IR *and* LR *invoke* bc-ask$_D$ *with identical parameters and* IR *finds a complete proof from local knowledge, then* LR *will find a complete proof from local knowledge as well.*

Proof Since *DP* will not automatically make any remote requests, if IR finds a complete proof of goal A **says** F, then there is a series of inference rules that, when applied to a subset of the locally known credentials, produces a proof of A **says** F. Theorem 1 shows that FC produces a proof of each formula for which a proof is derivable from locally known credentials, so a proof of A **says** F will be found by *DP* and returned immediately before any LR tactics are applied. □

Lemma 9 *If both* IR *and* LR *invoke* bc-ask$_D$ *with identical parameters, and if* IR *finds a proof with marker* m *then* LR *will find a proof with marker* m.

Proof We define the depth of a proof to be the depth of the tree that represents the series of inference rules that, when applied to the original goal, constitute a proof of the goal. To prove Lemma 9, we use induction over the depth d of the proof found by IR. Our induction hypothesis is that if both IR and LR invoke bc-ask$_D$ with identical parameters, Lemma 9 will hold for proofs with depth at most d.

Base case: The base case occurs when $d = 0$. Since Lemma 9 assumes that bc-ask$_D$ does not return a complete proof, we know that the base case represents the creation of a marker. This is handled by *DP* in a way that is independent of the tactic set.

Inductive case: For the inductive case, let the formula that IR and LR are attempting to prove be A $\mathbf{says}$ F. The proof of A $\mathbf{says}$ F that IR is able to find has depth $d + 1$ and contains marker m.

Let $((P = p_1 \wedge \ldots \wedge p_j), q)$ represent the first inference rule applied to A $\mathbf{says}$ F by IR. This rule is either a delegation rule or a standard (i.e., non-delegation) rule. Since the only manner in which IR and LR differ is in the rules dealing with delegation, if (P, q) is a standard rule, LR will apply this rule during the course of an exhaustive search. At this point, both IR and LR will attempt to prove all formulas in P. The proofs of these formulas can have depth at most d, so we can apply our induction hypothesis to show that LR will find a proof with marker m for this case.

If (P, q) represents a delegation rule, then P will consist of two formulas, p_l and p_r. If IR finds a proof of $q = A$ $\mathbf{says}$ F with marker m, then either (a) the proof of p_l contains m, or (b) p_l is provable from local knowledge and the proof of p_r contains m.

(a) Here we note that if IR finds a proof of p_l with marker m, then we can apply Lemma 7 to conclude that IR-NC finds a proof with marker m as well. The right tactic of LR differs from the corresponding inference rule in IR-NC only in that LR requires that p_l not be provable from local knowledge (as described in Section 6.2). Thus, if IR-NC investigates p_l, LR will apply a right tactic and investigate p_l as well. Since the proof of p_l found by IR can have depth at most d, we can apply our induction hypothesis to show that LR will find a proof of p_l containing marker m in this case.

(b) If p_l (the premise pertaining to delegation) is provable, then when LR applies a left tactic, we can apply Lemma 6 to show that both IR and LR will ultimately investigate the same right subgoal (e.g., B $\mathbf{says}$ F). The proof of this subgoal must have depth at most d, so we can apply our induction hypothesis to to show that LR will find a proof with marker m in this case. □

Theorem 4 *If* IR *finds a proof of goal F, then* LR *will find a proof of F as well.*

Proof sketch We prove Theorem 4 by induction over the recursion height (see Definition 2) of bc$_D$. Our induction hypothesis states that at recursion height h, any recursive call to bc$_D$ with environment ε made by LR will return a proof if a recursive call to bc$_D$ with environment ε made by IR returns a proof.

Base case: The base case occurs when the recursion height $= 0$. Since recursion of bc$_D$ occurs only when a proof involving a marker is found, we can conclude that, if IR tactics are able to find a complete proof, bc-ask$_D$ will return a proof that does not include a marker ([$3bcd$]). We can apply Lemma 8 to conclude that, when using LR tactics, bc-ask$_D$ will also return a proof.

Inductive case: For the inductive case, we let $h+1$ be the recursion height of the current invocation of bc$_D$. bc$_D$ recurses on [$12bcd$] only if the proof returned by bc-ask$_D$ ([$3bcd$]) contained a marker indicating that a remote request is necessary. From Lemma 9, we know that the marker returned by LR will be the same as the marker returned by IR. From this, we know that the remote request made by bc$_D$ on [$9bcd$] will have the same parameters in both the LR and IR scenarios. If we momentarily assume that the remote request returns the same response in both scenarios, then we can show that each of the parameters of the recursive call on [$12bcd$] in the LR scenario are the same as those of the IR scenario. Since the recursive call on [$12bcd$] must have height at most h, then we can apply our induction hypothesis to conclude that bc$_D$ will return find a proof using LR tactics if it is able to find one using IR tactics.

We now return to the assumption that remote requests made with the same environment in both scenarios will return the same result. This assumption can be relaxed via a proof that inducts over the remote request height of the distributed prover. This proof is analagous to that of Theorem 3. □