Efficient Proving for Practical Distributed Access-Control Systems*

Lujo Bauer[1], Scott Garriss[1], and Michael K. Reiter[2]

[1] Carnegie Mellon University
[2] University of North Carolina at Chapel Hill

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., [9,18]), recent work has imported logic into the system as a means to *implement* access control (e.g., [6]). 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** (Alice, Bob, resource)" or "K_{CA} **signed** K_{Alice} **speaksfor** K_{CA}.Alice"), and then inference rules are used to derive a proof that a required policy is satisfied (e.g., "Manager **says open**(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., [9,19,16,14]). 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.

* 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.

J. Biskup and J. Lopez (Eds.): ESORICS 2007, LNCS 4734, pp. 19–37, 2007.

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 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 4.2 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 5.

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 3.2) and systematically generating tactics (Section 3.3); (3) describe a technique for utilizing these pre-computed results to find proofs in dramatically less time than previous approaches (Section 3.3); and (4) evaluate our technique on a collection of policies representative of those used in practice (Section 4.1) and those indicative of a larger

$\phi ::= s \textbf{ signed } \phi' \mid p \textbf{ says } \phi'$ (s ranges over strings and p over principals)
$\phi' ::= \textbf{open } (s) \mid p \textbf{ speaksfor } p \mid \textbf{delegate}(p, p, s)$

$$\frac{pubkey \textbf{ signed } F}{\textbf{key}(pubkey) \textbf{ says } F} \text{ (SAYS-I)} \qquad \frac{A \textbf{ says } (B \textbf{ speaksfor } A) \quad B \textbf{ says } F}{A \textbf{ says } F} \text{ (SPEAKSFOR-E)}$$

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

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

Fig. 1. A sample access-control logic [4]

deployment (Section 4.2). In Section 5, we discuss the use of our techniques in the context of additional logics, systems and applications. Proofs of our theorems, and discussion of related work elided due to space constraints, can be found in our accompanying technical report [5].

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 2. 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 5.

If `pubkey` is a particular public key, then $\textbf{key}(\texttt{pubkey})$ is the principal that corresponds to that key. If Alice is a principal, we write Alice.`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 $\textbf{says } F$. 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` $\textbf{signed } F$, which can be transformed into a belief ($\textbf{key}(\texttt{pubkey}) \textbf{ says } F$) using the SAYS-I inference rule. To describe a resource that a client wants to access, we use the **open** constructor. A principal believes the formula $\textbf{open}(resource)$ if she thinks that it is OK to access $resource$.[1] Delegation is described with the **speaksfor** and **delegate** predicates. The formula Alice $\textbf{speaksfor}$ Bob indicates that Bob has delegated to Alice his authority to make access-control decisions about any resource. $\textbf{delegate}(\text{Bob}, \text{Alice}, resource)$ transfers to Alice only the authority to access the resource called $resource$.

[1] **open** takes a nonce as a second parameter, which we omit here for simplicity.

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 4.2). 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 **says** **open**(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} **signed** **delegate**(Dept, Alice, 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} **signed** **delegate**(Alice, Alice.machine-room, door1)); and, finally, (3) making individuals like Bob members of the group (K_{Alice} **signed** (Bob **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 **says** **open**(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 [21]) 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.

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 3.3) 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.

Precomputing delegation chains. A second insight is to locally precompute and cache delegation chains using two approaches: the well-studied *forward chaining* algorithm [22] 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 Charlie **says** F, then we can prove David **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 4.2). As we discuss in Section 3.3, 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 [13]. 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 3.1). Second, a path-compression algorithm (which we introduce in Section 3.2) 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 **says** **open**(door1)) (Section 3.3).

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.

3.1 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 [22] 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 $q \in KB$.*

If forward chaining is invoked on a knowledge base for which there is no fixed point, the algorithm is not guaranteed to terminate. Because of this, forward chaining is frequently restricted to Datalog knowledge bases, for which it can be shown to be complete [22]. Our logic includes some functions that are not representable in Datalog, but we show that these functions are crafted to not affect completeness. For a proof of Theorem 1 and all other theorems in this paper, please see our technical report [5].

3.2 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

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

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

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

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

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

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

24      foreach (p', q') ∈ paths
25          foreach (p'', q'') ∈ newPaths              /* Try to append all new paths
26              if((θ ← unify(q', p'')) ≠ ⊥)               to all previous paths. */
27                  c ← (subst(θ, p'), subst(θ, q''))
28                  paths ← union(c, paths)
```

Fig. 2. PC, an incremental path-compression algorithm

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, 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.

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.

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

For the proof of Theorem 2, please see our technical report [5]. 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).

3.3 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.

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. For a more detailed description, please see our technical report [5].

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. 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 5.

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.[2] We call tactics generated in this manner LR tactics.

[2] For completeness, if there are choice subgoals in both premises, one will be resolved and then the prover will be rerun (see [5] for details). In practice, we have yet to encounter a circumstance where a single round of proving was not sufficient.

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

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).

Fig. 3. Example construction of LR tactics from an inference rule

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 3 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.

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 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 [10] 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).

Completeness of LR. Despite greater efficiency, LR tactics have strictly greater proving ability than the depth-limited inference rules. We state this formally below.

Theorem 3. *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.*

For the proof of Theorem 3, please see our technical report [5]. 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.

4 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 4.2, 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/`).

4.1 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)

Fig. 4. 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)

Fig. 5. Credentials on Charlie's phone

Policy. The policy for this scenario is expressed in the credentials known to Alice and Charlie, shown in Figures 4 and 5. The first six credentials of Figure 4 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

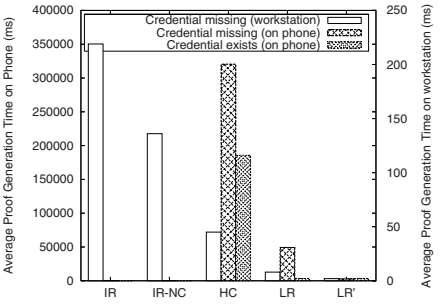

Fig. 6. Alice's prover generates complete proof or list of credentials that Alice can create

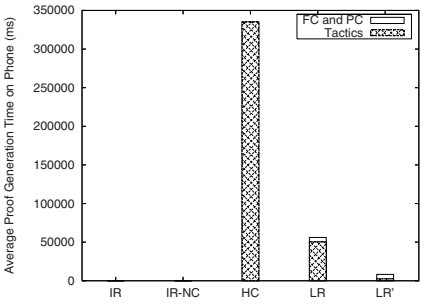

Fig. 7. Aggregate proving time: Charlie's before help request + Alice's + Charlie's after help request

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 5) 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 4 and 5 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 6 describes the proving performance experienced by Alice when she attempts to help Charlie. Alice wishes to delegate authority to Charlie by giving him membership in the group Alice.machine-room. 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 6 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 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 8 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.

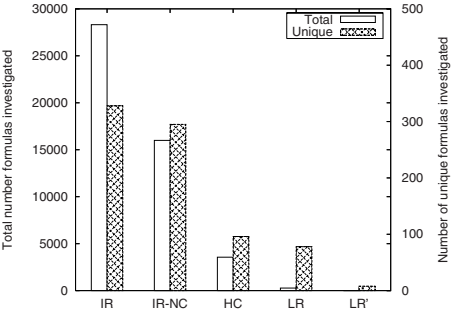

Fig. 8. Formulas investigated by Alice

4.2 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.

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 4.1.

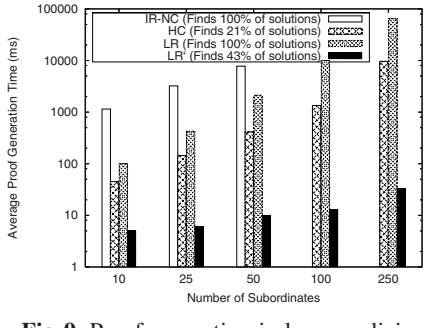

Fig. 9. Proof generation in larger policies

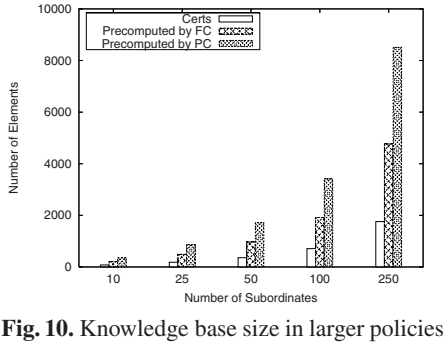

Fig. 10. Knowledge base size in larger policies

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 9 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 10. 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.

5 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., [8]).

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 3.1). This is sufficient to express most access-control logics, e.g., the logics of SD3 [17], Cassandra [7], and Binder [11], but is not sufficient to express higher-order 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 [1,19,16], the RT family of logics [20], Binder [11], Placeless Documents [3], and the domain-name service logic of SD3 [17], 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., [17,11,3]) 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 [12].

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 [15] allows users to dynamically create credentials. Properties (b) and (c) have been the focus of considerable previous work, notably SPKI [1,19,16], the DNS logic of SD3 [17], RT [20], and Cassandra [7]. 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.

6 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 5.

References

[1] Abadi, M.: On SDSI's linked local name spaces. Journal of Computer Security 6(1-2), 3–21 (1998)
[2] Appel, A.W., Felten, E.W.: Proof-carrying authentication. In: Proceedings of the 6th ACM Conference on Computer and Communications Security, ACM Press, New York (1999)
[3] Balfanz, D., Dean, D., Spreitzer, M.: A security infrastructure for distributed Java applications. In: Proceedings of the 2000 IEEE Symposium on Security & Privacy, IEEE Computer Society Press, Los Alamitos (2000)
[4] Bauer, L., Garriss, S., Reiter, M.K.: Distributed proving in acess-control systems. In: Proceedings of the 2005 IEEE Symposium on Security & Privacy, IEEE Computer Society Press, Los Alamitos (2005)
[5] Bauer, L., Garriss, S., Reiter, M.K.: Efficient proving for practical distributed access-control systems. Technical Report CMU-CyLab-06-015R, Carnegie Mellon University (2007)
[6] Bauer, L., Schneider, M.A., Felten, E.W.: A general and flexible access-control system for the Web. In: Proceedings of the 11th USENIX Security Symposium (2002)
[7] Becker, M., Sewell, P.: Cassandra: Flexible trust management, applied to electronic health records. In: Proceedings of the 17th IEEE Computer Security Foundations Workshop, IEEE Computer Society Press, Los Alamitos (2004)
[8] Blaze, M., Feigenbaum, J., Strauss, M.: Compliance checking in the PolicyMaker trust-management system. In: Hirschfeld, R. (ed.) FC 1998. LNCS, vol. 1465, Springer, Heidelberg (1998)
[9] Burrows, M., Abadi, M., Needham, R.: A logic of authentication. ACM Transactions on Computer Systems 8(1), 18–36 (1990)

[10] Chen, W., Warren, D.S.: Tabled evaluation with delaying for general logic programs. Journal of the ACM 43(1), 20–74 (1996)
[11] DeTreville, J.: Binder, a logic-based security language. In: Proceedings of the 2002 IEEE Symposium on Security and Privacy, IEEE Computer Society Press, Los Alamitos (2002)
[12] Ellison, C.M., Frantz, B., Lampson, B., Rivest, R.L., Thomas, B.M., Ylonen, T.: SPKI Certificate Theory, RFC2693 (1999)
[13] Felty, A.: Implementing tactics and tacticals in a higher-order logic programming language. Journal of Automated Reasoning 11(1), 43–81 (1993)
[14] Garg, D., Pfenning, F.: Non-interference in constructive authorization logic. In: CSFW'06. Proceedings of the 19th Computer Security Foundations Workshop (2006)
[15] Goffee, N.C., Kim, S.H., Smith, S., Taylor, P., Zhao, M., Marchesini, J.: Greenpass: Decentralized, PKI-based authorization for wireless LANs. In: Proceedings of the 3rd Annual PKI Research and Development Workshop (2004)
[16] Halpern, J., van der Meyden, R.: A logic for SDSI's linked local name spaces. Journal of Computer Security 9, 47–74 (2001)
[17] Jim, T.: SD3: A trust management system with certified evaluation. In: Proceedings of the 2001 IEEE Symposium on Security & Privacy, IEEE Computer Society Press, Los Alamitos (2001)
[18] Lampson, B., Abadi, M., Burrows, M., Wobber, E.: Authentication in distributed systems: Theory and practice. ACM Transactions on Computer Systems 10(4), 265–310 (1992)
[19] Li, N., Mitchell, J.C.: Understanding SPKI/SDSI using first-order logic. International Journal of Information Security (2004)
[20] Li, N., Mitchell, J.C., Winsborough, W.H.: Design of a role-based trust management framework. In: Proceedings of the 2002 IEEE Symposium on Security & Privacy, IEEE Computer Society Press, Los Alamitos (2002)
[21] Minami, K., Kotz, D.: Secure context-sensitive authorization. Journal of Pervasive and Mobile Computing 1(1) (2005)
[22] Russell, S., Norvig, P.: Artificial Intelligence, A Modern Approach, 2nd edn. Prentice Hall, Englewood Cliffs (2003)