Efficient, Usable Proof-Construction Strategies for Distributed Access-Control Systems

Scott Garriss

July 22, 2008

Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Contents

List of Tables

List of Figures

Abstract

Distributed access-control systems implemented using formal logic have several advantages over conventional access-control systems; namely, they allow for decentralized policy administration, can express a wide variety of policies without ambiguity, and provide greater assurance that granted accesses comply with access-control policy. Access is granted only if the system can construct a proof, in formal logic, that demonstrates that the access is authorized. The efficiency of the proof-construction algorithm is therefore of great interest, as it resides on the critical path to an access being granted. Any delay in proof construction will delay access, which can significantly impact the usability and effectiveness of the system.

However, it can be challenging to efficiently construct a proof in such a system. The credentials that encode access-control policy may be distributed among distant nodes in the system, or may not have been created yet. Users should be able to extend access-control policy in response to a requested access, and they should be guided through this process to ensure that the new credential will result in access being granted. Additionally, user input must be considered to ensure that, e.g., a request that Alice create a new credential does not disturb her at an inappropriate time.

The objective of this thesis is therefore to describe a suite of techniques that enable the efficient construction of a proof of access. We demonstrate that these techniques are practical using policies, data, and experience drawn from an experimental access-control system deployed at our university. This suite of techniques consists of three main components: an algorithm for distributing the proof construction process, an efficient proof-construction strategy that incorporates human interaction, and techniques for identifying and resolving misconfigurations in access-control policy before they delay or deny a legitimate access.

Distributed proof construction We introduce a distributed approach to proof construction and show that this approach substantially reduces the amount of communication required to construct a proof when compared to prior work, which employs a centralized approach with straightforward extensions for collecting credentials from remote nodes. These gains result from both ensuring that each component of the proof is assembled by the party with the most relevant knowledge and effectively utilizing a distributed cache. We show analytically that our approach will find a proof whenever the centralized approach will do so.

The distributed approach allows each party a great deal of flexibility as to how they attempt to construct a proof. This flexibility enables us to explore the efficient, usable proof-construction techniques that represent the second component of this thesis.

Efficient, usable proof construction Preliminary experience with our access-control testbed indicated that a practical proof-construction strategy must allow users to direct the proof search (as it might, e.g., involve bothering another user), identify situations in which the proof could be completed with the creation of a new credential, and to maximize the usage of locally cached credentials. We present a strategy for constructing proofs in such an environment and show that, through the effective use of precomputed results, we achieve dramatic improvements over prior work in terms of computational efficiency at the time of access. As before, we show that these gains do not entail any loss in proving ability. These techniques have been deployed for almost two years, and their efficiency is instrumental to the continued success of our experimental system.

Identifying and resolving policy misconfigurations A misconfiguration in access-control policy can result in a legitimate access being delayed or denied. This can be highly annoying to users, and may have severe consequences if timely access is critical. We show how rule mining techniques may be applied to access logs to identify potential misconfigurations in policy before they result in an access being denied. We also describe a technique by which past user behavior can be utilized to direct requests for policy corrections to the appropriate administrator. Using data collected from our testbed, we show that these techniques can correct a significant faction of misconfigurations before they impact a legitimate access, and that we can detect most of the discrepancies between the policy that an administrator implemented and the policy that was intended.

The three components of this thesis, when combined, represent a cohesive strategy for constructing proofs of access in an efficient and usable manner in a distributed access-control system that is implemented using formal logic. To our knowledge, this thesis represents the first such strategy to effectively address the requirements of such a system that is deployed and in active use.

Acknowledgements

This dissertation does not represent my work alone, but also the work of my family and countless friends and teachers who have helped me over the years. Here, I will attempt to thank those who most directly contributed to the success of my thesis, but "attempt" is the key word; I will surely omit someone to whom I am grateful.

My greatest debt of gratitude is to my parents, who taught me to study hard and finish what I start, always ensured that I had what I needed to succeed, endured my propensity to "argue with the sign post," and never let me settle for less than I was capable of. I hope that when given the opportunity to raise children, I will do the same.

Ginger, you have been a constant source of support throughout graduate school. You were there to listen when I needed someone to talk to, and you were there to give advice when I needed it. This medium is insufficient to express my gratitude.

My advisors, Mike Reiter and Lujo "The Hyphenator" Bauer, played an instrumental role in the development of my research career. Since arriving at CMU, I have become more adept at working on challenging problems, and they should receive much of the credit for this progression. I appreciate the countless hours they have spent reading drafts and offering candid feedback. Furthermore, their guidance has greatly increased the diversity of punctuation present in this thesis.

This thesis builds upon the Grey system, and as such, would not have been possible without the many people who contributed to the development of Grey. In particular, I would like to thank Jason Rouse for the long, and exceedingly frustrating, hours he spent making the initial prototype operational.

Lastly, I would like to thank all of my friends for making life in Pittsburgh enjoyable. Thanks to y'all (ahem, yinz), I now enjoy Pennsylvania beer and am the proud owner of a Terrible Towel. French fries on sandwiches, however, are still disgusting. Ahren, Casey, Jon, and Kathleen: thank you for always welcoming me into your homes and letting me eat your food. To everyone I hung out with, particularly Ahren, Bryan, Cody, Jim, Jon, and Mike A., thank you for providing a forum for research advice, tech support (or, more accurately, tech venting), car talk, and general hilarity.

Chapter 1

Introduction

The need to restrict access to precious resources is a common theme throughout human history. However, the recent proliferation of advanced computing technology has drastically increased both the number and variety of the resources that must be protected. This, in turn, makes it increasingly difficult to effectively create and enforce access-control policy. In particular, an access-control system must often guard access to resources that are distributed across a network and governed by policy that is created by multiple entities. The need for distributed policy creation can be the result of scale, i.e., there are too many resources for a single person to administer, or because the resources belong to a federation in which no single entity can dictate global access-control policy.

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., [26, 50]), recent work has imported logic into the system as a means to *implement* access control (e.g., [18]). In these systems, the resource monitor evaluating an access request requires a proof, in formal logic, that the access satisfies access-control policy. The use of formal logic offers several advantages, namely, that it enables an unambiguous specification of access-control policy, and that a proof conveys a greater degree of confidence in the correctness of an access-control decision.

In such a proof, digitally signed credentials are used to instantiate formulas of the logic (e.g., "K_{Alice} $\mathbf{signed\ delegate}(\mathsf{Alice}, \mathsf{Bob}, \mathsf{resource})$" or "$K_{\mathsf{CA}}$ $\mathbf{signed}$ K_{Alice} $\mathbf{speaksfor}$ $K_{\mathsf{CA}}.\mathsf{Alice}$"), and then inference rules are used to derive a proof that a required policy is satisfied (e.g., "$\mathsf{manager}$ $\mathbf{says}$ $\mathbf{open}(\mathsf{resource})$"). 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., [26, 57, 41, 36, 20, 30, 40]). Second, and of primary concern in this thesis, it must be possible to efficiently find proofs for accesses that should be allowed. Granting timely access is critical to a positive user experience; any unnecessary delays will damage users' perception of the system. As we will see in Chapter 3, the application of straightforward proof-search techniques to our scenario is so inefficient as to be unusable.

In the context of a practical distributed access-control system, there are several factors that make it difficult to efficiently construct a proof. First, the credentials that encode access-control policy are created by multiple entities and distributed throughout the system. The proof-construction process must therefore be capable of identifying and retrieving the needed credentials. Requests to distant parties to supply the needed credentials take time (especially if human interaction is required), so the proof-construction process must be designed to minimize the number of requests necessary to determine if access should be granted.

Second, the proof-construction process must consider human factors. A request for assistance from another principal may carry a social cost, and should be made only if absolutely necessary and with user approval. Similarly, as access-control policy often contains omissions, the proof-construction process should be able to recognize situations in which the creation of a new credential would enable the construction of a proof, and present the user with options for creating this credential. When there are different ways of proceeding in this manner, the user should be allowed to select which path to follow. This fundamentally alters the order in which the prover must explore the search space.

Third, access-control policy can contain omissions that prevent the construction of a proof for a legitimate access from existing credentials. While the proof-construction process may be able to suggest a credential that would resolve this omission at the time of access, this would involve an administrator's intervention, and the ensuing delay could be frustrating to the user attempting to gain access. Where possible, the system should detect such policy misconfigurations and prompt the appropriate administrator to resolve them before they result in a legitimate access being delayed or denied.

The objective of this thesis is therefore to describe a collection of new techniques for efficiently demonstrating that access should be granted in a distributed access-control system based on formal logic. Rather than devising a proving strategy customized to each application, we would prefer to develop a general proof-building strategy that is effective in a wide range of applications. Our approach consists of three components: (1) distributed proof construction to reduce the number of times a remote party must be contacted to construct a proof, (2) techniques for incorporating human guidance into the proof-construction process in a way that does not result in dramatic increases in computation, and (3) the use of data mining techniques to identify and resolve misconfigurations in access-control policy before they result in legitimate access being delayed or denied. The following thesis statement summarizes our contributions.

> **Thesis statement:** Access-control systems that require each request to include a formal proof of compliance with access-control policy can be made efficient enough for practical deployment through the use of distributed proving algorithms. These proof-construction algorithms leverage human intuition and provide guidance to users when policy changes are necessary.

We validate our techniques using experience and data drawn from the Grey system [13], which is 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 at Carnegie Mellon University for over two and a half years, 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. To our knowledge, the techniques presented in this thesis represent the first cohesive strategy for satisfying the requirements detailed in the preceding paragraphs.

1.1 Distributed Proof Construction

In a distributed access-control system, it is often necessary to request credentials from remote parties to demonstrate that access should be granted. As described above, these requests may be time consuming, and it is therefore desirable to minimize the number of such requests needed to construct a proof. In Chapter 2, we introduce a distributed proof-construction strategy and show that this strategy is more efficient than prior approaches in terms of network requests. Prior to our initial publication of these techniques [14], all works of which we are aware employed what we call an *eager* strategy, in which the party that needs to submit the proof[*] (the reference monitor or requesting client) generates it singlehandedly, retrieving only credentials from others when necessary. Instead, here we advocate a *lazy* strategy, in which a party attempting to gain access to a resource enlists the help of others to prove particular subgoals in the larger proof—versus merely retrieving credentials from them—yielding a proof that is assembled in a more distributed fashion.

[*]In contrast to our goals here, most systems do not submit a formal proof, but (possibly informal) evidence that a request should be granted.

A variant of this distributed approach to proof construction was concurrently proposed by Minami and Kotz [62] in the domain of context-sensitive authorization and subsequently adopted by PeerAccess [72]. Section 2.4 describes these works in further detail.

There are compelling reasons to depart from the eager strategy employed in previous works. Fundamentally, the eager strategy places a burden on the proof-construction strategy to request credentials without knowledge of what credentials are available or will be signed. As such, in systems where authority may be delegated dynamically and at user discretion, an eager strategy may request a credential from a user that the user will be unwilling to sign because it conveys too much authority, or that conveys too little authority and so dooms the user to be interrupted again later. For example, an access-control policy requiring $\mathsf{Alice}\ \textbf{says}\ \textbf{action}(X)$ in order to perform X (e.g., open a door) can be satisfied by a request $\mathsf{Bob}\ \textbf{says}\ \textbf{action}(X)$ if Alice signs $\mathsf{Bob}\ \textbf{speaksfor}\ \mathsf{Alice}$. However, as this conveys far more authority to Bob than merely the authority to perform X—namely, the ability to perform *any* action on behalf of Alice—Alice may refuse to sign it. Similarly, asking Alice for a weak credential, e.g., $K_{\mathsf{Alice}}\ \textbf{signed}\ (\mathsf{Bob}\ \textbf{says}\ \textbf{action}(X) \supset \mathsf{Alice}\ \textbf{says}\ \textbf{action}(X))$, precludes Alice from making more general statements that will save her from being interrupted later to approve another action Y for Bob. For example, Alice might instead add Bob to a group (e.g., $K_{\mathsf{Alice}}\ \textbf{signed}\ (\mathsf{Bob}\ \textbf{speaksfor}\ \mathsf{Alice.Students})$) to which she has already delegated the right to perform X (e.g., $\mathsf{Alice}\ \textbf{says}\ (\mathsf{Alice.Students}\ \textbf{says}\ \textbf{action}(X) \supset \mathsf{Alice}\ \textbf{says}\ \textbf{action}(X))$) as well as other actions. From this, Alice can then assemble a *proof* of $\mathsf{Alice}\ \textbf{says}\ (\mathsf{Bob}\ \textbf{says}\ \textbf{action}(X) \supset \mathsf{Alice}\ \textbf{says}\ \textbf{action}(X))$, which is exactly what was needed. More importantly, Alice need not be contacted the next time Bob needs to prove access to a resource to which $\mathsf{Alice.Students}$ are authorized.

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

1.2 Techniques for Efficient Proof Construction

In many cases, the credentials known to the principal attempting to gain access to a resource may be insufficient to prove that the access is authorized. The needed credentials may exist only on a distant node, or they may not exist at all due to an omission in the specified access-control policy. The prover's behavior in such situations can have a dramatic impact on the users' perception of the system.

For example, if Bob requests that Alice assist in the proof-construction process, the request may ultimately incur a social cost by, e.g., interrupting Alice during a meeting or waking her in the middle of the night. As a result, some requests cannot be made without human approval. Furthermore, if there are principals other than Alice who could also render assistance (e.g., the head of Alice's department), Bob may possess insight as to which principal is most appropriate to contact. This implies that the proof-construction strategy should incorporate human guidance so as to avoid following avenues that are unlikely to succeed or are socially unacceptable.

If Alice's policy prevents her from constructing a proof in response to Bob's request, the prover could simply fail. This, however, does not tell the Alice why the process failed, or what credential she must create to authorize Bob's access. Instead, it is preferable for the prover to identify the needed credential and suggest it to Alice. This helps Alice understand why the process failed, and gives her a means of implementing the needed policy if she desires.

Additionally, the prover should not solicit input from the user if a proof can be constructed from locally known credentials. The rationale for this requirement is obvious. However we will see that, when combined with the other requirements, it significantly impacts the manner in which the must prover explore the space of possible solutions.

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. Building upon the distributed proving framework introduced in Chapter 2, we show that our strategy offers dramatic improvements in the efficiency of proof construction relative to straightforward implementations, consequently making such systems significantly more useable in practice. As before, 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.

We evaluate the performance of our algorithm on policies drawn from our deployment and on larger, synthetically generated policies in Section 3.6. Additionally, we show empirically 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 3.7.

1.3 Detecting and Resolving Policy Misconfigurations

At some point, each of us has had a request to access a resource be denied that should have been granted. Such events are typically the result of a misconfiguration of access-control policy. Resolving these misconfigurations usually involves a human user to confirm that policy should be modified to permit the requested access. Misconfigurations are therefore often disruptive and time-consuming, and can be especially frustrating if the person who can change the policy cannot be reached when access is needed.

As a result, identifying and correcting misconfigurations before they result in the denial of a legitimate access request is essential to improving the usability of any access-control system. Eliminating all such misconfigurations in advance of accesses is arguably an unattainable goal (unless we invent a technique for reading users' minds). In Chapter 4, we set out to show, however, that *many* such misconfigurations *can* be eliminated in advance. Eliminating a misconfiguration is a two-step process: first we must identify a potential misconfiguration, then attempt to resolve it by contacting the most appropriate human.

Identifying misconfigurations Intuitively, identifying misconfigurations is possible because in most practical settings there is significant similarity in the policy that governs access to related resources. Consequently, the history of permitted accesses may shed light on which accesses that have not yet been attempted are likely to be consistent with policy.

The method we explore for identifying access-control misconfigurations is a data-mining technique called *association rule mining* [3]. This technique enables the inference of if-then rules from a collection of multi-attribute records. Intuitively, rule mining identifies subsets of attributes that appear in multiple records. These subsets are used to construct rules that suggest that if all but one of the attributes of a subset are present in a record, then the last attribute should also be present. We employ association rule mining to identify potential misconfigurations in access-control policy from a global view of access logs by representing each resource that is accessed as an attribute, and the set of resources accessed by an individual as a record. Records for which the mined rules do not hold represent potential misconfigurations.

Resolving misconfigurations Once a potential misconfiguration has been identified, we have to resolve the misconfiguration, which entails determining which human is best able to correct the access-control policy. In systems where policy is governed by a single administrator, this process is straightforward. In a distributed access-control system, however, it may not be clear which of potentially many users would be willing or able to extend the policy to grant the access. Since user interaction has a cost (in time and user aggravation), our technique must balance the desire to proactively resolve a misconfiguration with the desire to avoid unnecessary user interaction.

Our proposed resolution technique again relies on past user behavior; specifically, we determine which users have in the past created policy with respect to the particular user or resource in question, and suggest to those users that they correct the identified misconfiguration. In this way the misconfigurations can be resolved before they inconvenience the users that they affect. Compared to more reactive methods that attempt to resolve a misconfiguration only after an access that should be allowed fails, our technique can drastically reduce time-of-access latency and even the total time users spend interacting with the system without increasing the number of interruptions.

We evaluate our techniques using 16 months of data drawn from the Grey deployment. As expected, the performance of the methods we explore can be tuned to achieve a desired tradeoff between success in detecting and guiding the repair of misconfigurations, and the inconvenience to users of suggesting incorrect modifications to policy. For a particular, reasonable set of parameters, we correctly identify 58% of intended, but not yet implemented policy, i.e., 58% of the misconfigurations in the implemented policy. Using these predictions, our technique for resolving misconfigurations is able to proactively implement the needed policy for 44% of accesses that would otherwise have incurred a costly time-of-access delay. Each such correction results in significant savings in time-of-access latency. These gains are achieved without increasing the total time users spend interacting with the system.

1.4 Related Work

Distributed authorization has received considerable attention from the research community. Much of the related research, however, revolves around formalizing and analyzing the expressive power of authorization systems (c.f., [1, 6, 33, 58]), and only a fraction of it addresses the practical details and strategies for distributing and collecting credentials. Here we survey the landscape of related work by providing an overview of works that are broadly applicable to the entire thesis. We compare the technical details of our techniques to these works and to additional works that are related to an individual technique in the chapters in which the techniques are presented. Where appropriate in this section, we refer to a certificate as the digitally signed representation of a credential.

Taos The Taos operating system made two main contributions to distributed access control [73]: its access-control mechanism was inspired by a formal logic [2, 50]; and its access-control mechanism was built in at the OS, rather than application, level. The former quality inspired a greater degree of trust in the well-foundedness, and therefore correctness, of the implementation. The latter allowed the notion of identity to be embedded at a lower level, making it easier, for example, to reason about the security of communication channels within the OS.

In Taos, authority is initially derived from login credentials, and then partially or fully delegated via secure channels to other processes. A credential manager module builds, checks, and stores the credentials as they are passed around. A component of the operating system called the authentication agent determines whether a requesting process has the right to execute a particular action by querying the credential manager and referring to access-control lists (ACLs). A trusted certification authority (CA) maintains the mappings between cryptographic keys and the names used in ACLs.

PolicyMaker and KeyNote PolicyMaker [24] is a trust-management framework that blurs the distinction between policies and credentials by expressing them both as (possibly signed) programs. Determining whether a policy is satisfied involves executing the policy and the supplied credentials. Execution is local to the entity that is trying to verify whether a request is valid.

In the general case, allowing credentials to include arbitrary programs causes the evaluation of these credentials to become potentially intractable. However, by imposing constraints on credentials (in particular, by requiring each to be executable in polynomial time, monotonic, and authentic) it is possible to specify a polynomial-time algorithm for determining whether a set of credentials satisfies a policy [25]. These and other constraints led to the creation of KeyNote [23], which refines the ideas of PolicyMaker into a more practical system. The Strongman [48] architecture provides a means of enforcing policies specified in KeyNote in a distributed system.

Proof-Carrying Authorization The logical foundation of the Grey system is Proof-Carrying Authorization (PCA). Appel and Felten proposed PCA as a distributed authorization framework that uses a higher-order logic as a language for defining arbitrary application-specific access-control logics [6]. The underlying higher-order logic allows the application-specific logics to be remarkably expressive, but its lack of a decision procedure requires clients to construct and submit a proof that their request satisfies access-control policy. At the same time, proofs of access constructed in any application-specific logic can easily be verified by a simple, general checker [7]. The application-specific logic used by Grey is described in Chapter 2. In addition to Grey, PCA has also served as the foundation for an access-control system for web pages [18] and as a means for allowing disparate public key infrastructure technologies to interoperate [54].

Placeless Documents Balfanz et al. have developed a distributed access-control infrastructure for Java applications [8], one of the first implemented systems to be built around a sound formal core. Requests to access resources are accompanied by certificates that can be used to verify the validity of the request. The system does not specify, however, how certificates are collected or how a requester determines which certificates should be attached to a particular request; this is a focus of this thesis.

SD3 and QCM SD3 [45] is a trust-management system that further develops the idea of automatically distributing and fetching certificates that was introduced in QCM [39]. SD3 is implemented as middleware, shielding users from the details of using cryptographic primitives and certificate distribution. Unlike many other distributed authorization systems, but similarly to our approach, it produces easily verifiable proofs of access—this makes it possible for a potentially complex credential-collection algorithm to reside outside of the system's TCB. An SD3 query evaluator automatically fetches remote certificates needed to fulfill access requests. In addition, it allows certificates to be requested using wildcards and caches remote certificates after they have been fetched.

SPKI/SDSI SPKI 2.0 [34], a merger of the SPKI [33] and SDSI [66] efforts, is a digital-certificate scheme that inherits the binding of privileges to keys proposed in SPKI and the ability of SDSI to support locally defined namespaces. SPKI certificates are represented as tuples, and can bind names to keys, names to privileges, and privileges to keys. The authorization process for SPKI involves verifying the validity of certificates, translating the uses of names to a canonical form, and computing the intersection of the privileges described in authorization tuples.

SPKI has been modeled in formal logic [1, 57, 41], and has been implemented as an access-control mechanism for web pages [28, 61]. In the implemented system, the web server presents a web browser with the ACL protecting a requested page. It is the browser's responsibility to provide the server with a set of certificates which can be used to verify the browser's authority. The Greenpass system [38] modifies existing access-control methods for wireless networks to use SPKI credentials to allow users to delegate access to visitors.

Automated trust negotiation The scenario in which automated trust negotiation (ATN) protocols (e.g., [55, 21]) operate differs from that of standard authorization protocols in that the principal requesting access may not be willing to reveal particular credentials (or individual attributes of those credentials) to the resource monitor. The principal requesting access maintains a policy that governs what sensitive attributes may be revealed, as well as to whom and under what conditions they may be revealed. An ATN protocol must not only determine if the requesting principal has sufficient authority to access the resource, but also if the resource monitor is authorized to view all of the credentials that demonstrate that the access request should be granted. All ATN protocols of which we are aware support negotiations involving only two parties (the requesting principal and the resource monitor).

Context-sensitive authorization The objective of context-sensitive authorization (cf. [4]) is to permit access-control systems to make decisions on the basis of contextual information, such as the user's location. The sources of this information may be distributed across distant components of the system, and similarly to ATN, some of this information may be sensitive and should not be disseminated widely. Minami and Kotz describe a distributed algorithm for constructing a proof that access should be granted [62]. The Minami and Kotz technique allows sensitive credentials to be encrypted so that they can only be seen by the party in possession of the corresponding decryption key. This limits the disclosure of the credential as long as the recipient is trustworthy. The drawback to this approach is that the validity of a proof may not be self-evident; in situations where credentials are only disclosed to selected parties, these parties must be trusted to correctly derive facts from the private credentials without revealing the derivation itself. This implies that the proof construction algorithm must be part of the trusted computing base, whereas in the PCA approach, only the proof checker need be trusted.

Alternative access-control logics While a complete taxonomy of research in access-control logics is outside the scope of this thesis, we survey a few here. RT [58, 56] is a family of languages based on constraint Datalog that blend several desirable properties of role-based access control [68] and trust management [25]. In particular, Li et al. developed a centralized algorithm for discovering credential chains (the equivalent of proofs in our terminology) in an environment with a distributed credential store [59]. Li et al. define Datalog with constraints, which improves upon previous Datalog-based languages by allowing, among other things, the definition of structured resources (such as a folder hierarchy).

Becker and Sewell designed Cassandra [19], which is language that is also based on Datalog with constraints. They illustrate the practical significance of Cassandra's expressiveness by modelling policies for a national health-care system. SecPAL [20] demonstrates that further gains in expressiveness are possible using Datalog with constraints. Dillaway presents an access-control system for grid computing that was implemented using SecPAL [31]. DKAL [40] is an authorization language expressed in existential fixed-point logic rather than Datalog with constraints. This allows DKAL to be strictly more expressive than SecPAL without increasing the complexity of query evaluation. The focus of SecPAL and DKAL is the development of a new language; as such, they rely on external mechanisms to ensure that the needed credentials are present to evaluate a query.

In contrast to most authorization logics, which reason about time using an external mechanism, DeYoung et al. describe an authorization logic that explicitly models time [30]. They show that existing PCA-based systems can be implemented in their logic. However, the impact on efficiency of considering time during the proof search is not explored.

1.5 Thesis Structure

This thesis is structured as follows. Chapter 2 introduces and evaluates our distributed approach to proof construction. These techniques were previously published [14]; this thesis expands upon those results through additional analysis found in Section 2.3.7. Chapter 3 introduces and evaluates our techniques for efficiently constructing a proof in a manner that addresses several usability requirements. These techniques were previously published [15, 16]; this thesis includes additional evaluation regarding tabling (Section 3.6.3) and large caches (Section 3.6.4). Chapter 4 introduces and evaluates our techniques for detecting and resolving misconfigurations. These techniques were previously published [17]; this thesis additionally measures the performance of these techniques using partial data in Section 4.3. Chapter 5 contains concluding remarks.

Chapter 2

Distributed Proof Construction

In this chapter, we present a distributed algorithm for assembling a proof that a request satisfies an access-control policy expressed in a formal logic, in the tradition of Lampson et al. [50]. In prior work, a single party constructed the entire proof and requested credentials from other parties as needed. This lead to scenarios in which the party constructing the proof must request credentials without knowledge of what credentials exist or what credentials a remote party may be willing to create. We refer to this approach as the *eager* strategy. In contrast, here we propose a distributed *or lazy* strategy in which multiple parties collaborate to construct a proof. We show analytically that the lazy strategy succeeds in assembling a proof whenever a prover utilizing the eager strategy would do so. In addition, we show empirically that our algorithm significantly reduces the communication required to assemble a proof, which has direct implications in terms of both scalability and usability. We show that when combined with additional optimizations including caching and *automatic tactic generation*, which we introduce here, our algorithm retains its advantage, while achieving markedly improved performance. The techniques presented in this chapter serve as a foundation for the efficient, usable proving techniques that are used daily in the Grey deployment and are presented in Chapter 3.

The remainder of this chapter is structured as follows. We cover background in access-control logics and tactical theorem proving in Section 2.1. We detail our approach to distributed proof generation in Section 2.2. We evaluate our approach empirically and introduce optimizations including caching and automatic tactic generation in Section 3.6. We expand upon our previous publication of these results [14] through additional discussion of automatic tactic generation (Section 2.3.5) and practical considerations of our techniques (Section 2.3.7). We discuss related work in Section 2.4. We conclude in Section 2.5.

2.1 Background

To be able to precisely discuss the constructions of proofs of access, we first need to define a logic that will allow us to describe our access-control scenarios. The access-control logic we will use is straightforward and developed in the style of Lampson et al. [50]. However, we emphasize that our techniques are not specific to this logic.

2.1.1 Access-Control Logic

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

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

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

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

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

where s ranges over strings and p principals.

The formulas of our logic describe principals' beliefs. If Alice believes that the formula F is true, we write Alice **says** F. To indicate that she believes a formula F is true, a principal signs it with her private key—the resulting sequence of bits will be represented with the formula `pubkey` **signed** F.

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

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

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

$$\begin{aligned}\phi &::= s\ \textbf{signed}\ \phi' \mid p\ \textbf{says}\ \phi' \\ \phi' &::= \textbf{action}\ (s,\ s) \mid p\ \textbf{speaksfor}\ p \mid \\ &\quad\ \ \textbf{delegate}(p, p, s)\end{aligned}$$

where s ranges over strings and p principals.

Note that the **says** and **signed** predicates are the only formulas that can occur at top level. The inference rules for manipulating formulas are straightforward and are as follows:

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

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

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

$$\frac{A\ \textbf{says}\ (B\ \textbf{speaksfor}\ A.S) \quad B\ \textbf{says}\ F}{A.S\ \textbf{says}\ F} \qquad (\textsc{speaksfor-e}2)$$

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

While simple, this logic is sufficiently expressive to represent many practical policies. Bauer et al. found that when controlling access to physical resources, users were able to implement policies using this logic that more accurately represented their intentions than they could using traditional keys [11].

2.1.2 Tactical Theorem Provers

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

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

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

2.2 Distributed Proof Generation

2.2.1 Proving Strategies

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

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

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

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

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

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

In the lazy strategy, then, as soon as Alice's theorem prover produces a subgoal of the form A **says** F, Alice asks the node A (in the above example, Bob) to prove the goal for her. In other words, Alice is lazy, and asks for assistance as soon as she finds a subgoal that might be more easily solved by someone else. In Section 3.6 we demonstrate empirically the advantages of the lazy strategy.

Our prover assumes a cooperative environment in which a malicious node may easily prevent a proof from being found or cause a false proof to be generated. Our system adopts the approach of prior work (e.g., [6, 45]), in which the reference monitor verifies the proof before allowing access, which means that these attacks will merely result in access being denied.

2.2.2 A General Tactical Theorem Prover

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

Our proving algorithm, which is derived from a standard backward chaining algorithm (e.g., [67, p.288]), is shown in Figure 2.1. The proving algorithm, bc-ask, takes as input a list of goals, and returns either failure, if all the goals could not be satisfied, or a substitution for any free variables in the goals that allows all goals to be satisfied simultaneously. The algorithm finds a solution for the first goal and recursively determines if that solution can be used to produce a global solution. bc-ask proves a goal in one of two fashions: locally, by applying tactics from its knowledge base (Figure 2.1, lines 15–20); or remotely, by iteratively asking for help (lines 10–14).

The helper function subst takes as parameters a substitution and a formula, returning the formula after replacing its free variables as described by the substitution. compose takes as input two substitutions, θ_1 and θ_2, and returns a substitution θ' such that subst(θ',F) = subst(θ_2, subst(θ_1, F)). rpc$_l$ takes as input a function name and parameters and returns the result of invoking that function on the machine with address l. We assume that the network does not modify or delete data, and that all messages arrive in a finite amount of time. unify takes as input two formulas, F_1 and F_2, and determines if a substitution θ exists such that subst(θ, F_1) = subst(θ, F_2), i.e., it determines if F_1 and F_2 can be made equivalent through free-variable substitution. If such a substitution exists, unify returns it. A knowledge base, KB, consists of a list of tactics as described in Section 2.1.2. determine-location decides whether a formula F should be proved locally or remotely and, if remotely, by whom. Figure 2.2 shows an implementation of determine-location for the lazy strategy; an implementation for the eager strategy can be obtained by removing line 1 and

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

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

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

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

10       if ($l \neq localmachine$)
11           while (($\alpha \leftarrow$ rpc$_l$(bc-ask(first($goals$), $\theta$, $failures'$))) $\neq \perp$)    /* make remote request */
12               $failures' \leftarrow \alpha \cup failures'$           /* prevent $\alpha$ from being returned again */
13               $answer \leftarrow$ bc-ask(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 (($\theta' \leftarrow$ unify($q, q'$)) $\neq \perp$)   /* determine if tactic matches first goal */

17                   while (($\beta \leftarrow$ bc-ask($P$, compose($\theta', \theta$), $failures'$)) $\neq \perp$)   /* prove subgoals */

18                       $failures' \leftarrow \beta \cup failures'$          /* prevent $\beta$ from being returned again */
19                       $answer \leftarrow$ bc-ask(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 2.1: bc-ask, our proving algorithm

removing the if-then clause from line 2. When bc-ask is operating as a centralized prover, determine-location always returns *localmachine*.

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

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

Figure 2.2: Algorithm for determining the target of a request

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

Note that this algorithm does not explicitly generate a proof. However, it is straightforward to design the goal and tactics so that upon successful completion a free variable in the goal has been unified with the proof [18].

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

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

For the full proof, please see Appendix A.1. Informally: By close examination of the algorithm, we show by induction that bc-ask explores the same proof search space whether operating as a centralized prover or as a distributed prover. In particular, the centralized and distributed prover behave identically except when the distributed prover asks other nodes for help. In this case, we show that the distributed prover iteratively asks other nodes for help (lines 10–14) in exactly the manner that a centralized prover would consult its own tactics (lines 15–20).

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

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

2.2.3 Distributed Proving with Multiple Tactic Sets

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

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

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

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

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

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

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

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

2.3 Empirical Evaluation

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

We implemented our proving algorithm in Prolog, taking advantage of Prolog's built-in backward chaining. We simulated a distributed prover using a single Prolog instance by augmenting each fact in the prover's cache with an extra field that indicates the node to which the fact is known. Our implementation maintains the identity of the "local" node; that is, the node that is currently attempting to construct a proof. Only a single node may be local at any given time, and the prover may only use cached entries known to the local node. When the prover makes a remote request, the identity of the local node is changed to the recipient of the request. We note that our techniques are specific neither to Prolog nor to our choice of tactics and could be implemented in other automated theorem proving environments (e.g., [65]).

2.3.1 Constructing a Policy

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

To remedy this problem, we first undertook to map the physical access-control policy for rooms in our department's building (Figure 2.3). The policy reflects the hierarchical structure of authority in our department, which, as hierarchy is a recurring theme in human

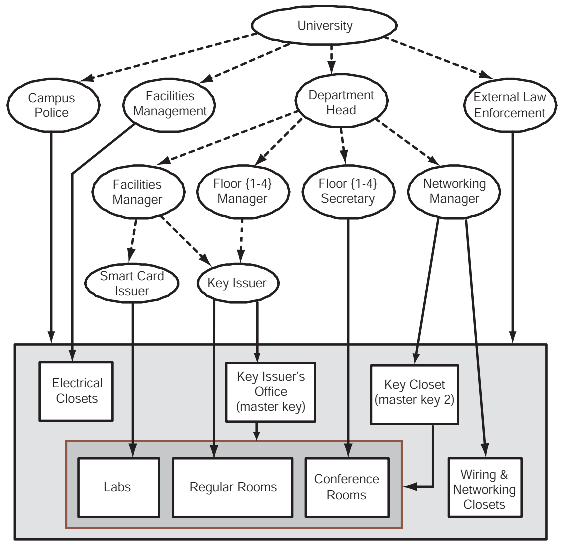


Figure 2.3: The authorization scheme for physical space in Hamerschlag Hall, home of the Carnegie Mellon Electrical & Computer Engineering Department

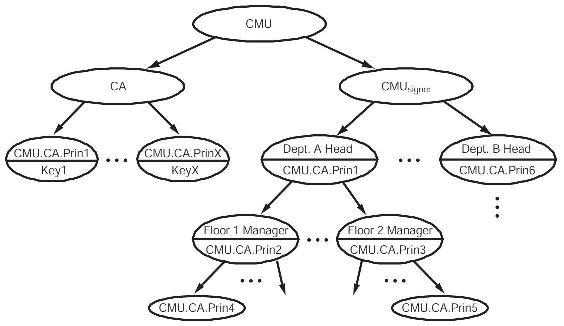


Figure 2.4: The authorization scheme for Hamerschlag Hall, modified for use in an digital access-control system

organization, leads us to believe that our policy approximates the policies employed by many other organizations. A close examination of this policy reveals that it contains elements that would be superfluous in a digital access-control system. For example, delegation of authority is conveyed either through physical tokens (the key issuer gives a user a key) or through the organizational hierarchy (the head of the department delegates to the floor manager the responsibility of managing access to all the rooms on a floor, but doesn't provide him with a physical token). In a digital access-control policy, delegation of authority is always explicitly represented; furthermore, in the digital domain it is unnecessary to have a policy include elements, such as the Key Issuer and Smart Card Issuer, whose sole purpose is the distribution of physical tokens. At the same time, a practical digital policy requires the mapping of keys to names. Universities typically have a registrar's office that performs similar bookkeeping; we add to the registrar the duties of a local certification authority. Another characteristic of physical access-control policies used in practice is the difficulty in maintaining the separation between users and the roles they inhabit (for example, the role of department head and the person who has that position). In a digital system, where delegation of authority is always explicit, this separation is easier to manage. Due to the importance of the university's key, we split it into a master key and a signing key. The private portion of the master key is intended to be kept offline and used only to periodically designate new signing keys. Figure 2.4 illustrates our derived policy.

We chose to structure the authorization hierarchy from the university to individual users as a complete tree. We describe a policy with a (j, k, l) tree to indicate that there are j department heads, k floor managers under each department head, and l users under each floor manager. We test our algorithms with several different (j, k, l) trees. We chose to use complete trees for simplicity only; the results obtained from unbalanced trees did not differ substantially. Specifically, we constructed 20 unbalanced trees with 253 principals each by randomly removing 216 nodes from a complete (3,5,30) tree. The performance of the initial access with both forms of caching enabled decreased by up to 4%, with an average decrease of 2%.

Each of the policies protecting a room requires that the university approve access to it (e.g., $\mathbf{key}(K_{CMU})$ **says action**(`room15`)). The proof that a user may access the room is based on a chain of certificates leading from K_{CMU} to the user himself. The proof also shows which inference rules (of the logic described in Section 2.1.1) need to be applied to the certificates and in what order to demonstrate that the certificates imply that access should be granted. Figure 2.5 shows an example proof that allows UserC to access *resource*, a resource controlled by K_{CMU}. In this example, the goal that must be proved is $\mathbf{key}(K_{CMU})$ **says action**$(\mathit{resource}, \mathit{nonce})$. $\mathcal{P}_1$–$\mathcal{P}_{11}$ represent the necessary certificates, and Lines 0–25 constitute the proof.

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

2.3.2 Evaluation Criteria

The primary criteria we use to evaluate the performance of the two proving strategies detailed in Section 2.2 is the number of requests made while attempting to construct a proof. Since requests in our system may ultimately cause an actual user to be queried to approve the creation of a certificate, the number of requests roughly approximates the required level of user interaction. Additionally, since much of the communication may be between devices with high-latency connections (such as cell phones connected via GPRS), the number of requests involved in generating a proof will be one of the dominant factors in determining the time necessary to generate a proof.

When running the simulations, the only principals who access resources are those located in the lowest level in the hierarchy. The resources they try to access are rooms on their

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

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

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

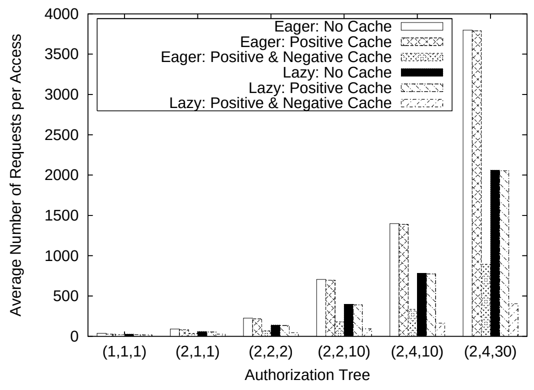
$\mathcal{P}_8 = K_{\mathsf{UserA}}$ **signed** $(\mathbf{delegate}(\mathbf{key}(K_{CMU}).DH_1,\ \mathbf{key}(K_{CMU}).DH_1.FM_1,\ \mathit{resource}))$
$\mathcal{P}_9 = K_{\mathsf{UserA}}$ **signed** $(\mathbf{key}(K_{CMU}).CA.\mathsf{UserB}$ **speaksfor** $\mathbf{key}(K_{CMU}).DH_1.FM_1)$

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

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

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

Figure 2.5: Proof of $\mathbf{key}(K_{CMU})$ **says** $\mathbf{action}(\mathit{resource},\ \mathit{nonce})$



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

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

Figure 2.6: Performance of initial access with different caching strategies

floor to which they are allowed access. Unless otherwise specified, the performance results reflect the average over all allowed combinations of users and resources.

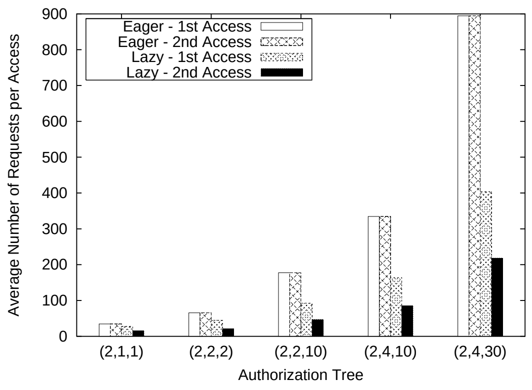
2.3.3 First Access

Figure 2.6 shows the average number of requests made by each proving strategy when first attempting to prove access to a resource. On average, lazy outperforms eager by between 25% and 45%, with the performance difference growing wider on larger authorization trees. However, the number of requests made is far too large for either strategy to be used in a practical setting. Upon further investigation, we discovered that more than half of all requests are redundant (that is, they are repetitions of previous requests), indicating that caching would offer a significant performance benefit.

Our initial intuition was to cache proofs of all successful subgoals found by the prover. However, as Figure 2.6 indicates, caching the results of successful proof requests offers surprisingly little performance benefit. We discovered that most of the redundant requests will, correctly, result in failure; that is, most of the redundant requests explore avenues that cannot and should not lead to a successful access. We modified the caching mechanism to cache failed results as well as positive results (also shown in Figure 2.6). This reduced the number of queries by up to 75% for both strategies.

2.3.4 Effects of Caching on a Second Access

Since all of the results discovered by the eager strategy are cached only by the principal who accessed the resource, the cache is of no benefit when another principal attempts to access a resource. The lazy scheme distributes work among multiple nodes, each of which can cache the subproofs it computes. In the lazy scheme, access of the same or a similar resource by a second, different principal will likely involve nodes that have cached the results of previous



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

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

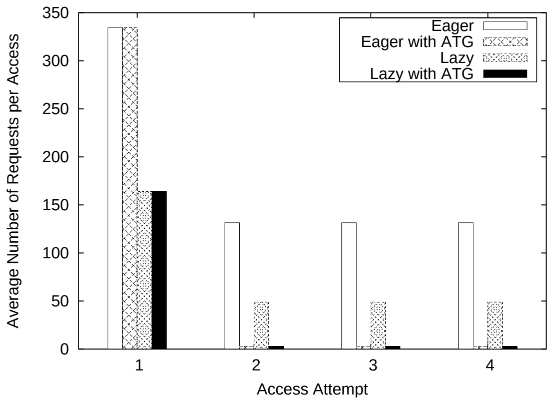
Figure 2.7: Performance of subsequent access to a different resource by a different principal

accesses. This enables the lazy strategy to take advantage of caching in a way that the eager strategy cannot, resulting in significant performance gains. To compute the average performance, we ran the simulation for every possible combination of principals making the first and second access. Figure 2.7 shows that the average case eager performance in the second access is identical to its performance in the first attempted, as expected. The figure also shows that caching on interior nodes in the lazy strategy decreases the number of requests made by the second access by approximately a factor of 2. The result is that lazy completes the second access with approximately one-fourth the number of requests of eager.

2.3.5 Automatic Tactic Generation

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

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



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

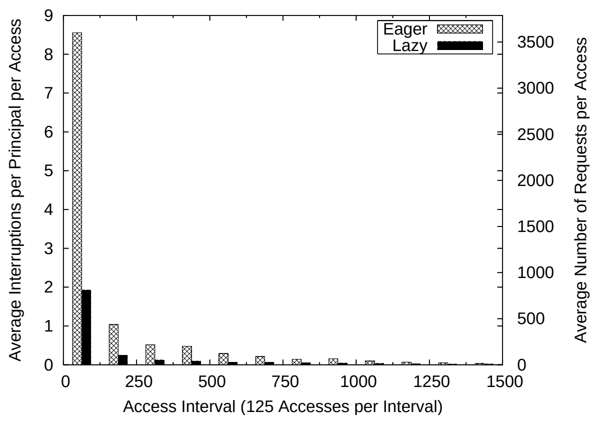
Lazy	No ATG		With ATG	
Access	Requests	STDEV	Requests	STDEV
1	164	88.2	164	88.2
2	49	26.2	3	0
3	49	26.2	3	0
4	49	26.2	3	0

Figure 2.8: Sequential access of four resources by same principal in a (2,4,10) tree

A common scenario in which automatic tactic generation is very useful is when attempting to access several rooms on the same floor. The policies protecting each of the rooms are likely to be very similar, since they probably belong to the same organizational unit and share the same administrator. Pure caching is not likely to help much because the rooms are all named differently, and proof of access explicitly mentions the resource for which it was generated. However, automatic tactic generation allows proofs to be computed very efficiently, as shown in Figure 2.8. After an initial access, ATG allows both the eager and the lazy strategy to complete subsequent proofs for different resources with a minimal number of requests. ATG is particularly effective in scenarios where proof search may result in requests to many principals. In scenarios where there are significantly fewer principals to ask, or users are cognizant of the shape of the policy, other heuristics or user intuition may be equally effective.

Should ATG fail to construct a proof, the standard proving algorithm must be run. In this case, each additional tactic constructed by ATG adds overhead to the proof search. As such, in scenarios where few proofs share a common shape, ATG could also produce a large number of tactics that are rarely useful, and the additional overhead incurred by applying tactics may negate the benefit of knowing the correct generalization. ATG should be disabled in scenarios where this problem occurs.

ATG can be viewed as a specific instance of *explanation-based generalization* (EBG) [64], which seeks to form generalized concepts from a single training example and domain knowledge. The contribution of ATG is to determine the scenarios in which it is useful to construct generalizations. We have found that generalizing proofs of intermediate subgoals produces such a large number of tactics that the overhead of considering each of these tactics as part of the proof search negates their benefit. As such, ATG constructs only the generalizations that we found to be useful, that is, generalizations of complete proofs of access.



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

Figure 2.9: Average of 10 simulations with 1500 random accesses in a (4,4,25) tree

In addition to reducing the number of requests, ATG can also improve the computational efficiency of proof search. As ATG abstracts the nonce from previous proofs, it can quickly find proofs for repeated accesses to the same resource using cached credentials. Thus, if the standard proof-construction strategy is inefficient, ATG can reduce the computation required by bypassing the inefficient search process. Our deployment benefited from ATG in this manner until it began relying on the techniques described in Chapter 3.

2.3.6 Simulating a User's Experience in a Deployed System

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

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

Figure 2.9 shows the performance of the system with all optimizations enabled, measured both as the average number of requests each principal has to answer per access attempt, and the total number of requests per access attempt. In this more realistic scenario, the lazy strategy continues to do well. During the first interval of 125 accesses, the lazy strategy is at least three times more efficient in the number of requests made. In subsequent intervals, caching and ATG quickly reduce the number of requests to minimal levels for both strategies. The greatest load on the system will occur in the initial stages of deployment, which is the stage at which the lazy strategy offers the greatest benefit.

2.3.7 Discussion

Several aspects of our distributed proving algorithm bear further consideration; we discuss them here.

Cache Size Larger caches require more memory and cause unification (our cache-lookup mechanism) to become slower. Here we consider the cache size of the root of the policy hierarchy (e.g., CMU in Figure 2.4), as it is likely to be among the largest of any cache in the system. This is an artifact of the lazy strategy; the root node is likely to receive a request for assistance corresponding to many first-time accesses. By providing assistance, the root node's cache will learn of the credentials needed to prove that each access is granted. The widespread use of ATG would reduce the number of proof requests directed to the root node; here we disable ATG so as to measure the worst-case scenario.

We measured the size of the root node's cache in a (10,10,50) tree (containing over 5000 users) after every user accessed all permitted resources. In this scenario, the root node's cache will contain 29,832 credentials, 4,910 positive facts, and 11 negative facts. The digital certificates that encapsulate the credentials would occupy approximately 37 MB of storage, based on an average certificate size of 1,302 bytes observed on our deployment's root node. However, the only logical representation of credential needs to be loaded into memory in order to construct proofs. Including both credentials and facts, the logical representation of the root node's cache (which must be resident in memory) occupies approximately 5MB.

An implicit assumption of this work is that requests for assistance are much slower than cache lookups. As cache lookups become slower as the size of the cache increases, there will ultimately be a point at which the cache becomes so large that this assumption will no longer hold. However, as requests for assistance may involve user interaction, we feel that this assumption will hold for systems of realistic size. We will perform a more in-depth analysis of the effects of caching after introducing the proof-construction techniques of Chapter 3.

Unnecessary requests As described, the positive cache contains only the proofs and corresponding credentials received in response to requests for assistance. It is possible that the credentials present in cache may be used to derive other formulas that are not cached. Since the lazy strategy never reasons about another principal's beliefs, a situation could arise in which Bob has the credentials necessary to prove CMU **says** F, but will nevertheless ask CMU for assistance. Chapter 3 discusses techniques that address this problem.

Applicability of negative cache Negative caching is useful in any scenario that involves redundant requests, but it is particularly effective in scenarios where the proof search is both exhaustive and automated, i.e., the scenario simulated above. However, in scenarios where requests for assistance are directed by humans (such as that of Chapter 3), there is likely to be less inefficiency in the search process. Humans may leverage their memory to avoid making identical requests, and their intuition to effectively direct queries for assistance and to determine when further search is futile. In such scenarios, negative caching could be used to discourage users from exploring avenues of proof search that have not been successful in the past, but a user should be given the option to override negative cache if she believes that a request that failed in the past is now likely to succeed (due to, e.g., the user receiving a promotion).

Consistency As with any form of caching, maintaining the consistency of the cache is a concern. Inconsistencies can arise in two ways: issued credentials are revoked, and new credentials are created that allow the derivation of formulas that were previously added to the negative cache. Mechanisms, such as the one presented by Minami and Kotz [63], can propagate updates though a system where cached entries depend on one another. However, this can only limit the duration of the cache inconsistency. This is a fundamental problem in distributed systems: the CAP theorem states that consistency, availability, and tolerance of network partitions cannot be achieved simultaneously [37]. We designed our system to remain available in the presence of partitions (due to, e.g., a device losing connectivity, a software crash, or an unavailable user), which implies that some degree of inconsistency must be tolerated. If requirements dictate that changes to the validity of facts be propagated instantly, the system can remain available only as long as all nodes are online.

If credentials are being rapidly issued and revoked, it is possible that a proof could be derived from credentials that were all valid at some point during the interval in which the proof was constructed, but were never valid simultaneously. Lee and Winslett refer to this as *incremental consistency* [53]. They provide algorithms that achieve stronger consistency guarantees through the use of commitments and additional validation steps. Lee et al. extend this work to provide consistency guarantees in context-sensitive environments where privacy concerns may result in portions of the proof tree remaining hidden [52]. Our system could incorporate these techniques in scenarios where incremental consistency is insufficient.

Human interaction An exhaustive search involves querying many principals that can potentially render assistance. Some principals, such as the root node or certification authority, are likely to be hosted on servers and to respond to requests automatically. However, many users may wish to respond to requests manually so that omissions in authored policy can be resolved before they lead to access being denied. As described in this chapter, an exhaustive search could involve interrupting many of these users with requests that they are unwilling to grant, thus diminishing the overall usability of the system.

In practice, this problem could be addressed by a combination of techniques. First, a user could indicate that the principal requesting assistance should not do so again, even for different resources or on behalf of other users requesting access. The principal requesting assistance would then create negatively cached facts general enough to describe the user's preference. Additionally, it is likely that the user attempting to gain access to a resource has some intuition as to which other user is most likely to render assistance. The proof-construction technique can incorporate this intuition to direct queries to the most appropriate principal without performing an exhaustive search. We have found this technique to be very effective in our deployment, though the increased user interaction poses new requirements as to how the prover should explore the search space on a single node prior to requesting assistance. We explore these requirements and our techniques for addressing them in the following chapter.

2.4 Related Work

In this section, we focus on the mechanisms by which related works make access-control decisions in a distributed system. These mechanisms fall, roughly speaking, into two categories: remote credential retrieval and distributed reasoning.

Remote credential retrieval Several existing distributed access-control systems support distributed knowledge bases by using a centralized algorithm to identify potentially useful credentials, then initiating queries to remote parties to retrieve those credentials if they exist. We highlight a few systems that utilize this approach here. Binder [29], PolicyMaker [24], and KeyNote [25, 23], 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. In fact, generalizing credentials in the style of PolicyMaker may as a side effect make it more difficult to determine how to go about locating a missing credential. Although credentials contain code to be executed and can be authored by different entities, the credentials are all collected by and executed in the local environment of the entity that is evaluating a policy. Hence, at evaluation time a credential cannot take advantage of any specialized knowledge present in the environment of the node on which the credential originated.

Placeless Documents defines a logic for enforcing access-control for distributed Java applications [8], but does not specify how remote credentials are to be retrieved. Once retrieved, however, credentials are cached by the recipient of the credential. SPKI is a syntax for digital certificates dealing with authentication [34] that has been used to implement access control for web pages [61] and wireless networks [38] in a manner that allows credentials to be collected from remote servers. RT is a language for defining trust where the provided search algorithm for evaluating access-control decisions is capable of retrieving remote credentials [59]. Winsborough and Li adapted this technique to retrieve credentials in an automated trust negotiation protocol [71].

SD3 is a distributed trust-management framework that extends the evaluation techniques of QCM [39, 46] and utilizes both a push and a pull model for migrating credentials between parties [45]. In addition, SD3 allows certificates to be requested using wildcards and caches remote certificates after they have been fetched. In this chapter, we investigate more powerful methods for fetching the needed certificates while allowing the authors of the certificates more control over which certificates are used.

Bauer et al. use the proof-carrying authorization framework [6] to guard access to web content; the client's theorem prover will retrieve remote credentials in the course of proof-generation as needed [18]. Cassandra provides a general framework for specifying distributed multi-domain policies and supports remote credential retrieval from locations specified in the credential [19]. SecPAL [20] and DKAL [40] improve upon the expressibility of Cassandra, while continuing to allow for the retrieval of remote credentials. The Strongman architecture specifies policy as KeyNote programs; credentials are stored on designated repositories, and clients download the needed certificates from the repository prior to requesting access [48]. Zhang et al. developed a usage-control authorization system for collaborative applications [78] in which 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 the works described above 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, which has received substantially less attention in literature, supports distributed reasoning about access-control policies. This approach allows the remote retrieval of subproofs rather than credentials, which we showed substantially reduces the number of remote requests necessary to complete a proof. Minami et al. use this approach for context-sensitive authorization queries [62], and extend it to provide distributed cache consistency in the face of certificate revocation [63]. 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 [54]. The notion of distributed proof construction is supported by PeerAccess [72] and MultiTrust [77]. PeerAccess relies on proof hints to direct queries, whereas MultiTrust relies on query routing rules. It is not clear how either proof hints or query routing rules would be created in a practical system. In contrast, our algorithm directs queries on the basis of the goal that the algorithm is attempting to prove.

2.5 Conclusion

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

The techniques presented here are designed for and evaluated in an environment where proof search is automated and an exhaustive search is desired. This does not, however, immediately yield a system that is practical in all scenarios. In particular, human interaction may be required to, e.g., request assistance, create a new credential, or terminate the search early. Chapter 3 describes the usability requirements imposed by such a scenario, and shows how they impact the manner in which the proof-construction algorithm should search for proofs. The distributed approach to proof construction presented in this chapter should be viewed as as a fundamental building block that enables us to develop the efficient, user-centric proof-construction strategies described in Chapter 3.

Chapter 3

Efficient Proving for Practical Distributed Access-Control Systems

In this chapter, 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. Our techniques build upon the distributed proof construction strategy described in Chapter 2 by specifying in detail how those techniques are to be used in environment where human factors weigh significantly upon proof construction. 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 the 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.

The contributions of this chapter are to: (1) identify the requirements of a proving algorithm in a distributed access-control system with dynamic credential creation (Section 3.1); (2) propose mechanisms for precomputing delegation chains (Section 3.4) and systematically generating tactics (Section 3.5.2); (3) describe a technique for utilizing these pre-computed results to find proofs in dramatically less time than previous approaches (Section 3.5); and (4) evaluate our technique on a collection of policies representative of those used in practice (Section 3.6.1) and those indicative of a larger deployment (Section 3.6.2). In addition to the results we previously published [15], this chapter evaluates our techniques in an environment with tabling (Section 3.6.3) and explores the potential side effects of our techniques (Section 3.6.4). In Section 3.7, we discuss the use of our techniques in the context of additional logics, systems, and applications.

3.1 Goals and Contributions

In this chapter, 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 sample logic described in Section 2.1.1. 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 3.7.

3.1.1 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 3.6.2). 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 [14] and [62]) 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) [14]. 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 3.5.2) in 2 seconds, and is able to find a provably complete set of options in well less than a minute.

3.1.2 Insights

We address the requirements outlined in Section 3.1.1 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 3.1.1.

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 3.5.1.

Precomputing delegation chains A second insight is to locally precompute and cache delegation chains using two approaches: the well-studied *forward chaining* algorithm [67] 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 3.6.2). As we discuss in Section 3.5.2, 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 [35]. 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.

3.2 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 3.3). Second, a path-compression algorithm (which we introduce in Section 3.4) 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 3.5).

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.

3.3 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 [67] 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 3 *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 [67]. Many access-control logics are a subset of, or can be translated into, Datalog (e.g., [45, 57]). 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: **key** and dot (**.**). The single inference rule that applies **key** is only able to match its premise against a credential. As a result, **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 [67].

3.4 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 sec-

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

PC(credential $f$)
    if (credToPath($f$) $= \perp$), return                /* If not a delegation, do nothing. */
    $(x, y) \leftarrow$ depends-on($f$)                   /* If input is a third-person
    if $(((x, y) \neq \perp) \wedge \neg((x, y) \in paths))$     delegation, add it to incompletePaths.*/
        $incompletePaths \leftarrow incompletePaths \cup (f, (x, y))$
        return

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

    foreach $(f', (x', y')) \in incompletePaths$          /* Check if new paths make any previously
        foreach $(p'', q'') \in paths$                       encountered third-person credentials
            if(($\theta \leftarrow$ unify($(x', y'), (p'', q'')$)) $\neq \perp$)     useful. */
                $(p', q') \leftarrow$ credToPath($f'$)
                add-path((subst($\theta, p'$), subst($\theta, q'$)))

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

    foreach $(p', q') \in paths$
        if(($\theta \leftarrow$ unify($q, p'$)) $\neq \perp$)     /* Try to prepend new path to
            $c \leftarrow$ (subst($\theta, p$), subst($\theta, q'$))     all previous paths. */
            $paths \leftarrow$ union($c, paths$)
            $newPaths \leftarrow$ union($c, paths$)

    foreach $(p', q') \in paths$
        foreach $(p'', q'') \in newPaths$                 /* Try to append all new paths
            if(($\theta \leftarrow$ unify($q', p''$)) $\neq \perp$)     to all previous paths. */
                $c \leftarrow$ (subst($\theta, p'$), subst($\theta, q''$))
                $paths \leftarrow$ union($c, paths$)
```

Figure 3.1: PC, an incremental path-compression algorithm

ond 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 3.1, 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 3.1). 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, union$((p, q), S)$ returns S if unify$(p, q) \neq \perp$, and $S \cup \{(p, q)\}$ otherwise.

3.4.1 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.2.

Theorem 4 *If* PC *has completed on KB, then for any A, B such that $A \neq B$, if for some F (B* **says** *$F \supset \ A$* **says** *F) in the context of KB, then (B* **says** *F, A* **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).

3.5 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.

3.5.1 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 Chapter 2 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 ($\textsf{bc-ask}_D$, shown in Figure 3.2) as a modification to the distributed backward chaining algorithm of Chapter 2. For ease of presentation, $\textsf{bc-ask}_D$ 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 bc-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.[*] 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.

bc-ask$_D$ operates by recursively decomposing the original goal into subgoals. The base case occurs when bc-ask$_D$ is invoked with an empty goal, in which case bc-ask$_D$ will determine if the current solution has been previously marked as a failure and return appropriately (Lines 3–4) of Figure 3.2. For non-empty goals, bc-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*), bc-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). bc-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, bc-ask$_D$ first performs the same function as the base case without recursing, then attempts to prove the remainder of the goals. If bc-ask$_D$ applied a tactic, it attempts to recursively prove the premises of that tactic (Line 19). If this results in a remote marker, bc-ask$_D$ suspends proving and returns (Line 21). Otherwise, bc-ask$_D$ attempts to recursively prove the remainder of the goals (Lines 22–23).

Locally creatable credentials We can extend bc-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 bc-ask$_D$ will attempt to complete the proof to determine if creating the credential will lead to a complete proof.

[*]In Figure 3.2, a marker is typed as a substitution—this is because bc-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 $\theta$,                             /* 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 $\perp$   /* $\theta$ 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 $\langle answer, [creds|q] \rangle$   /* 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 $\beta$ 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 3.2: $\mathsf{bc\text{-}ask}_D$, a delayed version of $\mathsf{bc\text{-}ask}$

Suspending proving for remote markers For theoretical completeness, $\mathsf{bc\text{-}ask}_D$ must suspend proving after the addition of a remote marker because 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.

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 A.3.

Theorem 5 *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.

3.5.2 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 3, 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 3.7.

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

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

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 3.3: Example construction of LR tactics from an inference rule

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.[†] 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 3.3 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. The particular right rule shown in Figure 3.3 can be applied repeatedly without making progress (as the first premise unifies with the conclusion of the rule). We employ rudimentary cycle detection to prevent repeated application of this rule without decreasing the the theoretical proving ability of our technique. In practice, each of these restrictions can be accomplished through additional parameters to $\mathsf{bc\text{-}ask}_D$ 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

[†]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.

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 [27] 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). We evaluate our techniques using tabling in Section 3.6.3.

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 A.4.

Theorem 6 *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.

3.6 Empirical Evaluation

The usability of the distributed access-control system as a whole depends on the timeliness with which it can generate a proof of access. Proof-generation time is influenced both by the number of requests for assistance and the amount of time it takes the local prover 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. The distributed proving techniques described in Chapter 2 coupled with the techniques described in this chapter for leveraging human intuition to direct queries are able to minimize the number of distributed proof requests necessary to construct a proof. Therefore, our evaluation focuses on the remaining metric: the time required by the local prover to either construct a complete proof or 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 in Chapter 2, which is indicative of a larger deployment. As explained in Section 3.6.2, 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/`).

3.6.1 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 3.4: 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)

Figure 3.5: Credentials on Charlie's phone

Policy The policy for this scenario is expressed in the credentials known to Alice and Charlie, shown in Figures 3.4 and 3.5. The first six credentials of Figure 3.4 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 access. The final credential is given to Alice when Charlie asks her for help: it indicates Charlie's desire to open door1.

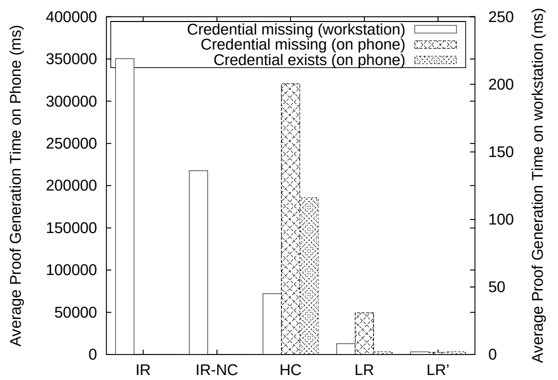


Figure 3.6: Alice's prover generates complete proof or list of credentials that Alice can create

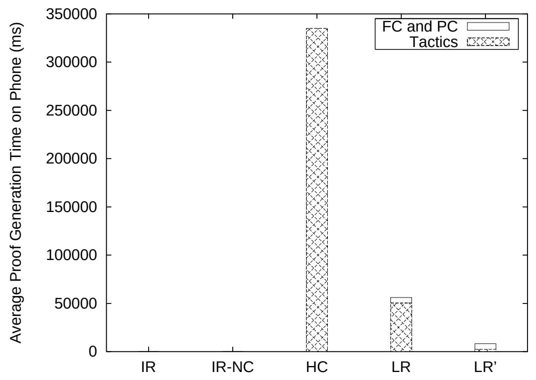


Figure 3.7: Aggregate proving time: Charlie's before help request + Alice's + Charlie's after help request

Charlie's policy (Figure 3.5) 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 3.4 and 3.5 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 3.6 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 3.6 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 3.7 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

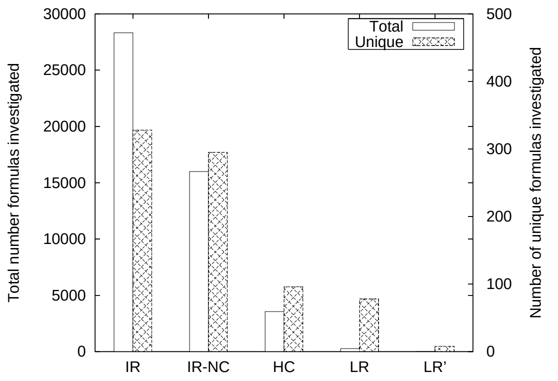


Figure 3.8: Number of formulas investigated by Alice

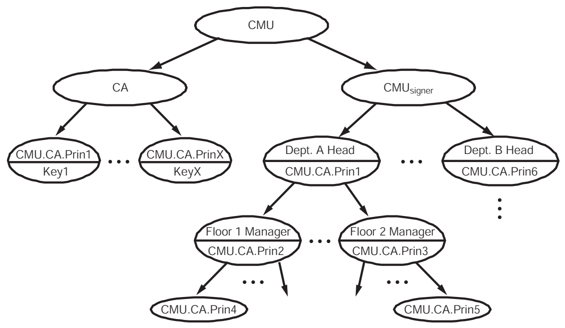

CMU
CA
CMUsigner
CMU.CA.Prin1 Key1
CMU.CA.PrinX KeyX
Dept. A Head CMU.CA.Prin1
Dept. B Head CMU.CA.Prin6
Floor 1 Manager CMU.CA.Prin2
Floor 2 Manager CMU.CA.Prin3
CMU.CA.Prin4
CMU.CA.Prin5

Figure 3.9: An access-control policy presented in Section 2.3.1

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 3.8 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.

3.6.2 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 Section 2.3.1. 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 3.6.1. 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 3.10 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 3.11. 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.

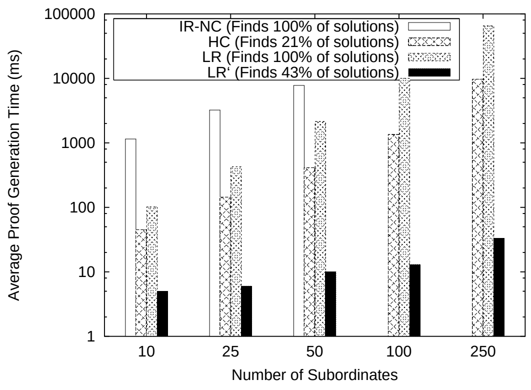


Figure 3.10: Proof generation in larger policies with missing credential

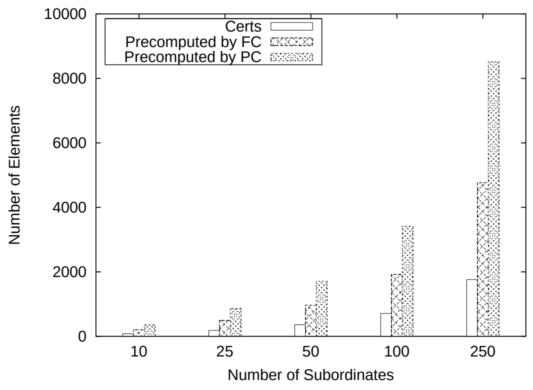


Figure 3.11: Size of knowledge base in larger policies with missing credential

3.6.3 Tabling Evaluation

Tabling is an alternative technique for eliminating redundancy within the local proof search [27]. When attempting to prove subgoal S, a tabled algorithm will attempt to find all possible solutions for S, and add the results to a table. Any subsequent attempts to prove S can then refer to the table rather than repeating the search process. Similarly, our LR tactics utilize results precomputed by forward chaining to avoid investigating subgoals at the time of access. However, our technique differs in that these results are computed prior to the time of access and that we additionally precompute all delegation chains using path compression. Due to the similarity of our techniques, it is worth investigating the benefit that our methods will provide in a tabled proof-search environment.

For this evaluation, we use XSB (`http://xsb.sourceforge.net/`) as our Prolog interpreter due to its support of tabling. All evaluations are run using the workstation described previously. With tabling enabled, we compare the two theoretically complete tactic sets, IR and LR. We omit IR-NC as tabling eliminates the cycles that IR-NC prevents. As described in Section 3.3, our logic is not a subset of Datalog; the SAYS-LN rule permits arbitrary expansion when explored in a backward fashion. This poses a problem for IR as it produces an infinitely large search space, even with tabling enabled. Therefore, it is not possible to use the inference rules as tactics in a tabled search environment without modification.

One way to prevent infinite expansion is with the depth-limiting mechanism described previously. However, the straightforward implementation of a depth-limiter involves an additional parameter to each proving predicate. This interacts poorly with tabling, as tabled results are then specific to a particular depth. Instead, we restrict the number of applications of SAYS-LN by limiting the degree to which a name can be nested. For example, if we restrict the degree of nesting to one, $A.B$ is a permissible name, whereas $A.B.C$ is not.

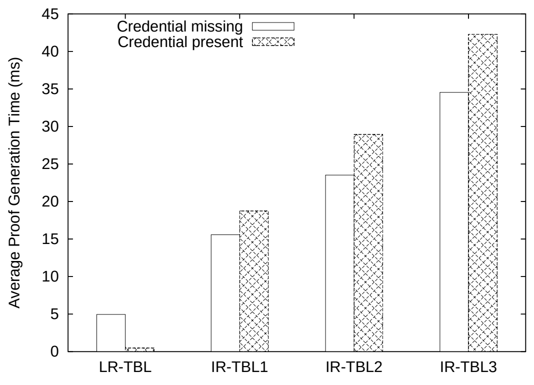


Figure 3.12: Proof-generation time in example policy with tabling enabled

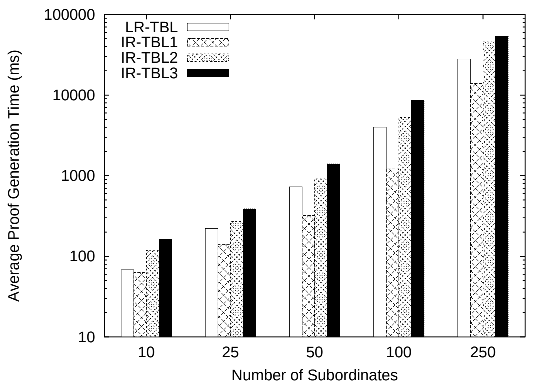


Figure 3.13: Proof-generation time in larger policies with missing credential and tabling enabled

IR-TBL1, IR-TBL2, and IR-TBL3 represent tactic sets that support nested names of degree one, two, and three.

Performance We first evaluate these strategies using the example policy drawn from our deployment, which is described in Section 3.6.1. Figure 3.12 describes the performance observed by Alice while she attempts to help Charlie. Figure 3.6 depicts the same scenario without tabling. Again, performance is shown for two situations: one in which Alice has not yet created the credential delegating authority to Charlie, and one in which she has. In both situations, we see that LR is dramatically faster than each IR variant. The gap is especially wide when the needed credential already exists, as for LR the vast majority of the work is done prior to the time of access, while IR, even with tabling enabled, must still derive the proof at the time of access.

Next, we evaluate these strategies using the larger policies described in Section 3.6.2 in the scenario where the needed credential is missing. Figure 3.13 shows that LR-TBL is more efficient than IR-TBL2 and IR-TBL3 despite the fact that the IR variants do not cover the entire search space of LR-TBL. The benefit of using LR tactics in these scenarios is significant, yet not as great as in the smaller example scenario. We hypothesize that this is because IR has to pay a greater initial penalty to populate the tables than does LR (due to the fact that LR can utilize precomputed results). The search space associated with the larger policies contains more opportunity to utilize tabled entries, thus the initial penalty is a much smaller percentage of the total computation.

However, the search space of IR-TBL1 is sufficiently constrained that it is quicker than LR-TBL. In the above scenarios, only one application of SAYS-LN is necessary and therefore IR-TBL1 is able to find all of the desired solutions. However, this does not mean that there is no meaningful loss in proving ability associated with restricting the use of SAYS-LN. For example, suppose that the department delegates authority to access the machine room to Dept.machine-room.staff, which is intended to describe all staff who should have access to the machine room. Suppose that Alice, who is in charge of the machine room, can speak on behalf of Dept.machine-room, but wishes to limit her authority on a day-to-day basis to the level of authority granted to staff. To accomplish this, Alice may wish to issue the credential Alice **says** (Dept.machine-room.staff **says** F). However, because this involves a name with two degrees of nesting, IR-TBL1 will not suggest this credential as a possibility to Alice. LR-TBL is both efficient and does not impose any restrictions on the manner in which the inference rules may be used.

When comparing the absolute times obtained using SWI Prolog without tabling (Figure 3.10) to those obtained on the same policies using XSB with tabling enabled (Figure 3.13), we see that they are very similar, whereas one would expect the tabled execution to be faster. This does not indicate that tabling is ineffective, but instead that SWI Prolog appears to be more heavily optimized than XSB. Simulations performed in XSB with tabling disabled are significantly slower than the equivalent simulations performed in SWI Prolog.

3.6.4 Effects of Caching

The primary reason that our techniques can efficiently find proofs at the time of access is that they utilize results precomputed by forward chaining and path compression. As a result, our techniques require a certain amount of processor time to compute and space to store these results. In this section, we measure the impact of these requirements for the scenarios presented above and additionally for several worst-case scenarios. The significance of this impact will vary depending on the platforms and policies used in a deployed system.

Forward chaining and path compression are not guided by any specific access attempt, and thus many of the precomputed results will be irrelevant for any particular access. This is generally an acceptable tradeoff, as the user's device, be it a phone or a computer, is likely idle most of the time, and so the computation required to find results that are not ultimately useful is transparent to the user. If, however, our techniques are employed on a device where a premium is placed on any computation (e.g., a server under heavy load or a device with stringent requirements on power consumption), then the cost of precomputing results must be considered. While we present aggregate precomputation times in Figures 3.14 and 3.16, this computation does not all occur at once, but incrementally as new credentials are added to the knowledge base. As the precomputed results can improve the efficiency of multiple time-of-access proof searches, the time-of-access savings can ultimately offset the cost of precomputation, resulting in less overall computation using our technique. As the break-

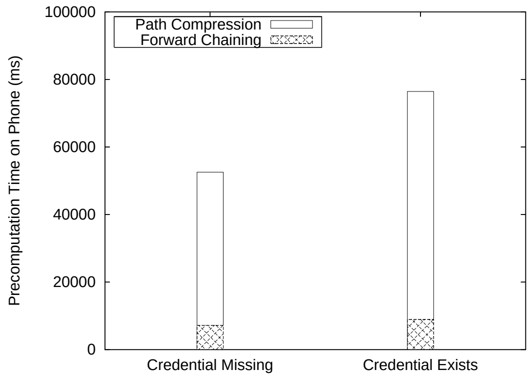


Figure 3.14: Precomputation time for example policy on phone

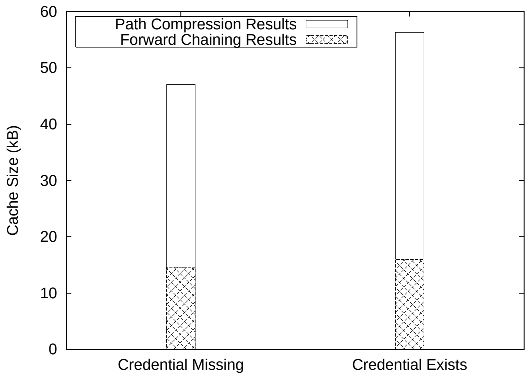


Figure 3.15: Cache size for example policy

even point can vary widely depending on the scenario, we do not investigate this tradeoff further.

To efficiently find proofs at the time of access, these precomputed results must all be loaded into the memory of the machine attempting to generate a proof. Should the size of the cache exceed the available memory, the resulting thrashing will diminish the efficiency of the proof search. Below, we measure the amount of space required to represent the knowledge base as uncompressed plain text. This representation includes only information necessary for reasoning; it does not include the digital certificates that encode the credentials.

Running example Figure 3.14 depicts the total time required to precompute results from the credentials known to Alice (see Figure 3.4) both in the scenario where Alice has not yet delegated authority to Bob and in the scenario where she has. The results shown were measured on the Nokia N70 using JIProlog. Comparing Figure 3.14 to Figure 3.6, we see that the amount of precomputation (52.5 seconds) is comparable to the amount of computation performed by LR on the critical path of access when the desired credential is missing (48.8 seconds). Additionally, the difference in time-of-access computation between LR and HC (271.7 seconds) is *greater* than the total amount of precomputation performed using our techniques (52.5 seconds). That is, for scenarios drawn from our deployment, our techniques would be more efficient than HC even if *all* computation was performed on the critical path to an access being granted.

Figure 3.15 shows that size of the knowledge base is 48 kB when Alice has not yet delegated authority to Bob, and 58 kB when she has. These cache sizes are well within the capabilities of a modern smartphone. Policies with similar structure but a greater number of credentials than our example policy have arisen in our deployment, but we have not observed any policy that exceeded the phone's memory or storage capacity.

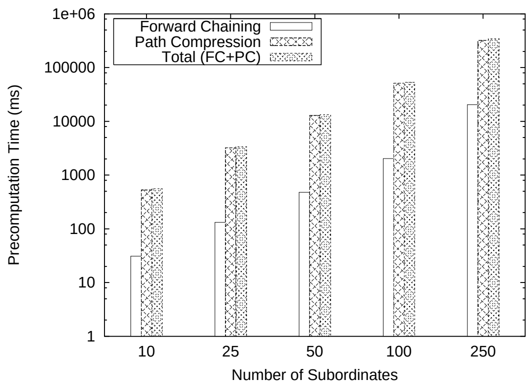


Figure 3.16: Precomputation time for larger policies

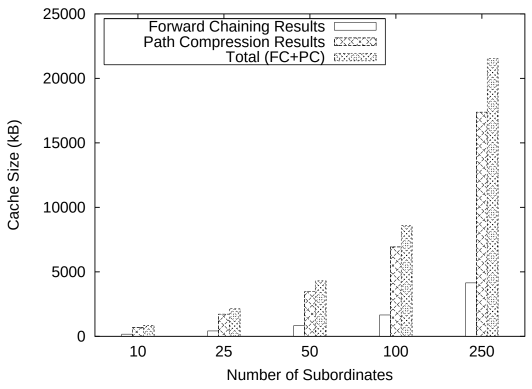


Figure 3.17: Cache size for larger policies

Large policies Figure 3.16 depicts the total precomputation time for the larger policies described in Section 3.6.2. The results are depicted on a log scale, and were collected on the workstation using SWI Prolog. The amount of precomputation is roughly an order of magnitude greater than the amount of computation that occurs on the critical path to access (Figure 3.10). In comparison to the policy of our running example, this policy is not only larger, but more complex. Unlike the example policy, in this policy, a certification authority assigns names to keys, roles are mapped to names, and delegations are made to roles. This results in a knowledge base with a far greater number of intermediate facts and paths than would be present in a policy that governs the same number of principals and resources, but without the complexity of a CA or roles.

Figure 3.17 shows the size, in kilobytes, of the knowledge base produced by forward chaining and path compression. Recalling Figure 3.11, which shows the number of entries in the knowledge base, we see that average size of each entry is roughly constant as the size of the policy increases. For the largest policy, the average size of a fact produced by forward chaining is 890 bytes, while the average size of a path produced by path compression is 2091 bytes. The size of each path is larger than that of each fact because each path must contain two formulas: a premise and a conclusion.

Worst case Over the life of the system, principals learn of new credentials through interacting with others to construct proofs of access. As long as requests for assistance are directed by human intuition, the size of a principal's cache will be bounded by the number of people she interacts with, in which case the cache size is unlikely to exceed those depicted in Figure 3.17.

However, in scenarios where the user does not know how to best direct the query, the default choice (according to the techniques of Chapter 2) is to ask the resource owner, who,

Policy	Users	Precomputation (minutes)			Cache Size (MB)		
		FC	PC	Total	FC	PC	Total
(4,4,25)	423	1.2	27.9	29.1	6.4	26.6	33.0
(5,10,50)	2558	46.9	1100.7	1147.6	39.3	165.0	204.3
(10,10,50)	5113	183.4	4347.9	4531.3	78.7	330.1	408.8

Table 3.1: Worst-case cache characteristics

in all policies described in this thesis, is the root node of the policy hierarchy. The root node, having delegated authority to manage the resource to other principals, is unlikely to know if credentials exist that would extend authority to the principal requesting access. If this is the case, the root node will in turn request further assistance, and then add any credentials included with the response to its knowledge base. Due to the volume of requests, the root node is likely to be a server that does not directly interact with users. As the techniques described in this chapter are designed around the need to interact with users, they are not optimized for use in such an environment. Nevertheless, it is instructive to evaluate our techniques in this environment to determine the extent to which they scale to larger knowledge bases.

The largest possible knowledge base will occur on the root node in a scenario in which human guidance is not used, and all requests are instead directed automatically. In this scenario, the root node would receive a request for assistance every time a user attempts to access a resource for which they do not have sufficient credentials to demonstrate the proper authority. After a sufficient number of accesses, the root node's cache would include the entire access-control policy of the system.

To measure the worst-case cache characteristics, we analyze the policy described in Figure 3.9 from the perspective of the root node (as opposed to the analysis in Sections 3.6.1–3.6.3, which was from the perspective of a manager). We consider three different policy sizes (using the notation described in Chapter 2): (4,4,25), (5,10,50), and (10,10,50), which represent organizations of 423, 2558, and 5113 users. Using the workstation described earlier, we simulate a scenario in which each principal accesses every resource permitted by the policy, and we measure the total precomputation time required to construct the root node's cache and the size of the resulting cache once the simulation completes. Table 3.1 shows the results.

Notably, the size of the cache is roughly linear with respect to the size of the policy, with the cache growing by approximately 0.08 MB for each additional user. However, the computation required to generate the cache does not scale as well. As the size of the knowledge base grows, the number of possibilities that FC and PC must investigate increases, as does the length of time required to perform each unification operation. These factors combine to cause the amount of precomputation to increase proportional to the square of the number of users. Thus, as policies grow, the cost of precomputing results using FC and PC will ultimately become prohibitive. The exact point at which this occurs will vary with system configuration, but we can extrapolate from our data to estimate that the root node of a policy with 50,000 users would require 300 days of precomputation to construct its knowledge base using FC and PC, which is clearly not practical.

In practice, however, it is likely that most queries will not be automatically directed to the root node, and so the cache sizes will not be as extreme as those presented here. For example, an organization with 50,000 users, if only 10% of requests are directed automatically, the effects of the root node's cache will be no worse than that of the (10,10,50) policy in Table 3.1.

Minimizing the impact of large caches Having shown that the amount of precomputation required by our techniques may pose problems in scenarios where a node's cache describes policy for tens of thousands of principals, we now describe three potential alternatives to addressing the problem.

The first alternative is to limit the number of credentials that a node may cache. This would limit the amount of precomputed state, which, in turn, would prevent the computation time required to process a new credential from escalating further. Once the cache contains the maximum number of credentials, each new credential must evict an old one. A variety of standard cache-replacement algorithms (e.g., least-recently used) may be used to select the credential to evict. Once that credential is evicted, the facts and paths that depend on that credential must be deleted as well. This can be accomplished efficiently by maintaining a separate data structure that maps credentials to the entries that depend on them. Restricting the number of credentials that a principal may cache will result in some additional requests for assistance that could have been avoided with a cache of unlimited size. We hypothesize that, with a properly chosen replacement strategy, this drawback will be offset by the decreased computation required from a smaller cache.

The second alternative is for the root node to not participate in the proving process directly, but to instead serve as a proxy for directing requests to a second tier of servers on the basis of the resource described in the request. The objective would be to partition the knowledge base that would have been maintained by the root node into subsets that contain as little overlap as possible, thus keeping the size small enough to enable efficient precomputation. To accomplish this, the root node would need a deterministic mapping from each resource to the server tasked with handling requests pertaining to that resource. For example, requests could be mapped based on the department to which the resource belongs. In some situations, an effective mapping may be discernable from credentials issued by the root node, while other situations may require some human configuration.

A third alternative is to eliminate the use of path compression based on the observation that PC requires substantially more precomputation than FC. It may therefore be possible to construct a variant of LR that utilizes only results precomputed by FC. However, as precomputed paths allow LR to apply an arbitrarily long delegation chain in a single step, the omission of PC will diminish the efficiency of proof search at the time of access, as the proposed LR variant could only apply delegation chains one step at a time on the basis of facts precomputed by FC.

Although the physical size of the cache scales linearly with the number of credentials, and therefore does not pose as great a problem as the required precomputation time, it is worth considering techniques for storing the cache more efficiently. At present, no attempt has been made to optimize the size of the physical representation of the cached knowledge base. In particular, each fact in the knowledge base contains the proof of that fact. As many of the facts are derived from other facts in the knowledge base, there is a great deal of redundancy between the proof terms of various facts. As the proof terms are not used for reasoning, they could be stored in a data structure that eliminates this redundancy. The proof term is generally much larger than the fact it proves, so the savings associated with this approach could be substantial.

3.6.5 Discussion

The efficiency of LR stems from the fact that, at the time of access, LR only needs to explore a subset of the search space, whereas other techniques must explore the entire search space. This implies that, in time-of-access efficiency, LR has a fundamental advantage over IR (including tabled versions of IR). The only situations where this advantage could erode are (1) if the cost of processing any credentials included with the request for assistance is so costly that it negates the advantage of using LR tactics, and (2) if the amount of precomputed state is so vast that searching for a result in the data structure of precomputed results is slower than deriving the result from credentials.

We believe that the first situation is unlikely to occur often in practice. In the vast majority of cases, the only credential that will need to be processed at the time of access is the credential indicating a principal's desire to access a resource (e.g., Credential 15 in Figure 3.5). This credential does not describe a path, and is therefore ignored by PC. Our evaluations show that the cost of processing this credential with FC is very small relative to the total proof search time of other strategies.

With regard to the second situation, our evaluations show that the amount of precomputed state in realistic scenarios is roughly linear in the number of credentials. This is insufficient to cause the search for a precomputed result using LR to be less efficient than deriving that result using IR. In comparing the efficiency of LR to IR in an environment with tabled execution, it is worth noting that LR is at a relative disadvantage because it is implemented in Prolog. Results produced by forward chaining and path compression are stored in the Prolog knowledge base and accessed by other Prolog predicates, whereas results computed via tabling are stored in a more efficient data structure that is incorporated into the execution environment. It is therefore likely that further efficiency gains in LR could result from incorporating the precomputed results into the tabling mechanism itself.

In Chapter 2, we introduced two forms of caching, positive and negative, and a technique called Automatic Tactic Generation (ATG). The results computed by FC and PC supercede the positive caching techniques described in Chapter 2, but the techniques described in this chapter do not directly employ negative caching or ATG. The objective of both negative caching and ATG is to reduce the number of remote requests necessary to construct a proof in a distributed setting, while the techniques described in this chapter are aimed at efficiently constructing a proof from local knowledge, or if one cannot be found, determining how to best proceed. Thus, if LR determines that remote assistance is necessary, it will output a list of appropriate principals to ask. At this point, negative caching could be employed to deemphasize requests that have previously failed before presenting the list to the user. Similarly, ATG could prioritize requests that match tactics generated from previous similar proofs. This application of ATG would differ from the previous chapter in that the automatically generated tactics would be employed after the standard proving process rather than before it. However, in an environment with human guidance, negative caching and ATG are useful primarily to provide assistance to the user. Effective human intuition can accomplish similar objectives, so while useful, negative caching and ATG are not as crucial in a human-guided proving environment as they are in the fully automated environment of Chapter 2.

3.7 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., [25]).

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 3.3). This is sufficient to express most access-control logics, e.g., the logics of SD3 [45], Cassandra [19], and Binder [29], 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 [6]. The general notion of delegation introduced in Definition 1 is conceptually very similar to that of the various logics that encode SPKI [1, 57, 41], the RT family of logics [58], Binder [29], Placeless Documents [8], and the domain-name service logic of SD3 [45], 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., [45, 29, 8]) 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 (Alice **says** F, Bob **says** F) and (Bob **says** **open**(door1), Charlie **says** **open**(door1)), we need to determine the intersection of F and **open**(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., **open**(door1). In some cases, a different algorithm may be appropriate: for SPKI, for example, the algorithm is a type of string intersection [34].

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 [38] allows users to dynamically create credentials. Properties (b) and (c) have been the focus of considerable previous work, notably SPKI [1, 57, 41], the DNS logic of SD3 [45], RT [58], and Cassandra [19]. 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_{Alice} **signed** **delegate**(Alice, Alice.machine-room, 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.

3.8 Related Work

Section 2.4 characterized related works by whether they operate over a distributed knowledge base using remote credential retrieval or distributed reasoning. Section 3.7 discussed the applicability of the techniques presented in this chapter to other logics, systems, and applications. The techniques presented in this chapter also make advances in terms of how queries are directed, how proofs are constructed, and how user interaction is incorporated into the system. In this section, we compare related work to each of these areas. However, we are aware of no previous algorithm that meets all of the requirements described in Section 3.1.1, and no works that analyze the performance of the distributed proving alternatives we consider in this chapter.

Directing queries PeerAccess provides a framework for describing policies that govern interactions between principals [72]. PeerAccess uses proof hints encoded in the knowledge base to guide whom a principal asks when local knowledge is insufficient to construct a proof. MultiTrust is an authorization framework that supports distributed proving, routing requests according to predefined query routing rules [77]. In many systems that support remote credential retrieval (e.g., [19, 45, 38, 48]), the description of the credential explicitly encodes its location. The distributed proof-construction technique that we described in Chapter 2 routes queries on the basis of the formula to be proved. Minami and Kotz route queries on the basis of a pre-defined integrity policy that specifies which principals are trusted to respond to each type of query [62]. In contrast, the approach we described in this chapter seeks to leverage the user's intuition when directing remote queries.

Proof-construction techniques Li et al. presented a distributed credential chain discovery algorithm for the RT_0 language [59]. This algorithm works both forwards and backwards using a graph representation of credential chains, and is capable of retrieving credentials from remote parties. However, unlike our techniques, all computation is done at the time of access and on a single node. Jajodia et al. designed a general framework for describing partially specified hierarchical access-control policies [44]. 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. designed Trust-X to negotiate trust between peers [21]. 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.

When used as a stand-alone proof-search algorithm, forward chaining can derive many facts that are not relevant to the query it is trying to solve. Magic Sets is an optimization in which the rules for forward chaining are rewritten at compile time to include constraints that prevent the derivation of many irrelevant facts [9]. This optimization does require some knowledge as to the types of query that forward chaining will attempt to answer, and as such, it is not immediately obvious how it would apply to our use of forward chaining, as we derive facts without knowledge of what query will ultimately need to be answered.

User interaction 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 [47]. 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 [38], 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. We engage the user to create new credentials as well as to approve and direct requests for assistance.

3.9 Conclusion

In this chapter we presented a new approach to generating proofs that accesses comply with access-control policy using the distributed proving framework described in Chapter 2. 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, will scale effectively to larger policies, and will continue to offer benefit in environments that support tabling. We determine that the precomputation required by our technique is reasonable for many practical scenarios, characterize the scenarios in which it is not, and discuss future directions for improving scalability. Our method will generalize to other security logics that exhibit the common properties detailed in Section 3.7.

The techniques presented in this chapter provide a means of efficiently searching for a proof at the time of access, and include provisions for allowing users to modify policy to ensure that the access will be granted. In other words, if a policy is misconfigured to deny a legitimate access, one benefit of these techniques is that they streamline the process by which a user may resolve this misconfiguration at the time of access. However computationally efficient this process may be, the required human intervention imposes a delay that can severely diminish users' perception of the system [12]. Identifying and resolving such misconfigurations before they result in an access being delayed or denied is the focus of Chapter 4.

Chapter 4

Detecting and Resolving Policy Misconfigurations

Access-control policy misconfigurations that cause requests to be erroneously denied can result in wasted time, user frustration and, in the context of particular applications (e.g., health care), very severe consequences. In this chapter, we apply association rule mining to the history of accesses to predict changes to access-control policies that are likely to be consistent with users' intentions, so that these changes can be instituted in advance of misconfigurations interfering with legitimate accesses. Instituting these changes requires consent of the appropriate administrator, of course, and so a primary contribution of our work is to automatically determine from whom to seek consent and to minimize the costs of doing so.

We evaluate the effectiveness of our techniques on data collected from Grey, an experimental access-control system that has been deployed and actively used at our institution to control access to offices for approximately two years [13]. This deployment is one in which misconfigurations that prolong access to an office substantially diminish the perceived usability of the system [12]. Data drawn from logs allows us to reconstruct a detailed scenario of how policy was exercised over the course of 10,911 attempted accesses, spanning approximately 14 months. This data includes accesses by 29 users to 25 physical doors and shows that there is a high degree of resource sharing (22 of the 25 resources were accessed by more than one user).

Additionally, Grey supports dynamic policy creation (using the techniques described in Chapter 3), meaning that a user can delegate her authority to open a door using her smartphone in response to a request to do so. This capability of Grey allows us to determine how users were able to resolve misconfigurations in real life. We augment this data with information collected from a user survey that asked what policy users were willing to implement should the need arise. Results from the survey allow us to determine precisely how users would resolve misconfigurations in scenarios that did not occur over the course of our deployment.

Using this data, we show that our methods can reduce the number of accesses that would have incurred costly time-of-access delays by 44%, and can correctly predict 58% of the intended policy. These gains are achieved without increasing the total amount of time users spend interacting with the system.

The contributions of this chapter are fourfold: (i) to develop techniques to identify potential misconfigurations (Section 4.1), (ii) to develop techniques to resolve misconfigurations once they have been identified (Section 4.2), (iii) to evaluate these techniques on a data set collected from a deployed system (Sections 4.1.3 and 4.2.3), and (iv) to evaluate the effectiveness of our techniques when only partial data is available (Section 4.3). These results expand upon our previous publication [17] by the addition of (iv).

4.1 Techniques for Identifying Misconfigurations

To identify potential policy misconfigurations, we first use *association rule mining* to detect statistical patterns, or *rules*, from a central database of previously observed accesses (Section 4.1.1). We then analyze our data using these rules to predict potential misconfigurations, or instances of the data for which the rules do not hold (Section 4.1.2). Once we determine whether or not the prediction was correct, we incorporate the result into a feedback mechanism to promote the continued use of rules that accurately reflect policy and prune rules that do not.

To illustrate the usefulness of this technique, consider the following scenario. Bob is a new student who is advised by Alice, a professor. Bob and Alice both work in a building where the same system controls access to Alice's office, Bob's office (which is shared with some of Alice's other students), a shared lab, and a machine room. When the department assigns Bob an office, it configures access-control policy to allow Bob to gain access to his office (e.g., by giving Bob the appropriate key). Though Alice is willing in principle to allow Bob access to the lab and machine room, both she and the department neglect to enact this policy.

The first time Bob attempts to access the shared lab space, he is denied access as a result of the misconfiguration, at which point he must contact Alice or the department to correct it. This process is intrusive and could potentially take minutes or even hours. However, the past actions of Bob's office-mates suggest that people who access Bob's office are very likely to also access the shared lab space and machine room. The techniques we describe here allow us to infer from Bob's access to his office that Bob is likely to need access to the lab space and machine room. Identifying this in advance allows for the misconfiguration to be corrected before it results a denied access and wasted time. Detecting the misconfiguration is accomplished using only the history of accesses in the system; the technique is independent of the underlying access-control mechanism, policy, and policy-specification language.

4.1.1 Association rule mining

The objective of association rule mining is to take a series of records that are characterized by a fixed number of attributes, e.g., boolean attributes A through D, and discover rules that model relationships between those attributes. Suppose that for 75% of the records where both A and B are true, D is also true. This property would give rise to the rule $A \wedge B \rightarrow D$. One measure of the quality of this rule is *confidence*, which is defined as the percentage of time that the conclusion is true given that the premises of the rule are true (75% for this example). Confidence represents the overall quality of a rule.

We employ the Apriori algorithm [3] to mine association rules, although other methods also exist. Apriori first builds all possible *itemsets*, or groups of attributes, that have occurred together in more than a certain fraction of the records. This fraction is known as the *support* of an itemset. For each of these itemsets (e.g., $A \wedge B \wedge D$), Apriori enumerates all subsets of the itemset (D, $B \wedge D$, etc.). Using each subset as the premise for a rule and the remainder of the itemset as the conclusion, Apriori calculates the confidence of the rule, and keeps only rules whose confidence exceeds a specified minimum.

In our context, each resource is represented by a boolean attribute. Each user in the system is represented by a record in which the attributes corresponding to the resources that the user has accessed are set to true. For example, if attributes A through D each represent a resource, the record $\langle$false, true, true, false$\rangle$ would represent a user who has accessed resources B and C.

In our scenario, a small number of high quality rules may identify statistically significant patterns, but on too small of a subset of the overall policy to be of much use. Thus, in tuning the output produced by Apriori, our objective is to balance the quality of the rules produced with the quantity of those rules. We achieve this by varying the minimum allowable confidence and support that a rule may have. Requiring a higher confidence biases the output towards rules that are true with high likelihood, while requiring higher support biases the output towards rules that describe patterns that occur with higher frequency. Section 4.1.3 describes the extent to which these parameters affect our ability to detect misconfigurations.

4.1.2 Using mined rules to make predictions

The rules output by Apriori must, in turn, be used to identify potential policy misconfigurations. A potential misconfiguration is a record for which the premises of the rule hold, but the conclusion does not. If a rule has a confidence of one, it implies that for all records which the premises of the rule hold, the conclusion of the rule holds as well. This means that every user who has accessed the resources represented by the premises of the rule has already accessed the resource mentioned in the conclusion. These rules do not allow us to identify any possible misconfigurations, so we ignore them. For each remaining rule, we identify the records for which the premise holds, but the conclusion does not. Each such record represents a potential misconfiguration; we predict that the user represented by that record should have access to the resource identified by the conclusion of the rule.

Feedback One limitation of using mined rules to predict policy is that a dataset may contain several patterns that are statistically significant (i.e., they produce rules whose confidence exceeds the minimum) that are nonetheless poor indicators of policy. For example, the rule (perimeter door $A \rightarrow$ office D) may have medium confidence because door A is physically close to office D, and is therefore used primarily by the professor who owns office D and by his students. However, should this rule be used for prediction, the professor will be asked to delegate authority to enter his office to everyone who accesses door A.

To prevent the system from making repeated predictions on the basis of poor rules, we introduce a feedback mechanism that scores rules on the correctness of the predictions they produce. A correct prediction is one that identifies a misconfiguration that a human is willing to repair. The idea is to penalize a rule when it results in an incorrect prediction, and to reward it when it results in a correct prediction. Rules whose score drops below a threshold are no longer considered when making predictions.

To illustrate how scores are computed, consider the following example. Suppose that four resources A through D are commonly accessed together. This implies that Apriori will construct the itemset $A \wedge B \wedge C \wedge D$. Suppose that Apriori constructs the rules $A \wedge B \rightarrow D$ and $A \wedge C \rightarrow D$ from that itemset. Our feedback mechanism keeps a pairwise score for each premise attribute and conclusion, that is, (A, D), (B, D), and (C, D). If the rule $A \wedge B \rightarrow D$ is used to make a correct prediction, the scores for (A, D) and (B, D) will be incremented. If the prediction was incorrect, those scores will be decremented. If $A \wedge B \rightarrow D$ produces four incorrect predictions and $A \wedge C \rightarrow D$ produces three correct predictions, the scores for (A, D), (B, D), and (C, D) will be -1, -4, and 3.

The score for a rule is the sum of the scores of the premise attribute, conclusion pairs. In the scenario described above, $A \wedge B \rightarrow D$ would have a score of -5, while $A \wedge C \rightarrow D$

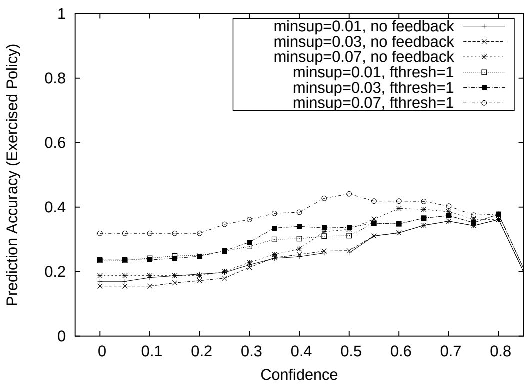


Figure 4.1: Prediction accuracy (exercised policy)

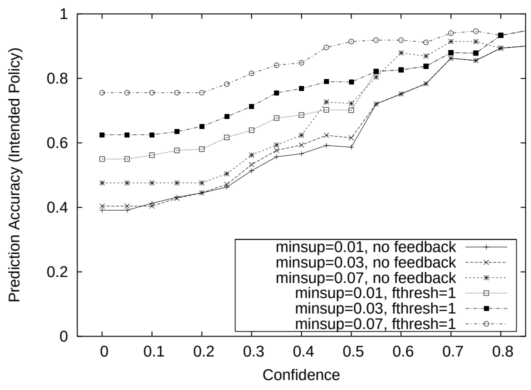


Figure 4.2: Prediction accuracy (intended policy)

would have a score of 2. If the threshold for pruning rules is 0, $A \wedge B \rightarrow D$ would be pruned from the set of rules used to make predictions.

The reason for employing this technique instead of a more straightforward system where each rule is scored independently is that an itemset will often result in many more rules than the example presented above (in fact, Apriori may produce a rule with every combination of A, B, and C as the premise). Our technique allows us to more quickly prune groups of similar rules that are poor indicators of policy.

This feedback strategy performs well in our evaluation. However, in a hypothetical scenario, a rule could acquire a high positive score, which would require that many failures occur before the rule is pruned. In situations where this is unacceptable, the system could employ alternative feedback strategies that only consider recent accesses or compute the feedback score as a percentage of successful predictions. In the unlikely event that a rule's feedback score is high but its continued use is problematic, an administrator can manually prune the rule.

4.1.3 Evaluation

In evaluating our prediction techniques we distinguish between several different types of policy. *Implemented policy* is the policy explicitly enacted via credentials that grant authority to users. Data from logs is sufficient to learn the entire implemented policy for our deployment environment. *Intended policy* includes implemented policy and policy that is consistent with users intentions but has not yet been enacted through credentials. *Exercised policy* is the subset of implemented policy that allowed the accesses that were observed in the logs. This is the only kind of policy that is used for making predictions; the other policy sets are used purely to evaluate the effectiveness of our methods. Finally, *unexercised policy* is the intended policy without the component that has been exercised.

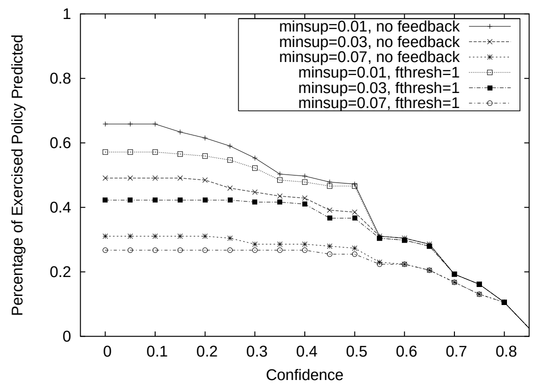


Figure 4.3: Coverage of exercised policy

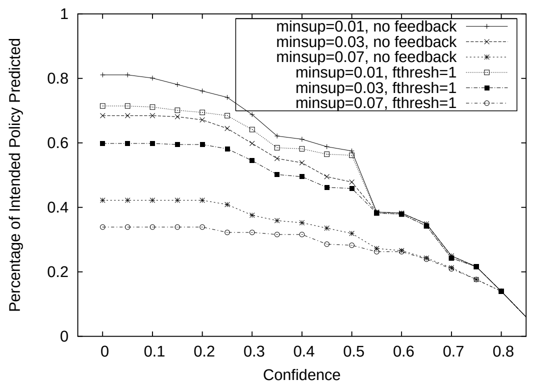


Figure 4.4: Coverage of intended policy

Since intended policy can only be partially gathered from system logs, we distributed a questionnaire to each user who acts as first-line administrator for a resource. The questionnaire listed all of the resources that the administrator had authority to delegate. For each such resource, the administrator was asked to select (from a list) the users to whom she would be willing to delegate authority. This allowed us to determine, for situations that did not occur in actual use of the system, whether the administrators would be willing to grant authority. These responses are used solely to analyze the accuracy of our predictions.

Our evaluations take place in a simulated environment defined by the usage data that we collected from our deployment. The logs describe 10,911 access attempts of 29 users to 25 doors. For each access attempt we logged who attempted to access which resource, when the attempt was made, and whether it succeeded; we refer to each such record as an *access event*. Several subsets of the implemented policy were completely or partially preconfigured, e.g., seven perimeter doors were preconfigured to be accessible to all users through a policy that used groups, and gaining access to these doors required no policy reconfiguration.

We replay the sequence of access events logged by our system and, after every event, attempt to predict which new policies are consistent with the accesses observed so far. The performance numbers we report are aggregated over the entire run of the simulation. A prediction is considered *accurate* with respect to exercised policy if the predicted policy is exercised in the data set for the first time after the the prediction was made. We also evaluate accuracy with respect to intended policy; here we count a prediction as accurate if it is consistent with intended policy that has not yet been exercised (regardless of whether it is ever exercised). Although accuracy of prediction is an important metric, in practice it is important to achieve both good accuracy and good *coverage*. Coverage describes the percentage of the target space predicted, i.e., the percentage of actual accesses present in the data set (or the percentage of intended policy) that we are able to predict.

Prediction accuracy We evaluate the accuracy of our predictions with respect to both exercised and intended policy. Accuracy is affected by tuning three parameters: the minimum confidence, *minconf*, and support, *minsup*, that a rule may have, and the feedback threshold, *fthresh*, that governs the minimum feedback score that a rule must have to be used in the prediction process (see Section 4.1.2). We evaluate three values for *minsup*: 0.01, 0.03, and 0.07, which represent supports of one, two, and three records. We evaluate confidence values ranging between 0.01 and 0.8. Values above 0.8 resulted in so few predictions that the accuracy was not meaningful.

In practice, the feedback score represents the result of previous attempts to resolve misconfigurations. However, the success rate of the resolution process depends on other factors, like our ability to efficiently detect whom to prompt to correct detected misconfigurations (see Section 4.2). To evaluate the accuracy of our predictions in isolation from the resolution process, we use *ideal feedback*, in which the scoring is based on what we know to be the intended policy in the system. We revert to the standard form of feedback when evaluating the resolution process in subsequent sections.

We evaluated *fthresh* values of -1, 0, and 1, using the technique described in Section 4.1.2 to score each rule. A rule is used to generate predictions only if it had no feedback score or if its feedback score was greater than or equal to the threshold value *fthresh*. For clarity, we present only the results obtained with *fthresh* set to 1, since that setting offered the greatest benefit.

Figures 4.1 and 4.2 show the prediction accuracy with respect to exercised and intended policy. As expected, including feedback in the prediction method improved accuracy for combinations of other parameters. Using rules with higher confidence and support parameters uniformly improves the accuracy of predictions with respect to intended policy, but the benefit with respect to exercised policy peaks at a confidence of around 0.5. Intuitively, this shows that while past behavior gives us more insight into intended policy, future accesses are not drawn uniformly from this space. We conjecture that a larger data set, in which the exercised policy covered a greater part of the intended policy, would show improved performance with respect to exercised policy.

The increased accuracy achieved by using higher-quality rules lowers the total number of predictions, as we will discuss in Section 4.1.3.

Prediction coverage We show prediction coverage for exercised and intended policy while varying the same parameters as when evaluating prediction accuracy. Our findings are shown in Figures 4.3 and 4.4. As expected, again, coverage decreases as we improve the accuracy of the rules. That is, the more accurate rules apply in a lower number of cases, and so predictions that would be made by a less accurate rule are missed. Interestingly, a sharp drop-off in coverage doesn't occur until confidence values are raised above 0.5, suggesting that values in the range between 0.3 and 0.5 may produce rules that are both accurate and have good coverage. With reasonable parameters (minsup=0.01, minconf=0.4, and fthresh=1) our predictions cover 48% of the exercised policy and 58% of the intended policy.

Accuracy over time In addition to measuring the aggregate accuracy of predictions across an entire run of the simulator, it is interesting to know how prediction accuracy varies as a function of time. To measure this, we compute the accuracy of predictions over intervals that contain 50 predictions each. Figure 4.5 shows these results for different values of *minconf* with *minsup* fixed at 0.01. Rules with a higher minimum confidence make fewer predictions and so result in fewer data points. Different values of *minsup* exhibit similar trends.

Somewhat surprisingly, we found that the predictions made early in the simulation are, roughly speaking, as accurate as those made later when more history is available to the rule-mining algorithm. We conjecture that this is because the initial data points, though relatively few in quantity, are of high quality. In other words, the early accesses are made by a small number of people and to a small number of shared resources, and so are representative of a very small but often exercised subset of the intended policy.

4.1.4 Discussion

We evaluated our ability to predict accesses that are consistent with policy with respect to both accuracy and coverage. Reasonably good performance was achieved using each metric, but, more importantly, we identified a combination of parameters for which the predictions were both reasonably accurate (i.e., not erroneous) and covered a large portion of the unexercised policy. Specifically, minimum confidence values between 0.3 and 0.5 achieved the best tradeoff. For these parameters, the increased accuracy resulting from our use of feedback to prune rules far outweighed the associated decrease in coverage.

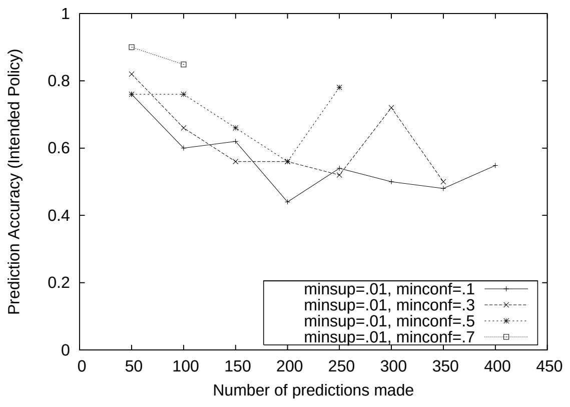


Figure 4.5: Prediction accuracy versus time

Varying the minimum support that a rule must have did not have as great an impact on results as the other parameters. The higher coverage that resulted from a minimum support value of 0.01 outweighed the increase in accuracy achieved by using higher values, and so for the evaluation in Section 4.2.3 we will fix the minimum support to 0.01.

Finally, we found that the predictions made with relatively few data points were roughly as accurate as predictions made with many more. Consequently, it appears that our methods would work well even in the early phases of adoption or deployment of a system that uses them.

One important question is the extent to which the success of our technique on our dataset will carry over to systems with different access patterns and policies. Our technique exploits the observation that principals with similar access patterns are often granted access to those resources via similar policies. Thus, in any system where users' access patterns imply characteristics of access-control policy, our technique is likely to provide benefit. Our technique will not be effective in a system where few shared resources exist or there is little overlap between the access patterns of individual users.

Our technique is effective as long as there is a discrepancy between implemented policy and intended policy. If a system is able to exactly implement the intended policy and the intended policy is fully known at the outset, then there are no misconfigurations to detect. However, regardless of the expressiveness of the policy language used by the system, neither of these conditions is likely to be met: the intended policy is often dynamic, i.e., developed in response to new situations; and even when the intended policy is not dynamic, it is rarely fully specified at the outset. Therefore, it seems likely that most systems will have misconfigurations.

4.2 Techniques for Repairing Misconfigurations

Once a potential misconfiguration has been identified, it is best if the policy is corrected as soon as possible to maximize the likelihood that the misconfiguration will not affect users. Since an identified misconfiguration might be erroneous, a user with the authority to modify policy must be queried to determine if the identified misconfiguration is consistent with the intended policy. If it is, then the user can repair the misconfiguration by altering or extending the policy. How the user elects to modify the policy is orthogonal to our work; it could entail changing an access-control list on a server or issuing digitally signed credentials that create a new group and delegate authority to that group. However, if there is only a single administrator in charge of policy, the question of which user to contact becomes trivial. Hence, we look at the more difficult problem of repairing misconfigurations in distributed access-control systems in which multiple users may have the ability to modify policy.

Following the example in Section 4.1, when Bob discovers that he cannot access the shared lab space, he must determine whom to ask for assistance in resolving the conflict. Since the lab space is shared, Alice may not be the only person with the authority to modify access-control policy. Professors Charlie and David may also be able to edit the policy governing the lab, but they may not be willing to grant authority to Bob. In this case, then, we would like the system to contact Alice and suggest to her that she amend policy to allow Bob access.

In this scenario, our previous work relied on Bob's intuition to direct queries to the most appropriate principals [15]. However, we would like to resolve such misconfigurations proactively, i.e., at a time when Bob is not actively interacting with the system. Contacting the user to obtain his intuition at a time when he is not already interacting with the system would necessarily involve an additional user interruption. Instead, we attempt to determine which users are most likely to be able and willing to resolve the misconfiguration by analyzing past user behavior.

4.2.1 Users to contact

The strategy that is most likely to succeed in repairing the misconfiguration is one that exhaustively queries all users who have the relevant authority, but this process is likely to embitter the users who are wantonly interrupted. Therefore, when deciding which users to contact, we must balance the desire to repair the misconfiguration with the desire to avoid unnecessary user interaction.

To determine which users have the authority to resolve a misconfiguration, we could analyze the implemented access-control policy. However, the language for expressing policy varies widely between systems, and we wish to design a technique that is not specific to any particular language. Instead, we determine who has authority to repair a misconfiguration by analyzing the observed behavior of users when they resolved past misconfigurations. This is possible because, in addition to logging past access attempts, our system maintains data about who users contacted when attempting to manually resolve misconfigurations. The intuition is that, because of similarities in the structure of access-control policy, principals who have rendered assistance in similar scenarios in the past are likely to provide assistance in the future, as well. For cases where that is insufficient, we also consider the users who have previously accessed the resource, as they may be allowed to redelegate authority.

4.2.2 Directing resolution requests

We propose four different strategies for constructing a candidate list of these principals based on past user behavior. Once this candidate list has been assembled, it is sorted in descending order by the number of times the principal has rendered assistance in previous scenarios.

Strategy 1: OU The candidate list consists of the principals who previously rendered assistance to Other Users (OU) when they attempted to gain access to the resource mentioned in the prediction.

Strategy 2: OR The candidate list consists of the principals who previously rendered assistance to the user mentioned in the prediction when that user accessed Other Resources (OR).

Strategy 3: U The candidate list consists of the Union (U) of the lists produced by the OU and OR strategies.

Strategy 4: UPPA The candidate list contains the list U plus the Principals who Previously Accessed (UPPA) the resource mentioned in the prediction. Since these principals have not previously rendered aid to anyone, they will be sorted to the end of the candidate list.

The last strategy aims to cover the case where the structure of the policy that would authorize the predicted access is slightly different than the policy that authorized previous accesses. For example, should the system predict that Bob will access Alice's office, past observations may show that Alice's first access required the department to reconfigure policy. However, the department is unlikely to authorize Bob, whereas Alice (who has previously accessed the office) may be willing to provide the needed delegation.

4.2.3 Evaluation

Our objective is to evaluate the ability of our resolution techniques to resolve misconfigurations using the simulated environment described in Section 4.1.3, and to determine to what extent our techniques affect the usability of the system.

Our ability to resolve misconfigurations is measured by our *success rate*, which is the percentage of misconfigurations that we can resolve. Proactively resolving misconfigurations decreases the number of *high-latency accesses*, or accesses where a misconfiguration must be corrected at the time of access. However, resolving misconfigurations does involve user input; this effect is measured by counting the number of *user interruptions*. Finally, the *total user interaction time* estimates the extent to which users must interact with the system, both with and without our techniques.

Success rate We define success rate to be the percentage of misconfigurations resolved, where a misconfiguration is a a discrepancy between implemented and intended policy. The process for resolving a misconfiguration consists of constructing a candidate list of principals

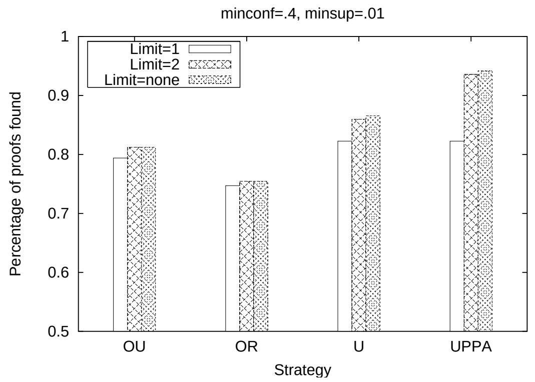


Figure 4.6: Success rate of resolution strategies

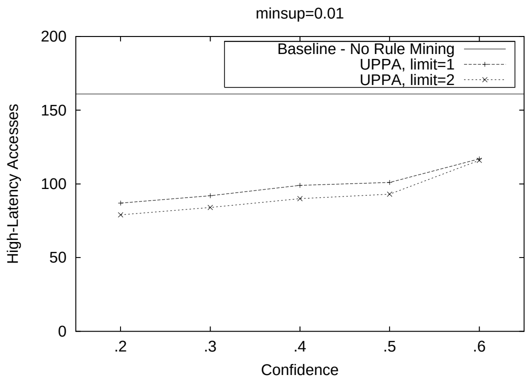


Figure 4.7: Number of high-latency accesses

to query, determining how many of those candidates to query, and performing the queries. We evaluate the success rate using the four different strategies for constructing a candidate list presented in Section 4.2.2 with three different limits on the number of candidates to query.

Figure 4.6 shows the success rates obtained with these strategies when *minconf* and *minsup* are set to 0.4 and 0.01. The U and UPPA strategies are more likely to succeed than OU (which consults those who helped other users access the same resource) or OR (which consults those who helped the same user access other resources). This is not surprising, because U combines the results from OU and OR, and UPPA further extends U (by consulting users who previously accessed the same resource). In all cases, there was a noticeable benefit if the top two candidates were consulted instead of just the top candidate, but asking candidates beyond the top two offered little additional benefit. When only the most likely candidate was consulted, UPPA and U were equivalent, since UPPA's extension of the candidate list involved only appending to it. There is of course an increased overhead cost when consulting more than just the top candidate; we analyze this cost below.

High-latency accesses If an access request is consistent with the intended access-control policy, but not with the implemented policy, then the user must attempt to resolve the misconfiguration prior to accessing the resource. The main cause of latency and inconvenience in this scenario is that human intervention is required to augment existing policy, and this intervention is on the critical path to an access being allowed. At best, this is inconvenient both to the person requesting access and the person who must repair the misconfiguration. At worst, the person who can resolve the misconfiguration may not be available (e.g., flying), and the delay for repairing the misconfiguration may be very lengthy. The data we collected from our deployment encompasses 212 time-of-access policy corrections; in 39 cases a user's access was delayed by more than 10 minutes, and in 21 by over an hour.

Here we evaluate the extent to which our prediction and resolution techniques reduce the number of these high-latency accesses. The amount of reduction depends on the extent to which the predictions cover exercised policy (Figure 4.3) and the ability of the resolution process to succeed in the cases where exercised policy is correctly predicted (Figure 4.6).

Figure 4.7 shows the extent to which our prediction and resolution techniques reduce the number of high-latency accesses. Due to its higher success rate, consulting two users with UPPA reduces high-latency accesses more than consulting only one. Lower minimum confidence thresholds yield greater reductions, because they produce greater coverage. More importantly, significant reductions can be achieved with confidence values that are likely to result in an acceptable amount of overhead: for example, for a minimum confidence value of 0.4, and a limit of two user consultations reduces the number of high-latency accesses by 44%.

User interruptions Proactively resolving a misconfiguration may involve proactively querying users to determine the intended policy. We consider an interruption to be any query directed to a user as part of the resolution process. This overstates the extent to which a user must be involved, because some queries directed to a user can be answered automatically by the user's phone in Grey, e.g., if the misconfiguration can be resolved by policy that is implemented, but has not yet been exercised. Our survey did not ask users how they would implement their intended policies, so our simulations consider only policy that was exercised over the course of our deployment.

Often, several misconfigurations are detected at the same time, and their corresponding resolution processes query the same user, resulting in multiple interruptions. We introduce an optimization, *batching*, that groups these queries into a batch, allowing the user to answer them all as part of a single interactive session. This optimization is motivated by the observation that the incremental cost of additional user interaction is less than that of the initial interruption.

Figure 4.8 shows the number of user interruptions for our data when consulting either one or two principals chosen by the UPPA strategy and varying the minimum confidence values for rules. Some principals, such as a department credential server, automatically respond to requests using implemented policy without requiring user interaction, and therefore these requests are not reflected in Figure 4.8. As expected, consulting two principals instead of one increases the number of interruptions (and the success rate, as previously discussed). In some scenarios, the first principal consulted is one that responds automatically, but the second consultation requires user interaction. This explains how consulting two principals can produce more than twice as many interruptions as consulting only a single principal. Batching is more effective when consulting two principals and for low minimum confidence

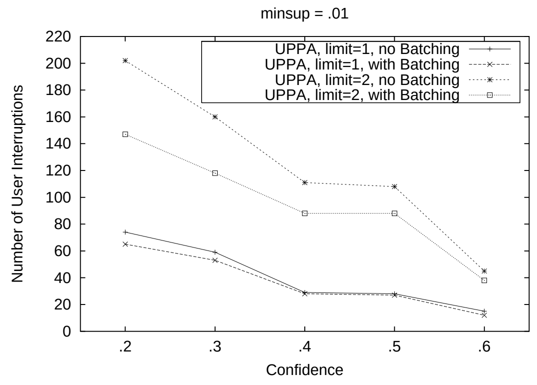


Figure 4.8: Proactive user interruptions

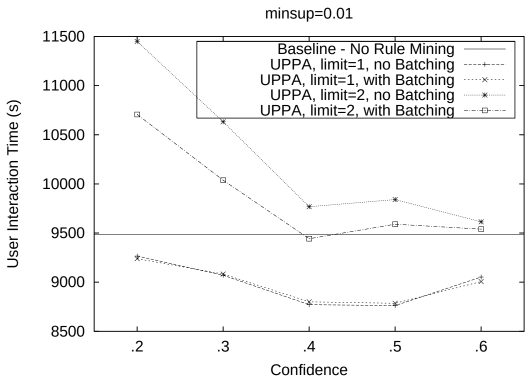


Figure 4.9: Total user interaction time

values; these settings result in more attempts to consult users, and thus more redundancy among those requests that can be reduced through batching.

Interestingly, minimum confidence values of 0.4 and 0.5 cause a marked decrease in the number of interruptions without significantly increasing the number of high-latency accesses (Figure 4.7).

User interaction time The results described thus far demonstrate that increased proactive user interaction will reduce the number of high-latency accesses, but it is difficult to determine when the associated costs of interaction outweigh the benefit of avoiding high-latency accesses.

Any attempt to quantify this tradeoff is necessarily an approximation; there are many factors that influence a user's perception of the system that cannot be measured precisely. High-latency accesses, for example, typically annoy users far more than interruptions of similar length that occur off the critical path to an access [12]. In this evaluation we measure only: (1) the delay observed by the user who is attempting to gain access while the policy is being reconfigured (how long Bob waits for access to Alice's office when Alice must reconfigure the policy); (2) the duration of the reconfiguration (how long Alice takes to reconfigure her policy to allow Bob access); and (3) the length of time required to obtain credentials that describe implemented policy when no reconfiguration is necessary. In our system, latency (1) ranged between 25 seconds and 18.6 hours, with a median of 98 seconds (the average, heavily influenced by a few outliers, was 53 minutes). The median time observed for latency (2) was 23 seconds, only 5.8 seconds of which was spent selecting the appropriate correction (the rest was Alice finding her phone, etc.). Finally, the average duration of latency (3) was 6.9 seconds. These particular latencies are specific to our system; we anticipate that they would be much higher (on the order of hours) in systems that do not offer integrated support for resolving misconfigurations. However, our results will hold as long as there remains a similar relationship between the latencies (e.g., delay incurred by a user waiting for access is much greater than delay incurred by a user modifying policy), which we feel is likely for the vast majority of access-control systems.

With these caveats in mind, we use our simulation results and the timings above to approximate the total time that all users would spend interacting with the system (accessing resources or creating and correcting policy). If a misconfiguration must be resolved on the critical path of access, the total user interaction time is the sum of latencies (1) and (2). If a misconfiguration is resolved prior to access, the total interaction time is simply latency (2), as the resulting credentials can be proactively distributed to the recipient of the delegation. Guided by data from our deployment, we weight the various latencies incurred by users as follows. Requests for credentials that occur on the critical path to an access being granted are are assumed to take 6.9 seconds if the request can be completed without user interaction, and 98 seconds otherwise. Whenever a query requires user interaction, we assume that the user takes 23 seconds to respond to the query. We assume that each additional question posed to the user as part of a batch takes 5.8 seconds.

We calculate the total time users would spend interacting with the system using the UPPA strategy and varying the minimum confidence level required of the rules. The results are shown in Figure 4.9. Restricting UPPA to consult a single user results in a slight time savings for all minimum confidence values tested. Allowing a second user to be consulted results in a small increase in total time for higher minimum confidence values and larger increase in total time for lower values. Notably, with a minimum confidence value of 0.4 and batching enabled, UPPA results in a slight overall time savings even if it is allowed to contact two principals.

4.2.4 Discussion

Each strategy for directing queries in our resolution mechanism is capable of resolving a majority of policy misconfigurations. In particular, the UPPA strategy, when allowed to consult two users, is able to resolve almost 95% of such misconfigurations. The proactive resolution of these misconfigurations results in a drastic reduction (between 40% and 50%) in the number of high-latency accesses. Consulting two users in the UPPA strategy is more effective at resolving misconfigurations than consulting a single user, but this increase in effectiveness comes at a cost of significantly more user interruptions. Batching can reduce the number of user interruptions by approximately 15% when consulting more than one user. When comparing a user consultation limit of two to a limit of one on the basis of total user interaction time, the time associated with the additional user interaction is compensated by the savings resulting from fewer high-latency accesses.

To summarize: Our results show that a significant reduction (44%) in the number of high-latency accesses can be achieved without increasing the total amount of time users spend interacting with the system.

Our technique for resolving misconfigurations is general in that it operates on the basis of observed behavior rather than inspection of access-control policy. It is therefore independent of both the underlying policy-specification language and the manner in which misconfigurations are repaired. It does, however, require that the system know who resolved previous misconfigurations. Since changes to policy are generally logged, we feel that this is a reasonable requirement.

The results describing the estimated user interaction time are necessarily specific to our system. However, our system provides integrated support for resolving misconfigurations at the time of access. Many systems do not support this functionality, and as a result, the duration of each high-latency access is likely to be dramatically higher. Our techniques significantly reduce the number of high-latency accesses, so the case for proactively resolving misconfigurations in such a scenario is likely to be even stronger than the one we present here.

4.3 Rule Mining with Partial Data

To this point, we assumed that all access logs are available to the rule-mining algorithm. However, some scenarios may restrict the available data to a subset of the overall data, e.g., to protect privacy or improve scalability. Specific users may prefer that their access logs be excluded from rule mining to prevent others from observing their behavior. Similarly, users may prefer that access to particular resources, such as a bathroom, not be logged. As the complexity of rule mining in our scenario grows with the number of resources, large organizations may find it necessary to mine rules using subsets of resources to improve scalability.

If data pertaining to a particular resource is withheld, the mined rules will not reference that resource, and therefore cannot predict accesses of that resource. Similarly, if a user's data is withheld, our technique will not predict any accesses for that user. However, removing large fragments of the input data can also have a detrimental effect on rule mining on the remaining data. In particular, consider two resources, A and B, that are governed by extremely similar policy (e.g., the rule $A \rightarrow B$ has high confidence), but are statistically unrelated to any other resources (e.g., $\forall C$, $C \rightarrow B$ has low confidence). If data pertaining to A is withheld, then it is unlikely that there will be many correct predictions regarding B, as the remaining rules will either not meet the minimum confidence requirement, or will be pruned by our feedback mechanism after making several incorrect predictions.

Removing user data can also severely degrade the performance of rule mining on the remaining data. Following the above example, if a significant fraction of the users who have accessed A are withheld from rule mining, the support of the itemset (A, B) could drop below *minsup*. Alternatively, the confidence of the rule $A \rightarrow B$ could fall below *minconf*. Either situation would prevent predictions of B based on accesses of A.

These above examples are worst-case scenarios. In our evaluation, we find that there is enough overlap between rules to compensate for the loss of subsets of the input data. In the context of our example, this implies that there is often a rule $C \rightarrow B$ that will predict many of the misconfigurations pertaining to B if the rule $A \rightarrow B$ is eliminated. In this section, we investigate the extent to which the removal of portions of the data impacts the performance of the remainder of the system. However, our dataset is drawn from a relatively small deployment, and it is therefore difficult to determine the extent to which these results imply properties about other policies.

4.3.1 Partitioning Strategy

With 25 users and 29 resources, our data can be divided into 2^{54} distinct subsets; it is therefore intractable to simulate all possibilities. Instead, we elect to partition the set of resources, and evaluate the performance when removing the data associated with one partition at a time. Our objective is to partition the data such that we can remove the data most and least likely to impact rule mining in order to estimate the maximum and minimum impact of withholding data. As patterns are formed when multiple users access the same resources, we would expect that removing data pertaining to the resources that were accessed by the most users to have a much greater impact than removing data pertaining to the resources that were accessed by the fewest users. Similarly, removing the data of the users who accessed the most resources should have a greater impact than removing the data of the users who accessed the fewest resources.

As the input data to the rule mining algorithm does not include the total number of accesses, we elect to partition our data on the basis of unique accesses as follows. We first sort the resources by the number of unique users that have accessed the resource. If two resources have been accessed by the same number of unique users, the total number of accesses is used to break the tie. We partition the sorted list into quintiles, and assign each quintile a letter designation A through E, with A representing the 20% of resources that have been accessed by the fewest unique users and so on.

For users, we perform a similar process, instead sorting by the number of unique resources that each user has accessed. Ties are broken using the total number of accesses made by a user. The users are also partitioned into quintiles, labeled F through J, with F representing the 20% of users who have accessed the fewest unique resources.

4.3.2 Evaluation

We present two metrics for each partitioning: the number of high-latency accesses and the total user interaction time.

In Section 4.2.3, we showed that our techniques could significantly reduce the number of high-latency accesses incurred by users of the system. This is a positive result, as high-latency accesses are inconvenient and frustrating to users. Here, we analyze the number of high-latency accesses to determine the extent to which removing a partition of data will diminish the benefit of our techniques. In Section 4.2.3, we also estimated that our techniques would not impose any net increase in the total time that users spend interacting with the system. Here, we investigate the extent to which this result holds when portions of the dataset are withheld from the rule-mining algorithm. We compute the user interaction time using the method described in Section 4.2.3.

For these simulations, we fixed all other parameters to values that yielded good results in the preceding sections. Namely, we fix *minconf* to 0.4, *minsup* to 0.01, *fthresh* to 1, the resolution strategy to UPPA with a limit of one request, and we disable batching as its benefit is minimal when limiting the resolution procedure to one request. Results obtained with a limit of two requests show similar results.

Resource Partitions To measure the impact of removing a partition, we count the number of high-latency accesses of resources in each resource partition. For example, if Partition A contains door1 and Alice's access of door1 incurs a costly time-of-access delay to resolve a misconfiguration, this will be reflected only in the the number of high-latency accesses for Partition A. Figure 4.10 depicts the net change in high-latency accesses incurred when a single partition is removed. The tics on the X axis indicate which partition of data was removed. Each X-axis tic has five bars, each representing the number of additional high-latency accesses that occur when that partition of data is withheld. For example, tic D in Figure 4.10 shows that the removal of data for Partition D results in no additional high-latency accesses of resources in Partitions A and C, one for Partition B, 14 for Partition D, and two for Partition E. The "all" tic on the X axis represents the scenario in which rule mining is not used at all. From this tic, we can see that rule mining with the given system parameters can prevent 62 high-latency accesses across all partitions.

Interestingly, the number of high-latency accesses of resources in Partitions A-C increases only slightly when rule mining is not used at all. This indicates that our techniques are primarily resolving misconfigurations pertaining to the resources in Partitions D and E; that is, the resources that are shared amongst the most users. Additionally, the removal of Partition D does not substantially increase the number of high-latency accesses to resources

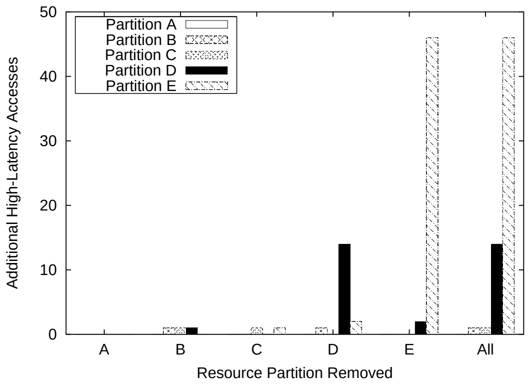


Figure 4.10: Additional high-latency accesses incurred by removal of each resource partition

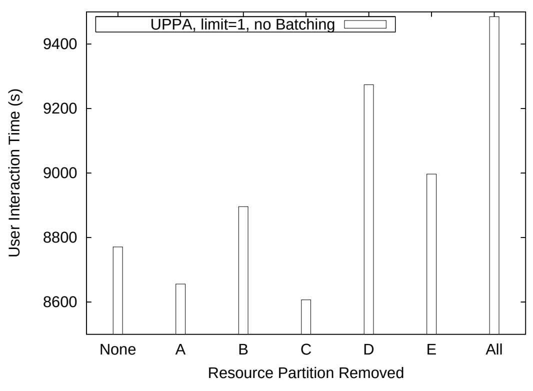


Figure 4.11: Total user interaction time with removal of each resource partition

in Partition E, and conversely, that removing Partition E produces only minor effects on Partition D.

Figure 4.11 shows the estimated user interaction time when no partitions are withheld (the "none" tic), when each resource partition is withheld, and when rule mining is not used at all (the "all" tic). For each resource partition, withholding that partition's data still results in an overall decrease in user interaction time relative to the baseline scenario without rule mining. Additionally, withholding Partitions A and C reduces the user interaction time by approximately 1% relative to the scenario where no partitions are removed. The data from these partitions results in the construction of several relatively inaccurate rules, so removing them will decrease the number of incorrect predictions that must be resolved. This gain in user interaction time comes at a cost of no additional high-latency accesses for Partition A, and two additional high-latency accesses for Partition C. Thus, the performance of the rule mining is strictly superior without Partition A in that it reduces user interaction time without incurring any additional high-latency accesses.

User Partitions We repeat the previous evaluation, but partition the dataset by users rather than resources. Again, we measure the number of high-latency accesses and total user interaction time. Figure 4.12 depicts the number of high-latency accesses incurred by removing one user partition at a time. If Alice is a member of Partition J, her high-latency access of `door1` will only be attributed to Partition J. In contrast to resource partitions, we see (by inspecting the "all" tic) that the benefit of using rule mining is spread more evenly across all partitions of users, though the users who access the most distinct resources still gain the greatest benefit. Again, the negative effects of removing a partition of users are largely constrained to the users included in that partition.

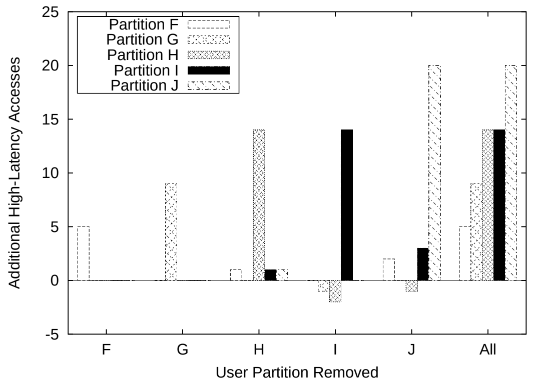


Figure 4.12: Additional high-latency accesses incurred by removal of each user partition

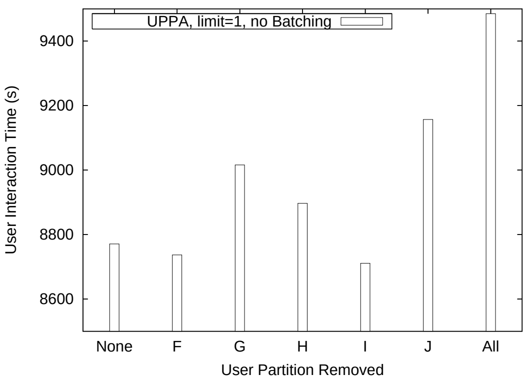


Figure 4.13: Total user interaction time with removal of each user partition

Interestingly, removing Partition I actually *decreases* the number of high-latency accesses that occur in Partitions G by two and Partition H by one. Similarly, removing Partition J decreases the high-latency accesses in Partition H by one. Removing a partition of data often changes the percentage of cases in which a rule holds (i.e., its confidence). In this manner, removing Partitions I and J increases the confidence of a few rules enough to exceed *minconf* and be used in the prediction process. These new rules identify several additional misconfigurations that result in a further decrease in the number of high-latency accesses.

The estimated user interaction time with each user partition withheld is shown in Figure 4.11. As was the case for resource partitions, the user interaction time with each user partition withheld remains lower than the baseline in which rule mining is not used. Removing Partitions F and I results in less user interaction than the scenario where all data is used for rule mining. Since the removal of Partitions F and I do not result in any additional high-latency accesses outside of their partitions, we can conclude that, in our dataset, these partitions are poor indicators of the policy for users in other partitions.

Partitioning based on total accesses We also evaluate a partitioning strategy that sorted based on total accesses rather than unique accesses. While the resulting partitions are similar, they are not identical. For example, one of the users who accessed the fewest unique resources (and was therefore located in Partition F) accessed these resources extremely often. When sorting on total accesses, this user would fall into Partition J. While specific results differ when partitioning by total accesses, we find that our overall conclusions held with this partitioning strategy as well.

4.3.3 Discussion

Given the partitioning strategies described above, rule mining continues to produce useful results even when a fifth of the input data is removed. In particular we found that removing Partitions A, F, and I results in the same or fewer high-latency accesses in other partitions as well as a slight decrease in the total user interaction time. The removal of the remaining partitions does result in additional high latency accesses outside of the removed partition; however, the effects are generally small. Removing Partition J results in the largest increase, with four additional high-latency accesses. Removing any one of Partitions B, D, E, G, H, or J slightly increases the user interaction time (by a maximum of 6% for Partition D), but in each case the interaction time remains below that of the scenario where rule mining is not used. These results indicate that, for our dataset, subsets of users or resources can be removed for privacy or scalability reasons without significantly impacting the experience of the remaining users.

4.4 Further Discussion

Large policies As our dataset represents a somewhat small deployment, an interesting question is to what extent our approach scales to larger systems. There are two aspects of scalability to consider: the extent to which rule mining can identify useful patterns in the logs of large systems and the computational complexity required to mine these patterns. As the number of resources a user may access will not increase proportionally with the size of the organization, the matrix of past accesses is likely be more sparsely populated in large systems than in ours. Thus, any resource that is shared by a large number of users will produce rules that are clearly distinctive in terms of confidence and support. However, data from large systems will, in absolute terms, contain a greater number of spurious patterns. Further research is necessary to determine whether tuning the rule mining parameters to ignore spurious patterns would negatively affect our ability to mine rules concerning resources shared by a small group of users.

As the complexity of mining rules grows with the number of attributes (i.e., resources in our system), scaling to environments with ever larger numbers of resources will eventually pose a problem to a centralized rule miner. However, recent algorithms can efficiently mine rules on datasets containing 10,000 attributes and 50,000 records [76]. In even larger organizations, the useful patterns are likely to be with respect to localized resources (e.g., resources for an individual lab or building). The data for rule mining process could therefore be partitioned along these boundaries and mined on different machines. In addition to improving scalability, decentralizing the data may mitigate users' privacy concerns and remove the requirement that a single entity have a global view of the entire system.

Data representation for rule mining In our approach, the input to the rule mining algorithm is simply a collection of boolean values, each indicating whether a particular user has accessed a particular resource. This representation, while effective in our scenario, excludes other potentially useful information. In particular, this representation does not include how often a user accessed a resource, the time of the most recent access, or any explicit structure between resources. This information is relevant to the prediction of future accesses, and if included, could improve our ability to proactively resolve misconfigurations.

For example, consider a scenario in which both Alice and Bob have each accessed resources A and B many times. Should Alice access resource C once, our technique would predict that Bob is now likely to access C. This prediction may be incorrect if Alice was granted temporary access to C due to exceptional circumstances. In this case, the discrepancy between the number of times Alice accessed A and B and the number of times she accessed C might indicate that the rule $A \land B \to C$ should be ignored until Alice has accessed C a few more times. The drawback to this approach is that delaying predictions until more data becomes available could impede our ability to resolve misconfigurations before the time of access.

Additionally, the time elapsed since a user last accessed a resource can indicate the relevance of a rule mined from that data. Continuing the above example, the rule $A \land B \to C$ is more likely to reflect current access-control policy if Alice accessed C yesterday than if she accessed C over a year ago. As described, our techniques do not distinguish between the two scenarios. Thus, while Alice's ability to access C may be temporary, our technique assumes that access is granted permanently. To counter this, access data could be discarded after it reaches a certain age. This would allow the system to adapt to situations in which access is revoked after a period of time. The length of this interval would need to be carefully selected to balance the desire to mine rules from as much data as possible with the desire to quickly adapt when access is revoked.

The third type of information that our data representation does not capture is any explicit structure between resources. For example, given that Alice is authorized to write to dir/fileA, the properties of the file system may imply that she is also authorized to list the contents of dir, but do not imply anything about her ability to write to dir/fileB. Our techniques treat dir, dir/fileA, and dir/fileB as independent resources. This approach has two drawbacks: (1) it requires logs of multiple accesses to construct rules that are obvious from the structure of the resources (e.g., dir/fileA $\to$ dir) and (2) by considering each resource independently, it may be unable to mine more general trends, e.g., dir $\to$ doorA. This could occur if Alice writes to dir/fileA, Bob writes to dir/fileB, and both Alice and Bob access doorA. When each resource is considered in isolation, there is insufficient data to discover the relationship between dir and doorA. One way of addressing this is to consider the structure of the resources when constructing the data that is given to the rule mining algorithm, e.g., each write to dir/fileA could also count as a listing of dir. Further investigation is necessary to determine if this approach is sufficiently general to cover all possible resource structures.

Directing resolution requests Although our techniques for directing resolution requests are relatively simple, we showed that they were able to successfully resolve up to 95% of the correctly identified misconfigurations. However, some scenarios may necessitate more advanced heuristics for directing requests. For example, consider a scenario in which multiple administrators govern a single resource, but each administrator delegates only to a non-overlapping subset of users. The resource could be a lab space shared between the research groups of two professors. Each professor is willing to delegate authority to access the lab to his or her students, but not to the students of the other professor. If we predict that a new student will access the lab, our technique would simply direct the resolution request to the administrator who has resolved the most misconfigurations in the past, which would be the professor with the larger research group. A more advanced heuristic might also consider the new student's access history and the rule used to make the prediction to determine which group the student belongs to.

Another drawback of directing resolution requests to the administrator who has resolved the most related requests in the past is that it effectively penalizes an administrator for providing assistance. This problem is particularly acute in a scenario where several administrators share equal responsibility for a resource. One possible way to address this concern would be to create a group containing the most helpful administrators in the candidate list, then randomly select an administrator from that group. If the group selection parameters are chosen correctly, this approach could ensure that requests are spread equally among the administrators responsible for the resource.

4.5 Related Work

Related work falls, broadly speaking, into three categories: works that use similar data mining or machine learning techniques to analyze access-control policy, works that focus on assisting users in policy administration, and distributed access-control systems to which our techniques are applicable.

Policy analysis The objectives of our work are related to those of *role mining* (e.g., [49, 69]), in which machine-learning techniques are used to extract roles from implemented policy. These roles may then serve as a guide when migrating a legacy system to one that supports role-based access control. Vaidya et al. formally define the role mining problem and bound its complexity [70]. In contrast, our techniques do not seek to characterize the implemented access-control policy. Instead, we aim to discover the portions of intended access-control policy that have not been implemented. Additionally, our techniques operate on the basis of observed behavior and are agnostic to the policy-specification language.

Many tools for empirical access-control policy analysis have been developed for firewalls (e.g., [10, 60, 42, 74, 5, 75]). These tools generally provide ways to test or validate firewall policy against administrator intentions or other rules. Our work differs from these in multiple ways. First, since in the firewall setting there is typically one authority for the proper access-control policy (the human administrator or a high-level specification of that policy), there is no analog in that domain to a central concern here, namely determining with whom to inquire about a potential policy change and minimizing the costs for doing so. Second, because it has the benefit of a policy authority that can be consulted freely, firewall analysis has focused on detecting traffic permitted in violation of policy, i.e., to improve security, at least as much as what additional traffic should be allowed. The central technique we employ here, namely learning from allowed accesses to predict others that are likely to be intended, focuses exclusively on improving the *usability* of discretionary access controls granted in a least-privilege manner.

The use of data-mining algorithms for detecting misconfigurations has recently been treated in several domains. Perhaps most closely related to our work is Minerals [51], a system which uses association rule mining to detect router misconfigurations. By applying association rule mining to router configuration files in a network, Minerals detected misconfigurations such as router interfaces using private IP addresses that should have been deleted, user accounts missing passwords, and BGP errors that could result in unintentionally providing transit service or that could open routers to attack. The work of El-Arini and Killourhy [32] similarly seeks to detect router misconfigurations as statistical anomalies within a Bayesian framework. As in the aforementioned works in firewall analysis, though, whom to consult about apparent configuration errors and minimizing the costs of doing so were not issues considered in these works; these issues are central to ours, however.

Policy administration Jaeger et al. seek to characterize the ways in which implemented policy may conflict with constraints that specify what accesses are prohibited [43]. Their tool, Gokyo, allows the administrator to view the conflicts, and determine the least complex way of correcting them, possibly by marking them as explicit exceptions to specified policy. This tool could assist users of our techniques to resolve misconfigurations without introducing conflicts.

Many accesses in the context of health-care systems are granted by exception rather than because they are consistent with implemented access-control policy [22]. Using access-control logs that are annotated with whether or not an access was granted by exception, PRIMA uses heuristics to extract the exceptions that occur often enough to warrant further attention [22]. An administrator must then determine if the exception represents a misconfiguration. Applied frequently, PRIMA will help administrators to refine the implemented policy so that it expands to include exceptions that are consistent with the intended policy. As with our work, success is measured in terms of the coverage of intended policy. Our techniques differ from theirs in that we use data mining to identify exceptions and that we attempt to correct the implemented policy prior to the time of access.

Distributed access-control systems Our techniques operate using only data that describes past access attempts and who was contacted to resolve any misconfigurations. As such, our techniques should apply to systems employing a wide variety of access-control frameworks, such as RBAC [68], SPKI/SDSI [66], RT [56], or PCA [6]. We utilize observed behavior to determine to whom a resolution request should be directed. Some systems, such as PeerAccess [72], explicitly encode hints as to how queries should be directed. Such hints could be considered in conjunction with observed behavior.

Though we have conducted our study in the context of Grey, there are numerous systems that we believe are well equipped to utilize the techniques we develop here. In particular, similar to Grey's support for dynamic delegation, many other systems enable retrieving credentials remotely from other parties as a means for satisfying access-control policy (e.g., [24, 45, 48, 38, 19, 56]). These mechanisms could be used to drive the correction of access-control misconfigurations once they are detected, though they do not detect those misconfigurations or predict how they might be corrected, as we have studied here. Trust-X [21] proposes a mechanism in which subsets of policy cached from previous access-control decisions can be used to more quickly grant access in the future. This is effective when the implemented policies that govern different access requests share many common components. Our technique is complementary in that we focus on addressing the discrepancy between implemented policy and intended policy.

4.6 Conclusion

In various application contexts (e.g., health-care systems), the consequences of unnecessary delays for access to information can be severe. In such settings, it essential to eliminate access-control policy misconfigurations in advance of attempted accesses; even in less critical environments, doing so can greatly improve the usability of the access-control system. In this chapter we have shown how to eliminate a large percentage of such misconfigurations in advance of attempted accesses using a data-mining technique called association rule mining.

We demonstrated that by doing so, we can greatly reduce the critical-path delays to accesses, while inducing little additional overall work on users of the system. Specifically, we showed, using data from a deployed access-control system, that our methods can reduce the number of accesses that would have incurred a costly time-of-access delay by 44%, and can correctly predict 58% of the intended policy. These gains are achieved without increasing the total amount of time users spend interacting with the system. To accomplish these results, we contributed both new uses of rule mining and novel approaches for determining the users to which to make suggestions for changing policy. These results should be applicable to a wide range of discretionary access-control systems and settings, but particularly for systems that provide automated support for resolving misconfigurations via dynamic delegation.

Chapter 5

Conclusion

In this thesis, we present a collection of techniques for efficiently demonstrating that a request to access a resource should be granted in a distributed access-control system that is based on formal logic. It is imperative that this process complete in a timely fashion, as unnecessary delays can severely diminish the usability of the system [12]. However, in the context of a practical distributed access-control system, there are several factors that make it difficult to construct a proof. First, the credentials that encode access-control policy are distributed among the nodes of the system. Second, the proof-construction algorithm must consider human factors, such as the social cost of making a request for assistance and the need to guide users to create new credentials reactively in response to a query. Third, the system should attempt to detect misconfigurations in access-control policy and try to resolve them before they result in a legitimate access being delayed or denied.

Our techniques consist of (i) a procedure for distributing the process of constructing a proof, (ii) a method for efficiently constructing proofs in a way that exposes choice points, allows reactive credential creation, and fully utilizes locally known credentials, and (iii) a technique for proactively identifying and resolving misconfigurations in access-control policy.

Distributed proof construction We present a technique by which the proof construction process itself is distributed between parties. This approach, which we term *lazy* is advantageous in that the proof search is performed by the party most likely to either already have or be willing to create the credentials needed to demonstrate that a formula is true. We show analytically that the proving ability of the lazy approach is strictly greater than a centralized (or *eager*) approach that retrieves remote credentials as needed. We show empirically that the lazy approach requires significantly fewer requests for assistance to construct a proof than does the eager strategy, and that on subsequent queries, the advantage of the lazy strategy widens as it is better able to make use of cached results than the eager strategy.

Importantly, the lazy strategy enables a remote party a great deal of freedom as to how to proceed with the proof search. This flexibility can, for example, allow the remote node to use an alternative tactic set or to incorporate guidance provided by the user of the remote node.

Efficient, usable proof construction Preliminary experience from our deployment led us to define three requirements that a proof-construction strategy must meet in order to provide a positive user experience. First, a proof-construction strategy must notify users when they could create a credential that would allow a proof to be constructed. Second, choice points should be exposed to users to allow them to guide the proof search in the manner that they deem to be the most appropriate. Third, the proof-construction strategy should always find a proof if one can be derived from local knowledge.

Using our distributed proving technique as a foundation, we present a novel strategy for constructing proofs that meets each of the above requirements. This strategy makes use of three key insights: we utilize a delayed proof search in conjunction with user input to minimize the number of expensive proof steps (such as asking for remote assistance), we precompute all possible facts using the well-studied forward-chaining algorithm and all possible paths using a path compression algorithm which we introduce in this thesis, and we present a technique for systematically generating tactics from the inference rules of the logic so that tactics no longer need to be hand crafted.

As with our distributed approach to proof construction, we show analytically that these techniques offer strictly superior proving ability to a centralized approach. We show empirically that our techniques yield dramatic gains in efficiency when compared to our previous hand-tuned approach using both policies drawn from our deployment and policies indicative of a much larger deployment. Our analysis shows that our techniques offer benefit in an environment with tabling, and the side-effects of precomputation are reasonable for many practical scenarios.

Identifying and resolving policy misconfigurations We present techniques for identifying and resolving misconfigurations in access-control policy before those misconfigurations result in legitimate access being delayed or denied. Based on the observation that access-control policy exhibits patterns, and that these patterns are observable from logs of past accesses, we apply association rule mining to mine rules from these logs. Situations in which the mined rules do not hold represent potential misconfigurations. Once a potential misconfiguration has been identified, we utilize past user behavior to determine the most appropriate principal to contact to determine if policy should be modified, and if so, to perform the appropriate modification.

We evaluate our techniques using 14 months of data drawn from the Grey deployment. We find that, for reasonable parameter values, we can correctly identify 58% of the misconfigurations in the intended access-control policy. Proactively resolving these misconfigurations allows us to eliminate 44% of the accesses that would incur a lengthy delay to correct a misconfiguration. In addition, we find that our techniques are resilient to removing portions of the collected data. In particular, the impact of removing data pertaining to a subset of users or resources is largely born by the members of that subset.

The difficulty of constructing a proof demonstrating that access should be granted is a significant impediment the deployment of distributed access-control systems based on formal logic. This thesis presents a collection of techniques for constructing such proofs in a manner that meets the various requirements of practical system. To our knowledge, ours is the first such work to do so. We demonstrate analytically that the proving ability of our techniques is strictly greater than centralized solutions, and we show empirically that our techniques offer significant improvement in many practical scenarios. We demonstrate the practicality of our work through integration into the Grey system, where the techniques proposed in Chapters 2 and 3 have been in active use for over two years.

Bibliography

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

[2] Martin Abadi, Michael Burrows, Butler Lampson, and Gordon D. Plotkin. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(4):706–734, September 1993.

[3] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules. In *Proceedings 20th International Conference on Very Large Data Bases, VLDB*, 1994.

[4] Jalal Al-Muhtadi, Anand Ranganathan, Roy Campbell, and M. Dennis Mickunas. Cerberus: a context-aware security scheme for smart spaces. In *Proceedings of the First IEEE International Conference on Pervasive Computing and Communications*, 2003.

[5] Ehab S. Al-Shaer and Hazem H. Hamed. Discovery of policy anomalies in distributed firewalls. In *Proceedings of the 23rd INFOCOM*, March 2004.

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

[7] Andrew W. Appel, Neophytos G. Michael, Aaron Stump, and Roberto Virga. A trustworthy proof checker. *Journal of Automated Reasoning*, 31:231–260, 2003.

[8] Dirk Balfanz, Drew Dean, and Mike Spreitzer. A security infrastructure for distributed Java applications. In *Proceedings of the 2000 IEEE Symposium on Security & Privacy*, 2000.

[9] Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D. Ullman. Magic sets and other strange ways to implement logic programs (extended abstract). In *Proceedings of the fifth ACM SIGACT-SIGMOD symposium on Principles of Database Systems*, 1985.

[10] Yair Bartal, Alain J. Mayer, Kobbi Nissim, and Avishai Wool. Firmato: A novel firewall management toolkit. In *Proceedings of the 1999 IEEE Symposium on Security*

and Privacy, May 1999.

[11] Lujo Bauer, Lorrie Cranor, Robert W. Reeder, Michael K. Reiter, and Kami Vaniea. A user study of policy creation in a flexible access-control system. In *CHI 2008: Conference on Human Factors in Computing Systems*, April 2008.

[12] Lujo Bauer, Lorrie F. Cranor, Michael K. Reiter, and Kami Vaniea. Lessons learned from the deployment of a smartphone-based access-control system. In *Proceedings of the 3rd Symposium on Usable Privacy and Security*, July 2007.

[13] Lujo Bauer, Scott Garriss, Jonathan M. McCune, Michael K. Reiter, Jason Rouse, and Peter Rutenbar. Device-enabled authorization in the Grey system. In *Information Security: 8th International Conference, ISC 2005 (Lecture Notes in Computer Science 3650)*, pages 63–81, 2005.

[14] Lujo Bauer, Scott Garriss, and Michael K. Reiter. Distributed proving in acess-control systems. In *Proceedings of the 2005 IEEE Symposium on Security & Privacy*, 2005.

[15] Lujo Bauer, Scott Garriss, and Michael K. Reiter. Efficient proving for practical distributed access-control systems. In *Proceedings of the 12th European Symposium on Research in Computer Security (ESORICS)*, 2007.

[16] Lujo Bauer, Scott Garriss, and Michael K. Reiter. Efficient proving for practical distributed access-control systems. Technical Report CMU-CyLab-06-015R, Carnegie Mellon University, 2007.

[17] Lujo Bauer, Scott Garriss, and Michael K. Reiter. Detecting and resolving policy misconfigurations in access-control systems. In *Proceedings of the 13th ACM Symposium on Access Control Models and Technologies (SACMAT)*, 2008.

[18] Lujo Bauer, Michael A. Schneider, and Edward W. Felten. A general and flexible access-control system for the Web. In *Proceedings of the 11th USENIX Security Symposium*, 2002.

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

[20] Moritz Y. Becker, Cedric Fournet, and Andrew D. Gordon. Design and semantics of a decentralized authorization language. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium (CSF)*, 2007.

[21] Elisa Bertino, Elena Ferrari, and Anna Cinzia Squicciarini. Trust-X: A peer to peer framework for trust establishment. *IEEE Transactions on Knowledge and Data Engineering*, 16(7):827–842, 2004.

[22] Rafae Bhatti and Tyrone Grandison. Towards improved privacy policy coverage in healthcare using policy refinement. In *Proceedings of the 4th VLDB Workshop on Secure Data Management*, 2007.

[23] Matt Blaze, Joan Feigenbaum, John Ioannidis, and Angelos D. Keromytis. *The KeyNote trust-management system, version 2*, 1999. IETF RFC 2704.

[24] Matt Blaze, Joan Feigenbaum, and Jack Lacy. Decentralized trust management. In *Proceedings of the 1996 IEEE Symposium on Security & Privacy*, 1996.

[25] Matt Blaze, Joan Feigenbaum, and Martin 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.

[26] Michael Burrows, Martín Abadi, and Roger Needham. A logic of authentication. *ACM Transactions on Computer Systems*, 8(1):18–36, 1990.

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

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

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

[30] Henry DeYoung, Deepak Garg, and Frank Pfenning. An authorization logic with explicit time. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium (CSF)*, 2008.

[31] Blair Dillaway. A unified approach to trust, delegation, and authorization in large-scale grids. Technical report, Microsoft, 2006.

[32] Khalid El-Arini and Kevin Killourhy. Bayesian detection of router configuration anomalies. In *Proceedings of the 2005 ACM SIGCOMM Workshop on Mining Network Data*, August 2005.

[33] Carl M. Ellison, Bill Frantz, Butler Lampson, Ronald L. Rivest, Brian M. Thomas, and Tatu Ylonen. Simple public key certificate. Internet Engineering Task Force Draft IETF, 1997.

[34] Carl M. Ellison, Bill Frantz, Butler Lampson, Ronald L. Rivest, Brian M. Thomas, and Tatu Ylonen. *SPKI Certificate Theory*, 1999. RFC2693.

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

[36] Deepak Garg and Frank Pfenning. Non-interference in constructive authorization logic.

In *Proceedings of the 19th Computer Security Foundations Workshop (CSFW'06)*, 2006.

[37] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):51–59, June 2002.

[38] 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.

[39] Carl A. Gunter and Trevor Jim. Policy-directed certificate retrieval. *Software—Practice and Experience*, 30(15):1609–1640, December 2000.

[40] Yuri Gurevich and Itay Neeman. Dkal: Distributed-knowledge authorization language. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium (CSF)*, 2008.

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

[42] Scott Hazelhurst, Adi Attar, and Raymond Sinnappan. Algorithms for improving the dependability of firewall and filter rule lists. In *Proceedings of the 2000 International Conference on Dependable Systems and Networks*, June 2000.

[43] Trent Jaeger, Antony Edwards, and Xioalan Zhang. Policy management using access control spaces. *ACM Transaction on Information and System Security*, 6(3):327–364, 2003.

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

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

[46] Trevor Jim and Dan 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.

[47] Apu Kapadia, Geetanjali Sampemane, and Roy 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.

[48] Angelos D. Keromytis, Sotiris Ioannidis, Michael B. Greenwald, and Jonathan M. Smith. The STRONGMAN architecture. In *Third DARPA Information Survivability Conference and Exposition*, 2003.

[49] Martin Kuhlmann, Dalia Shohat, and Gerhard Schimpf. Role mining—revealing business roles for security administration using data mining technology. In *Proceedings of the 8th ACM Symposium on Access Control Models and Technologies (SACMAT)*, June 2003.

[50] Butler Lampson, Martin Abadi, Michael Burrows, and Edward Wobber. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems*, 10(4):265–310, 1992.

[51] Franck Le, Sihyung Lee, Tina Wong, Hyong S. Kim, and Darrell Newcomb. Minerals: Using data mining to detect router misconfigurations. In *MineNet '06: Proceedings of the 2006 SIGCOMM Workshop on Mining Network Data*, pages 293–298, 2006.

[52] Adam J. Lee, Kazuhiro Minami, and Marianne Winslett. Lightweight consistency enforcement schemes for distributed proofs with hidden subtrees. In *Proceedings of the 12th ACM Symposium on Access Control Models and Technologies (SACMAT)*, 2007.

[53] Adam J. Lee and Marianne Winslett. Safety and consistency in policy-based authorization systems. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, 2006.

[54] Chris Lesniewski-Laas, Bryan Ford, Jacob Strauss, Robert Morris, and M. Frans Kaashoek. Alpaca: Extensible authorization for distributed services. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, 2007.

[55] Jiangtou Li, Ninghui Li, and William H. Winsborough. Automated trust negotiation using cryptographic credentials. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, 2005.

[56] Ninghui Li and John C. Mitchell. Rt: A role-based trust-management framework. In *Proceedings of The Third DARPA Information Survivability Conference and Exposition*, 2003.

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

[58] Ninghui Li, John C. Mitchell, and William H. Winsborough. Design of a role-based trust management framework. In *Proceedings of the 2002 IEEE Symposium on Security & Privacy*, 2002.

[59] Ninghui Li, William H. Winsborough, and John C. Mitchell. Distributed credential chain discovery in trust management. *Journal of Computer Security*, 11(1):35–86, 2003.

[60] Alain Mayer, Avishai Wool, and Elisha Ziskind. Fang: A firewall analysis engine. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, May 2000.

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

[62] Kazuhiro Minami and David Kotz. Secure context-sensitive authorization. *Journal of Pervasive and Mobile Computing*, 1(1), 2005.

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

[64] Tom M. Mitchell, Richard M. Keller, and Smadar T. Kedar-Cabelli. Explanation-based generalization: A unifying view. *Machine Learning*, 1(1):47–80, 1986.

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

[66] Ronald L. Rivest and Butler Lampson. SDSI—A simple distributed security infrastructure. Presented at CRYPTO'96 Rumpsession, April 1996.

[67] Stuart Russell and Peter Norvig. *Artificial Intelligence, A Modern Approach*. Prentice Hall, second edition, 2003.

[68] Ravi Sandhu, Edward Coyne, Hal Feinstein, and Charles Youman. Role-based access control models. *IEEE Computer*, 29(2), 1996.

[69] Jurgen Schlegelmilch and Ulrike Steffens. Role mining with ORCA. In *Proceedings of the 10th ACM Symposium on Access Control Models and Technologies (SACMAT)*, June 2005.

[70] Jaideep Vaidya, Vijayalakshmi Atluri, and Qi Guo. The role mining problem: Finding a minimal descriptive set of roles. In *Proceedings of the 12th ACM Symposium on Access Control Models and Technologies (SACMAT)*, 2007.

[71] William H. Winsborough and Ninghui Li. Towards practical automated trust negotiation. In *Proceedings of the Third International Workshop on Policies for Distributed Systems and Networks (Policy 2002)*, 2002.

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

[73] Edward Wobber, Martin Abadi, Michael Burrows, and Butler Lampson. Authentication in the Taos operating system. *ACM Transactions on Computer Systems*, 12(1):3–32, 1994.

[74] Avishai Wool. Architecting the Lumeta firewall analyzer. In *Proceedings of the 10th USENIX Security Symposium*, 2001.

[75] Lihua Yuan, Jianning Mai, Zhendong Su, Hao Chen, Chen-Nee Chuah, and Prasant

Mohapatra. FIREMAN: A toolkit for FIREwall modeling and ANalysis. In *Proceedings of the 2006 IEEE Symposium on Security & Privacy*, 2006.

[76] Yubo Yuan and Tingzhu Huang. A matrix algorithm for mining association rules. In *International Conference on Intelligent Computing (ICIC)*, 2005.

[77] Charles Zhang and Marianne Winslett. Multitrust: An authorization framework with customizable distributed proof construction. In *Proceedings of the Joint Workshop on Foundations of Computer Security, Automated Reasoning for Security Protocol Analysis, and Issues in the Theory of Security (FCS-ARSPA-WITS'08)*, 2008.

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

Appendix A

Proof of Theorems

A.1 Proof of Termination for a Distributed Prover

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

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

A.1.1 Lemma 2

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

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

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

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

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

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

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

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

8r Since DP made the RPC to [8d].l, [8r].l is $\mathit{localmachine}$.

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

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

15r Since [8r].$l = \mathit{localmachine}$, the body of this else statement will always be executed.

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

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

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

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

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

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

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

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

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

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

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

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

A.1.2 Lemma 3

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

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

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

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

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

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

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

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

In both cases, [c].*failures* = [d].*failures*, and so [9c].$\mathit{failures}'$ = [9d].$\mathit{failures}'$.

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

1 is an assumption of the current lemma as well.

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

3 is true by the definition of Case A.

4 is true by our induction hypothesis.

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

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

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

□

A.1.3 Theorem 1

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

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

A.2 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 3.4, 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 unify$(a, b) \neq \perp$ then we will write $a = b$ and use a and b interchangeably. We let [kp] represent line k in PC, and [ka] represent line k in add-path.

Lemma 4 *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*$, (\text{unify}(y', x) \neq \perp) \supset (x', y) \in$ *paths*.

Proof We prove Lemma 4 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*, $(\text{unify}(y', x) \neq \perp) \supset (x', y) \in$ *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 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 $\text{unify}(q, p') \neq \bot$ 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 $\text{unify}(q', p'') \neq \bot$.

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 4 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 add-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 add-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 4), 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, $(x', p) \in \mathit{paths}$ and from Step 2.2 we know that by the time [24a] is reached, $(p, y) \in \mathit{newPaths}$. 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 add-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 $(q, y) \in \mathit{paths}$, Observation 1 tells us that by the time [24a] is reached, $(p, y) \in \mathit{newPaths}$. Since $(x', p) \in \mathit{paths}$ and $(p, y) \in \mathit{newPaths}$, Observation 2 tells us that when add-path terminates, $(x', y) \in \mathit{paths}$ 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 $\forall (x, y) \in \mathit{paths}$, $\forall (x', y') \in \mathit{paths}$, $(\mathsf{unify}(y', x) \neq \bot) \supset (x', y) \in \mathit{paths}$ holds for the base case and the inductive case, we can conclude that Lemma 4 holds. □

Lemma 5 *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 $(p, q) \in \mathit{paths}$.*
2. *If $((p, q) \leftarrow \mathsf{credToPath}(c_j)) \neq \bot$, $\pi \leftarrow \mathsf{depends\text{-}on}(c_j)$ and $\pi \in \mathit{paths}$, $\mathsf{add\text{-}path}((p, q))$ has been invoked and $(p, q) \in \mathit{paths}$.*

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 5 by induction over the number of invocations i of PC. Our induction hypothesis is that conditions 1-2 of Lemma 5 (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 $\pi \in \mathit{paths}$ as is required by the second condition of Lemma 5. 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 $\pi \notin \mathit{paths}$, 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 $c_j \in \mathit{incompletePaths}$ (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 $\pi \in \mathit{paths}$.

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 $\mathcal{C}$. □

Theorem 4 *If* PC *has completed on KB, then for any* A, B *such that* $A \neq B$*, if for some* F $(B$ **says** $F \supset\ A$ **says** $F)$ *in the context of KB, then* $(B$ **says** F, A **says** $F) \in KB$*.*

Proof If $(B$ **says** $F \supset\ A$ **says** $F)$ is true, then there must exist a set of delegation credentials from which we can conclude $(B$ **says** $F \supset\ A$ **says** $F)$. Since all credentials are given as input to PC, from Lemma 5 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 4 to show that, from the set of credentials given to add-path, all possible paths have been constructed, thus proving Theorem 4. □

A.3 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 in Figure 2.1. For the purpose of formal comparison, we assume that all caching optimizations described in Chapter 2 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 bc-ask$_I$. bc-ask$_I$ outputs either a complete proof or $\perp$, while bc-ask$_D$ (shown in Figure 3.2) may additionally output a marker indicating a choice subgoal that needs to be proved. As such, a wrapper mechanism must be used to repeatedly invoke bc-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 bc-ask$_I$ is an example. To construct a distributed prover using bc-ask$_D$, we define a wrapper, bc$_D$ (shown in Figure A.1) 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 bc-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 bc-ask$_D$, $[kbcd]$ represent line k in bc$_D$, and $[ki]$ represent line k in bc-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.

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.*

```
0   ⟨substitution, credential[ ]⟩ $\mathsf{bc}_D$(          /* returns a substitution */
        list $goals$,                                        /* list of conjuncts forming a query */
        substitution $\theta$,                               /* current substitution, initially empty */
        set $failures$)                                      /* set of substitutions that are known
                                                                not to produce a complete solution */
1       local set $markers$, $failures'$
2       $failures' \leftarrow failures$
3       while $((\langle \beta, creds\rangle \leftarrow \mathsf{bc\text{-}ask}_D(goals, \theta, failures')) \neq \bot)$   /* find all solutions */
4         if $\mathsf{notMarker}(\beta)$, return $\langle \beta, creds\rangle$      /* if complete proof found, return*/
5         $markers \leftarrow markers \cup \{\beta\}$
6         $failures \leftarrow failures \cup \{\beta\}$

7       for each $m \in markers$
8         $\langle f, \theta, failures'\rangle \leftarrow m$
9         while$((\langle \alpha, creds\rangle \leftarrow \mathsf{rpc}_l(\mathsf{bc}_D(f, \theta, failures'))) \neq \bot)$
10            $failures' \leftarrow \alpha \cup failures'$
11            $\mathsf{addToKB}(creds)$
12            $\langle \beta, creds\rangle \leftarrow \mathsf{bc}_D(goals, \theta, failures)$
13            if $(\beta \neq \bot)$, return $\langle \beta, creds\rangle$
14      return $\langle \bot, null\rangle$
```

Figure A.1: bc_D, a wrapper to process partial proofs returned by $\mathsf{bc\text{-}ask}_D$

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 6 *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 3.7, 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 7 *For any goal G and knowledge base KB, $\mathsf{bc\text{-}ask}_D$ using tactic set $\mathcal{T}$ will find a proof of G without making any remote requests if $\mathsf{bc\text{-}ask}_I$ using $\mathcal{T}$ will find a proof of G without making any remote requests.*

Proof sketch If $\mathsf{bc\text{-}ask}_I$ 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 $\mathsf{determine\text{-}location}(A) \neq \mathit{localmachine}$. Our induction hypothesis states that if both $\mathsf{bc\text{-}ask}_I$ and $\mathsf{bc\text{-}ask}_D$ make a recursive call with identical environments that will have recursion height h, then the recursive $\mathsf{bc\text{-}ask}_D$ call will return the same result as the recursive $\mathsf{bc\text{-}ask}_I$ call.

Base case: The base case is when the recursion height $= 0$, which occurs when $\mathit{goals} = [\,]$. Since an assumption of this case is that all input parameters to $\mathsf{bc\text{-}ask}_D$ are the same as $\mathsf{bc\text{-}ask}_I$, by inspection of the algorithms ([3d]-[4d], [5i]-[6i]), $\mathsf{bc\text{-}ask}_D$ and $\mathsf{bc\text{-}ask}_I$ will both return $\perp$ if $\theta \in \mathit{failures}$, or θ otherwise.

Inductive case: For the inductive case, we assume that, at recursion height $h+1$, $\mathsf{bc\text{-}ask}_D$ was invoked with the same parameters as $\mathsf{bc\text{-}ask}_I$. Since, by the assumptions of this lemma, $\mathsf{bc\text{-}ask}_I$ 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, $\mathsf{bc\text{-}ask}_I$ will continue to [17i].

Case 1: If (P, q) is a tactic, then P will be non-empty, causing $\mathsf{bc\text{-}ask}_D$ 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 $\mathit{failures}'$, 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 $[22d].\mathit{answer} = [19i].\mathit{answer}$. If $\mathit{answer} \neq \perp$, it will be returned in both scenarios, otherwise $\mathsf{bc\text{-}ask}_D$ and $\mathsf{bc\text{-}ask}_I$ will continue to the next iteration of [19d] and [17i]. With $[22d].\mathit{answer} = [19i].\mathit{answer}$ for each iteration, if $\mathsf{bc\text{-}ask}_I$ returns a solution, $\mathsf{bc\text{-}ask}_D$ will also. Otherwise, $\mathsf{bc\text{-}ask}_D$ and $\mathsf{bc\text{-}ask}_I$ 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 $\mathsf{compose}(\theta', \theta) \in \mathit{failures}'$ ([5i]), and $\beta = \mathsf{compose}(\theta', \theta)$ ([6i]) otherwise. Note that β is added to $\mathit{failures}'$ 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 $\phi \in \mathit{failures}'$ (where $\phi = \mathsf{compose}(\theta', \theta)$ from [13d]) then [16d]-[18d] will not be executed. $\mathsf{bc\text{-}ask}_D$ will then proceed to try the next element in the knowledge base ([11d]), which is the same as the behavior of $\mathsf{bc\text{-}ask}_I$ when $[17i].\beta = \perp$. If $[15d].\phi$ is not in $\mathit{failures}'$, then [16d]-[18d] will be executed. Since ϕ is not modified between [13d] and [17d], $[17d].\phi = \mathsf{compose}(\theta', \theta) = [19i].\beta$. 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 $[17d].\mathit{answer} = [19i].\mathit{answer}$. From this, we can conclude that if $\mathsf{bc\text{-}ask}_I$ finds a proof, $\mathsf{bc\text{-}ask}_D$ will find a proof as well. □

Lemma 8 *For any distributed proving node attempting to prove goal G with knowledge base KB, bc_D will find a proof of G if $\mathsf{bc\text{-}ask}_I$ 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 8 via induction over the number of remote requests r that a local prover using $\mathsf{bc\text{-}ask}_I$ makes to complete the proof. Our induction hypothesis states that for all queries such that $\mathsf{bc\text{-}ask}_I$ finds a proof with $r-1$ requests, 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 7.
Inductive Case: We prove the inductive case by (1) showing that bc_D will eventually make a remote request that is identical to the initial remote request made by $\mathsf{bc\text{-}ask}_I$, (2) showing that when bc_D re-runs the entire query after the remote request finishes ([12*bcd*]), 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), $\mathsf{bc\text{-}ask}_D$ will recurse with the same parameters that $\mathsf{bc\text{-}ask}_I$ recurses with after $\mathsf{bc\text{-}ask}_I$ finishes its initial remote request.

Step 1: By an argument analogous to that of Lemma 7, we assert that $\mathsf{bc\text{-}ask}_D$ will behave identically to $\mathsf{bc\text{-}ask}_I$ until the point at which $\mathsf{bc\text{-}ask}_I$ encounters a goal for which it needs to make a remote request ([11*i*]). At this point, $\mathsf{bc\text{-}ask}_D$ will construct a marker ([9*d*]) containing the same parameters as $\mathsf{bc\text{-}ask}_I$ would use to make the remote request.

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

Step 2: By inspection of $\mathsf{bc\text{-}ask}_D$ (in particular, [18*d*] and [23*d*]), 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 ([9*bcd*]) 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 [11*bcd*], so from Lemma 6 we can conclude that the local prover can now generate a complete proof of f.

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

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

The knowledge base KB' used by $\mathsf{bc\text{-}ask}_D$ was initially identical to the knowledge base KB used by $\mathsf{bc\text{-}ask}_I$. However, bc_D added the credentials returned by the remote request to KB' ([11bcd]), resulting in a KB' such that $KB \subset KB'$. By Lemma 6, we can conclude that $\mathsf{bc\text{-}ask}_D$ will eventually find the same proof using KB' as it finds using KB. Thus, if we can prove Lemma 8 when $\mathsf{bc\text{-}ask}_D$ uses KB, the result will hold when $\mathsf{bc\text{-}ask}_D$ uses KB'. From the previous paragraph, we know that all parameters to the recursive $\mathsf{bc\text{-}ask}_D$ 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 $\mathsf{bc\text{-}ask}_D$ is identical to that of $\mathsf{bc\text{-}ask}_I$.

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 $\mathsf{bc\text{-}ask}_D$ will find a proof if $\mathsf{bc\text{-}ask}_I$ is able to find a proof. □

Theorem 5 *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 $\mathsf{bc\text{-}ask}_D$) will produce a complete proof if $\mathsf{bc\text{-}ask}_I$ produces a complete proof. We prove Theorem 5 by induction over the remote request height of the proof. Our induction hypothesis states that if $\mathsf{bc\text{-}ask}_I$ and bc_D are invoked with parameters P (which include goal G), and $\mathsf{bc\text{-}ask}_I$ 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 $\mathsf{bc\text{-}ask}_I$ does not make any remote requests, we can apply Lemma 7 to conclude that bc_D will produce a proof if $\mathsf{bc\text{-}ask}_I$ produces a proof.

Inductive Case: For the inductive case, we note that any remote requests made by $\mathsf{bc\text{-}ask}_I$ operating at request height $h+1$ must have height at most h. Lemma 8 proves that bc_D will find a proof of G if $\mathsf{bc\text{-}ask}_I$ 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 $\mathsf{bc\text{-}ask}_I$ 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 8 which allows us to conclude that bc_D will find a proof if $\mathsf{bc\text{-}ask}_I$ finds a proof with request height $h+1$. □

A.4 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 $\mathsf{bc\text{-}ask}_D$ (Figure 3.2) and

bc_D (Figure A.1), 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.[*]

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.

Lemma 9 *Consider the case in which* IR *is given the query* A **says** F *and each of the first series of recursive* $\mathsf{bc\text{-}ask}_D$ *calls made by* IR *is an application of a delegation rule for which the left premise is provable, and the next recursive* $\mathsf{bc\text{-}ask}_D$ *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 $\mathsf{bc\text{-}ask}_D$ calls made by IR are applications of a delegation rule for which the first premise is provable, and the $(r+1)$'th recursive $\mathsf{bc\text{-}ask}_D$ 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 4, 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 10 *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 3.6, we refer to the version of IR with cycle prevention as $\mathsf{IR\text{-}NC}$. 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 $\mathsf{bc\text{-}ask}_D$ 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 $\mathsf{bc\text{-}ask}_D$ 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 $\mathsf{IR\text{-}NC}$ and IR is that $\mathsf{IR\text{-}NC}$ eliminates cycles, $\mathsf{IR\text{-}NC}$ 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 $\mathsf{IR\text{-}NC}$ will find a proof of F on the initial attempt as well. □

Lemma 11 *If both* IR *and* LR *invoke* $\mathsf{bc\text{-}ask}_D$ *with identical parameters and* IR *finds a complete proof from local knowledge, then* LR *will find a complete proof from local knowledge as well.*

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

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 3 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 12 *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 12, 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 12 will hold for proofs with depth at most d.
Base case: The base case occurs when $d = 0$. Since Lemma 12 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 **says** F. The proof of A **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 **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 $q = A$ **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 10 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 3.5.2). 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 9 to show that both IR and LR will ultimately investigate the same right subgoal (e.g., B **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 6 *If* IR *finds a proof of goal* F*, then* LR *will find a proof of* F *as well.*

Proof sketch We prove Theorem 6 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, $\mathsf{bc\text{-}ask}_D$ will return a proof that does not include a marker ([3*bcd*]). We can apply Lemma 11 to conclude that, when using LR tactics, $\mathsf{bc\text{-}ask}_D$ 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 [12*bcd*] only if the proof returned by $\mathsf{bc\text{-}ask}_D$ ([3*bcd*]) contained a marker indicating that a remote request is necessary. From Lemma 12, 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 [9*bcd*] 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 [12*bcd*] in the LR scenario are the same as those of the IR scenario. Since the recursive call on [12*bcd*] 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 analogous to that of Theorem 5. $\square$