# Secure Context-sensitive Authorization

A Thesis

Submitted to the Faculty

in partial fulfillment of the requirements for the

degree of

Doctor of Philosophy

in

Computer Science

by

Kazuhiro Minami

DARTMOUTH COLLEGE

Hanover, New Hampshire

February, 2006

Examining Committee:

_____

(chair) David Kotz

_____

Sean Smith

_____

George Cybenko

_____

Mustaque Ahamad

_____

Charles Barlowe
Dean of Graduate Studies

# Abstract

Pervasive computing leads to an increased integration between the real world and the computational world, and many applications in pervasive computing adapt to the user's context, such as the location of the user and relevant devices, the presence of other people, light or sound conditions, or available network bandwidth, to meet a user's continuously changing requirements without taking explicit input from the users.

We consider a class of applications that wish to consider a user's context when deciding whether to authorize a user's access to important physical or information resources. Such a context-sensitive authorization scheme is necessary when a mobile user moves across multiple administrative domains where they are not registered in advance. Also, users interacting with their environment need a non-intrusive way to access resources, and clues about their context may be useful input into authorization policies for these resources. Existing systems for context-sensitive authorization take a logic-based approach, because a logical language makes it possible to define a context model where a contextual fact is expressed with a boolean predicate and to derive higher-level context information and authorization decisions from contextual facts.

However, those existing context-sensitive authorization systems have a central server that collects context information, and evaluates policies to make authorization decisions on behalf of a resource owner. A centralized solution assumes that all resource owners trust the server to make correct decisions, and all users trust the server not to disclose private context information. In many realistic applications of pervasive computing, however, the resources, users, and sources of context information are inherently distributed among many organizations that do not necessarily trust each other. Resource owners may not trust the integrity of context information produced by another domain, and context sensors may not trust others with the confidentiality of data they provide about users.

In this thesis, we present a secure distributed proof system for context-sensitive authorization. Our system enables multiple hosts to evaluate an authorization query in a peer-to-peer way, while preserving the confidentiality and integrity policies of mutually untrusted

principals running those hosts. We also develop a novel caching and revocation mechanism to support context-sensitive policies that refer to information in dozens of different administrative domains. Contributions of this thesis include the definition of fine-grained security policies that specify trust relations among principals in terms of information confidentiality and integrity, the design and implementation of a secure distributed proof system, a proof for the correctness of our algorithm, and a performance evaluation showing that the amortized performance of our system scales to dozens of servers in different domains.

# Acknowledgments

I want to thank my parents for their love and consistent encouragement.

I am extremely grateful to my adviser, David Kotz, for his clear guidance toward the completion of my degree. Without his support, I would never have finished a Ph.D degree. Through discussions and collaborations with him, I have learned a lot to be a better researcher. He was always willing to support me when I needed his help.

I also thank Sean Smith, George Cybenko, and Mustaque Ahamod for serving on my the thesis committee. Their feedback on the draft of my thesis helped me improve the thesis significantly. I am particularly thankful for Sean's detailed comments that helped me discuss my thesis in a broader context.

I also want to thank Bob Gray and Ron Peterson, for their incredible patience helping me in the D'Agent project in the early years of my Ph.D life. I was able to acquire many practical skills, such as Unix shell programming, through the interactions with them. Ron also gave me invaluable feedback on my initial ideas for my thesis.

Special thanks to my officemates who made my working environment lively: Guanling Chen and Anarb Paul. Guanling and I had been working on the same project, Solar, and shared the difficulties and success of the project together. Anarb was willing to listen to my research ideas and gave me valuable feed back.

I thank my fellow students and postdoctoral fellows that made my time at Dartmouth an enjoyable experience: Tristan Henderson, Ming Li, David Wagner, and Meiyuan Zhao.

Most of all, thanks to my wife Yoko for her loving support and patience for the past six years. I also thank my sons, Ryosuke and Keisuke, whose joy and excitement always energize me.

## Funding

This research program is a part of the Institute for Security Technology Studies, supported under Award number 2000-DT-CX-K001 from the U.S. Department of Homeland Security, Science and Technology Directorate. This work is also part of the Center for Mobile Computing at Dartmouth College, and has been supported by IBM, Cisco Systems, NSF grant EIA-98-02068, and DARPA Award number F30602-98-2-0107. Points of view in this document are those of the author and do not necessarily represent the official position of the U.S. Department of Homeland Security or its Science and Technology Directorate, or any of the other sponsors.

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

Pervasive computing, also called ubiquitous computing, leads to an increased integration between the real world and the computational world [117]. Our daily environment will be embedded with hundreds of thousands of devices, which have computation and communication capabilities, and people can concentrate on their task by interacting with those devices in such a smart environment, rather than working with general-purpose desktop computers. The vision of pervasive computing has been increasingly feasible as a crucial set of hardware components have been developed [116] over the last decade. There are many hand-held and wearable devices commercially available, and those can (or could) interact with each other through wireless networking technologies, such as IEEE 802.11b and Bluetooth. The advent of sensor networks [94] will lead to environments with thousands or millions of embedded sensors that can provide detailed context information.

Successful applications become *context-aware*, able to automatically adapt to the changing conditions in which they execute. Context-aware applications use context information, such as the location of the user and relevant devices, the presence of other people, light or sound conditions, or available network bandwidth, to meet a user's continuously changing requirements without taking explicit input from the users. Therefore, a system in pervasive

computing needs to define rules or policies to specify how it dynamically adapts itself to a changing user's context. For example, Schilit [104] applies if-then rules to change the behavior of their system to adapt to a user's changing context.

Besides defining triggering actions, many systems [28, 49, 95, 104] in pervasive computing apply a logic-based language to derive high-level context information from raw sensor data. Many context-aware applications need high-level context information (e.g., a user's attending a meeting) to grasp the user's intent and provide services that meet his requirements appropriately. A logic-based language is suitable for this purpose, because we can use the same logic-based language to express both contextual facts that correspond to raw sensor data and rules that derive higher-level context information. For example, suppose that a location-tracking system keeps track of the location of Bob's PDA and that location is expressed as $location(pda12, sudikoff)$, which means that $pda12$ is in the sudikoff building. Suppose also that the fact $owner(Bob, pda12)$ is known to the system. Then, Bob's location could be derived using the rule $location(P, L) \leftarrow owner(P, D), location(D, L)$, which derives the location of a user $P$ from the location of device $D$ that belongs to user $P$.

One promising application of the logic-based approach is a context-sensitive authorization system [5, 32] that considers a requester's context as well as his identity to make a granting decision; the system derives the granting decision (true or false) with a set of rules encoding policies and facts encoding context information. A context-sensitive authorization scheme is necessary when mobile users move across multiple administrative domains in which a different authority defines security policies, because those users are not registered into systems in each domain in advance. Also, users interacting with their environments need a non-intrusive way to access resources, and clues about their context may be useful input into authorization policies for these resources.

For example, imagine a "smart meeting room" with many embedded devices. The

current whiteboard may be replaced with a networked virtual whiteboard. The meeting attendees may read or write to it using their PDAs or other small devices in this room via a wireless network. Since it is not practical to restrict the networked access to the board physically as we do today, we need a new mechanism to restrict access to such resources. The current access-control mechanism, based on a static access-control list (ACL), does not work for this purpose, because it is impossible to modify the ACL for each meeting. One reasonable policy, corresponding to one for a physical whiteboard, may be to allow people in the meeting room to access the board only during the time slot specified in the meeting calendar file. Thus, the authorization policy is location and time dependent, i.e., if the principal $P$ is in the room during the meeting time, $P$ is granted to access the virtual whiteboard. There are many other devices in the meeting room, such as the air conditioner, projector, and light-level controller, which may be managed with similar context-sensitive policies.

We can use context-sensitive policies to handle exceptional situations. Consider a health-care application, in which policy dictates that medical records may be accessed only by a patient's primary physician. A different doctor in the emergency room, however, needs to access the patient's medical record if the condition of that patient is critical. In this case, we may specify the context-sensitive authorization policy using the location of the patient, a list of nearby doctors defined by some notion of proximity, and current medical sensor data about the patient such as heart rate and blood pressure.

Finally, context-sensitive policies are useful for an emergency-response system [58, 100] that contains an information dissemination infrastructure for responders in a disastrous incident. The system collects information on disaster status and casualties in an incident from biomedical and environmental sensors and provides that information to the first responders and the command centers. The first responders are responsible for rescuing and evacuating casualties, and they obtain information on the situation of the incident and med-

ical conditions of the casualties in that incident with their wireless portable or wearable devices. However, access to the information on the incident must be limited on a need-to-know basis to prevent an attack by an adversary who exploits the information on the incident. Since the responders need to access information or resources managed by different state or local agencies, we adopt context-sensitive authorization policies that consider a responder's location, or severity of casualties near that responder, to grant access to critical information (such as the location of other responders or status of various equipment) or to a resource (such as a vehicle or other rescue equipment). In summary, context-sensitive authorization allows users to define a flexible policy by referring to environmental variables, attributes of objects and users, and both static and dynamic relationships between objects and between users.

## 1.1 Research challenges

When we apply context-sensitive policies to systems for pervasive computing, we must address the issue of information sharing among different organizations, because each sensing system that keeps track of a user's context information is owned by a different organization. Most organizations possess a certain physical boundary inside which they themselves can exclusively deploy sensing systems into their facilities. Therefore, a context-aware application that keeps track of a mobile user that crosses the organizational boundaries needs to obtain context information from multiple organizations to keep track of the user's context continuously. For example, imagine a large office building where there are sensors managed by the city, the building owner, the companies leasing space, and the individual employees. A location-aware application in that building might need to access multiple indoor location tracking systems in different organizations. Furthermore, there are many situations where sensing systems in the same area are managed by different organizations.

For example, the facility department of a company, which manages an office environment, controls the condition of light and temperature in the building, while the computer service department keeps track of the location of users that carry a wireless device.

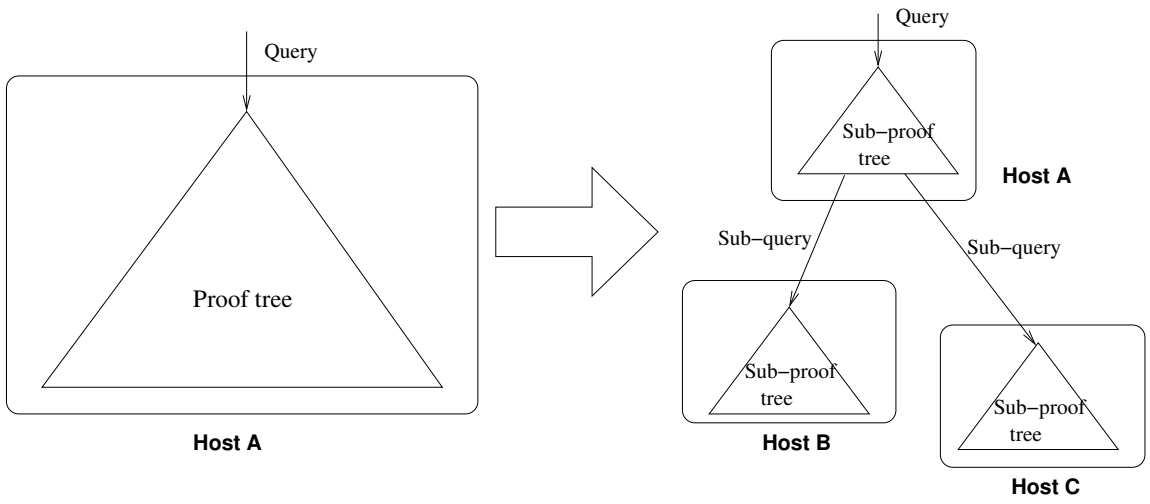
However, existing context-sensitive authorization systems [5, 8, 24, 32, 57, 88, 112] have a central server that collects context information, and evaluates policies to make authorization decisions on behalf of a resource owner. A centralized solution assumes that all resource owners trust the server to make correct decisions, and all users trust the server not to disclose private context information. Without such a universally trusted principal, it is impossible to build a system that maintains the global knowledge of all the context information. Cross-domain authentication [15, 25, 43] supports user authentication across multiple administrative domains. However, it is not address the issue of assigning privileges for accessing local resources to foreign users.

To achieve such information sharing among organizations, we must address two trust issues. First, each administrative domain (organization) defines confidentiality policies to protect information in that domain. It is necessary for an administrator of a location tracking system to protect users' location privacy [13, 88], for example. Another example is a server that maintains patients' medical records. The server must respect HIPAA rules about the confidentiality of medical records [1]. Therefore, a requester must satisfy the confidentiality policies of an information provider to access the requested information. Second, each administrative domain defines integrity policies that specify whether to trust information from other domains in terms of the integrity (correctness) of that information. Because context information is computed from raw sensor data, it inherently involves uncertainty. It is, therefore, important for each domain to choose reliable sources of information to derive correct context information.

## 1.2 Secure distributed proof system

In this thesis, we propose a secure distributed proof system that makes an authorization decision based on context-sensitive policies, addressing the issue of information sharing among multiple administrative domains. We take a logic-based approach for an authorization system [60, 124, 125]. When a client requests access to a resource, the resource owner constructs a logical proof that derives a granting decision. If such a proof is successfully constructed, access may be granted; otherwise access is denied.

To define the problem of information sharing precisely, we make a few assumptions. First, context information and rules that express authorization polices are distributed across servers in different administrative domains. There are other rules that derive higher-level context information. Second, an administrator of each domain defines confidentiality policies to protect context information and rules in that domain. Third, an administrator of each domain defines integrity policies that express the administrator's trust in the correctness of rules and facts in other domains. Although a method for obtaining accurate or trustworthy context information [51] is an important research agenda in pervasive computing, we assume that each administrator knows which information in other domains are trusted in terms of its integrity and is thus able to define integrity policies properly. When an administrator trusts the integrity of information in other domains, that means that the administrators agree on the semantics of the logical expression of that information. Therefore, our thesis does not address the issue of how to agree on the semantics of information among multiple parties. Furthermore, trust relationships in terms of confidentiality and integrity are defined by *principals*, each of which represents a specific user or organization, and we assume that each host is associated with one principal (e.g., the owner of a PDA, or the manager of a server). In practice, one principal may manage multiple devices, but for generality here we assume one principal per device.




(a) Centralized authorization server (b) Decentralized multiple authorization servers

Figure 1.1: Decentralized evaluation of an authorization query. The proof of a query is decomposed into sub-proofs and produced on distributed multiple hosts. On the left, Host A generates a whole proof on a centralized server. On the right, Host A, B, and C produce only a subtree of the proof.

Our major research goal is to build a distributed proof system that obtains an authorization decision by collaborating multiple mutually untrusted principals in a peer-to-peer way; that is, if a principal that handles an authorization query does not have all of the necessary information, the principal could send subsequent queries to other principals, and those principals handle those queries in the same way. The core of our approach is a proof decomposition in a distributed environment according to each principal's integrity policies; that is, rather than depending on a central trusted server (Figure 1.1a), we decompose a proof into sub-proofs produced by multiple hosts (Figure 1.1b). This collaboration is only possible if the querier can trust the integrity of other hosts (to provide correct facts and to properly evaluate rules) and if the other hosts can trust the querier with confidential facts.

In this thesis, we show that our secure distributed proof system meets three key goals by presenting our algorithm, the design of the system, and the results of our experiments that measure the performance of our system. These goals are:

**Confidentiality:** Information used for making an authorization decision is protected according to confidentiality policies defined by the owner of that information.

**Integrity:** Each principal receives a proof that satisfies his integrity policies from a principal that handles the query.

**Scalability:** Our target application could involves dozens of principals in different domains, and our system, therefore, should work with that many principals with reasonable performance.

## 1.3 Contributions

We summarize the contributions of this dissertation below.

- We introduce fine-grained security policies to formally define trust relations among principals in terms of information confidentiality and integrity.

- We show that it is possible to derive an authorization decision, even though authorization rules and context information referred to by those rules are distributed across multiple domains and are protected with different confidentiality policies. We develop a distributed algorithm that enables multiple principals to construct a proof in a peer-to-peer way, while preserving the integrity and confidentiality policies of those principals. We also prove the correctness of our algorithm.

- We design and implement a system that supports the distributed algorithm to study the performance of the system. We also develop a caching and revocation mechanism based on capabilities. Our revocation mechanism revokes all the cached information that depends on an initially revoked fact across the servers.

- We measure the performance of our system and show that the amortized performance of our system scales to a large proof that spans across dozens of servers. Our system is, therefore, suitable to a context-aware application in which a user's privileges must be continuously monitored. The results of our experiments with a wide range of parameters should serve as guidelines about the worst-case performance of a systems that adopts our techniques.

## 1.4 Dissertation outline

The rest of this thesis is organized as follows.

Chapter 2 introduces our authorization rule language and also defines integrity and confidentiality policies that formally define trust relations between principals.

Chapter 3 describes the design of a secure distributed proof system for a simple case in which confidentiality and integrity policies are defined only on facts. We introduce key ideas that enables secure distributed proving.

Chapter 4 extends the design of the system in Chapter 3 for a general case in which policies are defined on rules as well. We provide a proof for the correctness of the algorithm.

Chapter 5 describes the design of our caching and revocation mechanism based on capabilities.

Chapter 6 presents our experimental results that demonstrate the effectiveness of our caching mechanism.

Chapter 7 compares our system with related work in rule-based authorization, context-sensitive authorization, and distributed proof systems.

Chapter 8 discusses limitations of our system and suggestions for future work.

Finally, Chapter 9 concludes.

# Chapter 2

# Security policies

In this section, we describe our language for defining authorization policies that protect resources and introduce the concept of a proof tree, which is constructed when making an authorization decision. Next, we introduce *confidentiality policies* that protect both authorization policies and context information and *integrity policies* that specify trust in the correctness of those information. We also extend the notion of integrity to a proof tree.

## 2.1 Authorization rule language

In rule-based authorization systems, authorization policies are represented as logical expressions. We express authorization policies with Horn clauses since they are expressive enough to support policies in existing rule-based authorization systems [5, 8, 32]. We do not use a general first-order logic, which is not decidable in general. The syntax of a Horn clause is $b \leftarrow a_1 \wedge a_2 \ldots \wedge a_n$ , which says that simple statements called *atoms* $a_1$ through $a_n$, if all true, imply $b$. The atom $b$ is called the *head* of the clause, and the atoms $a_1, \ldots, a_n$ the *body* of the clause. An atom is formed from a predicate symbol followed by a parenthesized list of variables and constants. Our authorization policies do not support negation

of atoms because it is impossible to collect complete information in a distributed environment. Also, we do not support atoms that take general functions as parameters. When there is no atoms in the body of a Horn clause, the atom in the body of the clause represents a fact, which is true without any conditions. For example, we can express the contextual fact "Bob is in Hanover" as $location(Bob, Hanover)$. We assume that each principal stores its authorization policies and facts in a knowledge base. We express both contextual and static facts in the same way. The difference is that contextual facts in the knowledge base are updated dynamically.

**Example authorization rules.** The teams responding to a large-scale disaster are coordinated by experts drawn from multiple disciplines (fire, police, medical) and often multiple jurisdictions (city, state, federal). Increasingly, incident commanders use software to assist with incident management and situational awareness. The National Incident Management System (NIMS) [58] defines clear roles for the many participants in a large-scale response, so role-based access control (RBAC) [103] is a natural basis for protecting resources in an incident management system (IMS). Such an IMS needs to dynamically link people, resources, and information from multiple domains, providing information to those who need it in a time of crisis.

We give an example scenario[1] in which roles in such an incident management system is defined based on contextual situation of responders. Suppose that an incident occurs in an airport. There is a surveillance camera image server managed by the airport, and the chief of operations (*bob*) wishes to use the camera images to improve his awareness of the situation. Figure 2.1 shows a set of rules that define the airport's policy to grant access to the camera resource, which allows the local police chief access to the images whenever he is in the airport, as determined by either his Wi-Fi network connection or by the GPS

[1]Note that this is not an official scenario defined by NIMS.

tracking device in his radio. Rule 1 says that principal $P$ must hold the role *operation_chief* to be granted, and rule 2 defines the two conditions to hold that role. The first condition specifies the prerequisite role *police_chief* in a police department, and the second requires principal $P$ to be in the airport. Rules 3–5 specify how we derive the location of principal $P$ from the raw location information of a device.

## 2.2 Proof tree

To make an authorization decision, we must check whether a proof tree for query $?grant(P)$ can be constructed with a given set of rules and facts. The proof tree consists of nodes that represent rules (or facts) and edges that represent the unification that replaces an atom in the body of a rule in a parent node with the atoms in the body of a rule or a fact in a child node. Every leaf node contains a fact that has no atom in its body.

Given the facts listed in Figure 2.1, we can construct the proof tree shown in Figure 2.2 by unifying the query with the first four rules, substituting variables as needed. We return to this example in Sections 3.6 and 4.7 to explain how we construct this proof in a distributed fashion.

## 2.3 Rule patterns

We first introduce the notion of *rule patterns*, which are mechanisms for expressing confidentiality and integrity policies on rules and facts in a knowledge base. A rule pattern is just a regular Horn clause to be unified with a rule or a fact in the knowledge base. We use a rule pattern to specify to which rules and facts a given policy is applied, because it is infeasible to specify a policy on each instance of a rule or a fact that contains variables. A rule pattern is associated with a set of rules or facts that match it through *unification*, a pattern-matching

Rules:

$$grant(P) \leftarrow role(P, \textsf{operation\_chief}) \quad (2.1)$$

$$role(P, \textsf{operation\_chief}) \leftarrow roleIn(P, \textsf{police\_chief}, \textsf{police\_dept}) \wedge location(P, airport) \quad (2.2)$$

$$location(P, L) \leftarrow owner(P, D) \wedge location(D, L) \quad (2.3)$$

$$location(D, L) \leftarrow wifi(D, A) \wedge in(A, L) \quad (2.4)$$

$$location(D, L) \leftarrow gps(D, X, Y) \wedge closeTo(X, Y, L) \quad (2.5)$$

Facts:

$roleIn(bob, \textsf{police\_chief}, \textsf{police\_dept}).$ Bob is chief of the local police department. (2.6)

$owner(bob, pda15).$ Bob owns device pda15. (2.7)

$wifi(pda15, ap39).$ pda15 is associated with access point ap39. (2.8)

$in(ap39, airport).$ Access point ap39 is at the airport. (2.9)

Figure 2.1: Sample set of rules. We use uppercase for variables and lowercase for constants and names.

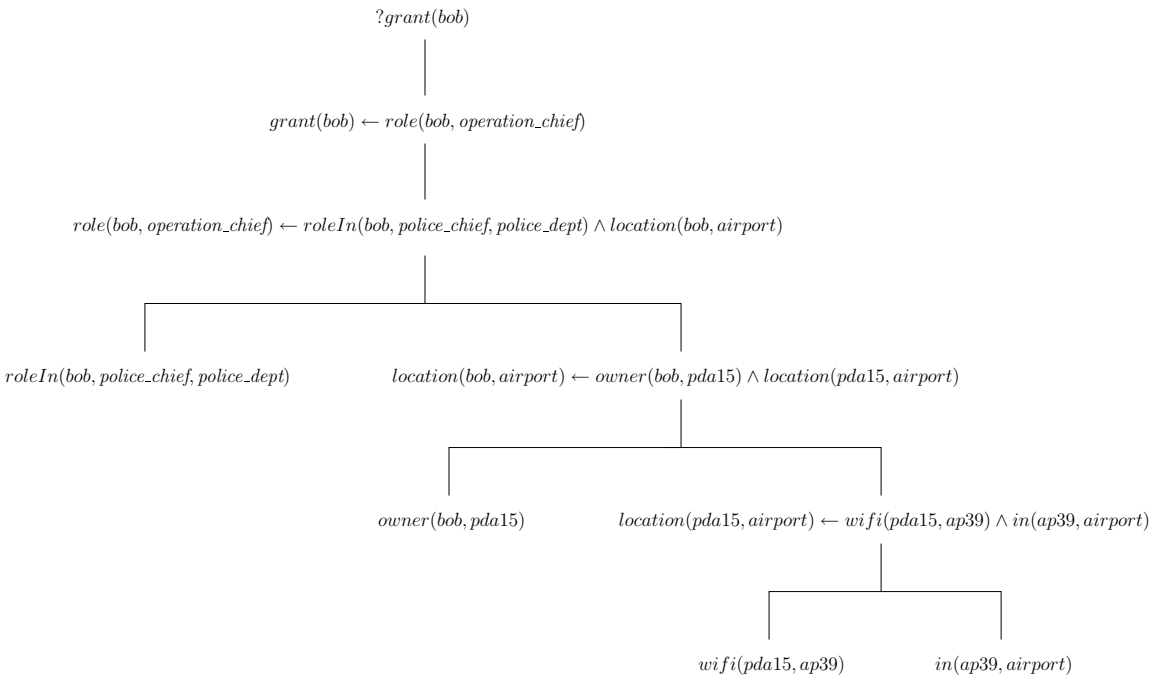



Figure 2.2: Example proof tree based on the rules in Figure 2.1.

process that makes a rule pattern and an actual rule in the knowledge base identical by instantiating variables in the rule pattern. For example, the rule pattern $location(bob, X)$ is matched with the fact $location(bob, hanover)$ in the knowledge base, because the variable $X$ can be instantiated to $hanover$. It does not match with the fact $location(alice, hanover)$, however. The rule pattern $role(X, Y) \leftarrow occupation(X, Y) \wedge location(X, hospital)$ can be matched with the rule below by instantiating $X$ to $P$ and $Y$ to $physician$.

$$role(P, physician) \leftarrow occupation(P, physician) \wedge location(P, hospital)$$

A principal may define as many security policies as it sees fit to define. Each security policy $(rp, t)$ is represented as a rule pattern $rp$ and a set of trusted principals $t$. There are two kinds of security policies: integrity policies and confidentiality policies.

## 2.4 Integrity policies

Integrity policies express trust in the correctness of rules and facts. Our definition is based on information flow theory [14, 134], which focuses on confidence in the accuracy of information rather than modification of information. When a principal $p_i$ defines the integrity policy $(rp, t)$ it means that $p_i$ trusts those principals in $t$, a set that we denote $trust_i(rp)$, to be correct in whatever rules or facts match pattern $rp$. We use subscript $i$ in the trust policy to denote which principal defines the policy.

The integrity of a fact means that the boolean value representing a fact is correct. For example, if principal $p_0$ includes principal $p_1$ in its $trust_0(loc(P, X))$, then principal $p_0$ believes that $p_1$'s evaluation (true or false) of a location query of the form $?loc(P, X)$ (e.g., $?loc(bob, hanover)$) is correct. On the other hand, the integrity of a rule means that the rule itself is able to correctly derive a new fact. For example, if principal $p_0$ includes principal

$p_1$ in its rule pattern $trust_0(loc(P, X) \leftarrow WiFi(P, Y) \wedge in(Y, X))$, then $p_0$ believes that $p_1$'s rule $loc(bob, X) \leftarrow WiFi(bob, Y) \wedge in(Y, X)$ is a correct rule to resolve the query of the form $?loc(bob, hanover)$. In other words, principal $p_0$ believes that it is valid to replace the query $loc(bob, hanover)$ with two sub-queries $?WiFi(bob, Y)$ and $?in(Y, hanover)$. Principal $p_0$ can verify that principal $p_1$ applied the rule correctly to derive the conclusion by checking the integrity of a proof tree defined in Section 2.4.1.

Notice that trust on a fact is a stronger notion than trust on a rule. Trust on a fact implicitly trusts the rules used to derive that fact. For example, the trust on the rule pattern $loc(X, Y)$ implicitly indicates trust of any rule whose head can be unified with $loc(X, Y)$.

### 2.4.1 Integrity of a proof tree

A principal trusts the integrity of a proof tree (that is, believes its result) for a query if it is consistent with its integrity policies. We formally define the integrity of a proof tree from the viewpoint of an initial querier principal $p_0$ inductively as follows. Suppose that principal $p_0$ issues a query $q$ to principal $p_1$.

**Base case (single-node tree):** If the proof from principal $p_1$ contains a query $q$'s result, and principal $p_0$ has an integrity policy $(rp, t)$ such that rule pattern $rp$ matches query $q$ and $p_1$ belongs to the set of principals $t$, then $p_0$ trusts the results of the proof tree.

**Induction step:** If the proof from $p_1$ contains a proof tree whose root node represents a rule $r$, the head of rule $r$ matches query $q$, $p_0$ has an integrity policy $(rp, t)$ such that rule pattern $rp$ matches $r$ and $p_1$ belongs to the set of principals $t$, and $p_0$ trusts the integrity of the subproof trees under the root node representing $r$, then $p_0$ trusts the proof tree.

## 2.5 Confidentiality policies

Confidentiality policies protect facts and rules in a principal's knowledge base. A fact must be protected if it contains confidential information. A rule must be protected if confidential information may be inferred from reading the rule. For example, the rule $grant(P) \leftarrow loc(bob, sudikoff)$ says that any principal $P$ is granted access when $bob$ is at the location of *sudikoff* building. If a request is granted, the requester may infer that bob is at Sudikoff, which might not be public knowledge.

When a principal $p_i$ defines the confidentiality policy $(rp, t)$, it means that $p_i$ trusts those principals in $t$, which we often refer to as the access control list $acl_i(rp)$, with facts or rules matching rule pattern $rp$. Principal $p_0$ only responds to a query $q$ from principal $p_1$ if there exists a rule pattern $rp$ that can be unified with the query $q$ and principal $p_1$ belongs to $acl_0(rp)$. For example, suppose that principal $p_0$ defines the policy $acl_0(location(bob, L)) = \{p_1, p_2\}$; principal $p_0$ responds to a query $?location(bob, hanover)$ from principal $p_1$, because rule pattern $location(bob, L)$ matches with $location(bob, hanover)$.

## 2.6 Assumptions

In this thesis, we make a few assumptions to maintain our focus on the confidentiality and integrity issues in a distributed context-sensitive authorization system. First, integrity policies of every principal are public knowledge. Second, each principal can gain knowledge about whether it has a privilege of accessing information on a given fact from other principals. It is possible for a principal to infer that knowledge by issuing a query about that fact to those principals. If the requesting principal receives a query result, that principal possesses the privilege. If the request is denied, that means that the principal does not possess the privilege. However, the denial of a query request does not reveal any information on

whether the principal that receives the query actually has knowledge about the fact in the query or not. Third, we assume that each principal's confidentiality policies on rules are not public knowledge because a confidentiality policy on a rule discloses the fact that the principal possesses a rule that matches the rule pattern of that confidentiality policy. Third, a public-key infrastructure is available and every principal can obtain the public key of other participants, so that they can establish secure channels with a session key and verify the authenticity of messages with digital signatures.

For purposes of simplifying our explanation, we consider the basic case that supports security policies only on facts first in Chapter 3, and then the general case that supports security policies on facts and rules in Chapter 4.

# Chapter 3

# Secure distributed proof system for the basic case

In this chapter, we describe our secure distributed proof system for the basic case that supports security policies only on facts. We first describe the architecture of the system and introduce the notion of proof decomposition in a distributed environment. We then cover a mechanism for enforcing confidentiality policies and give a distributed algorithm for evaluating a logical query in a peer-to-peer way.

## 3.1 Architecture

With no central server to make authorization decisions, we use multiple hosts that are administered by different principals. Without loss of generality, we assume that each host $i$ is administered by a different principal $p_i$, although in many realistic environments there may be principals that own or manage many hosts. Each host stores a local copy of its principal's integrity and confidentiality policies. Each host provides an interface for handling queries from remote hosts, and may ask other hosts to resolve any subqueries necessary. In

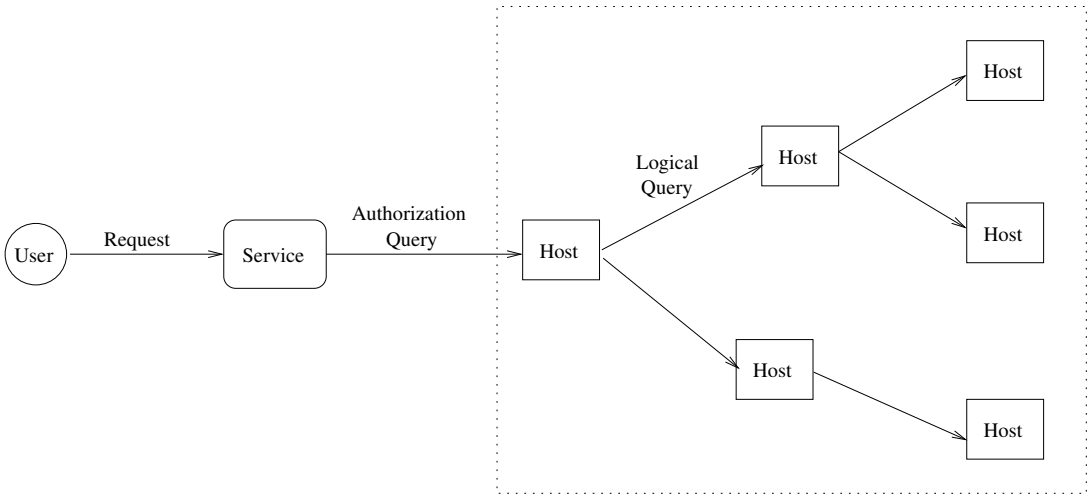



Figure 3.1: The hosts enclosed in the dotted lines make an authorization decision in a collaborative way.

Figure 3.1, a user sends a request to the server that provides some service, and the server issues an authorization query to a host it chooses in order to make a granting decision. If the host that receives the query does not have all the rules and facts for making the decision, it issues subsequent queries to other hosts, and this process could be iterated recursively. There is no single host that manages other hosts, and multiple hosts instead handle a query in a peer-to-peer way.

All the hosts have an identical structure, which is shown in Figure 3.2. The query handler handles queries from other hosts and enforces the local confidentiality policies. The inference engine constructs a proof tree for a given query based on the rules and facts in the local knowledge base. If some query cannot be evaluated locally, the inference engine issues a remote query to another host through the query issuer. The inference engine is created per query by the query handler; multiple queries are handled concurrently by multiple threads of the inference engine objects. The query issuer refers to its local integrity policies to choose a principal whose evaluation of the query is trusted; the integrity policies serve as a directory service to choose a principal to which it sends a query. The query issuer receives a reply and checks its integrity based on the integrity policies. The event handler

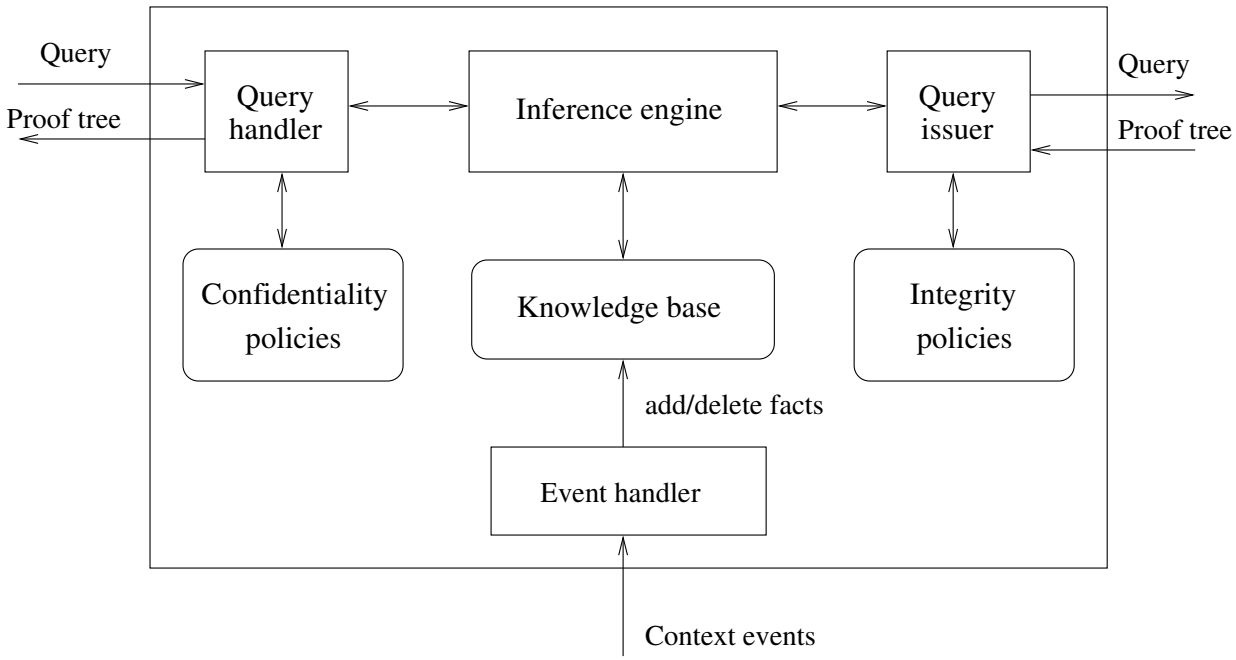



Figure 3.2: Structure of a host.

converts events that contain new context information into corresponding facts and updates the knowledge base; these events may be delivered by a context-dissemination service such as Solar [27].

## 3.2 Proof object

The response to a query is a *proof* object represented as $(p_r, n, (value)_{K_r})$, where $p_r$ is a receiver principal. Figure 3.3 shows the grammar of a proof object. The proof object contains a nonce $n$ that is attached with the query to prevent replay attacks by an adversary that is capable of intercepting the encrypted messages between principals. In the following discussion, we omit the nonce field in the proof object for brevity.

The $value$ is a query result, which is a boolean value (*TRUE* or *FALSE*), a conjunction of boolean values, or the value *REJECT*. The value *REJECT* is used when a given query is not handled because the querier principal does not satisfy the handler principal's

$$
\begin{aligned}
\underbrace{< \textit{proof} >} \quad &::= \quad \text{`('} < \textit{receiver} > \text{`,'} < \textit{nonce} > \text{`,'} < \textit{proofs} > \text{`)'} \\
< \textit{proofs} > \quad &::= \quad \text{`('} < \textit{value\_pair} > (\wedge \ < \textit{value\_pair} >) * \text{`)'} \mid \text{`('} < \textbf{value} > \text{`)'} \\
< \textit{value\_pair} > \quad &::= \quad \text{`('} < \textit{receiver} > \text{`,'} < \textbf{value} > \text{`)'} \\
< \textit{receiver} > \quad &::= \quad < \textit{string} > \\
< \textit{query} > \quad &::= \quad < \textit{atom} > \\
< \textit{atom} > \quad &::= \quad < \textit{predicate} > \text{`('} < \textit{args} > \text{`)'} \\
< \textit{predicate} > \quad &::= \quad < \textit{string} > \\
< \textit{args} > \quad &::= \quad < \textit{arg} > (\text{`,'} < \textit{arg} >) * \\
< \textit{arg} > \quad &::= \quad < \textit{string} > \\
< \textbf{value} > \quad &::= \quad \text{`}TRUE\text{'} \mid \text{`}FALSE\text{'} \mid \text{`}REJECT\text{'} \\
< \textit{nonce} > \quad &::= \quad < \textit{number} > \\
< \textit{number} > \quad &::= \quad < \textit{number} >< \textit{digit} > \mid < \textit{digit} > \\
< \textit{digit} > \quad &::= \quad 0|1|2|3|4|5|6|7|8|9
\end{aligned}
$$

Figure 3.3: Representation of a proof for the base case. The item *value*, shown with bold fonts, is encrypted with a public key of a principal that receives the proof. The item *proof*, shown with an underbrace, is digitally signed with a private key of a principal that constructs a proof.

confidentiality policies. Otherwise, the handler principal constructs a proof tree locally, then includes the query's result (*TRUE* or *FALSE*) in the proof object. (We name the returned object a *proof object* because, in the general case in Section 4, it contains a proof tree that shows how the query result is derived.) The receiver principal $p_r$ might not be the principal that issues query $q$ (we explain why, below), and, therefore, the name of the receiver principal needs to be included in the proof object, so that the receiver principal can decrypt an encrypted value. As indicated with the notation $(value)_{K_r}$, the value must be encrypted with receiver principal $p_r$'s public key $K_r$ to enforce the confidentiality policies of the publisher principal. The public key encryption is performed to prevent intermediate principals from reading the value. Furthermore, the whole proof object is transmitted via a secure channel established with a session key between a querier and a handler principal to prevent an eavesdropper from reading the content of the proof object. We assume that the two principals share the symmetric key via a protocol using public-key operations when the querier and handler principal authenticate with each other for the first time. The digital signature of a whole proof signed by a handler principal ensures *non-repudiability* of the handler; that is, the handler principal is not able to falsely deny later that it sent the proof.

A principal $p_0$ that handles query $q_0$ might issue subqueries to other principals, and the returned proofs from those principals might contain encrypted query results that principal $p_0$ cannot decrypt. Therefore, the query $q_0$'s result depends on the encrypted values in the proofs for the subqueries that $p_0$ issues, and principal $p_0$ returns a proof for query $q_0$ that contains the query results for the subqueries as follows. Suppose that principal $p_0$ issues subqueries $q_i$ for $i = 0, \ldots, n-1$, and receives several $pf_i = (p_{r(i)}, (value_i)_{K_{r(i)}})$ where $p_{r(i)}$ is the receiver principal of the proof, $value_i$ is the query $q_i$'s result, and $K_{r(i)}$ is principal $p_{r(i)}$'s public key. The query $q_0$'s result is *TRUE* only if $p_0$ can verify that $value_i$ is *TRUE* for all $i$ in the proof. If any $pf_i$ that principal $p_0$ can decrypt contains a *FALSE* value, $p_0$ returns a simple proof $(p_r, (FALSE)_{K_r})$, where $p_r$ is a receiver principal chosen

by $p_0$. Otherwise, if there are some subproofs that $p_0$ cannot decrypt (because $r(i) \neq 0$), then principal $p_0$ returns the proof $(p_r, (\wedge_i(p_{r(i)}, (value_i)_{K_{r(i)}}))_{K_r})$ for all $r(i) \neq 0$, as a response to query $q_0$. We do not need to include the digital signatures of the embedded subproofs in the proof, because a digital signature is only used to ensure *non-repudiability* of the principal that sends the whole proof. In Section 4.1, a digital signature is also used to check the authenticity of a subproof in a whole proof. The proof contains the concatenated subproofs encrypted with public key $K_r$. The query result of the proof is *TRUE* if the conjunction of all the $value_i$ (i.e., $\wedge_i(value_i)$) is *TRUE*.

## 3.3 Decomposition of a proof tree

When a querier issues a query to a principal that satisfies the querier's integrity policies for that query, the principal that handles the query only returns a proof that contains the query's result (*TRUE*, *FALSE*, or *REJECT*), and the proof tree that derives the query's result does not have to be disclosed to the querier. If multiple principals are involved in processing a query, no single principal obtains all the rules and facts in the proof tree of the original query. Instead, the proof tree for the query is decomposed into multiple *subtrees* evaluated by different principals in a distributed environment. In other words, there is no single principal that maintains a whole proof; instead, each principal maintains a subproof of the whole proof.

Figure 3.4 describes such collaboration between a querier and a handler hosts. Suppose that host $A$ run by principal Alice, who owns a projector, receives an authorization query $?grant(Dave, projector)$ that asks whether Dave is granted access to that projector. Since Alice's authorization policy in her knowledge base refers to a requester's location (i.e., $location(P, room112)$), Alice issues a query $?location(Dave, room112)$ to host $B$ run by Bob. Alice chooses Bob, because Bob satisfies Alice's integrity policies for queries

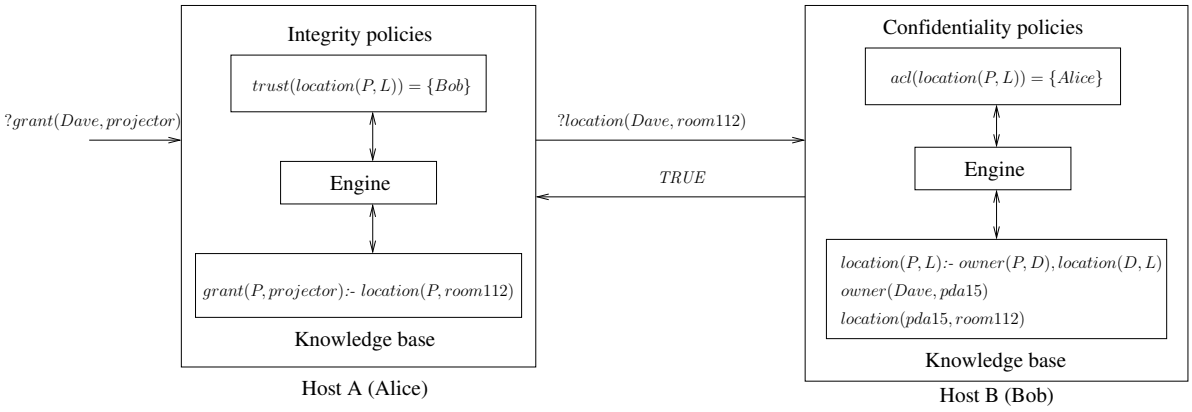



Figure 3.4: Remote query between two principals. Alice is a principal that owns a projector, and Bob is a principal who runs a location server. We only show a query result in Bob's reply, omitting other fields.

that matches a rule pattern $location(P, L)$ as defined in her integrity policies. Bob processes the query from Alice, because Alice similarly satisfies Bob's confidentiality policies for queries that matches a rule pattern $location(P, L)$. Bob derives that Dave is in *room112* from the location of his device using the facts $location(pda15, room112)$ and $owner(Bob, pda15)$. However, he only needs to return a query result $TRUE$ that states that $location(Dave, room112)$ is true, because Alice believes Bob's statement about people's location (i.e., $location(P, L)$) according to her integrity policies [1].

The proof of the query is thus decomposed into two subproofs maintained by Alice and Bob respectively. Alice's proof contains only a root node that states $location(Dave, room112)$ is true, and Dave's proof consists of a root node that contains the rule and two leaf nodes that contain $owner(Dave, pda15)$ and $location(pda15, room112)$. In general, Bob could return a proof tree that contains multiple nodes. If Alice only trusts Bob's rule that derives Bob's location instead of Bob's fact, he would need to submit a larger proof tree to satisfy Alice's integrity policies as we discuss in Chapter 4.

[1] In the real world, Dave might lose his device or leave it in his office momentarily, and, as a result, Bob incorrectly reports Dave's location to Alice. However, Alice is responsible for deciding whether she should believe location information from Bob, and how Alice finds trustworthy principals is out of the scope of our thesis.

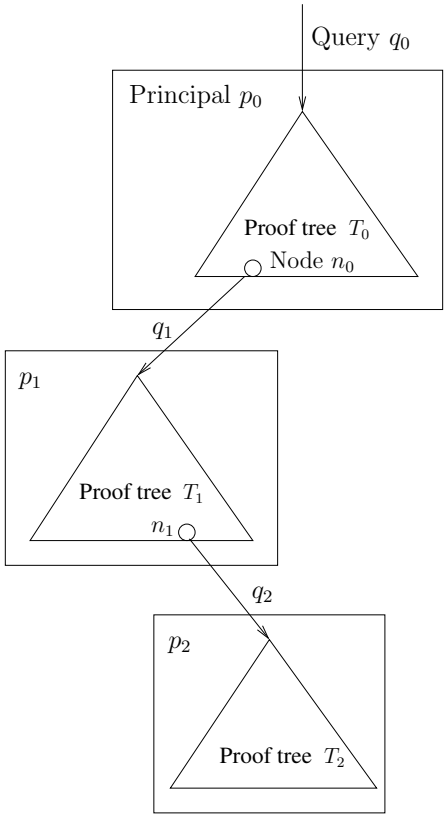



Figure 3.5: Decomposed proof tree. Principals $p_0$,$p_1$,and $p_2$ construct a proof tree for query $q_0$ in a distributed way. Nodes $n_0$ and $n_1$ are leaf nodes of proof trees $T_0$ and $T_1$ respectively. Principal $p_0$ that handles query $q_0$ issues query $q_1$ to principal $p_1$ to obtain the fact in node $n_0$, and principal $p_1$ similarly issues query $q_2$ to principal $p_2$.

In general, a proof could be decomposed into sub-proofs maintained by multiple principals. Figure 3.5 shows that the proof tree for query $q_0$ is constructed by principal $p_0$, $p_1$, and $p_2$ in a distributed way. Principal $p_0$ receives query $q_0$ and issues subquery $q_1$ to principal $p_1$ to construct a proof tree $T_0$, and principal $p_1$ similarly issues query $q_2$ to principal $p_2$ to construct a proof tree $T_1$. The facts or rules in the proof trees $T_0$, $T_1$, and $T_2$ are not disclosed to other principals; the result of evaluating each proof tree is returned to the querier as a boolean value or conjunction of encrypted boolean values.

**Example.** Figure 3.6 shows the proofs in the evaluation of the query $?grant(bob)$, involving $p_1$, $p_2$ and $p_3$. The query $?grant(bob)$ from principal $p_0$ to $p_1$ is decomposed into two

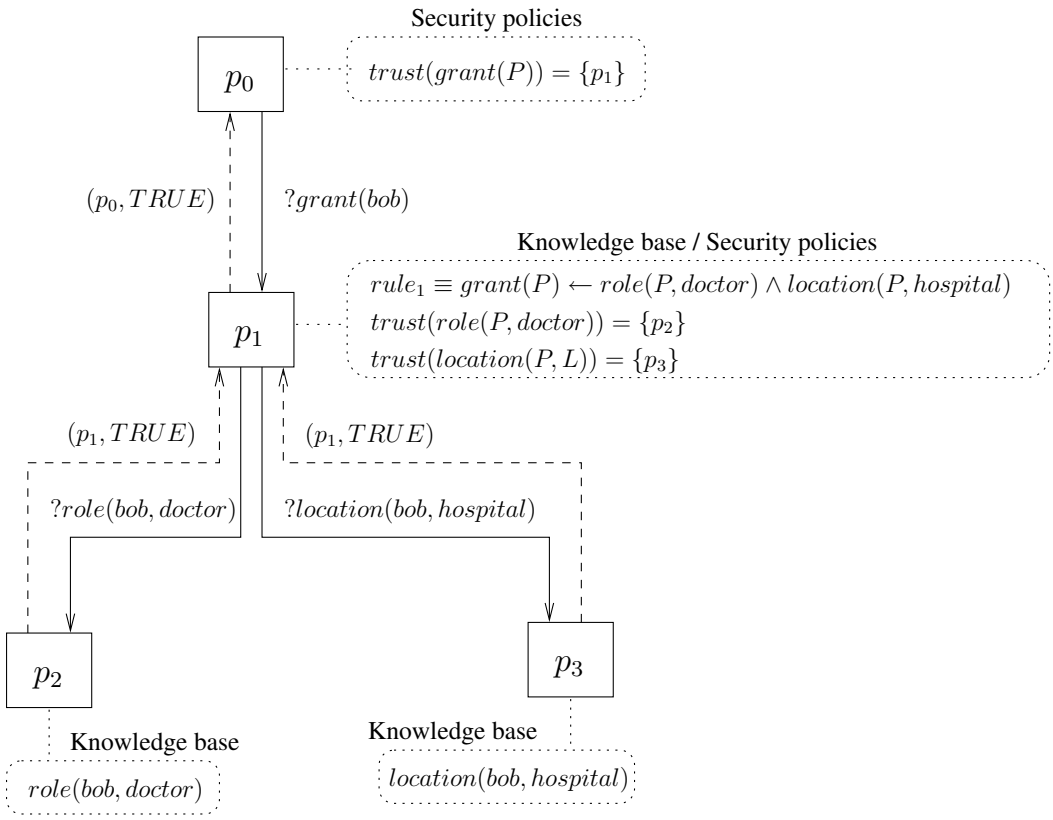



Figure 3.6: The solid arrows are labeled with queries and the dashed arrows are labeled with returned proofs. The rounded rectangles with dotted lines represent the knowledge bases and security policies of those principals respectively.

sub-queries $?role(bob, doctor)$ and $?location(bob, hospital)$ according to the rule $rule_1 \equiv grant(X) \leftarrow role(X, doctor) \wedge location(X, hospital)$, and those subqueries are handled by principal $p_2$ and $p_3$ respectively. Principal $p_2$ has the matching fact $role(bob, doctor)$ in its knowledge base and returns the proof $(p_1, TRUE)$ to principal $p_1$. Principal $p_3$ also returns the proof $(p_1, TRUE)$. Principal $p_1$ trusts the integrity of the proofs from $p_2$ and $p_3$ according to its integrity policies, and internally constructs the proof tree that contains the rule $rule_1$ as a root node and the facts $role(bob, doctor)$ and $location(bob, hospital)$ as its children nodes. Principal $p_1$ concludes that the statement $grant(bob)$ is *true* and returns the proof $(p_0, TRUE)$.

## 3.4 Enforcement of confidentiality policies

The enforcement of each principal's confidentiality policies is different from that in many existing authorization systems, which check the privileges of a requester principal before divulging information directly to the requester. In our system, a principal that publishes a proof chooses the receiver of the proof from a list of upstream principals in the whole proof tree; a query is appended with an ordered list of upstream principals that could receive the query result. Therefore, it is possible to obtain an answer for a query even when a querier principal does not satisfy the handler principal's confidentiality policies. The principal may make that choice because its confidentiality policy does not allow it to divulge the information to the querier, but may allow the information to be released to another principal further up the tree. The encrypted result will become part of the querier's response up the tree; eventually the receiver principal may decrypt the result and compute the conjunction to see whether the tree is *true*.

We formally define the ordered list of upstream principals as follows. We say that a principal *represents* a proof-tree node when a rule or a fact contained in that node is published by that principal. We denote the principal that represents node $n$ as $rep(n)$, and the ordered list of principals that represent a corresponding ordered list of nodes $s$ as $rep(s)$. Suppose that principal $p$ represents a node $n$ in a proof tree. We denote the ordered list of nodes on the path from the root of the proof tree to $n$, excluding $n$, as $upstream\_nodes(n)$. That is, the nodes are ordered from the root node downward.

The list of upstream principals for $p$ is defined as $rep(upstream\_nodes(n))$, which we denote as $receivers(p)$ for brevity, in the case of a specific tree. In Figure 3.7, principal $p_0$'s issuing query $q_0$ causes principals $p_1$ and $p_2$ to issue subqueries $q_1, q_2$ and $q_3$. Principal $p_3$'s list $receivers(p_3)$ is $< p_0, p_1, p_2 >$, for example.

When a publisher principal chooses a receiver from the list $receivers(p)$, the receiver

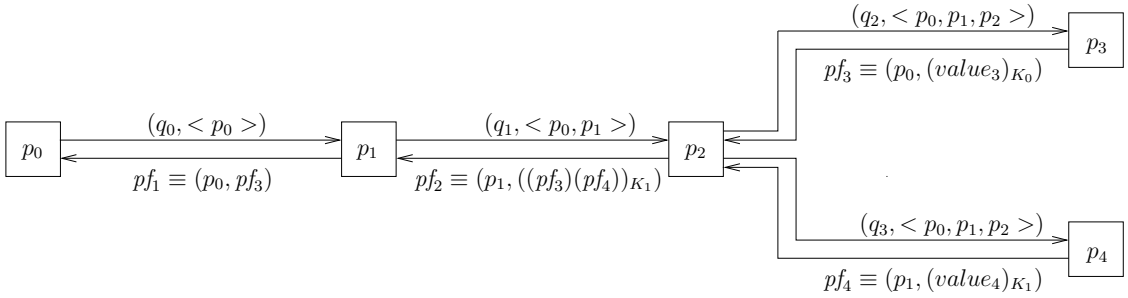



Figure 3.7: Principal $p_0$'s query $q_0$ is handled by principals $p_1, p_2, p_3$, and $p_4$ in a distributed way. Principal $p_i$ handles query $q_{i-1}$, and returns the proof $pf_i$, for $i = 1$ to 4. Each query is attached with an ordered list of upstream principals.

must satisfy the following two conditions. First, it must satisfy the publisher's confidentiality policies. For example, suppose that principal $p_4$ chooses $p_1$ as the receiver of query $q_3$'s result. Principal $p_1$ must satisfy $p_4$'s confidentiality policies for query $q_3$; that is, $p_4$ must have a confidentiality policy $(rp, t)$ where rule pattern $rp$ matches query $q_3$ and principal $p_1$ belongs to a set of principals $t$.

Second, the receiver principal must satisfy the constraints due to recursive encryption of a proof at each principal. A principal that handles a query might issue subqueries to other principals. If that principal cannot decrypt the query results in those subproofs, it includes the subproofs into its proof and encrypts them with the public key of a receiver principal. This recursive encryption is necessary to prevent an untrusted intermediate principal on the path towards the receiver from knowing the query result by decrypting some subproof whose query result is *FALSE*. Because such embedded encrypted subproofs are encrypted recursively by intermediate principals until they reach their receiving principals, the intermediate principals have to make sure that their encryption on embedded subproofs are decrypted when the proof reaches the receiving principals of the subproofs. Otherwise, the embedded subproofs pass the receiving principals without being decrypted, and the

proof fails.

In Figure 3.7, principal $p_3$ chooses $p_0$ as the receiver of proof $\mathit{pf}_3 \equiv (p_0, (value_3)_{K_0})$ where $value_3$ is query $q_2$'s result and $K_0$ is $p_0$'s public key, and $p_4$ chooses $p_1$ as the receiver of proof $\mathit{pf}_4$. Principal $p_2$ embeds those proofs from $p_3$ and $p_4$ into proof $pf_2$, because $p_2$ cannot decrypt those proofs but knows that the query result for query $q_1$ is the conjunction of the query results in those encrypted proofs. Suppose that both principal $p_0$ and $p_1$ in $receivers(p_2)$ satisfy the first condition; they satisfy $p_2$'s confidentiality policies for query $q_1$. Principal $p_2$ must choose $p_1$ as the receiver to satisfy the second condition. Notice that principal $p_2$ encrypts the proofs from principals $p_3$ and $p_4$ with principal $p_1$'s public key. This recursive encryption is necessary to prevent an attack by malicious upstream principals as we discuss in Section 3.4.

When principal $p_1$ decrypts and evaluates the proof $\mathit{pf}_4$, if the result is true then $p_1$ only embeds $pf_3$ into proof $pf_1$, which is decrypted by principal $p_0$; otherwise, $p_2$ drops the proof $pf_3$ and return a proof that contains a *FALSE* value. If principal $p_2$ chooses $p_0$ as the receiver of proof $pf_2$ instead, the proof $pf_4$, which is embedded in proof $pf_2$, is forwarded to $p_0$ without being decrypted by $p_1$ and the proof is not usable by $p_0$.

In general, a proof contains any number of encrypted subproofs. Suppose that principal $p_i$'s list $receivers(p_i)$ is $< p_0, \ldots, p_{i-1} >$, and $p_i$ returns proof $pf_i$ that contains subproofs $pf_j$ for $j = 0, \ldots, n-1$ to principal $p_k$. Let $p_{r(j)}$ be the receiver principal for proof $pf_j$, and $index(p, s)$ be the function that returns $p$'s index in the ordered list $s$. The second condition for selecting a receiver is stated as follows.

$$\forall j \ ((index(p_{r(j)}, receivers(p_i)) \leq index(p_k, receivers(p_i))) \vee (r(j) = i))$$

If there is more than one principal that satisfies the above two conditions, principal $p_k$ chooses the principal of the minimum index (closest to the root). This guideline is impor-

tant not to narrow the choices of the receivers made by the upstream principals. Note that the proof fails if the path to the root does not permit these decryptions and validations; the failure results because the integrity and confidentiality policies of the principals involved will not allow the necessary information sharing. We, therefore, anticipate that an addition of rules or policies by principals would increase the false negative rate of authorization decisions.

**Attack by colluding principals.** A principal must encrypt proofs from other principals with the public key of a receiver principal when the proof to the receiver principal contains those proofs, as we see in the example in Figure 3.7. This recursive encryption is necessary to prevent an attack by colluding upstream principals that modify a receivers list attached with a query.

We describe such an attack with an example in Figure 3.8. Suppose that principals $p_1$ and $p_2$ are malicious colluding principals. When $p_1$ issues a query $q_1$, $p_1$ produces the incorrect receivers list $< p_1, p_0 >$ instead of the correct list $< p_0, p_1 >$. Another malicious principal $p_2$ issues a query $q_2$ with the receivers list $< p_1, p_0, p_2 >$; principal $p_2$ just appends itself at the end of the receivers list obtained from $p_1$. Principal $p_1$ needs $p_2$'s cooperation, because if $p_2$ is not a colluding principal, $p_2$ detects $p_1$'s modification because $p_1$ is not at the end of the receivers list. When principal $p_2$ sends a query to $p_3$, $p_3$ cannot detect that the receivers list is incorrect because principal $p_2$ operated legitimately with the exception of overlooking $p_1$'s incorrect receivers list.

Suppose that principal $p_3$ chooses $p_0$ as the receiver of $p_3$'s proof, which contains proofs obtained from principals $p_4$ and $p_5$ and that principals $p_4$ and $p_5$ choose principals $p_0$ and $p_1$ as the receivers of their proofs respectively. If principal $p_3$ does not encrypt the proofs from $p_4$ and $p_5$ with $p_0$'s public key, principal $p_1$, who receives those proofs before $p_0$ does, decrypts the proof from principal $p_5$. Principal $p_1$ knows the query result for $q_2$ without a

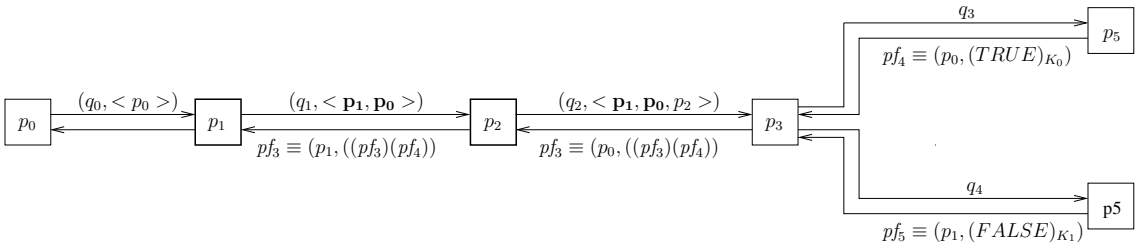



Figure 3.8: Principals $p_1$ and $p_2$ are malicious colluding principals. Principal $p_1$ produces the incorrect receivers list with a query $q_1$. The elements ordered incorrectly in the receivers list are shown with bold fonts. For brevity, we omit the receivers list for queries $q_3$ and $q_4$.

legitimate privilege if $p_5$'s proof contains a $FALSE$ value, because the result of $q_2$ is the conjunction of the values contained in the proofs from $p_4$ and $p_5$. If $p_5$'s proof contains a $TRUE$ value, $p_1$ is still not sure about the query result, though.

## 3.5 Algorithms for the base case

Each host (run by some principal) provides an interface HANDLEREMOTEQUERY for handling a query from a remote host. It takes as parameters a query string $q$, a list of upstream principals *receivers* defined in Section 3.4, and a querier principal's integrity policies $i\_policies$, as shown in Figure 3.9. The function HANDLEREMOTEQUERY calls the function GENERATEPROOF to obtain a proof.

Figure 3.10 shows the algorithm for the function GENERATEPROOF, run on principal $p_1$'s host to build a proof while enforcing confidentiality policies of the handler principal $p_1$. The function takes several parameters: principal $p_0$ that issues a query, principal $p_1$ that handles a query, a query string $q$, a nonce $n$, a list of upstream principals $receivers$ for $p_1$ (i.e., $receivers(p_1)$), $p_0$'s integrity policies $i\_policies_0$, $p_1$'s integrity policies $i\_policies_1$, $p_1$'s confidentiality policies $c\_policies_1$, and $p_1$'s knowledge base $KB_1$. The querier prin-

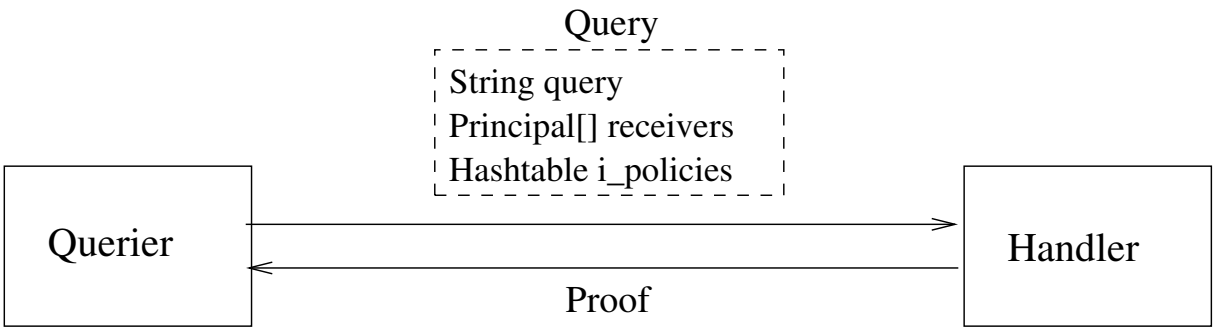



Figure 3.9: Query interface.

cipal $p_0$'s integrity policies are provided to the function as a parameter for principal $p_1$ to avoid wasting effort by constructing a proof that does not satisfy the querier $p_0$'s integrity policies; a principal $p_0$ later uses its integrity policies $i\_policies_0$ to check the integrity of a proof returned from $p_1$.

If $p_0$ is an initial querier, it includes itself into the list $receivers$.

Lines 2–3 check whether there is any principal in the list $receivers$ that satisfies the handler principal $p_1$'s confidentiality policies. The principals that belong to the intersection of $receivers$ and the union of the access-control lists in $p_1$'s confidentiality policies for query $q$ are eligible to receive a proof from $p_1$. We treat the ordered list $receivers$ as a set in line 2, and denote the result set as $s$. If there is no such principal (i.e., the set $s$ is empty), line 4 returns a proof with a *REJECT* value to querier principal $p_0$.

Line 5 sets the receiver principal of a proof in the case that the query result in the proof is obtained locally. The chosen receiver is the principal that belongs to list $s$ and has the minimum index in the ordered list $receivers$. We choose that principal with the function $minIndex(s, receivers)$ in line 5. Let $index(p, receivers)$ be the function that returns the index of a principal $p$ in the list $receivers$. We define $minIndex(s, receivers)$ as the function that returns a principal $p$ that satisfies the following conditions.

$$p \in s \wedge \forall p' \mid ((p' \in s) \wedge (index(p, receivers) \leq index(p', receivers)))$$

We use the symbol '|' to denote "such that" in our algorithm for brevity. Line 7 checks whether the handler principal $p_1$ satisfies the querier $p_0$'s integrity policies. If not, line 8 returns a proof with a *FALSE* value to principal $p_r$. Line 9 checks whether query $q$ matches fact $f$ in $p_1$'s knowledge base. If so, line 10 returns a proof with a *TRUE* value to principal $p_r$.

Lines 11–19 cover the case that query $q$ matches the head of rule $r$ in $p_1$'s knowledge base. Line 12 unifies query $q$ and rule $r \equiv A \leftarrow B_1, \ldots, B_k$, resulting in the instantiated rule $A' \leftarrow B'_1, \ldots, B'_k$. Lines 13–14 obtain subproofs for the subqueries $B'_1, \ldots, B'_k$ iteratively. Line 15 sets the conjunction of all the subproofs into a variable *pfs*, and line 16 decrypts subproofs that can be decrypted by principal $p_1$ by calling the function DECRYPTPROOFS in Figure 3.11. The function DECRYPTPROOFS constructs a proof that contains subproofs that are not decrypted. If a proof is not constructed, it returns a *NULL* value.

When lines 7–18 fail to construct a proof that derives query $q$, our algorithm does not return a proof that contains *FALSE* immediately. Instead, it tries to obtain a proof from a remote principal in lines 21–32. Line 21 creates an empty set $accessed$ that will contain a set of remote principals. Line 22 appends $p_1$ into the $receivers$ list and sets it into another ordered list $rcvrs$, using the function $concat$ that appends a principal in the second parameter into the list in the first parameter. The ordered list $rcvrs$ is used as one of the parameters for a remote query below. Lines 22–31 obtain a proof for query $q$ from a remote principal. Line 22 checks whether there is any principal $p_l$ that satisfies $p_1$'s integrity policies for query $q$ and does not belong to the set $accessed$. The set $accessed$ maintains a list of principals to which the remote query is already issued. Line 23 calls the function ISSUEREMOTEQUERY with the remote principal $p_l$, the query $q$, the nonce *n*, the ordered list $rcvrs$, and the integrity policies $i\_policies_1$ as parameters, and that function returns a proof from principal $p_l$. Line 24 checks whether the proof contains the same nonce and query given as parameters of the remote query, and line 25 checks whether principal $p_1$

is the receiver of that proof. Lines 26–28 handle the case that a set of proofs*pfs* actually contains just a value *TRUE* or *FALSE*. If that holds true, line 26 checks whether the proof contains a *TRUE* value. If so, line 27 returns a proof that contains a *TRUE* value to principal $p_r$; otherwise, it returns a proof that contains a *FALSE* value similarly. Line 29 decrypts the proof by calling the function DECRYPTPROOFS. If the returned proof $pf'$ is not $NULL$, line 31 returns that proof; otherwise, line 32 puts principal $p_l$ into the set $accessed$. This process is iterated until the while loop in lines 22–32 obtains a proof or finishes trying all the remote principals. Finally, line 33 returns a proof with a *FALSE* value, since there is no other way to construct a proof for query $q$.

**Algorithm for decrypting a set of proofs.** Figure 3.11 shows the algorithm for decrypting a set of proofs obtained from remote principals. The function DECRYPTPROOFS takes as parameters a principal $p_c$ whose private key is used to decrypt the proofs, a set of proofs $pfs$ obtained from remote principals, a default receiver principal chosen by principal $p_c$, and a set of principals $s$ to whom principal $p_c$ could send a proof, and the ordered list $receivers$.

Line 1 checks whether the format of the proofs $pfs$ is valid, and line 2 checks whether all the subproofs that can be decrypted by principal $p_c$ contains a *TRUE* value. If that holds true, line 3 returns a proof with a *TRUE* value to principal $p_r$. Line 4 checks whether all the subproofs decrypted by $p_c$ contain a *TRUE* value, and if so, line 5 checks whether there is some principal $p_{r'}$ that satisfies the constraint due to the recursive encryption we describe in Section 3.4; that is, $p_{r'}$'s index in the ordered list $receivers$ must be greater than or equal to the index of $p_{r(i)}$ in $receivers$ if $r(i) \neq 1$. If there is such a principal $p_{r'}$, line 6 returns a proof containing the subproofs whose values could not be decrypted by $p_1$ with principal $p_{r'}$ as the recipient. If all the above attempts to return a proof, line 7 returns a *NULL* value, which means that the function fails to return a proof.

We extend the algorithms in this section to support the general case in Section 4.5 and

GENERATEPROOF$(p_0, p_1, q, n, receivers, i_policies_0, i_policies_1, c_policies_1, KB_1)$

1 $\triangleright$ Check whether there is any principal in $receivers$ that satisfies $p_1$'s confidentiality policies.
2 $s \leftarrow receivers \cap (\bigcup_i t_i)$ for all policies $(rp_i, t_i) \in c\_policies_1$ where $rp_i$ matches $q$
3 **if** $s = \emptyset$ $\triangleright$ if set $s$ is empty.
4 **then return** $(p_0, n, (REJECT)_{K_0})$
5 $p_r \leftarrow minIndex(s, receivers)$
6 $\triangleright$ Check whether principal $p_1$ satisfies querier $p_0$'s integrity policies.
7 **if** $\neg(\exists$ policy $p = (rp, t) \mid ((p \in i\_policies_0) \wedge (rp$ matches $q) \wedge (p_1 \in t)))$
8 **then return** $(p_r, n, (FALSE)_{K_r})$
9 **if** $\exists$ fact $f \mid ((f \in KB_1) \wedge (f$ matches $q))$
10 **then return** $(p_r, n, (TRUE)_{K_r})$
11 **elseif** $\exists$ rule $r \equiv A \leftarrow B_1, \ldots, B_k \mid ((r \in KB_1) \wedge (A$ matches $q))$
12 **then** unify $q$ and $A \leftarrow B_1, \ldots, B_k$, resulting in $A' \leftarrow B'_1, \ldots, B'_k$
13 **for** $i \leftarrow 1$ **to** $k$
14 **do** $pf_i \leftarrow$ GENERATEPROOF$(p_1, p_1, B'_i, receivers, i_policies_1, i_policies_1, c_policies_1, KB_1)$
where $pf_i = (p_{r(i)}, (value_i)_{K_{r(i)}})$, and $r(i)$ is the receiver principal of $pf_i$
15 $pfs \leftarrow \wedge_1^n pf_i$
16 $pf \leftarrow$ DECRYPTPROOFS$(p_1, pfs, p_r, s, receivers)$
17 **if** $pf \neq NULL$
18 **then return** $pf$
19 $\triangleright$ If we fail to construct a proof that derives the query locally, we try to obtain a proof from a remote principal.
20 $accessed \leftarrow \emptyset$
21 $rcvs \leftarrow concat(receivers, p_1)$
22 **while** $\exists$ principal $p_l$ $((p_l \notin accessed) \wedge \exists$ policy $p = (rp, t)$ $((p \in i\_policies_1) \wedge (rp$ matches $q) \wedge (p_l \in t)))$
23 **do** $pf \leftarrow$ ISSUEREMOTEQUERY$(p_l, q, n, rcvs, i_policies_1)$ where $pf = (p_{r'}, q', n', (pfs_{K_1})$
24 **if** $q = q' \wedge n = n'$
25 **then if** $p_{r'} = p_1$
26 **then if** $pfs = TRUE$
27 **then return** $(p_r, n, (TRUE)_{K_r})$
28 **else return** $(p_r, n, (FALSE)_{K_r})$
29 **else** $pf' \leftarrow$ DECRYPTPROOFS$(p_1, pfs, p_r, s, receivers)$
30 **if** $pf' \neq NULL$
31 **then return** $pf'$
32 **else** $accessed \leftarrow accessed \cup \{p_l\}$
33 **return** $(p_r, n, (FALSE)_{K_r})$

Figure 3.10: Algorithm for generating a proof.

DECRYPTPROOFS$(p_c, pfs, p_r, s, receivers)$

1 **if** $pfs = \wedge_1^n pf_i$ where $pf_i = (p_{r(i)}, val_i)$
2 **then if** $\forall i\ ((pf_i = (p_c, (val_i)_{K_c})) \wedge\ (val_i = TRUE))$
3 **then return** $(p_r, (TRUE)_{K_r})$
4 **elseif** $\forall i\ ((pf_i = (p_{r(i)}, (value_i)_{K_{r(i)}})) \wedge (((r(i) \neq c) \vee ((r(i) = c) \wedge (value_i = TRUE))))$
5 **then if** $\exists\ p_{r'} \mid (\forall i\ (((p_{r'} \in s) \wedge (index(p_{r(i)}, receivers) \leq index(p_{r'}, receivers)) \wedge (r(i) \neq c)) \vee (r(i) = 1)))$
6 **then return** $(p_{r'}, (\wedge_i\ pf_i)_{K_{r'}})$
for all $i$ where $pf_i = (p_{r(i)}, (value_i)_{K_{r(i)}}) \wedge (r(i) \neq c)$
7 **else return** $NULL$

Figure 3.11: Algorithm for decrypting a set of proofs.

provide the correctness proof in Section 4.6.

## 3.6 Example application

Consider again our initial example of an incident management system (IMS) shown in Figure 2.1; a centralized server would produce the proof tree in Figure 2.2. Figure 3.12 shows how user *bob* (principal $p_0$) requests images from the surveillance camera image server managed by the airport (principal $p_1$). Bob's request is handled by multiple principals $p_1, p_2, \ldots, p_7$. In Figure 3.12, every principal issues queries to the principals that satisfy its integrity policies, and every querier except for principal $p_2$ satisfies the confidentiality policies of the principals to which it sends the queries. Principal $p_2$ does not satisfy $p_4$'s confidentiality policies for query $?location(bob, airport)$, because $p_2$ is temporarily assigned to manage the role server for the incident, and thus principal $p_4$ does not establish a long-term trust relation with principal $p_2$. Fortunately, $p_1$ that runs the surveillance camera image server satisfies $p_4$'s confidentiality policies, principal $p_4$ encrypts the query result with $p_1$'s public key, and principal $p_2$ embeds $p_4$'s proof into its own proof, then returns it to $p_1$. Principal $p_1$ decrypts the query result in the proof from $p_2$, but it is not aware of the fact that the query result is created by principal $p_4$.

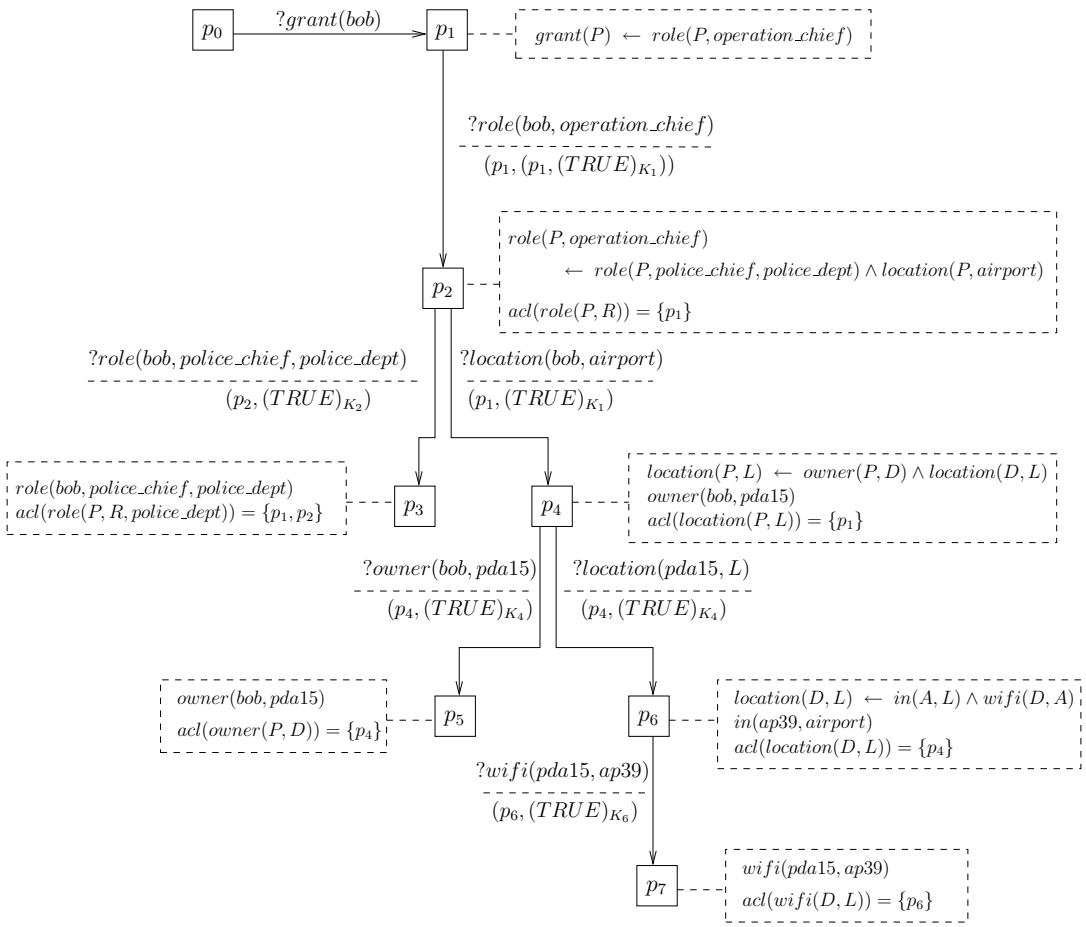



Figure 3.12: Example of an emergency response system. Principal $p_0$ is a first responder whose role is "operation_chief". Principal $p_1$ represents a surveillance camera image server. Principal $p_2$ is the role membership server of an incident management system (IMS). Principal $p_3$ is the role membership server of a police department. Principal $p_4$ represents a location-tracking service. The arrows represent the flow of queries among the principals. Each arrow is labeled with a query and a returned proof. The query is shown above the dashed line; the proof is shown below the line. Each principal's rules, facts and confidentiality policies are shown in a dashed rectangle.

# Chapter 4

# Secure distributed proof system for the general case

In this chapter, we extend our proof system so that it supports confidentiality and integrity policies on rules as well as on facts. A proof contains a proof tree that describes the derivation of the query's result if the evaluation of a query is *true*, instead of simply the result *TRUE*, in order to satisfy a querier principal's integrity policies. This situation occurs when the querier principal does not trust the integrity of the query result from the handler principal, but trusts handler's rule that is used to decompose the query into subqueries. We first describe the representation of a proof that contains a proof tree and then explain how the enforcement mechanisms for confidentiality and integrity policies should be extended to support policies on rules. Finally, we give an extended algorithm that supports a proof that contains a proof tree and provide its soundness proof.

## 4.1 Representation of a proof

We represent a *proof* using nested parentheses based on the grammar in Figure 4.1. A proof contains five fields: a sender principal, a receiver principal, a query, a nonce, and a proof tree optionally encrypted for a receiver. The sender is the principal that publishes a proof, and the receiver is the intended receiver of the proof. The query is a query string for which the proof is constructed, the nonce is a random number chosen by a querier principal, and the proof tree represents how the evaluation result for the query is derived.

The hierarchical structure of a proof tree is built by embedding subproofs into a proof recursively. That is, the proof contains a proof tree that consists of a root node (representing a rule) of the proof tree and the subproofs that contain the subproof trees under the root node. Therefore, each node in a proof tree described in Section 2.2 has a corresponding proof (or an embedded subproof) that contains it as the root node of its proof tree. If a proof contains a single-node proof tree, it only contains a query result or a set of proofs whose query results are encrypted as described in Section 3.4. The digital signature of a proof is attached with the proof as in the basic case in Chapter 3, but the main purpose is to check the integrity of a proof that contains subproofs produced by multiple principals. The digital signature also ensures *non-repudiability* of the sender principal. If a query result depends on encrypted values, it is represented as a set of value pairs that consist of a receiver principal and an encrypted query result, as we describe in Section 4.6.

The first four fields in a proof are necessary to verify the integrity of its proof tree. The sender's identity is necessary to check the authenticity of a proof by checking a digital signature attached with the proof. To verify a proof, one must verify the integrity of all the embedded subproofs in that proof, which are published by different principals. Therefore, every principal that publishes the subproof needs to attach a digital signature with it. We omit the digital signature of a proof from our syntax in Figure 4.1 for brevity. The receiver's

$$
\begin{array}{rcl}
< proofs > & ::= & < proof > (< proof >) * \\
< proof > & ::= & \text{`('} < sender >, < receiver >, < query >, < nonce >, < proof\_tree > \text{`)'} \\
< proof\_tree > & ::= & \text{`('} < rule\_cert >, \text{`('} < proofs > \text{`)'`)'} \mid < proofs > \mid < value\_pairs > \\
 & & \mid < value > \\
< sender > & ::= & < identifier > \\
< receiver > & ::= & < identifier > \\
< query > & ::= & < atom > \\
< atom > & ::= & < predicate > \text{`('} < args > \text{`)'} \\
< predicate > & ::= & < identifier > \\
< args > & ::= & < arg > (, < arg >) * \\
< arg > & ::= & < identifier > \\
< nonce > & ::= & < number > \\
< rule\_cert > & ::= & \text{`('} < rule >, < signer > \text{`)'} \\
< rule > & ::= & < head > \leftarrow < body > \\
< head > & ::= & < atom > \\
< body > & ::= & < atom > \ (\wedge \ < atom >) * \\
< signer > & ::= & < identifier > \\
< value\_pairs > & ::= & < value\_pair > (< value\_pair >) * \\
< value\_pair > & ::= & \text{`('} < receiver >, < value > \text{`)'} \\
< value > & ::= & \text{`}TRUE\text{'} \mid \text{`}FALSE\text{'} \mid \text{`}REJECT\text{'} \\
< identifier > & ::= & < string > \\
< string > & ::= & < string >< character > \mid < character > \\
< character > & ::= & a| \ldots |z|A| \ldots |Z|0|1|2|3|4|5|6|7|8|9 \\
< number > & ::= & < number >< digit > \mid < digit > \\
< digit > & ::= & 0|1|2|3|4|5|6|7|8|9
\end{array}
$$

Figure 4.1: Grammar for a proof. A sender principal attaches a digital signature with its publishing proof, and optionally encrypts the *proof_tree* field of a proof. We, however, omit the digital signatures and encryptions from our syntax. The major differences from the grammar in Figure 3.3 is that a proof has the *sender* field and that the proof contains a *proof tree* instead of a boolean value.

identity is necessary when a proof tree is encrypted by the receiver's public key as we discuss in Section 3.4. The nonce is necessary to prevent a malicious principal from reusing a proof for an identical query at an earlier time.

When we verify the integrity of the query result in a proof, we check the principal that signs the proof. However, when we also verify the integrity of a rule in a proof, we check the principal that defines that rule. That principal may be different from the one that applies the rule to handle a query. Therefore, the rule is paired with the principal that defines it so that the principal that receives a proof can obtain the digitally signed certificate of that rule separately to check the integrity of the rule. For example, if principal $p_0$ issues a query to $p_1$ and obtains a proof that contains a rule $r$ from $p_1$. However, that rule could be actually defined by another principal $p_2$ specified in the *signer* field of the derivation rule for *rule_cert* in Figure 4.1. Principal $p_0$ can check the authenticity of the rule by downloading a certificate that $p_2$ signed on the rule.

**Example.** The example in Figure 4.2 is a modification of Figure 3.6. Principal $p_0$ has different integrity policies, and, as a result, principal $p_1$ returns a proof that contains a proof tree. Principal $p_0$ does not trust the integrity of $p_1$ to evaluate the query $?grant(bob)$, but does trust the integrity of $rule_1$. Principal $p_1$ constructs a proof that consists of the rule $rule_1$ as a root node and the sub-proofs $proof_2$ and $proof_3$ as leaf nodes and returns it to principal $p_0$. The proof tree constructed by principal $p_1$ is trusted by principal $p_0$ because principal $p_0$ trusts $rule_1$ in principal $p_1$ and the facts $role(bob, doctor)$ and $location(bob, hospital)$ in principals $p_2$ and $p_3$ respectively, according to its integrity policies. The proofs $proof_2$ and $proof_3$ embedded in proof $proof_1$ must be attached with the digital signatures signed by principals $p_2$ and $p_3$ so that principal $p_0$ can check their authenticity.

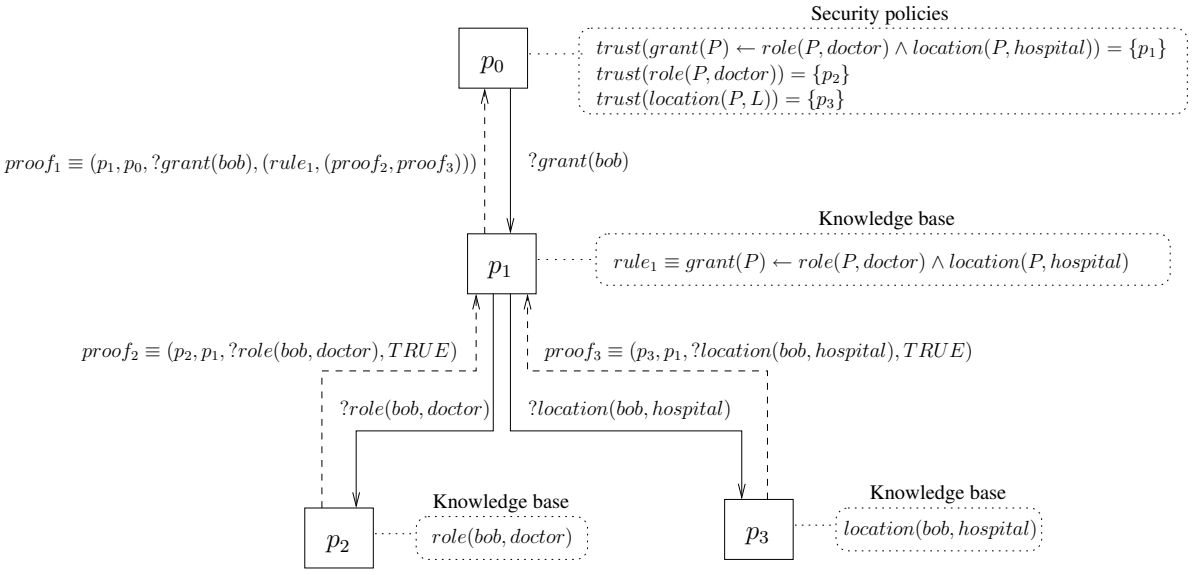



Figure 4.2: Construction of a proof tree. The solid arrows are labeled with queries and the dashed arrows are labeled with returned proof trees. The rounded rectangles with dotted lines represent the knowledge bases or security policies of those principals respectively. We omit nonce and digital signatures in the proofs for brevity.

## 4.2 Decomposition of proof trees

In the general case, a response to a query is a proof that contains a proof tree that satisfies the integrity policies of a querier. If the integrity of the principal that handles a query is trusted by the querier, it only returns a single-node proof tree that contains a query result. If there are such principals participating in evaluating a query, the whole proof tree is decomposed into several subtrees and is evaluated by those principals in a distributed way. The facts and rules used for evaluating a subtree do not have to be disclosed to a querier principal.

In Figure 4.3, principals $p_0, p_1, \ldots, p_{10}$ are the participants in evaluating a query, and each arrow shows how a proof tree flows from one principal to another. We show only the fields for a sender and a receiver principals for brevity, omitting other fields. The dashed lines show which principal's integrity policies are applied to the principals enclosed in the lines. Because principal $p_0$ trusts principal $p_2$ and $p_3$ in terms of the integrity of the given

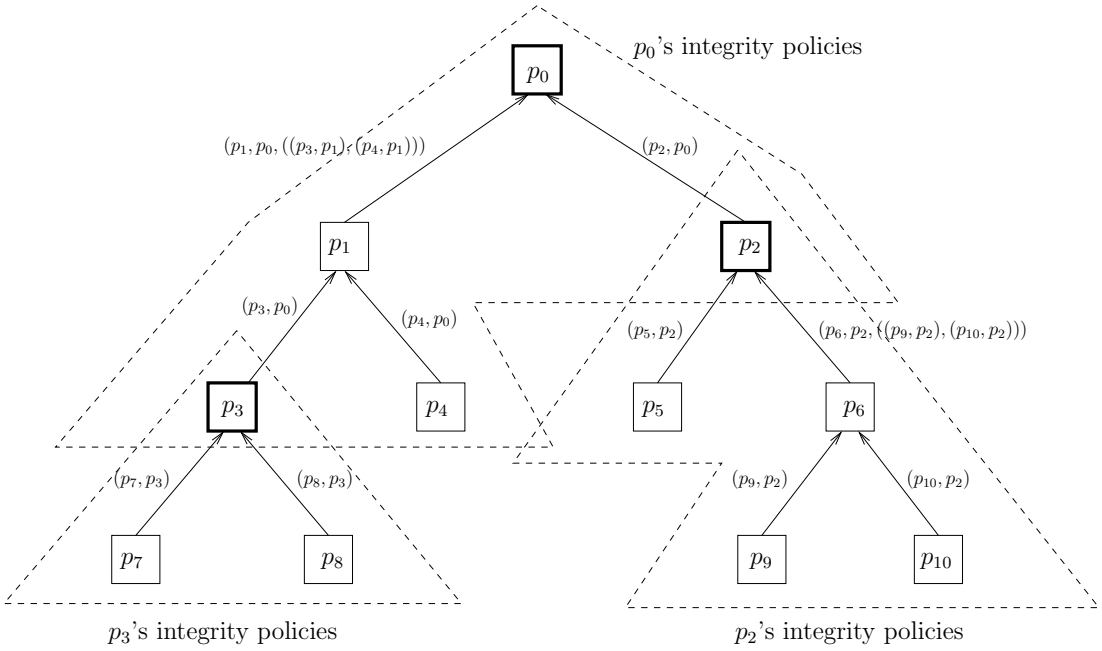



Figure 4.3: Example of subproofs. Principals $p_0, \ldots, p_{10}$ are the participants in evaluating a query. Each arrow shows how a proof tree flows from one principal to another. Each arrow is labeled with the pair of a sender and a receiver principals in a proof, omitting the other fields of the proof for brevity. The dashed lines show which principal's integrity policies are applied to the principals enclosed in the lines. The principals $p_0$, $p_2$, and $p_3$ that represent the root node of the nested subtrees are enclosed in the thick rectangles.

queries; it is possible to evaluate the query at $p_0$, $p_2$, and $p_3$ rather than collecting all the rules and facts at $p_0$. Principals $p_2$ and $p_3$ construct a proof tree locally based on their own integrity policies, and return only a single-node proof tree that contains a query result. Therefore, principal $p_0$ does not know how the query results from $p_2$ and $p_3$ are derived. A principal that issues a query attaches its local or a upstream principal's integrity policies with the query so that those policies are shared with downstream principals as we explain in Section 4.5.

## 4.3 Enforcement of confidentiality policies

We apply the same mechanism for enforcing confidentiality policies as in Section 3.4. The only difference is that a receiver principal must be an upstream principal that evaluates a proof subtree. We, therefore, define a set of principals $receivers(p)$ whose members are eligible to receive principal $p$'s proof as follows.

Suppose that in a proof tree there is a sequence of nodes $n_0, n_1, \ldots, n_k$ on the path from the root $n_0$ to node $n_k$ in the proof tree, and principal $p_i$ represents node $n_i$ and handles query $q_{i-1}$ from $p_{i-1}$ for $i = 1$ to $k$. Principal $p_i$ where $i < k$ belongs to the set $receivers(p_k)$ if it satisfies either of the following two conditions.

- Principal $p_i$ is $p_0$.
- There exists principal $p_l$ that belongs to $receivers(p_k)$, $p_l$ has an integrity policy $(rp, t)$ such that rule pattern $rp$ matches query $q_{i-1}$ and $p_i$ belongs to the set of principals $t$ (i.e., $trust(rp)$), and there is no other principal $p_j$ (where $l < j < i$), that satisfies this condition.

**Example.** We explain the above definition of $receivers(p_k)$ with an example in Figure 4.4. Suppose that principal $p_0$ issues a query $?grant(Bob)$ to $p_1$, and principal $p_1$ and $p_2$ issue subsequent queries $?role(Bob, doctor)$ and $?location(Bob, hospital)$ respectively. We decide the members of $receivers(p_3)$ as follows. Since principal $p_0$ satisfies the first condition, principal $p_0$ belongs to $receivers(p_3)$. Next, we consider the second condition. Principal $p_0$ defines an integrity policy $p_2 \in trust_0(role(P, R))$ (i.e., $(role(P, R), \{p_2\})$). The rule pattern $role(P, R)$ of the integrity policy matches query $role(Bob, hospital)$ that $p_2$ handles, and $p_2$ belongs to the trust list of the integrity policy. In addition, principal $p_1$ does not belong to $receivers(p_3)$, because principal $p_0$ only trusts $p_1$'s rule that is used to

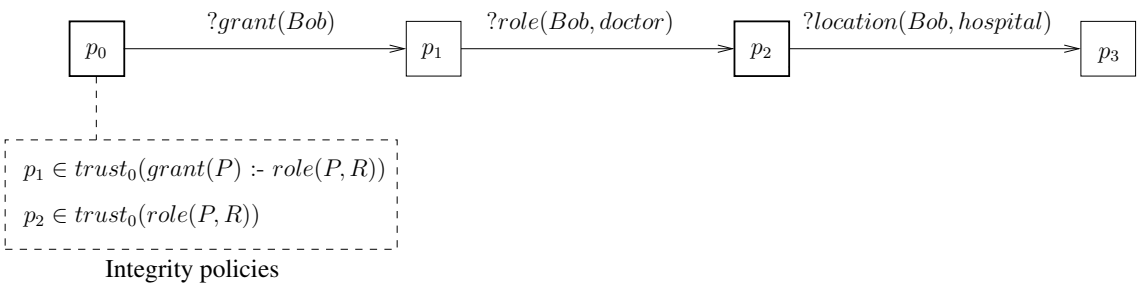



Figure 4.4: Example of a receivers list.

handle $p_0$'s query. Therefore, principal $p_2$ satisfies the second condition, and two principals $p_0$ and $p_2$ belong to $receivers(p_3)$.

Notice that our new definition does not change the definition of $receivers(p)$ in Section 3.4, because every principal issues a query to a principal that it trusts in terms of the integrity of evaluating the query. That is, if a querier principal $p_{i-1}$ in $receivers(p)$ issues query $q_{i-1}$ to $p_i$, $p_i$ belongs to $receivers(p)$ as well because $p_i$ satisfies the second condition above. In other words, all the upstream principals of $p$ belong to the set $receivers(p)$.

## 4.4 Hybrid encryption

When our system performs a public-key encryption, it first generates a symmetric key randomly and encrypts a proof with that symmetric key. We only use a public key to encrypt the symmetric key. To decrypt a proof, the system first decrypts the symmetric key with the receiver's private key and then decrypts the proof with that symmetric key.

## 4.5 Algorithms

Each host provides the same remote interface for handling a remote query. We describe the extended version of the function GENERATEPROOF, and then introduce the function CHECKPROOFINTEGRITY that checks the integrity of a proof tree that contains rules as

intermediate nodes.

**Algorithm for constructing a proof.** In Figure 4.6, we extend the algorithm in Figure 3.10 to support security policies on rules. There are a few modifications as follows. First, a proof has additional fields such as a sender principal, a query string, and a nonce according to the representation of a proof in Section 4.1. We use the parameter names $rcvrs$, $i\_pol_0$, $i\_pol_1$, and $c\_pol_1$ instead of $receivers$, $i\_policies_0$, $i\_policies_1$, and $c\_policies_1$ respectively for compactness.

Second, the algorithm handles a proof from a remote principal that contains multiple subproofs. The query result of the proof is the conjunction of the query results of all the embedded subproofs. The query result is *TRUE* if all the query results of the embedded subproofs have a *TRUE* value; otherwise, it is *FALSE*. Lines 15–19 construct a proof from the proofs $pf_i$ for $i = 1$ to $k$ obtained by calling the function GENERATEPROOF in line 14. Each proof $pf_i$ contains either a query result or multiple subproofs. Therefore, the query result of the proof is the conjunction of the query results of proofs $pf_i$ for $i = 1$ to $k$, and, if proof $pf_i$ contains multiple subproofs, its query result is represented as the conjunction of those embedded subproofs. Line 15 checks whether the handler principal $p_1$ can read all the query results of proof $pf_i$, and all the query results are a *TRUE* value. The query result of the proof is *TRUE* if the proof contains a *TRUE* value or all the embedded subproofs contains a *TRUE* value. If the condition in line 15 holds true, line 16 returns a proof with a *TRUE* value. Line 17 handles the case that principal $p_1$ cannot decrypt all the query results in the proofs $pf_i$ for $i = 1$ to $k$ and all the decrypted query results have a *TRUE* value. If so, line 19 checks whether there is a principal $p_{r'}$ that satisfies the constraint due to recursive encryption. We need to consider all the receiver principals of the embedded subproofs as well. If there exists such a principal $p_{r'}$, line 19 returns a proof that contains the subproofs whose query results cannot be decrypted by principal $p_1$.

GENERATEPROOF$(p_0, p_1, q, n, rcvrs, i_pol_0, i_pol_1, c_pol_1, KB_1)$

1 $\triangleright$ Check whether there is any principal in $rcvrs$ that satisfies $p_1$'s confidentiality policies.
2 $s \leftarrow rcvrs \cap (\bigcup_i t_i)$ for all policies $(rp_i, t_i) \in c\_pol_1$ where $rp_i$ matches $q$
3 **if** $s = \emptyset \triangleright$ if set $s$ is empty.
4 **then return** $(p_1, p_0, q, n, (\textit{REJECT})_{K_0})$
5 $p_r \leftarrow minIndex(s, rcvrs)$
6 $\triangleright$ Check whether principal $p_1$ satisfies querier $p_0$'s integrity policies.
7 **if** $\exists$ policy $p = (rp, t) \mid ((p \in i\_pol_0) \wedge (rp \text{ matches } q) \wedge (p_1 \in t))$
8 **then** append $p_1$ to $rcvrs$
9 **if** $\exists$ fact $f \mid ((f \in KB_1) \wedge (f \text{ matches } q))$
10 **then return** $(p_1, p_r, q, n, (\textit{TRUE})_{K_r})$
11 **elseif** $\exists$ rule $r \equiv A \leftarrow B_1, \ldots, B_k \mid ((r \in KB_1) \wedge (A \text{ matches } q))$
12 **then** unify $q$ and $A \leftarrow B_1, \ldots, B_k$, resulting in $A' \leftarrow B'_1, \ldots, B'_k$
13 **for** $i \leftarrow 1$ **to** $k$
14 **do** $pf_i \leftarrow$ GENERATEPROOF$(p_1, p_1, B'_i, n, rcvrs, i_pol_1, i_pol_1, c_pol_1, KB_1)$
where $pf_i = (p_{s(i)}, p_{r(i)}, B'_i, n, (pt_i)_{K_{r(i)}})$,
$s(i)$ and $r(i)$ are sender and receiver principals of $pf_i$ respectively.
15 **if** $\forall i \; ((r(i) = 1) \wedge (((pt_i = val_i) \wedge (val_i = \textit{TRUE}))$
$\vee((pt_i = \wedge_j (p_{r(i,j)}, (val_{ij})_{K_{r(i,j)}})) \wedge \forall j \; ((r(i,j) = 1) \wedge (val_{ij} = \textit{TRUE})))))$
16 **then return** $(p_1, p_r, q, n, (\textit{TRUE})_{K_r})$
17 **elseif** $\forall i \; ((r(i) \neq 1) \vee ((r(i) = 1) \wedge (((pt_i = val_i) \wedge (val_i = \textit{TRUE}))$
$\vee \; ((pt_i = \wedge_j (p_{r(i,j)}, (val_{ij})_{K_{r(i,j)}}))$
$\wedge(\forall j \; ((r(i,j) \neq 1) \vee ((r(i,j) = 1) \wedge (val_{ij} = \textit{TRUE}))))))))$
18 **then if** $\exists \; p_{r'} \; ((p_{r'} \in s) \wedge (\forall i \; ((r(i) = 1) \vee \; (((pt_i = val_i)$
$\wedge(index(p_{r(i)}, rcvrs) \leq index(p_{r'}, rcvrs))) \vee ((pt_i = \wedge_j (p_{r(i,j)}, (val_{ij})_{K_{r(i,j)}}))$
$\wedge(\forall j \; ((r(i,j) = 1) \vee (index(p_{r(i,j)}, rcvrs) \leq index(p_{r'}, rcvrs))))))))))$
19 **then return** $(p_1, p_{r'}, q, n, ((\wedge_i \; (p_{r(i)}, pt_i))(\wedge_{ij} \; (p_{r(i,j)}, (pt_{ij})_{K_{r(i,j)}})))_{K_{r'}})$
where $((pf_i = (p_{s(i)}, p_{r(i)}, B'_i, n, (pt_i)_{K_{r(i)}}) \wedge (r(i) \neq 1))$
$\vee((r(i) = 0) \wedge (pt_i = \wedge_j (p_{r(i,j)}, (val_{ij})_{K_{r(i,j)}})) \wedge r(i,j) \neq 1))$
20 $\triangleright$ Return a proof with rules that satisfy principal $p_0$'s integrity policies and $p_1$'s confidentiality policies.
21 **return** GENERATEPROOFWITHRULES$(p_0, p_1, q, n, rcvrs, i_pol_0, i_pol_1, c_pol_1, KB_1)$

Figure 4.5: Algorithm for generating a proof.

We discuss the remaining algorithm in the separate function GENERATEPROOFWITH-RULES that is called in line 21.

Third, we handle the case that principal $p_1$ is not trusted by $p_0$ in terms of the evaluation of a query, but $p_1$'s rule, which matches the query, is trusted by principal $p_0$, in lines 1–7. Line 21 checks whether there is a rule $R \equiv A \leftarrow B_1 \wedge \ldots \wedge B_k$ in $p_1$'s knowledge base whose head $A$ matches query $q$ and querier principal $p_0$ satisfies $p_1$'s confidentiality policies for rule $R$. If there is such rule $R$, line 2 checks whether querier $p_0$ has an integrity policy $p = (rp', t')$ that trusts the integrity of $p_1$'s rule $R$. Line 3 unifies query $q$ and rule $R$ resulting $R' \equiv A' \leftarrow B'_1, \ldots, B'_k$. Lines 4–5 obtain the proofs for the atoms $B'_1, \ldots, B'_k$ iteratively. Line 6 checks whether there is a receiver principal $p_{r'}$ in the set of principals $rcvrs$ that satisfies the constraints due to recursive encryption described in Section 3.4. If that holds true, we return the proof that contains rule $R'$ as the root node of the proof tree, and the proofs for $B'_1, \ldots, B'_k$ as the subproofs under the root node. The proof tree must contain the proofs whose proof trees are decrypted by $p_1$ to satisfy the receiver principal's integrity policies.

Fourth, when principal $p_1$ tries to construct a proof by issuing a remote query, we need to check whether querier principal $p_0$ trusts the integrity of the query result from handler principal $p_1$. Line 11 checks that condition by checking whether $p_1$ belongs to $rcvrs$, because line 8 in the function GENERATEPROOF appends $p_1$ to $rcvrs$ if the condition holds. If that holds true, line 12 issues a remote query with $p_1$'s integrity policies $i\_pol_1$. That is, $p_1$'s integrity policies are applied to the succeeding queries. Line 13 checks the integrity of the returned proof by calling the function CHECKPROOFINTEGRITY. The function returns a pair of a boolean value (*true* or *false*) and a simplified proof as we explain below. If the proof satisfies $p_1$'s integrity policies, line 15 returns the proof returned by the function CHECKPROOFINTEGRITY. If $p_1$ does not belong to $rcvrs$, line 17 issues a remote query with principal $p_0$'s integrity policies; that is, the querier principal's integrity policies are ap-

GENERATEPROOFWITHRULES$(p_0, p_1, q, n, rcvrs, i_pol_0, i_pol_1, c_pol_1, KB_1)$

1 **if** ($\exists$ rule $R \mid ((R \in KB_1) \wedge (R \equiv A \leftarrow B_1 \wedge \ldots \wedge B_k) \wedge (A$ matches $q)))$
   $\wedge(\exists$ policy $p \mid ((p \in c\_policies_1) \wedge (p = (rp, t)) \wedge rp$ matches rule $R)))$
2  **then if** $\exists$ policy $p' = (rp', t') \mid ((p' \in i\_pol_0) \wedge (rp'$ matches $R) \wedge (p_1 \in t'))$
3  **then** unify $q$ and rule $R$ resulting $R' \equiv A' \leftarrow B'_1, \ldots, B'_k$
4  **for** $i \leftarrow 1$ **to** $k$
5  **do** $pf_i \leftarrow$ GENERATEPROOF$(p_1, p_1, B'_i, n, rcvrs, i_pol_0, i_pol_1, c_pol_1, KB_1)$
   where $pf_i = (p_{s(i)}, p_{r(i)}, B'_i, n, (pt_i)_{K_{r(i)}})$, and
   $s(i)$ and $r(i)$ are sender and receiver principals of $pf_i$
6  **if** $\exists p_{r'}\ ((p_{r'} \in s) \wedge (\forall i\ (index(p_{r(i)}, rcvrs) \leq index(p_{r'}, rcvrs))))$
7  **then return** $(p_1, p_{r'}, q, n, ((R', p_c), \wedge_i pf_i)_{K_{r'}})$
   where $p_c$ is a signer principal of rule $R$
8  $\triangleright$ If we fail to construct a proof that derives the query locally,
   $\triangleright$ we try to obtain a proof from a remote principal.
9  *accessed* $\leftarrow \emptyset$
10 **while** $\exists$ principal $p_l$ that is capable of handling query $q$ and does not belong to the set $accessed$
11  **do if** $p_1 \in rcvrs$
12  **then** *proof* $\leftarrow$ ISSUEREMOTEQUERY$(p_l, q, rcvrs, i_pol_1)$
13  $(trusted, proof') \leftarrow$ CHECKPROOFINTEGRITY$(p_1, q, n, proof, i_pol_1)$
14  **if** $trusted$
15  **then return** *proof'*
16  **else** *accessed* $\leftarrow$ *accessed* $\cup \{p_l\}$
17  **else** *proof* $\leftarrow$ ISSUEREMOTEQUERY$(p_l, q, rcvrs, i_pol_0)$
18  **return** *proof*
19 **return** $(p_1, p_r, q, n, (FALSE)_{K_r})$

Figure 4.6: Algorithm for generating a proof that contains rules as intermediate nodes.

plied to the succeeding queries. Line 18 returns the proof returned by the function without checking its integrity. In other words, only principals trusted by their querier principals in terms of the integrity of their query results need to enforce their integrity policies on proofs received from remote principals. Finally, line 19 returns a proof with a *FALSE* value, since there is no other way to construct a proof for query $q$.

**Algorithm for checking the integrity of a proof.** The function CHECKPROOFINTEGRITY in Figure 4.7 checks whether a proof satisfies given integrity policies, based on the definition given in Section 2.4.1. It takes as parameters principal $p_c$ that checks the integrity of the proof, query string $q$, nonce $n_c$, proof *pf*, and $p_c$'s integrity policies $i\_policies_c$. The function also converts the hierarchical proof tree in a proof into a flat one that contains

encrypted query results in the leaf nodes; that is, all the intermediate nodes are removed from the proof tree while checking the integrity of those nodes.

Line 1 checks whether nonce $n$ in the proof *pf* is same as the nonce $n_c$ for the query. We assume that the authenticity of the proof *pf* is also checked by checking the digital signature attached with the proof in line 1. If that is not true, line 2 returns *false* with no proof tree. Line 3 checks whether $p_c$ trusts the integrity of principal $p_s$'s evaluating query $q$. If that holds true, line 4 returns *true* with the proof given as a parameter. Line 5 checks whether principal $p_c$ can decrypt the proof (i.e., principal $p_r$ is a receiver principal of the proof *pf*) and reads rule $R$ at the root of the proof tree. Notice that all the variables in rule $R$ is unified with some constants when the proof tree was constructed. Line 6 checks whether rule $r$ signed by principal $p_d$ satisfies $p_c$'s integrity policies. If that holds true, lines 7–11 check whether all the proofs for the atoms of rule $R$ satisfies $p_c$'s integrity policies by calling the function CHECKPROOFINTEGRITY recursively. If all the proofs satisfy the integrity policies, line 11 returns *true* with the proof that contains the concatenation of the subproofs that correspond to the leaf nodes of the initial proof tree.

Notice that it is necessary for the principal that checks the integrity of a proof to be able to read all the rules in the intermediate nodes of the proof tree.

## 4.6 Soundness of the algorithm

We show that our algorithm constructs a proof tree only if the confidentiality and integrity policies of every participating principal are satisfied.[1] We give the proof for the general case below, which covers the basic case in Chapter 3 as its special case. We separate the proof into two parts: the proof on confidentiality policies, and the proof on integrity policies.

[1]The other way (completeness of the algorithm) does not hold, as we discuss in Section 8.1, and we leave it as our future work.

CHECKPROOFINTEGRITY$(p_c, q, n_c, \mathit{pf}, i_policies_c)$

1 **if** $\neg((\mathit{pf} = (p_s, p_r, q, n, (\mathit{pt})_{K_r})) \wedge (n_c = n))$
2 **then return** $(false, \mathit{NULL})$
3 **if** $(\exists$ policy $p = (rp, t) \mid ((p \in i\_policies_c) \wedge (rp$ matches query $q) \wedge (p_s \in t)))$
4 **then return** $(true, \mathit{pf})$
5 **elseif** $((r = c) \wedge (\mathit{pt} = ((R, p_d), (\wedge_{i=1}^{n}\ \mathit{pf}_i))$
where $R$ is a rule, $p_d$ is the signer principal of $R$, and $\mathit{pf}_i$ for $i = 1$ to $n$ are subproofs.
6 **then if** $\exists$ policy $p = (rp, t) \mid ((p \in i\_policies_c) \wedge (rp$ matches rule $R) \wedge (p_d \in t)$
$\wedge$(principal $p_c$ holds a valid digital signature for $R$ signed by $p_d))$
where $R \equiv A \leftarrow B_1 \wedge \ldots \wedge B_k$
7 **then for** $i \leftarrow 1$ **to** $k$
8 **do** $(trust, pf_i') =$ CHECKPROOFINTEGRITY$(p_c, B_i, \mathit{pf}_i, i_policies_c)$
9 **if** $\neg trust$
10 **then return** $(false, \mathit{NULL})$
11 **return** $(true, (p_c, p_c, q, n, (\wedge_i\ pf_i')_{K_c}))$
12 **else return** $(false, \mathit{NULL})$
13 **else return** $(false, \mathit{NULL})$

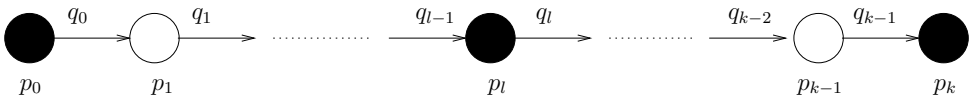
Figure 4.7: Algorithm for checking proof integrity.

### 4.6.1 Proof for confidentiality policies

We prove that our algorithm constructs a proof tree only if the confidentiality policies of every participating principal are satisfied by induction below.

**Base case:** We first show that our claim holds in the case of a single-node proof tree. Suppose that principal $p_0$ makes query $q$ to principal $p_1$, and $p_1$, which does not issue any subqueries, returns a proof whose proof tree only contains a root node. We only need to show that $p_1$'s confidentiality policies are satisfied, because $p_0$ does not disclose any information in its knowledge base to $p_1$. To satisfy $p_1$'s confidentiality policies, $p_1$ must have a confidentiality policy $(rp, t)$ such that rule pattern $rp$ matches query $q$ and $p_1$ belongs to the set $t$. The function GENERATEPROOF in Figure 4.6 ensures this condition in line 3. Therefore, principals $p_0$ and $p_1$ construct a proof only if their confidentiality policies are satisfied.


Case 1: Only principal $p_0$ belongs to the set $receivers(p_k)$.




Case 2: Some intermediate principal $p_l$ belongs to the set $receivers(p_k)$ as well.

Figure 4.8: Linear proof trees with and without an intermediate principal that belongs to the set $receivers(p_k)$. Black circles denote principals that belong to $receivers(p_k)$, and white circles denote principals that does not belong to $receivers(p_k)$. Each circle is labeled with a principal name, and each arrow is labeled with a query name.

**Induction step:** We next show that, if our claim holds for a proof tree whose depth is less than $k$, then it also holds for a proof tree of depth $k$. (The base case above considers a tree of depth 1.) Without loss of generality, we consider the case that a proof tree is linear. Because our algorithm for enforcing confidentiality policies on each node depends only on the nodes on the path from that node to the root in a proof tree; the node is not aware of the existence of the nodes in other branches of the proof tree.

Suppose that there is a linear tree of depth $k$ where nodes $n_0, \ldots, n_k$ are ordered from the root to the leaf. Let $p_0, \ldots, p_k$ be the principals that represent nodes $n_0, \ldots, n_k$ respectively, and $q_0, \ldots, q_{k-1}$ be the queries, where $q_i$ is the query by $p_i$ to $p_{i+1}$. When principal $p_0$ issues query $q_0$ to $p_1$, we consider two cases in Figure 4.8. In case 1, only principal $p_0$ belongs to a set of principals $receivers(p_k)$ defined in Section 4.3. In case 2, there are some other principals in the set $receivers(p_k)$ besides principal $p_0$.

We first consider case 1. Because principal $p_1$ does not belong to $receivers(p_k)$, prin-

cipal $p_2$ cannot distinguish query $q_1$ issued by principal $p_1$ from $q_1$ issued by principal $p_0$ instead, because all the parameters in those queries are same in both cases; the set $receivers$ contains only principal $p_0$ in both cases. The same can be observed for $p_2, \ldots, p_k$. In the latter case, by the induction hypothesis, our algorithm ensures that a proof tree for query $q_1$ is constructed by principals $p_2, \ldots, p_k$ if their confidentiality policies are satisfied. Because principals $p_2, \ldots, p_k$ do not distinguish the former case from the latter, our algorithm ensures that their confidentiality policies are preserved in the former case as well. The function GENERATEPROOF in Figure 4.6 ensures principal $p_1$'s confidentiality policies in lines 3. Principal $p_0$'s confidentiality policies are vacuously satisfied because $p_0$ does not disclose any information. We, therefore, prove that our algorithm ensures the confidentiality policies of the principals $p_0, \ldots, p_k$ with a proof tree of depth $k$ in case 1.

We next consider case 2. Without loss of generality, we assume that there is a single principal $p_l$ in $receivers(p_k)$ between principal $p_0$ and $p_k$. There are two subcases to be considered. In the first, subcase 2a, principal $p_l$ can decrypt all the nodes $n_{l+1}, \ldots, n_k$ in the proof tree for query $q_l$; that is, principals $p_{l+1}, \ldots, p_k$ choose $p_l$ as the receiver of their returning proofs. Because principals $p_{l+1}, \ldots, p_k$ do not choose $p_0$ from $receivers(p_j) = \{p_0, p_l\}$ for $j = l + 1$ to $k$ as the receiver principal of their proofs respectively, their algorithm works in the same way as the case where the set $receivers(p_j) = \{p_l\}$ for $j = l$ to $k$. Therefore, by the induction hypothesis, our algorithm ensures the confidentiality policies of $p_{l+1}, \ldots, p_k$. Because principal $p_l$ returns a proof with a single-node proof tree, principals $p_0, \ldots, p_{l-1}$ are not aware of the fact that principal $p_l$ issues query $q_l$ for handling query $q_{l-1}$. Therefore, by the induction hypothesis, our algorithm ensures the confidentiality policies of $p_0, \ldots, p_{l-1}$. Principal $p_l$'s confidentiality policies are also satisfied because our algorithm for enforcing confidentiality policies on $p_l$ works in the same way as the case that $p_l$ does not issue any subqueries and constructs a single-node proof tree responding to query $q_{l-1}$, because there is no constraint on $p_l$ due to recursive encryption because $p_l$ can

decrypt all the nodes in the proof from $p_{l+1}$ to $p_k$. Therefore, our claim holds for subcase 2a.

The second subcase 2b is that principal $p_l$ cannot decrypt some nodes in the proof tree received from $p_{l+1}$. If principal $p_l$ cannot decrypt node $n_m$ between $n_l$ and $n_k$ (i.e., $l < m < k$), the proof tree does not satisfy $p_l$'s integrity policies, and the proof fails. We, therefore, only consider the case that $p_l$ cannot decrypt leaf node $n_k$ only. When node $n_k$ chooses $p_0$ as a receiver principal, our algorithm for enforcing confidentiality policies works for nodes $n_1, \ldots, n_{k-1}$ in the same way as the case that node $n_k$ is omitted (i.e., principal $p_{k-1}$ does not issue query $q_{k-1}$ to $p_k$) because $p_k$'s proof encrypted with principal $p_0$'s public key does not interfere with the processes of principals $p_1, \ldots, p_{k-1}$ for choosing a receiver principal of their proofs from the set $receivers = \{p_0, p_l\}$ or $\{p_0\}$. The depth of the tree with nodes $n_1, \ldots, n_{k-1}$ is $k - 1$. Therefore, by the induction hypothesis, our algorithm ensures that a proof tree is constructed only when the confidentiality policies of principals $p_1, \ldots, p_{k-1}$ are satisfied. Principal $p_0$'s confidentiality policies are satisfied vacuously, and $p_k$'s confidentiality policies of principal $p_k$ are also satisfied because our algorithm on $p_k$ works in the same way as the case that $p_k$ constructs a proof tree of a single depth responding to query $q_{k-1}$ issued by principal $p_0$. Therefore, our algorithm ensures that a proof tree is constructed only when the confidentiality policies of every principal is satisfied. We cover all the cases in terms of confidentiality policies and conclude the proof.

### 4.6.2 Proof for integrity policies

We prove that our algorithm constructs a proof tree only if the integrity policies of every participating principal are satisfied by induction below.

**Base case:** We first show that our claim holds in the case of a single-node proof tree. Suppose that principal $p_0$ makes query $q_0$ to principal $p_1$, and $p_1$, which does not issue any subqueries, returns a single-node proof tree. We only need to show that $p_0$'s integrity poli-

cies are satisfied, because $p_0$ does not disclose any information in its knowledge base. To satisfy $p_0$'s integrity policies, $p_0$ must have an integrity policy $(rp, t)$ such that rule pattern $rp$ matches query $q$ and $p_1$ belongs to set $t$. Line 31 in $p_0$'s function GENERATEPROOF in Figure 4.6 obtains a proof from $p_1$ by calling the function ISSUEREMOTEQUERY, and line 32 in the function calls the function CHECKPROOFINTEGRITY whose line 3 ensures that the proof satisfies the above condition. Therefore, principals $p_0$ and $p_1$ construct a proof if their integrity policies are satisfied.

**Induction step:** We next show that if our claim holds for a proof tree whose depth is less than $k$, then it also holds for a proof tree of depth $k$. We consider the case that a proof tree is linear as we do in Section 4.6.1, because we can check the integrity of a proof tree by checking whether every path from the root to each leaf node satisfies given integrity policies. This claim is proved by induction as follows. The base case holds because there is only a single node in a proof tree. Suppose that our claim holds for a proof tree of depth $k-1$. By induction hypothesis, each subtree of depth $k-1$ satisfies the integrity policies of every participating principal if every path from the root node to each leaf node satisfies the integrity policies. If every path from the root node to each leaf node in the proof tree of depth $k$ satisfies integrity policies, the root node must satisfy the policies as well. According to our definition of the integrity of a proof tree in Section 2.4.1, a proof tree of depth $k$ satisfies given integrity policies if the root node and all the subtrees of depth $k-1$ under the root node satisfy the integrity policies. Therefore, our claim holds for the proof tree of depth $k$, and we conclude the proof of the above claim.

We assume the same linear proof tree in Section 4.6.1; that is, there is a linear tree of length $k$ where nodes $n_0, \ldots, n_k$ are ordered from the root to the leaf. Let $p_0, \ldots, p_k$ be the principals that represent nodes $n_0, \ldots, n_k$ respectively, and $q_0, \ldots, q_{k-1}$ be the queries as before. When principal $p_0$ issues query $q_0$ to $p_1$, we consider the same two cases in

Figure 4.8.

We first consider case 1. Because principal $p_1$ does not belong to the set $receivers(p_k)$, principal $p_2$ cannot distinguish query $q_1$ issued by principal $p_1$ from $q_1$ issued by principal $p_0$ instead, because all the parameters in those queries are same in both cases. In the latter case, by the induction hypothesis, our algorithm ensures that a proof tree for query $q_1$ is constructed by principals $p_2, \ldots, p_k$ if their integrity policies are satisfied. Because principals $p_2, \ldots, p_k$ do not distinguish the former case from the latter, our algorithm ensures their integrity policies in the former case as well. Principal $p_1$ checks the integrity of the proof from principal $p_2$ in the same way regardless of whether $p_1$'s issuing query $q_1$ is for handling query $q_0$ or not. Therefore, by the induction hypothesis, $p_1$'s integrity policies are satisfied. Principal $p_0$ checks the integrity of the proof from principal $p_1$ with the function CHECKPROOFINTEGRITY as follows. The integrity of the rule in node $n_1$ is ensured in line 6, and, by the induction hypothesis, the integrity of the subtree of depth $k-1$ from principal $p_2$ is ensured in line 8 by checking the integrity of the proof tree whose root node is $n_2$ by calling the function CHECKPROOFINTEGRITY recursively. Therefore, the function ensures that $p_0$'s integrity policies are satisfied with the proof tree from node $n_1$. We, therefore, prove that our algorithm ensures the integrity policies of the principals $p_0, \ldots, p_k$ with a proof tree of depth $k$ in case 1.

We next consider case 2. Without loss of generality, we assume that there is a principal $p_l$ in $receivers(p_k)$ between principal $p_0$ and $p_k$. There are two subcases to be considered. In the first, subcase 2a, the subproof from principal $p_l$ is a single-node proof tree that contains a query's result; principals $p_{l+1}, \ldots, p_k$ choose $p_l$ as the receiver of their nodes. Because principals $p_0, \ldots, p_{l-1}$ are not aware of the fact that principal $p_l$ issues query $q_l$, by the induction hypothesis, the integrity policies of principals $p_0, \ldots, p_{l-1}$ are satisfied. The fact that principal $p_0$ belongs to the list $receivers(p_l)$ of query $q_l$ does not change the behaviors of principals $p_{l+1}, \ldots, p_k$ for handling query $q_l$. Because our algorithm works

for principals $p_l, \ldots, p_k$ in the same way that principal issues query $q_l$ independently, by the induction hypothesis, our algorithm ensures that principal $p_l$'s integrity policies are satisfied for subcase 2a.

The second case 2b is that a proof from principal $p_l$ contains node $n_k$ whose proof tree is encrypted with $p_0$'s public key, as it could be done in line 19 of the function GENERATEPROOF in Figure 4.6. The proof from $p_l$ does not contain any other encrypted nodes because $p_l$ needs to read the nodes $n_{l+1}, \ldots, n_{k-1}$ to check the integrity of the proof from $p_{l+1}$. Principal $p_l$ checks whether the rules in nodes $n_{l+1}, \ldots, n_{k-1}$ satisfies $p_l$'s integrity policies, which is done in line 6 of the function CHECKPROOFINTEGRITY in Figure 4.7. If principal $p_l$ cannot decrypt all the nodes $n_{l+1}, \ldots, n_{k-1}$, $p_l$ returns a proof that contains *FALSE* because its failure to check the integrity of the proof, and, therefore, the proof tree for query $q_0$ is not constructed. Because principals $p_0, \ldots, p_{l-1}$ cannot distinguish whether the encrypted boolean value in the proof from $p_l$ is generated by principal $p_l$ or its descendant principal $p_k$, by the induction hypothesis, our algorithm ensures that the integrity polices of principals $p_0, \ldots, p_{l-1}$ are satisfied if $p_0$ accepts a proof tree whose leaf node $n_l$ contains an encrypted boolean value in node $n_k$.

We next consider the integrity policies of principals $p_l, \ldots, p_k$. In order for principal $p_l$ to check the integrity of the proof from principal $p_{l+1}$, $p_l$ must read all the intermediate nodes $n_{l+1}, \ldots, n_{k-1}$ in that proof. Therefore, principals $p_{l+1}, \ldots, p_{k-1}$ must choose $p_l$ as the receiver principal of their returning proofs. Principal $p_{l+1}, \ldots, p_{k-1}$ work in the same way as the case that principal $p_l$ issues query $q_l$ without receiving $q_{l-1}$ so, by the induction hypothesis, their integrity policies are preserved. Principal $p_k$'s integrity policies are satisfied vacuously. Principal $p_l$'s algorithm for enforcing integrity policies does not read the encrypted value in node $n_k$ and works in the same way regardless of returning a proof to $p_{k-1}$ or not. Therefore, by the induction hypothesis, $p_l$'s integrity policies are also preserved. We cover all the cases and conclude the proof.

## 4.7 Example application

We revisit the example of an incident management system (IMS); in Figure 3.12, every querier principal trusts the integrity of the principal that handles its query in terms of the correctness of the query's result. This time, we have some principals that define security policies on rules as well as facts.

Figure 4.9 shows how user *bob* (principal $p_0$) requests images from the surveillance camera image server managed by the airport (principal $p_1$). Principal $p_1$ agrees with the policy for role $operation\_chief$, that is, principal $p_2$'s rule below satisfies $p_1$'s integrity policies.

$$role(P, operation\_chief) \leftarrow roleIn(P, police\_chief, police\_dept) \wedge in(P, airport)$$

Therefore, principal $p_2$ that runs the role-membership server of IMS uses that rule to evaluate a query $role(bob, operation\_chief)$. However, principal $p_1$ does not trust the answer from principal $p_2$, since $p_2$ is temporarily assigned to manage the role server for the incident, and thus principal $p_1$ does not establish a long-term trust relation with principal $p_2$. Fortunately, principal $p_2$ trusts the role-membership server of the police department and the location tracking service run by principals $p_3$ and $p_4$ respectively, because those are long-running existing services. Principal $p_2$ is thus able to return a proof tree that contains the proofs from principal $p_3$ and $p_4$, and principal $p_1$ trusts that proof. The proof tree also satisfies the confidentiality policies of principals $p_2$, $p_3$ and $p_4$. Principal $p_4$ only returns the evaluation result of the query $?location(bob, airport)$ because it belongs to $trust(location(P, L)) = \{p_4\}$ defined by principal $p_1$.

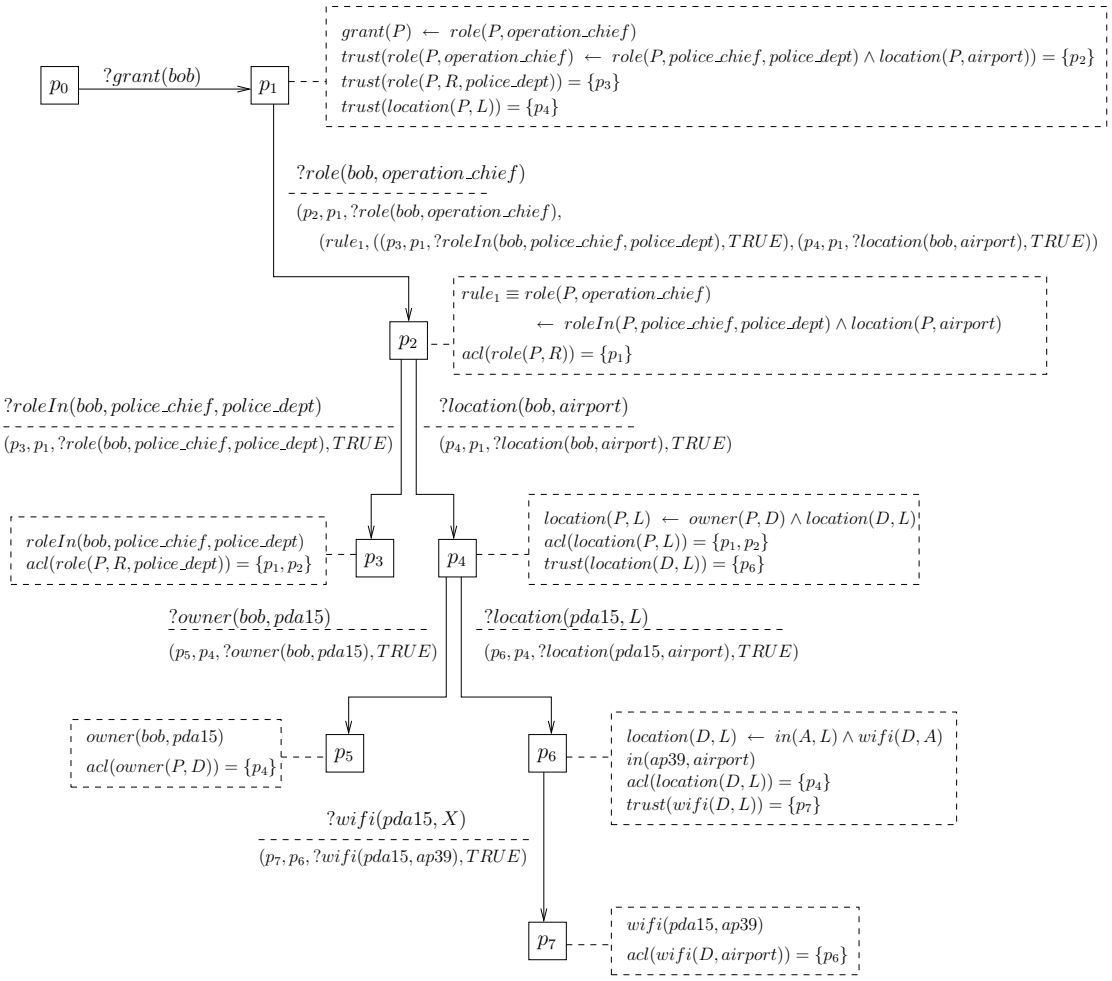



Figure 4.9: Example of an emergency response system. Principal $p_0$ is a first responder whose role is "operation_chief". Principal $p_1$ represents a surveillance camera image server. Principal $p_2$ is the role membership server of an incident management system (IMS). Principal $p_3$ is the role membership server of a police department. Principal $p_4$ represents a location-tracking service. The arrows represent the flow of queries among the principals. Each arrow is labeled with a query and a returned proof tree. The query is shown above the dashed line; the proof is shown below the line. Each principal's rules, facts and policies are shown in a dashed rectangle.

# Chapter 5

# Caching and revocation mechanism

In this chapter, we describe a caching and revocation mechanism that improves the performance of our system. Our caching mechanism supports both positive and negative query results and avoids issuing remote queries, which otherwise cause long latency due to cryptographic operations and the transmission of data over a network.

Our revocation mechanism ensures the freshness of cached information. To be sure to produce a query result only with information that is generated recently, we need an efficient mechanism that revokes obsolete cached information. Unlike existing revocation methods [84, 87, 135] in which only an issuer of a certificate can revoke it, our scheme must allow multiple hosts to revoke a given cached result because the result might depend on (contextual) facts maintained by different hosts. Therefore, we develop a revocation mechanism based on capabilities [110] in a distributed environment.

The rest of this chapter is organized as follows. Section 5.1 introduces our approach for capability-based revocation, and Section 5.2 describes the design of our caching and revocation mechanism. Section 5.3 describes the issue of race conditions among components in our system and presents our synchronization mechanism to solve that issue. Section 5.4 discusses additional requirements for supporting negative caching. Finally, Section 5.5 shows

a mechanism that keeps cached information fresh under the presence of an adversary who is capable of intercepting revocation messages.

## 5.1 Capability-based revocation

Our caching mechanism allows each principal to cache a fact derived from a constructed proof, and such a proof for a query could contain rules and facts published by multiple different principals. Therefore, the derived fact from that proof must be revoked if any information in the proof becomes invalid; that is, there might be multiple principals that are eligible to revoke a cached fact. We, therefore, developed a revocation mechanism based on capabilities [110] so that all the principals involved in constructing a proof may revoke the derived result from the proof.

We extend the representation of a proof in Section 4.1 to associate a query result or a rule in a proof with a capability, which is a large random number; that is, the nonterminals $rule$ and $value\_pair$ in Figure 4.1 additionally include the nonterminal $capability$, and their derivations are thus modified as follows.

$$
\begin{aligned}
< rule > \quad &::= \quad \text{`('} < head > \leftarrow < body >, < capability > \text{`)'} \\
< value\_pair > \quad &::= \quad \text{`('} < receiver >, < value >, < capability > \text{`)'} \\
< capability > \quad &::= \quad < number >
\end{aligned}
$$

The nonterminal $rule$ includes the $capability$ to associate a rule with a capability, and, similarly, the nonterminal $value\_pair$ includes the $capability$ to associate it with a fact. Thus, each node in a proof is associated with a capability generated by the principal who publishes the information in that node. Since a principal who publishes a proof encrypts the query result and capability together with a receiver principal's public key, the capability is a shared secret between the publisher and the receiver of the proof node. Therefore, the

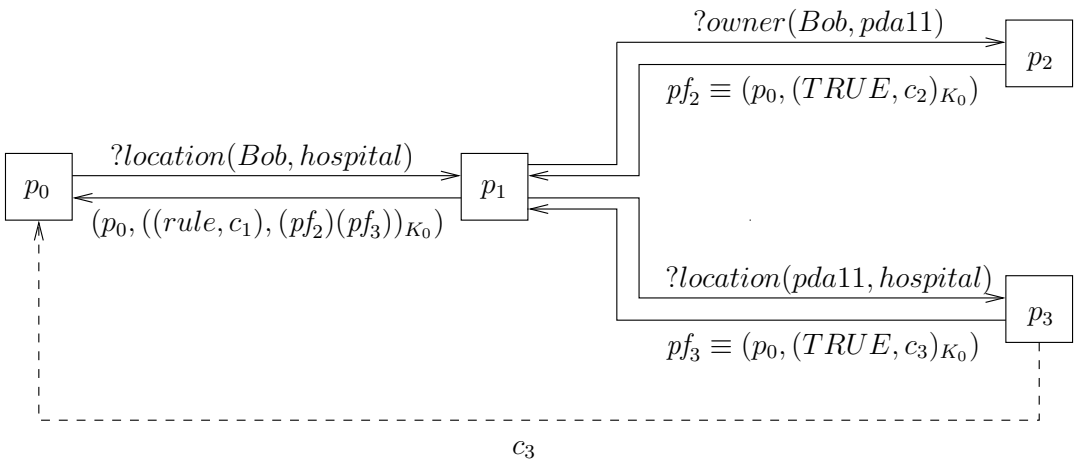



Figure 5.1: Capability-based revocation. The dashed line represents a revocation message sent by principal $p_3$.

principal who sent the proof can later revoke the fact or rule in the proof by sending the capability to the receiver principal. The sender principal of the revocation message does not need to authenticate itself to the receiver principal who maintains the cached information.

Figure 5.1 describes our revocation scheme in a distributed environment. A principal $p_0$ issues a query $?location(Bob, hospital)$, and a principal $p_1$ returns a proof that consists of a rule node produced by $p_1$ and two leaf nodes produced by $p_2$ and $p_3$ respectively. A principal $p_0$ caches the fact $location(Bob, hospital)$ derived from the received proof. Since principals $p_1$, $p_2$, and $p_3$ contribute to constructing the proof tree, they all should be eligible to revoke $p_0$'s cached fact. Therefore, each principal $p_i$ for $i = 1, 2, 3$ includes a capability $c_i$ into his produced node so that $p_i$ can revoke the proof later. A principal $p_0$ who caches the fact $location(Bob, hospital)$ associates it with the capabilities $c_1$, $c_2$, and $c_3$ obtained from the proof. Since principal $p_3$ chose $p_0$, not $p_1$, as a receiver of his proof $pf_3$, $p_3$ revokes his proof by sending a capability $c_3$ directly to principal $p_0$. Receiving that revocation message, a principal $p_0$ removes the cached fact associated with the capability $c_3$. Principals $p_1$ and $p_2$ could revoke the same cached fact in the same way. Our capability-based revocation does not involve any public-key operations, which are computationally expensive, because

a revocation message can be directly sent to a principal that maintains a cached fact. When our system constructs a proof tree responding to an authorization query, public-key encryptions are necessary to prevent intermediate principals between a sender and a receiver principal from reading the sender's query result. Furthermore, a revocation message does not need to be signed by a sender principal, because the capability is known only to the sender and the receiver of the revocation message. However, our revocation mechanism does need symmetric-key encryption for transmitting some revocation messages, since a revocation message contains an additional capability, which must be protected from eavesdroppers, when we revoke a negative query result, as we discuss in Section 5.4.

## 5.2 Design of a caching and revocation mechanism

Our revocation mechanism is based on a publisher-subscriber model; that is, a querier principal subscribes to a handler principal that handles his query, and the handler principal sends a revocation message when the query result becomes invalid. This process might occur recursively until all the cached facts that depend on the invalidated fact are revoked across the network. Figure 5.2 shows the structure of our caching and revocation mechanism and the message flow among the components when a cached fact is revoked. Each server consists of two components (an inference engine and a revocation handler) and four data structures (a subscribers list, a dependencies list, a subscription list, and a knowledge base). For brevity, we omit the query handler and query issuer that appear in Figure 3.2 from Figure 5.2. The inference engine is responsible for constructing a proof and caching query results obtained from other principals with the knowledge base and the subscription list. The engine also maintains information on other principals who issue a query to the engine with the subscribers and dependencies list so that the engine can revoke cached results in remote hosts. When principal $p_1$ receives a query $q_0$ from principal $p_0$, $p_1$'s inference

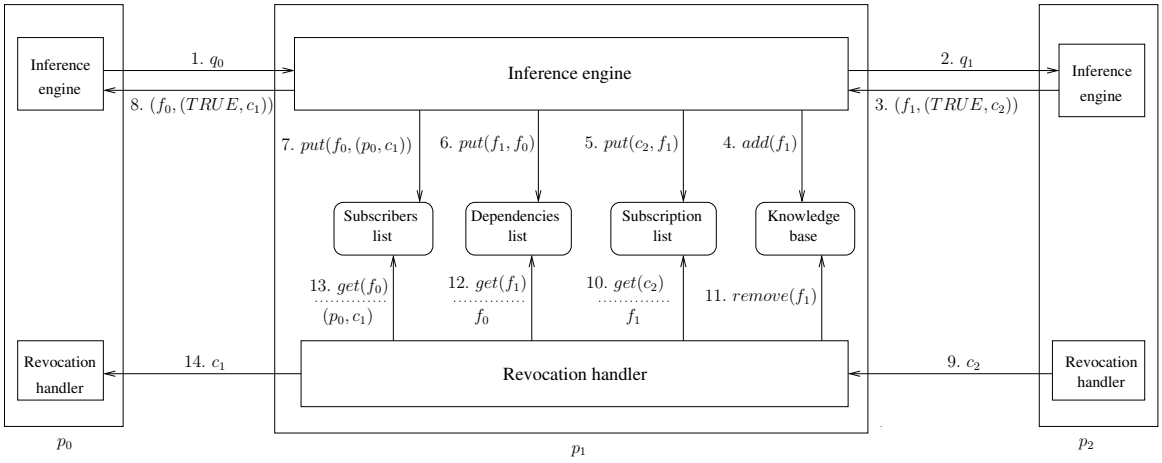



Figure 5.2: Structure of a caching and revocation mechanism. We omit the data structures from the servers of $p_0$ and $p_2$ for brevity. The number at the beginning of each message represents the sequence of the entire revocation process. The return value of a message is shown under a dotted line in the messages 10, 12, and 13.

engine constructs a proof tree for a fact $f_0$, which is unified with $q_0$, and issues a subsequent query $q_1$ to principal $p_2$, and $p_2$ returns a proof tree whose root node contains the unified fact $f_1$ and the pair of a query result $TRUE$ and a capability $c_2$. Note that facts $f_0$ and $f_1$ are identical to queries $q_0$ and $q_1$ respectively if those queries do not contain any variables. Principal $p_1$ stores $f_1$ as a fact into its knowledge base and also puts a key-value pair $(c_2, f_1)$ into the subscription list (a hash table). Notice that we use the same knowledge base to store cached results as well as local rules and facts. After constructing a proof tree for $f_0$, the engine stores the pair $(f_1, f_0)$, which represents $f_0$'s dependency on $f_1$, and a nested tuple $(f_0, (p_0, c_1))$ into the dependencies and subscribers list respectively. The nested tuple $(f_0, (p_0, c_1))$ expresses an if-then rule stating that a capability $c_1$ must be sent to principal $p_0$ if fact $f_0$ becomes invalid. The inference engine finishes handling query $q_0$ by returning a proof tree whose root node contains fact $f_0$ and the pair of a query result $TRUE$ and a capability $c_1$.

The revocation process occurs when principal $p_2$ sends a revocation message that contains a capability $c_2$. Principal $p_1$'s revocation handler receives the message, obtains a fact

to be revoked with capability $c_2$ from the subscription list, and removes fact $f_1$ from the knowledge base. Next, the revocation handler obtains fact $f_0$, which depends on $f_1$, from the dependencies list and then accesses the subscribers list to obtain a capability $c_1$ for revoking principal $p_0$'s cached fact $f_0$, and sends $c_1$ to $p_0$'s revocation handler. The same process is repeated on $p_0$'s server.

If a capability is a shared secret that is only used once, a capability does not need to be encrypted as we explain in this section. In Section 5.4 below, though, we add support for caching negative results and in that case we do need an encrypted channel for this message.

## 5.3 Synchronization mechanism

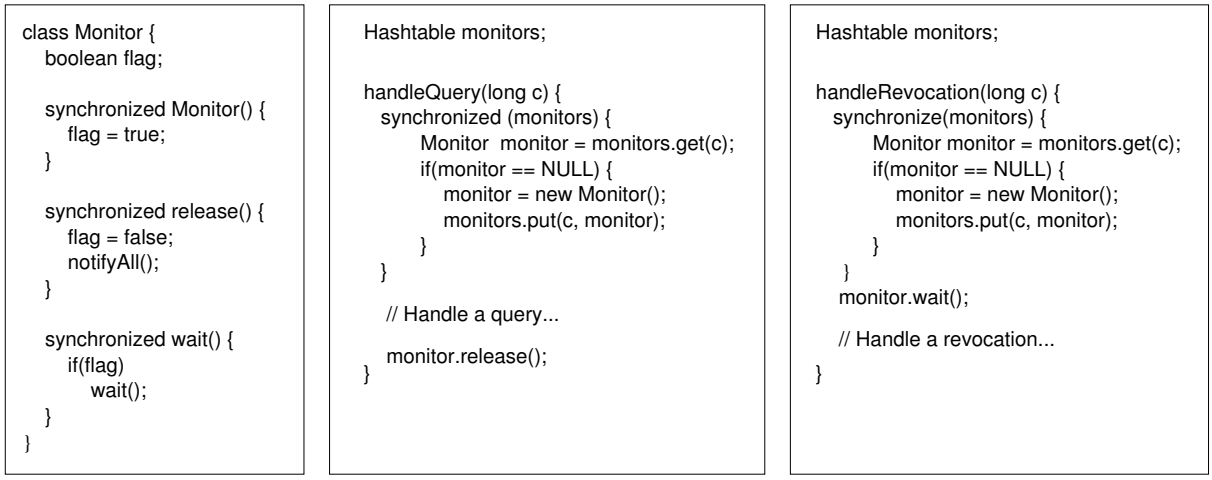
There is a race condition to be resolved between the inference engine and the revocation handler, because both modules access the four data structures in Figure 5.2. For example, it is possible that the revocation handler accesses the subscription list with a capability $c$ that revokes a fact $f$ before the inference engine writes the subscription information (i.e., $(c, f)$) to that list.

However, we cannot use a coarse mutual exclusion mechanism that allows the thread of the inference engine to block other threads' access to the data structure while processing a query, since a deadlock occurs when the engine issues a remote query that causes a closed cycle of subsequent queries by remote servers. For example, if a downstream server that receives a subsequent query issues a remote query back to the server of the inference engine, a new thread that is created to handle that query blocks because the inference engine on that server already obtains a lock on the data structures, which the new thread needs to access. Thus, the inference engine would wait for a reply for the remote query forever.

We, therefore, built a fine-grained synchronization mechanism that ensures that the engine that receives a proof-tree node with capability $c$ updates the data structures before

the revocation handler that receives a capability $c$ accesses them. To achieve this, our system creates a monitor object per capability and maintains a pair of a capability and a monitor object in a hash table. Figure 5.3 shows the pseudo code for the synchronization algorithm for the inference engine and the revocation handler. We use a monitor object in Figure 5.3a to block the revocation handler until the inference engine finishes handling a query. All the methods of this monitor object are an atomic operation, and thus more than two threads execute a thread concurrently. Since only the inference engine needs to block the other thread (i.e., the revocation handler), the lock of the monitor is set when it is created and is released when the inference engine calls the $release$ method after it finishes handling the query. The $notifyAll$ function in the $release$ method sends a signal to awaken all the threads that are waiting on the monitor object. The revocation handler thread that calls the $wait$ method is blocked until the lock is released.

Since a monitor object must be created dynamically either by the inference engine or the revocation handler, we need to synchronize access to a global hash table that maintains a set of monitor objects with a capability as a key. Figure 5.3b shows the $handleQuery$ method of the inference engine. The method first checks whether there exists a monitor object for a given capability. If the object does not exist, it creates a new monitor object and puts it into the hash table. The access to the hash table in the $synchronized$ block is mutually exclusive among all the threads; the code inside the synchronized block is an atomic operation on the hash table. The method then continues to process a query and releases the lock of the object after handling the query. Figure 5.3c shows the $handleRevocation$ method of the revocation handler. It also creates a monitor object for a given capability, if necessary. Then, it calls the $wait$ method to wait for the inference engine to finish handling the query. If the inference engine has already finished it, the $wait$ method does not block the thread of the revocation handler.



```
class Monitor {
   boolean flag;

   synchronized Monitor() {
      flag = true;
   }

   synchronized release() {
      flag = false;
      notifyAll();
   }

   synchronized wait() {
      if(flag)
         wait();
   }
}
```

(a) Monitor object

```
Hashtable monitors;

handleQuery(long c) {
  synchronized (monitors) {
       Monitor  monitor = monitors.get(c);
       if(monitor == NULL) {
          monitor = new Monitor();
          monitors.put(c, monitor);
       }
  }

   // Handle a query...

   monitor.release();
}
```

(b) the handleQuery method of the inference engine

```
Hashtable monitors;

handleRevocation(long c) {
  synchronize(monitors) {
       Monitor monitor = monitors.get(c);
       if(monitor == NULL) {
          monitor = new Monitor();
          monitors.put(c, monitor);
       }
   }
   monitor.wait();

   // Handle a revocation...
}
```

(c) the handleRevocation method of the revocation handler

Figure 5.3: Synchronization algorithm for the inference engine and the revocation handler. We omit parameters that are not relevant to our synchronization mechanism from Figure 5.3a and b.

## 5.4 Negative caching

Our system also supports caching negative facts (i.e., facts that are false), because a principal cannot return a negative result when he does not find any matched fact for the query in the local knowledge base; another principal might have a fact that matches with the query. To make a negative decision locally, a principal must cache a negative result after the attempt to obtain the queried fact from remote principals fails.

To support negative caching, each principal maintains the same set of data structures in Figure 5.2; that is, each server maintains another knowledge base that stores negative facts. The semantics of a negative revocation are different from that of positive caching; that is, when a cached negative fact is revoked, that fact can be cached as a positive fact. (On the other hand, to revoke a positive fact does not necessary mean that the revoked fact is no longer true; there might be another proof that derives the fact. Note that if a host that maintains a positive fact has first-hand knowledge about the validity of that fact without checking with other hosts, that host could convert the revoked positive fact into a negative

cached fact.)

When a negative fact is revoked, we must find an entry $(c, f)$ in the negative subscription list, where $c$ is a capability and $f$ is the revoked fact, and move it to the subscription list for positive cached facts. However, we cannot use the same capability $c$ for the entry in the positive list, because it might cause inconsistency about the subscription information between the sender and receiver of a revocation message in the case where the revocation message is lost. For example, suppose that we use the same capability for a switched positive cached fact. When a principal who sends another principal a revocation message for a negative cached fact, the sender principal moves the corresponding subscription information from the subscribers list of negative facts to that of positive facts. However, if the receiver principal does not receive the message because of a network failure, the receiver principal continues to maintain the negative cached fact, which is supposed to be revoked. When the sender principal later sends a revocation message that revokes the switched positive cached fact, the receiver principal revokes the negative cached fact instead. Thus, the inconsistency about the cached information occurs.

Therefore, a revocation message for a negative cached result needs to contain a new capability to revoke the switched positive cached result. Since the new capability must be a shared secret between a sender and a receiver of the revocation message, we need to encrypt the message with a shared key between those two parties. However, to establish a symmetric secure channel for all the pairs of two principals that participate in our system requires $n^2$ symmetric keys, where $n$ is the number of participating principals. To avoid this key-management challenge, our system encrypts a revocation message with the same randomly generated symmetric key that is used to encrypt the proof node that contains the cached result as we describe in Section 4.4; that is, each server records the capabilities in a received proof with a symmetric key that is used to decrypt that proof. Suppose that the proof contains a node with a capability $c_n$ and was encrypted with a symmetric key $K$ when

the server receives it. A server stores a $(c_n, K)$ pair in a hash table to handle a revocation message $(c_n, (c_n, c_p)_K)$ where $c_n$ is a capability that revokes a current negative result, $c_p$ is a capability that revokes the switched positive result in the future, and $K$ is the symmetric key associated with $c_n$. When a server receives this message, it first obtains a symmetric key $K$ corresponding to the capability $c_n$ in the message from the hash table, and decrypts $(c_n, c_p)_K$ with that key. If the first element of the decrypted tuple is same as the capability in the first field of the revocation message, the server considers that revocation message valid and revokes the corresponding fact. We continue to use the key $K$ if we later revoke the fact that corresponds to the capability $c_p$. In a way, the symmetric key $K$ is a real capability and the capability $c_n$ as an indirect reference to $K$.

The dependencies list for negative facts maintains dependencies between a fact in a query and facts that were tried but, being false, were not used to construct a proof for the query; that is, it might be possible to construct a proof for the query if one of those facts exists in the positive knowledge base. When a server receives notice of revocation for a negative cached fact, it tries again to build proofs that were not constructed because of that negative cached fact. We only use facts in the local knowledge base to construct those proofs, because issuing remote queries could make latency for revocation significantly longer. On the other hand, the latency for processing a query only with a local knowledge base is negligible compared with that for transmitting data over a network or performing cryptographic operations as we see in Chapter 6.

## 5.5 Timeliness of cached information

Our system must ensure that all the cached facts meet a given timeliness condition; all the timestamps associated with cached facts must be within a given interval between the current time and a recent past time. To simply keep the latest messages does not guarantee the

freshness of the cached facts because some hosts might crash or an adversary might intercept revocation messages so a server would make an incorrect decision based on obsolete cached information.

We, therefore, develop a mechanism that ensures the freshness of cached positive and negative facts obtained from remote servers. The updater thread on each server periodically sends each subscriber in a subscribers list a message that updates the timestamp of a cached fact by sending the capability with a new timestamp. We assume that all the server clocks are approximately synchronized. Since the server sends the same capability to refresh the same cached fact repeatedly, the updater thread encrypts the message with the same symmetric key that would be used to send a revocation message for revoking that cached fact. The watcher thread on another server receives that message and updates the timestamp of the fact in a subscription list. The watcher thread must synchronize with the inference engine using the same synchronization method we describe in Section 5.3. If the watcher thread finds any subscription with an old timestamp (possibly because an adversary intercepts revocation messages), it discards that subscription and initiates the revocation process described in Section 5.2.

# Chapter 6

# Experimental results

In this chapter, we present the experimental results for our system. Many context-aware applications, such as an emergency-response system in which responders continuously access information on an incident over a period from a few hours to several days, need to keep track of a user's privileges continuously. Therefore, our focus is to show that our caching mechanism significantly improves amortized performance of our system, while ensuring the freshness of cached information.

We used a 27-node cluster connected with a Gigabit Ethernet. Each node had two 2.8GHz Intel XEONs and 4GB RAM, and ran RedHat Linux 9 and Sun Microsystem's Java runtime (v1.5.0-hotspot). Our system had approximately 12,000 lines of Java code, extending the Prolog engine XProlog [114]. We used the Java Cryptographic Extension (JCE) framework, which provides RSA [99] and Triple-DES (TDES) [33] cryptographic operations.

The algorithm of the RSA operations in JCE conforms to the encryption scheme RSAES-PKCS1-v1_5 in Public-key Cryptography Standard (PKCS) #1 [93]. We used a 1024-bit public key whose public exponent is fixed to 65537 in our experiments. PKCS #1 uses the EME-PKCS1-v1_5 padding method [93] to encrypt or sign a message. The RSA sign-

ing operation uses MD5 [98] to compute the hash value of a message. We used Outer-CBC TDES in EDE mode [33] to perform symmetric key operations. The length of our DES keys was 192 bits, and the padding operation in TDES operations conforms to RFC 1423 [96]. Note that our system is not particularly optimized for performance and that all the code (including the cryptography library) is written in Java. Nevertheless, our experimental results show that our system with the caching mechanism is usable without being turned for performance. It is possible to improve the performance of our system by using a more efficient cryptography library such the OpenSSL toolkit [90].

We assume that each principal obtains the public keys of the principals when it defines its integrity policies and establish a session key with each of those principals before handling queries; that is, the performance cost of PKI is not included in our experiments.

In Section 6.1, we present a detailed study about the performance of our system without our caching mechanism. In Section 6.2, we compare the latency for handling a query with and without our caching mechanism. Finally, in Section 6.3, we show the results of the experiments that measured the latency for revoking a cached fact.

## 6.1 Analysis of performance overhead

We first analyze the performance overhead of our system without the caching mechanism. Our first experiment measured the latency of the system with two hosts that did not use the caching mechanism. One host maintains a rule $a0(P) \leftarrow a00(P)$, and the other host maintains a fact $a00(bob)$. When the former host receives a query $?a0(bob)$, it issues a remote query $?a00(bob)$ to the other host.

We measured the wall-clock latency for handling a query and also the latency of each cryptographic operation in that process. The measurement is iterated one hundred times, and we report the average and standard deviation of the measurements. Table 6.1 shows

the results. Note that some of the operations in Table 6.1 involve multiple cryptographic operations: The "TDES encryption on a returning proof" in Table 6.1 involves three TDES operations for encrypting a proof node, a proof tree, and a proof itself recursively. Similarly, the "TDES decryption on a received proof" involves three TDES operations. the "RSA encryption on DES keys" involves two encryption operations: one is an encryption of a DES key for a proof node and the other is that for a proof tree. Similarly, RSA decryption on DES keys involves two decryption operations. Also, note that our current implementation encodes a proof as a set of Java objects that correspond to the string representation of the proof in Section 4.1. The size of the Java objects was much larger than that of the corresponding string. For example, the size of strings for a query and a proof in this experiment was less than 124 bytes.

As we see in Table 6.1, public key operations consumed most of the processing time. On host 0, the "RSA decryption on a DES key" from host 1 used 53% of the local processing time. The "TDES decryption on a proof" also used another 22% of the time. On host 1, the "RSA encryption on DES keys" used 22% of the local processing time, and signing a proof with a RSA private key used another 17%. These results indicate that our caching scheme should improve the performance because a successful cache hit avoids all of these public key operations.

Table 6.2 compares the basic cost of each cryptographic operation per operation and per byte. When we compare per-byte operations, TDES operations were a few orders of magnitude faster than RSA operations. The TDES operations on a proof was an order of magnitude faster than that on a query. One possible reason is that the cost for computing a key schedule in a TDES operation becomes relatively larger portion of the total cost, as the data to be encrypted or decrypted gets smaller. It is not clear why the TDES encryption of a query is slightly faster than the TDES decryption of the query. The RSA encryption with a public key was not significantly faster than RSA encryption with a private key due

| | host 0 | | | host 1 | | |
|---|---|---|---|---|---|---|
| | latency | std. dev. | ratio | latency | std.dev | ratio |
| Total latency | 138.1 | 10.5 | | 85.2 | 5.1 | |
| Issue remote queries | 87.9 | 5.5 | | 0.0 | 0.0 | |
| Local computation | 50.2 | 6.8 | 1.00 | 85.2 | 5.1 | 1.00 |
| Read objects from a remote host | 0.0 | 0.0 | 0.00 | 41.2 | 0.7 | 0.48 |
| TDES decryption on a received query | 0.0 | 0.0 | 0.00 | 2.2 | 0.4 | 0.03 |
| TDES encryption on a returning proof | 0.0 | 0.0 | 0.00 | 10.9 | 3.1 | 0.12 |
| TDES decryption on a received proof | 10.9 | 3.1 | 0.22 | 0.0 | 0.0 | 0.00 |
| TDES encryption on an issued query | 1.2 | 0.9 | 0.02 | 0.0 | 0.0 | 0.00 |
| RSA decryption on DES keys | 26.6 | 1.2 | 0.53 | 0.0 | 0.0 | 0.00 |
| RSA encryption on DES keys | 0.0 | 0.0 | 0.00 | 18.7 | 2.0 | 0.22 |
| Create a RSA signature for a proof | 0.0 | 0.0 | 0.00 | 14.4 | 1.5 | 0.17 |
| Verify a RSA signatures for a proof | 2.2 | 0.9 | 0.04 | 0.0 | 0.0 | 0.00 |

Table 6.1: Average latency for processing a query with two hosts without caching capability. The latency is measured in milliseconds. The *std. dev* columns show the standard deviation of each measurement. The *ratio* columns show the ratio of the latency of each primitive operation compared with the total local processing time.

to the inefficient implementation of Java cryptography library (JCE). To initialize a Java Cipher object that provides a method for performing a public-key operation took much of the time, and the operation of modulo exponentiation in a public-key operation was indeed an order of magnitude faster than that in a private-key operation. In RSA encryption on a DES key, to initialize a Cipher object takes 8.3 milliseconds and modulo exponentiation takes 1.0 milliseconds. On the other hand, in RSA decryption on a DES key, the latency for initializing a Cipher object is negligible and modular exponentiation takes 9.3 milliseconds. An operation for signing a proof is slightly slower than an operation for RSA decryption because the former involves the computation of the MD5 hash value of a proof.

We next study the performance impact of changing the integrity policies that are sent to a handler principal along with a query. We compared the latency for handling the query in the example of the emergency response system in Figure 3.12 and Figure 4.9. In both cases, a set of principals $p_i$ (for $i = 1, \ldots, 7$) handled the same query. However, the query that principal $p_1$ issued to $p_2$ was attached with different integrity policies in those two cases.

| | byte length | latency | latency per byte |
|---|---|---|---|
| TDES encryption on a query | 723 | 1.2 | 0.0017 |
| TDES decryption on a query | 723 | 2.2 | 0.003 |
| TDES encryption on a proof | 34184 | 11.3 | 0.00033 |
| TDES decryption on a proof | 34184 | 11.3 | 0.00033 |
| RSA encryption on a DES key | 128 | 9.3 | 0.07 |
| RSA decryption on a DES key | 128 | 13.3 | 0.1 |
| Create a RSA signature for a proof | 128 | 14.4 | 0.11 |
| Verify a RSA signatures for a proof | 128 | 2.2 | 0.017 |

Table 6.2: Basic costs of cryptographic operations. The latency is measured in milliseconds. The first column shows the byte length of data on which an operation is performed. A single TDES encryption and decryption on a proof involves the encryption of a proof node, a proof tree, and a proof. The byte lengths of those objects were 6507 bytes, 13579 bytes, and 14098 bytes respectively, and their total length was 34184 bytes.

In Figure 3.12, all the principals that issue a query trust a handler principal in terms of the integrity of a query result. For example, responding to a query from principal $p_1$, principal $p_2$ returns a proof that contains only a root node. In Figure 4.9, however, principal $p_1$ only trusts the integrity of $p_2$'s rule that is used to handle $p_1$'s query. Thus, principal $p_2$ returns a proof that consists of a root node that contains $p_2$'s rule and two leaf fact nodes produced by principals $p_3$ and $p_4$ respectively. The other principals' integrity policies are same as in Figure 3.12.

We measured wall-clock latency for handling a query one hundred times and report the average of the measurements. We also measured the latency of each primitive operation on each server as we did in the previous experiment. Table 6.3 and Table 6.4 show the results for the cases in Figure 3.12 and Figure 4.9, respectively. The standard deviation of the total latency in the cases of Figure 3.12 and Figure 4.9 was 57.4 and 65.4 milliseconds respectively.

In this experiment, each principal $p_i$ ran on a separate host $i$. We also show in Figure 6.1 the comparison of the local processing time of each host in these two cases. The total latency of host $0$ in Table 6.4 was 105 milliseconds longer than that in Table 6.3, largely due

to two cryptographic operations performed on host $1$. One was the RSA decryption on DES keys. Table 6.4 shows that the latency for that operation on host $1$ was 78.9 milliseconds, while the corresponding latency in Table 6.3 is only 26.6 milliseconds. We see that the latency for RSA decryption on TDES keys was approximately proportional to the number of DES keys contained in a proof; that is, host $1$ in Figure 4.9 decrypted three TDES keys in a proof, while host $1$ in Figure 3.12 decrypted only one DES key.

The other operation that contributed to the increase of the latency was the decryption of a received proof with a DES key. The latency for that operation in Table 6.4 is 50.5 milliseconds, while the corresponding latency in Table 6.3 was 10.1 milliseconds. The latency increased because the proof in the former case was larger than that in the latter case; that is the proof in the former case contained three nodes, while the latter only contained a single root node.

In contrast, the local processing time at host $2$ in Table 6.4 was shorter than that in Table 6.3, because principal $p_2$ in Figure 4.9 forwarded the proofs from principals $p_3$ and $p_4$ to principal $p_1$ by including them into the proof returned to $p_1$ without decrypting those proofs. Therefore, there was no operation for RSA decryption on DES keys on host $2$ in Figure 4.9, as we see in Table 6.4. Principal $p_2$ did not have to check the digital signatures of the proofs from $p_3$ and $p_4$. However, when comparing Table 6.4 to Table 6.3, we see that the increase of the latency in host $1$ was larger than the decrease of the latency in host $2$ as we see in Figure 6.1. A large proof that is transmitted across hosts has an effect of increasing processing time of a particular host that receives that proof, because the receiving host needs to check the integrity of the large proof by traversing each node. Therefore, the imbalance of workload among hosts is likely to increase total latency for processing a query. The results for hosts $3, 4, 5, 6$, and $7$ are almost identical in both cases, since their operations are same in both cases except for the receiver principals of principal $p_4$'s result.

| | host 0 | | host 1 | | host 2 | | host 3 | |
|---|---|---|---|---|---|---|---|---|
| | latency | ratio | latency | ratio | latency | ratio | latency | ratio |
| Total latency | 950.0 | | 898.2 | | 762.7 | | 85.9 | |
| Issue remote queries | 901.3 | | 766.4 | | 584.0 | | 0.0 | |
| Local computation | 48.7 | 1.00 | 131.9 | 1.00 | 178.7 | 1.00 | 85.9 | 1.00 |
| Read objects from remote hosts | 0.0 | 0.00 | 41.2 | 0.31 | 41.3 | 0.23 | 41.3 | 0.48 |
| TDES decryption on received queries | 0.0 | 0.00 | 1.2 | 0.01 | 1.7 | 0.01 | 2.5 | 0.03 |
| TDES encryption on returning proofs | 0.0 | 0.00 | 3.5 | 0.03 | 3.2 | 0.02 | 3.4 | 0.04 |
| TDES decryption on received proofs | 9.6 | 0.20 | 10.1 | 0.08 | 19.5 | 0.11 | 0.0 | 0.00 |
| TDES encryption on issued queries | 1.2 | 0.02 | 0.8 | 0.01 | 2.1 | 0.01 | 0.0 | 0.00 |
| RSA decryption on DES keys | 26.7 | 0.55 | 26.6 | 0.20 | 53.0 | 0.30 | 0.0 | 0.00 |
| RSA encryption on DES keys | 0.0 | 0.00 | 18.2 | 0.14 | 18.0 | 0.10 | 18.9 | 0.22 |
| Create RSA signatures for proofs | 0.0 | 0.00 | 14.2 | 0.11 | 14.1 | 0.08 | 14.7 | 0.17 |
| Verify RSA signatures for proofs | 2.2 | 0.05 | 2.2 | 0.02 | 4.2 | 0.02 | 0.0 | 0.00 |
| | host 4 | | host 5 | | host 6 | | host 7 | |
| | latency | ratio | latency | ratio | latency | ratio | latency | ratio |
| Total latency | 492.2 | | 85.8 | | 221.6 | | 85.3 | |
| Issue remote queries | 313.8 | | 0.0 | | 88.6 | | 0.0 | |
| Local computation | 178.4 | 1.00 | 85.8 | 1.00 | 133.0 | 1.00 | 85.3 | 1.00 |
| Read objects from remote hosts | 41.1 | 0.23 | 41.4 | 0.48 | 41.5 | 0.31 | 41.4 | 0.49 |
| TDES decryption on received queries | 1.4 | 0.01 | 2.2 | 0.03 | 1.6 | 0.01 | 2.1 | 0.02 |
| TDES encryption for returning proofs | 3.3 | 0.02 | 3.3 | 0.04 | 3.7 | 0.03 | 3.1 | 0.04 |
| TDES decryption for received proofs | 20.6 | 0.12 | 0.0 | 0.00 | 10.2 | 0.08 | 0.0 | 0.00 |
| TDES encryption on issued queries | 1.9 | 0.01 | 0.0 | 0.00 | 0.8 | 0.01 | 0.0 | 0.00 |
| RSA decryption on DES keys | 53.4 | 0.30 | 0.0 | 0.00 | 26.3 | 0.20 | 0.0 | 0.00 |
| RSA encryption on DES keys | 18.1 | 0.10 | 19.2 | 0.22 | 18.4 | 0.14 | 18.9 | 0.22 |
| Create RSA signatures for proofs | 14.1 | 0.08 | 14.7 | 0.17 | 14.3 | 0.11 | 14.7 | 0.17 |
| Verify RSA signatures for proofs | 4.3 | 0.02 | 0.0 | 0.00 | 2.1 | 0.02 | 0.0 | 0.00 |

Table 6.3: Average latency for processing a query in Figure 3.12. Every principal who issues a query trusts a principal that handles that query in terms of the integrity of a fact in that query.

| | host 0 | | host 1 | | host 2 | | host 3 | |
|---|---|---|---|---|---|---|---|---|
| | latency | ratio | latency | ratio | latency | ratio | latency | ratio |
| Total latency | 1055.0 | | 1003.5 | | 725.2 | | 85.6 | |
| Issue remote queries | 1006.1 | | 752.2 | | 584.3 | | 0.0 | |
| Local computation | 48.9 | 1.00 | 251.3 | 1.00 | 141.0 | 1.00 | 85.6 | 1.00 |
| Read objects from remote hosts | 0.0 | 0.00 | 40.9 | 0.16 | 41.1 | 0.29 | 41.1 | 0.48 |
| TDES decryption on received queries | 0.0 | 0.00 | 1.6 | 0.01 | 2.9 | 0.02 | 2.4 | 0.03 |
| TDES encryption on returning proofs | 0.0 | 0.00 | 3.4 | 0.01 | 17.6 | 0.12 | 3.3 | 0.04 |
| TDES decryption on received proofs | 9.7 | 0.20 | 50.5 | 0.20 | 8.5 | 0.06 | 0.0 | 0.00 |
| TDES encryption on issued queries | 1.3 | 0.03 | 0.9 | 0.00 | 1.9 | 0.01 | 0.0 | 0.00 |
| RSA decryption on DES keys | 26.6 | 0.54 | 78.9 | 0.31 | 0.0 | 0.00 | 0.0 | 0.00 |
| RSA encryption on DES keys | 0.0 | 0.00 | 18.0 | 0.07 | 26.0 | 0.18 | 18.9 | 0.22 |
| Create RSA signatures for proofs | 0.0 | 0.00 | 14.2 | 0.06 | 15.9 | 0.11 | 14.5 | 0.17 |
| Verify RSA signatures for proofs | 2.2 | 0.04 | 7.0 | 0.03 | 0.0 | 0.00 | 0.0 | 0.00 |
| | host 4 | | host 5 | | host 6 | | host 7 | |
| | latency | ratio | latency | ratio | latency | ratio | latency | ratio |
| Total latency | 492.2 | | 85.7 | | 221.1 | | 85.5 | |
| Issue remote queries | 313.2 | | 0.0 | | 88.8 | | 0.0 | |
| Local computation | 178.8 | 1.00 | 85.7 | 1.00 | 132.3 | 1.00 | 85.5 | 1.00 |
| Read objects from remote hosts | 41.1 | 0.23 | 41.3 | 0.48 | 41.3 | 0.31 | 41.5 | 0.49 |
| TDES decryption on received queries | 1.6 | 0.01 | 2.5 | 0.03 | 1.7 | 0.01 | 2.2 | 0.03 |
| TDES encryption for returning proofs | 3.3 | 0.02 | 3.1 | 0.04 | 3.6 | 0.03 | 3.0 | 0.04 |
| TDES decryption for received proofs | 20.1 | 0.11 | 0.0 | 0.00 | 9.8 | 0.07 | 0.0 | 0.00 |
| TDES encryption on issued queries | 1.8 | 0.01 | 0.0 | 0.00 | 0.8 | 0.01 | 0.0 | 0.00 |
| RSA decryption on DES keys | 52.3 | 0.29 | 0.0 | 0.00 | 26.3 | 0.20 | 0.0 | 0.00 |
| RSA encryption on DES keys | 18.6 | 0.10 | 18.8 | 0.22 | 18.7 | 0.14 | 18.9 | 0.22 |
| Create RSA signatures for proofs | 14.6 | 0.08 | 14.4 | 0.17 | 14.3 | 0.11 | 14.7 | 0.17 |
| Verify RSA signatures for proofs | 4.1 | 0.02 | 0.0 | 0.00 | 2.2 | 0.02 | 0.0 | 0.00 |

Table 6.4: Average latency for processing a query in Figure 4.9. Principal $p_1$ who issues a query to $p_2$ trusts only the integrity of $p_2$'s rule, which is used to handle $p_1$'s query.

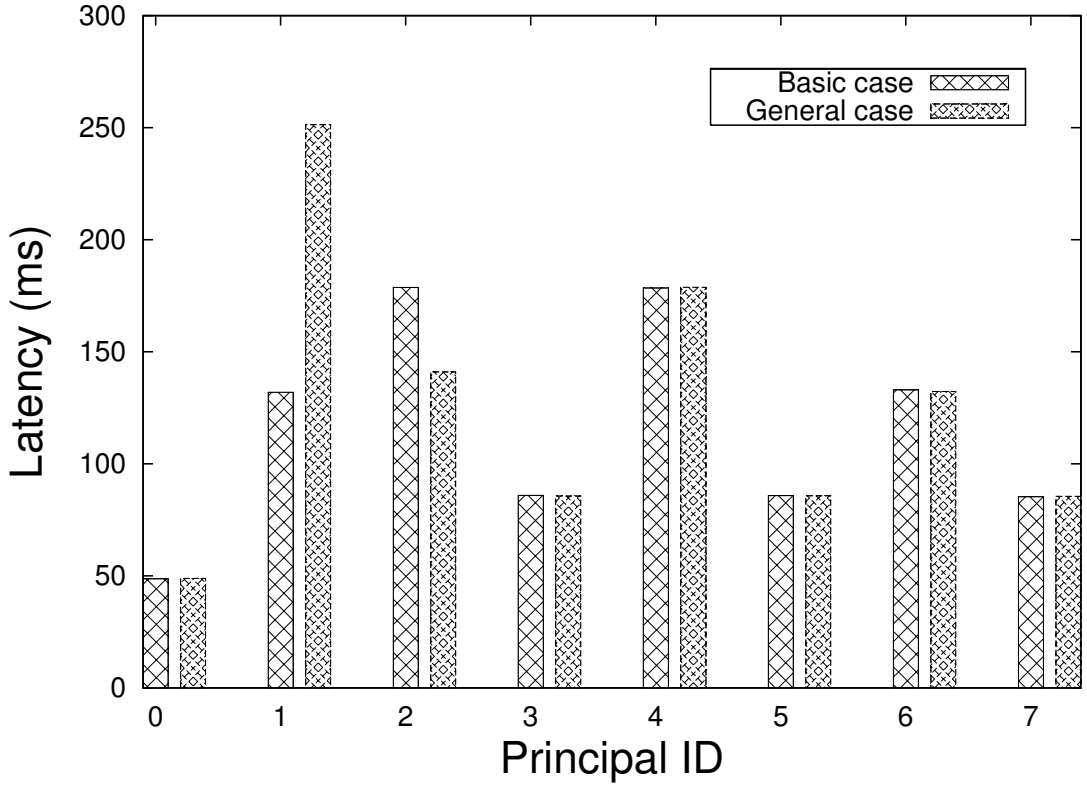



Figure 6.1: Comparison of local processing time at each principal in Figure 3.12 (basic case) and Figure 4.9 (general case).

## 6.2 Latency for handling queries

We next measured the latency for handling a query with different-size proof trees to evaluate the scalability of our caching scheme. We performed our experiments with 27 servers run by different principals; those servers could correspond to 27 different agencies in the emergency-response system. Our test program generated authorization, confidentiality, and integrity policies for those principals so that our system constructs a proof tree of a given size (i.e., the number of nodes in the proof tree) for a given query. Each query takes the form of $?grant(P, R)$ where $P$ is a principal and $R$ is a resource. The body of each rule takes the form of $a_0(c_0), \ldots, a_{n-1}(c_{n-1})$ where $a_i$ for $i = 0$ to $n - 1$ is a predicate symbol and $c_i$ for $i = 0$ to $n - 1$ is a constant. The size of the domain of predicate symbols is 1,000, and the size of the domain of constants is 20. There are possibly 20,000 different atoms in authorization policies that our test program generates, and it is, therefore, unlikely that a cache hit occurs when a query is evaluated for the first time. The generated policies ensures that every querier principal satisfied the confidentiality policies of a handler principal to which it issued a query, and the handler principal satisfied the querier principal's integrity policies. Those policies are independent of any particular application; that is, our test program chose the topology of a proof tree randomly. However, we conducted our experiment up to a proof tree with 50 nodes, which we believe is significantly larger than that in most applications, and, therefore, our results should provide guidelines about the worst-case latency of those applications. We prepared the facts and rules to allow ten different proof trees of each size, and in the experiment a given host issued a sequence of ten different queries of that size.

Our latency measurements also include the performance overhead for handling revocation messages. While measuring latency for handling queries 100 times, our test driver program updated all the facts in the knowledge bases dynamically. We assumed the extreme

case that all the facts in each proof tree are dynamic contextual facts, and updated every fact 20 times per second during the experiments. We believe that this update frequency is much faster than most context-aware applications need. The test driver updated the knowledge bases so that half of the queries in our experiments succeeded in constructing a proof tree, because the average latency could vary significantly, depending on how many of the queries succeeded in constructing a proof. (The latency for building a proof tree is longer than that for failing to build the tree, because the inference engine stops constructing a proof tree when it fails to resolve its current goal.)

Figure 6.2(a) compares query-handling latency under five different conditions; each data point is an average of 100 runs. In the *No caching, with RSA* case, each server did not cache any results obtained from other servers and used public-key operations for encrypting DES keys and signing proof trees. *No caching, with TDES* is same as the first case, except that every pair of the principals shared a secret DES key and used it to attach a message authentication code (MAC) using Keyed-Hashing for Message Authentication (HMAC-MD5) hashing algorithm [12, 65] to authenticate an encrypted proof[1]. We included this case to show that to use symmetric key operations instead of public-key operations (assuming that all the pair of the principals share a secret key) does not solve the problem of the long latency for handling a query. In the *Warm caching* case, every server cached results from other servers and we used only the latency data after the first round of ten different queries to compute average latency. In the *Cold caching* case, every server cached results from other servers and all the latency data including that of the initial round of queries were used to compute average latency. In the *Local processing* case, all the rules and facts were stored in a single server. Therefore, there was no remote query involved, and no encryption.

The two cases without caching show significantly longer latency than the other three

[1]A collision attack on MD5 is not possible because a malicious third party does not know a secret key shared by a sender principal and a receiver principal.

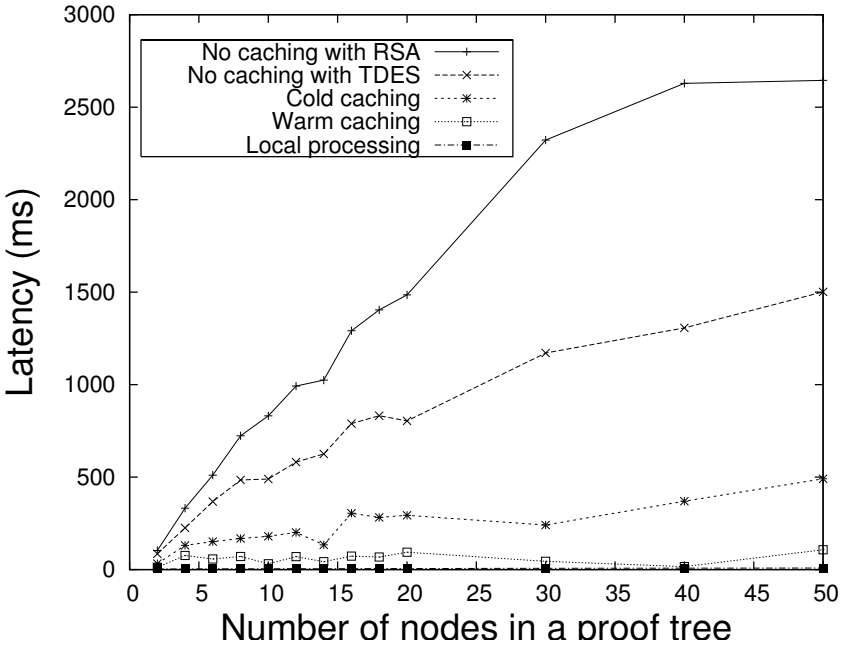
cases, although using MD5 and TDES operations rather than RSA reduced the latency 15 – 50%. The latency grew longer than 500 ms when a proof tree contained more than ten nodes. Figure 6.2(b) shows the same latency results for the other three cases, omitting the case of no caching. The latency of the cold caching case was 5 to 20 times longer than that of the warm caching, because the initial queries require the whole process of constructing a proof tree as in the case of no caching. The latency of the warm caching case was 2 to 15 times higher than that of the local processing case. The reason for the longer latency is a cache miss due to a revocation of a positive cached fact. However, the latency of the warm caching case was 3 to 23 times faster than the cold caching case, and thus we could improve the performance by prefetching query results in advance.

Although we conducted the experiments in a cluster with low-latency connections, our implementation to some extent simulates a low-latency network by encoding a proof as a set of Java objects, which is much larger than the corresponding string representation. Also, our caching mechanism could improve the performance of the system even more drastically in a wireless environment with low bandwidth and high data-loss ratio, because to handle a query with local cache is a common case for a long-running continuous query. The mechanism in Section 5.5 refreshes cached information periodically, and thus prevents making false positive decisions due to a disconnected wireless network.

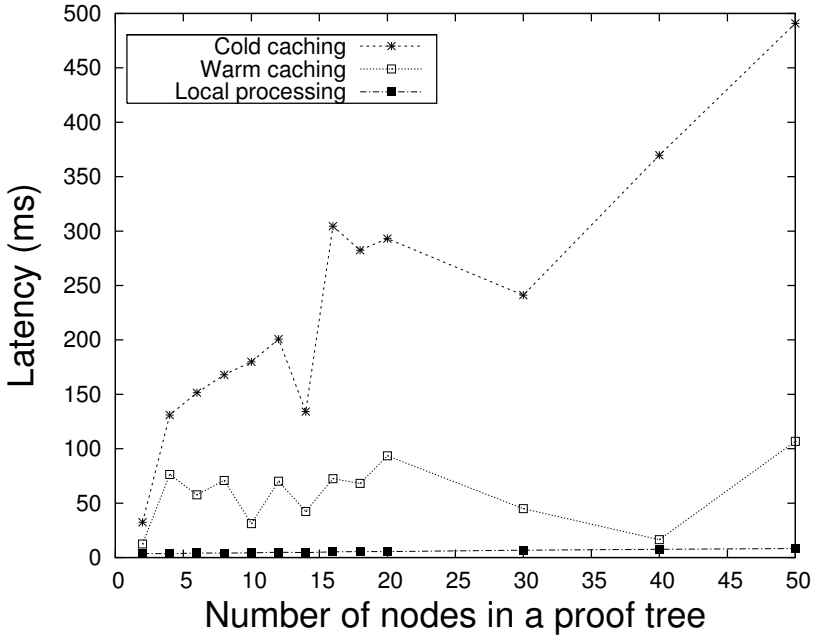
## 6.3 Latency for revoking cached facts

We measured the latency for revoking cached facts with another experiment. We used linear proof trees of various depths to measure the latency between the moment the test driver sent an event that updated a fact in the knowledge base and the moment that the test driver received the notification of a revoked cached fact from the root server that handles queries from the test driver. We conducted the same experiment 100 times and report the

average of the measurements. Figure 6.3 shows the latency for revoking cached facts with four different frequencies for updating the knowledge bases. The results show that the latency increased linearly as the depth of a proof tree grows. The latency slightly increased as the period for publishing an event decreased. The system handled 100 events per second with the latency less than 600 ms and a proof tree of depth 10.




(a) Five cases including no-caching cases.




(b) Three cases without no-caching cases.

Figure 6.2: Latency for handling queries.

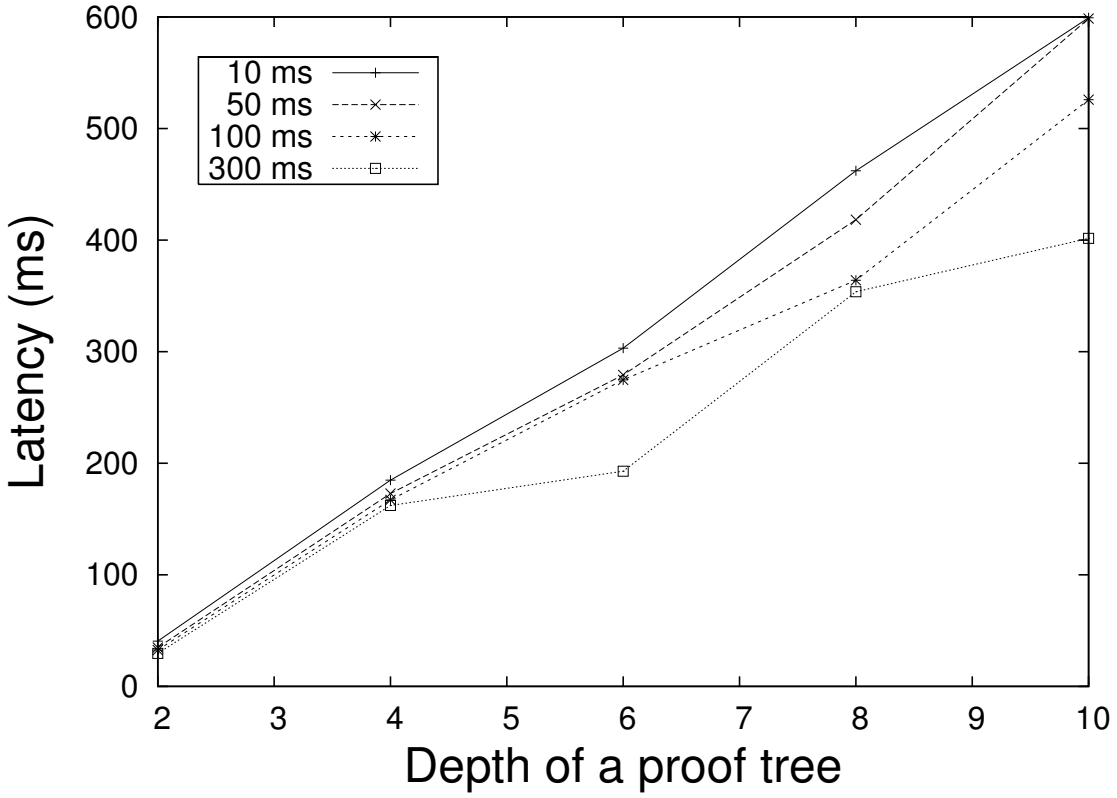



Figure 6.3: Average latency for revoking cached facts. Each curve represents a different period between fact updates in the knowledge bases, in milliseconds.

# Chapter 7

# Related work

In this chapter, we first cover existing context-sensitive authorization systems that consider context information to make an authorization decision. Those systems provide a rule-based authorization language that allows a policy maker to refer to context information in a central server. In Section 7.2, we cover distributed authorization systems that fetch information from remote servers while making an authorization decision. In Section 7.3, we cover trust-management systems that support an authorization language that makes it possible to express trust relations among principals. In Section 7.4, we cover trust negotiation systems, which are considered to be special cases of the distributed authorization systems where a client and a server exchange credentials to establish trust. The trust negotiation systems are similar to our systems in that they provide explicit mechanisms for protecting confidentiality policies. We clarify that those systems have different objectives and thus provide different mechanisms. In Section 7.5, we cover several variants of two-party secure function evaluation (SFE) that protects a client's private credentials from a server in a trust management system. Finally, in Section 7.6, we cover systems that support caching mechanisms for an inference engine.

## 7.1 Context-sensitive authorization systems

Although others have developed context-sensitive authorization systems, they all use a trusted central server that maintains authorization policies and context information. In those authorization systems, it is implicitly assumed that all the context information providers trust the central server to protect their information properly. The major difference among the existing context-sensitive authorization systems is an authorization language they support. Cerberus [5] uses a first-order logic to express policies and context information. Generalized RBAC (GRBAC) [30, 31, 32], OASIS [8, 52], and Tripathi's resource discovery system [112] adopt role-based access control (RBAC) [103], and use a propositional logic to define context sensitive roles. Myles' [88] language is written in XML. Mostéfaoui [86] and Apu [62] represent context information as a decision tree. Masone [82] developed a role definition language (RDL) based on SQL. We cover each existing context-sensitive authorization system below.

Cerberus [5] allows principals to define context-sensitive policies based on first-order logic. It expresses context information with context predicates such as "Location" and "Temperature", similar to our approach. Cerberus has a monolithic context infrastructure that contains current and historical context information, and a single inference engine evaluates all the authorization decisions.

Myles [88] provides an XML-based authorization language for defining privacy policies that protect users' location information, and those privacy policies could be context-sensitive. A user registers a validator that maintains the user's privacy policy into a location server. When the location server receives a request for accessing the user's location, the server asks the user's validator to make an authorization decision. That validator could access other validators to obtain information to handle the query from the location server. They assume that all validators trust with each other in terms of the confidentiality and

integrity of information.

Users must trust a set of validators that collect context information and make authorization decisions. Hulsebosch [57] developed an access-control system that supports context-sensitive policies. The major objective of their system is to allow users to access a service anonymously; a user provides his context information instead of his identity to be granted access to a service. However, their scheme requires a central context broker server that anonymizes the users' context information, and users must trust that broker server not to disclose their identifiable context information. There is no detail about their authorization language in their paper. They suggest adopting several verification methods that ensure the accuracy of context information, but no detail is covered in the paper.

Generalized RBAC (GRBAC) [30, 31, 32] introduces the environmental role (ERole) to achieve context-aware authorization. Their approach is based on the concept of Role-based access-control (RBAC). Constraints on environmental (context) variables can be defined with a Prolog-like logic language. Authorization is based on an ordinary role and an ERole; in effect, the ERole is an additional condition to be satisfied for an authorization decision. GRBAC has a central context management service that maintains a snapshot of current environmental conditions and has a central authorization engine that interacts with that service. They conducted experiments to measure latency for handling an authorization query with a policy that contains varying number of environmental roles and studied the effect of caching the environmental roles' evaluations. However, each of their measurements only involves a single rule, which is a conjunction of RRoles, maintained in a central authorization server.

Tripathi [112] presents a secure resource discovery system for pervasive computing. The major objective of their system is to allow a mobile user to access resources in different domains seamlessly by binding the resource names with the resources dynamically according to the user's context. Each resource in an environment is protected based on

context-sensitive role-based access control scheme. This approach is similar to Covington's ERole [30, 32]; contextual conditions are defined as preconditions to be satisfied before performing a requested operation. There is no detail about the architecture and design of the system in their paper.

OASIS [8, 52] is an RBAC system that can evaluate contextual conditions at both role-activation time and access time. The context conditions are expressed as context predicates in the Horn clauses of role-activation rules. OASIS has a central object-relational database that stores context predicates and also has a central authorization engine.

Mostéfaoui [85] proposes a role-based authorization system that considers context information. A centralized context engine collects context information from other context servers. The engine authenticates context servers to obtain only trusted information. However, there is no detail about the design of the system and security policies. Mostéfaoui [86] later proposes to express context-sensitive authorization policies with a contextual graph instead of a logical language. The context graph, which is similar to a decision tree, consists of nodes that represent contextual conditions and edges that specify the order to evaluate those conditions. Each path from a starting node to a goal node specifies the conditions that must be satisfied to get a privilege. Their focus is to make policies understandable, and there is no detail about the design and implementation of a system that supports policies represented as a contextual graph.

Masone [82] developed a role definition language (RDL) for defining context-sensitive roles. Unlike other context-sensitive role-based authorization systems that use logical expressions to specify conditions for a role membership, RDL applies mathematical set notation to define a set of role members. A role definition in RDL is converted into event handlers that stores context events into a relational database and role membership publishers that publish a list of role members every time the list of members has changed.

KNOW [62] is a user-feedback mechanism for a context-sensitive authorization system.

KNOW uses Ordered Binary Decision Diagrams (OBDDs) to express context-sensitive policies. An OBDD has two kinds of nodes: terminal and non-terminal. Non-terminal nodes represent atomic propositions, which are conditions in authorization policies, and terminal nodes contain a boolean value TRUE or FALSE. Each path from a root node to a terminal node that contains a TRUE value specifies a set of propositions to be satisfied to be granted access. When a user's request is denied, KNOW traverses the OBDD and finds a set of propositions that must be turned true. If there are multiple such alternative sets, it uses cost functions to compare the usefulness to the user. KNOW protects authorization policies with meta-policies, which corresponds to our confidentiality policies, and avoids disclosing confidential policies to the user who receives feedback. Since KNOW maintains all the policies in a central server, KNOW's approach is not applicable to a distributed proof system like ours.

In summary, none of the earlier approaches distribute authorization policies and context information into multiple administrative domains and explicitly handle trust: confidentiality and integrity. Also, there is no performance study of the earlier systems except for Generalized RBAC (GRBAC) [31].

## 7.2 Distributed authorization systems

In this section, we cover distributed authorization systems in which an authorization server fetches information from remote servers while making an authorization decision. Those systems are distributed in the sense that a policy maker can define policies that refer to information or other policies on remote servers. However, the evaluation process of authorization is centralized: a central server that collects all the necessary information from remote servers makes the authorization decision.

The International Telecommunication Union-Telecommunication Standardization Sec-

tor (ITU-T) X.812 [128] defines a general framework for distributed authorization systems we discuss in this section. X.812 does not specify particular mechanisms for implementing the framework. The framework is designed to support control of access according to context information (e.g., time of attempted access and the location of a requester) in a decentralized environment where multiple security domains are involved. An access-control system in the architecture consists of an Access Control Decision Function (ADF) and an Access Control Enforcement Function (AEF). An ADF function makes access control decisions by applying access-control policies, and an AEF function enforces the decisions made by the ADF. To make an authorization decision, ADF collects from other system entities Access Control Information (ACI), such as context information and attributes of a requester and a resource. In the X.812 framework, it is possible that an ADF or an AEF consists of multiple entities in different security domains. However, the X.812 framework does not define a trust model to exchange information between those entities.

Shibboleth [106], a project of Internet2 [59] and the Middleware Architecture Committee for Education (MACE) [79], defines architectures and policy structures that enable cross-institutional sharing of web resources. In Shibboleth, a service provider makes an authorization decision based on the attributes of a requester. When a service provider receives a request from a user at another institute, it contacts the identity provider at the origin site (where the requester resides) and obtains the attributes of the user who made the request. In Shibboleth, the message formats and protocols between service providers and identity providers are based on the Security Assertion Markup Language (SAML) [101, 102]. To protect users' privacy, Shibboleth requires that control of attribute release to service providers to be available to users as well as administrators. Each user defines attribute release policies that specify which attributes may be disclosed to which service providers, and the attribute authority module in the identity provider enforces those policies when attributes of a user is requested by a service provider. Shibboleth also allows a user to access

a web resource anonymously through the use of a pseudonymous identifier. Shibboleth defines a standard set of attributes that are widely used in the institutes of higher education. Shibboleth does not specify any particular authorization language. Therefore, each service provider needs to convert attribute assertion messages in SAML to statements in its own authorization language.

SPADE [89] provides Shibboleth with an authorization system that limits access to users' attributes in identity providers. SPADE collaborates with the attribute authority module in an identity provider to provide a mechanism that retrieves an attribute release policy (ARP) and makes an access-control decision for the user's attributes requested by a service provider. SPADE supports delegation of authority with SPKI/SDSI [36, 97] to maintain ARPs in a decentralized way; a principal with a senior role in an organization can delegate authority to define ARPs to junior principals by issuing SPKI delegation certificates. SPADE combines the ARPs defined by multiple principals in a delegation chain and derives authorization policies for each attribute of a user. We cover SPKI/SDSI in Section 7.3 in more detail.

Woo [123, 124] proposes a logic-based approach to support authorization in a distributed environment. Woo defines an authorization language based on a first-order logic to separate policies from mechanisms that depend on a particular implementation. Woo's language is designed to express relationships among users' privileges. For example, when we define a policy that allows principal $P$ who possesses a privilege for reading a document $a$ to read another document $b$, the policy could be defined as follows.

$$read(P, a) \Rightarrow read(P, b)$$

where $read$ is a predicate that denotes the read privilege. Their language also supports predicates that express the state of a system, such as CPU load, and it is, therefore, possible

to define authorization policies based on a limited set of dynamic attributes. Since Woo's language supports rules that derive negative rights as well as positive rights, an authorization engine that evaluate the rules could produce two conflicting decisions: *grant* and *deny*. Woo does not discuss how such a conflict should be resolved. Although Woo claims that it is possible to manage authorization policies and facts in different administrative domains, there is no detail about mechanism for sharing those policies across the domains. Woo later developed a distributed authorization system [125] where a central authorization server makes an authorization decision on behalf of a resource owner. The server collects all the information that is necessary to make an authorization decision from other servers. Authorization policies are represented as a generalized access-control list (GACL) where authorization policies are associated with each field in an access-control list.

Jajodia [60] proposes a mechanism for resolving conflicting decisions in a Prolog-like logic-based language. They provide a set of built-in predicates to define rules to resolve the conflicts.

The OASIS eXtensible Access Control Markup Language (XACML) 2.0 [129] is an authorization language based on the Extensible Markup Language (XML). XACML enables a policy maker to define an attribute-based authorization policy: a request from a user contains the attributes of the requester and the requested resource, and only the policies whose attributes match the attributes in the request are evaluated to make an authorization decision. In XACML, it is possible to define additional constraints as boolean conditions combined with disjunction and conjunction operators; a policy in XACML is equivalent to a statement in propositional logic. A policy can refer to sub-policies defined by other servers, and, therefore, the policy can contain nested sub-policies of an arbitrary depth. Since multiple policies for a given request might derive different authorization decisions, XACML provides a mechanism for resolving such a conflict using a policy-combining algorithm.

XACML also defines a distributed architecture that involves four types of system entities: a Policy Enforcement Point (PEP), a Policy Decision Point (PDP), a Policy Information Point (PIP), and a Policy Administration Point (PAP). When a PEP receives a request from a user, it enforces an authorization decision made by a PDP. When it receives a request from a PEP, a PDP makes an authorization decision by evaluating applicable policies and returns a response that contains a granting decision. The PDP obtains authorization policies and attribute values from PAPs and PIPs respectively. XACML defines the formats of a request and a response messages between a PEP and a PDP. However, it does not specify a protocol for transporting messages between the two parties. Furthermore, how a PDP collects policies and attributes from PIPs and PAPs is outside the scope of the XACML 2.0 specification.

The architecture of XACML is decentralized in the sense that authorization policies are defined and maintained by different parties and that those policies are enforced at multiple PEPs. However, for a given request, a single PDP needs to collect the policies and attributes that are necessary to make an authorization decision. That is, the PDP must retrieve all the sub-policies that are referred to in the authorization policies for the request. The current XACML 2.0 specification does not standardize either a mechanism for resolving a policy reference to the corresponding policy in a remote PAP or a mechanism for delegating the evaluations of sub-policies to other PDPs. It is up to each implementor whether to support a mechanism that enables multiple PDPs to evaluate an authorization query in a collaborative way. Sun's XACML implementation [109] provides a mechanism that enables a PDP to delegate the evaluation of a sub-policy to another PDP. However, there is no explicit mechanisms for specifying integrity and confidentiality policies.

Collaboration among system entities in XACML needs a trust model among those entities. However, mechanisms for trust establishment between system entities are outside the scope of the XACML 2.0 specification and is left open to each implementer. For ex-

ample, XACML does not provide a way to specify integrity and confidentiality policies on authorization policies maintained in PAPs, as discussed in Section 9.2 of the XACML 2.0 specification [129]. Lorch [78] reports an experience applying XACML to other distributed authorization systems, such as Shibboleth [106] and PRIMA [77], and concludes that XACML's authorization language is flexible enough to support policies in those systems.

The Security Assertion Markup Language (SAML) [101, 102] is an XML-based framework for exchanging security information between online business partners. SAML defines XML-based common message formats for security assertions, such as authentication statements, attribute statements, and authorization decision statements. In addition, SAML defines an application-layer protocol for exchanging security assertions and specifies bindings that detail how the SAML protocols are mapped onto transport protocols such as Simple Object Access Protocol (SOAP) 1.1 [107]. However, SAML does not define syntax for encoding a proof that consists of policies (rules) and attributes (facts) as our system does.

XACML and SAML are complimentary because XACML only defines a request and response format for a PEP and a PDP and does not specify protocols for obtaining attributes and policies. SAML could be used to implement a protocol that transports messages between a PEP and a PDP. Also, a PDP can obtain attribute assertions from PIPs via a protocol in SAML.

There are several distributed authorization systems based on XACML and SAML. Cardea [68] is a distributed authorization system that adopts XACML for evaluating an authorization request and SAML for exchanging security assertions among system entities. Since the collection of attributes is outside the scope of the XACML specification, Cardea introduces a SAML PDP that handles a SAML authorization request and constructs a XACML authorization request, which is passed to an XACML PDP that actually evaluates authorization policies. The SAML PDP is responsible for collecting attributes from

Attribute Authorities via a protocol defined in SAML. Cardea assumes that a SAML PDP maintains a mapping table between a resource requested in an authorization request and a set of attributes; that is, an SAML PDP knows which attributes an XACML PDP needs in order to evaluate a given authorization request, and includes those attributes into the authorization request to the XACML PDP. Since an XACML PDP is not capable of retrieving attributes that are referred to in its policies, an SAML PDP must know a set of policies that are applied to a given request in advance and it must include all the necessary attributes in the XACML authorization request. The XACML PDP in a single host makes an authorization decision with all the necessary policies and attributes. In Cardea, an SAML PDP relies on an external directory service, such as a Lightweight Directory Access Protocol (LDAP) [67] server, to locate trusted attribute authorities and trusted XACML PDPs.

Stowe [108] extends Sun's XACML implementation [109] to support a mechanism for PDP collaboration. In PDP collaboration, a PDP that receives an authorization query forwards that query to multiple external PDPs and makes an authorization decision by combining the responses from those external PDPs. Each PDP maintains a list of trusted external PDPs to which it forwards a received query. In Stowe's scheme, each PDP applies the same list of trusted PDPs to all different queries. Mazzuca [83] later extends Stowe's system into the eXtensible Distributed Access Control (XDAC) system where multiple PDPs contribute to an authorization decision. To overcome the incompatibility between SAML and XACML, syntax for SAML assertions is extended to include XACML attributes and results. First, a PDP can retrieve a policy from another PDP. Second, a PDP issues an authorization query to other PDPs as in Stowe's PDP collaboration. Mazzuca discussed the issue of confidential policies and attributes, which is involved in the collaboration of PDPs. First, a PDP might want to keep a requested policy from another PDP confidential. Second, when a PDP issues a query to another PDP, some confidential attributes might be passed along with the query to the receiving PDP. However, the system does not provide mecha-

nism for defining and enforcing confidentiality policies on policies and attributes. The trust model of the XDAC system is same as that of Stowe's system: each PDP maintains a truststore that contains certificates of trusted PDPs. Similarly, each PEP maintains a truststore of trusted PDPs. The XDAC system caches recent decisions on a PEP's local memory. The cached decisions are maintained in a linked list. The PEP specifies the lifetime of each element and the maximum number of elements in the cache. There is no explicit mechanism for revoking a cached decision when an attribute that is used to derive the decision changes its value.

Akenti [111] is an authorization service, which corresponds to a Policy Decision Point (PDP) in XACML, that supports X.509 [127] identified users. An authorization policy in Akenti consists of a set of policy certificates issued by multiple authorities in different security domains, but the Akenti policy engine needs to collect all the relevant certificates to make an authorization decision for the user and the resource. The engine finds all the necessary certificates by searching in the URLs specified in the policy certificates. In Akenti, all the policies and attributes of a user or a resource are expressed in XML.

PRIMA [77] is an authorization system in a grid environment where resources are nodes with computational capability. PRIMA uses X.509 attribute certificates [127] to represent user privileges. The architecture of PRIMA is similar to those of XACML and ITU-T X.812 [128]; a policy decision point collects certificates and makes authorization decisions.

In summary, distributed authorization systems provide a mechanism for fetching policies and other information from remote servers. The architecture of those systems except for Sun's XACML [109] are centralized in the sense that there is a single server that collects all the information for making an authorization decision. Those systems do not provide explicit mechanism for establishing trust among principals in terms of the confidentiality and integrity of sharing information.

## 7.3 Trust management systems

Trust management systems are distributed authorization systems in which authorization policies express trust relations among principals. Blaze [19] first introduced the notion of a trust management system that determines whether a set of credentials satisfy authorization policies for the given request. In a trust management system, the process of authentication is integrated with that of authorization, and there is no clear distinction between those processes. When a trust management system converts credentials issued by remote principals to statements in its authorization language, those statements are explicitly associated with the issuers of those credentials in the authorization language of the trust management system. For example, in PolicyMaker [19, 20] and KeyNote [18], a policy consists of a statement and the public key of a principal that issues that statement. Trust management systems based on logic [3, 6, 34, 61, 74] introduce the modal operator, *says*, to specify the issuer of a statement. Others also provide some language constructs to represent the issuer of a statement explicitly.

The earlier trust management systems [3, 19, 36, 74] only provide the functionality for evaluating an authorization query, assuming that all the necessary credentials issued by remote principals are collected in a single server and are converted into statements in their authorization languages. Subsequent trust management systems incorporate those functionalities into their systems. KeyNote [18] performs the verification and conversion of credentials. Bauer's distributed proving system [9], Binder [34], REFEREE [29], SD3 [61], and the Trust Policy Language (TPL) support a mechanism for fetching credentials from remote servers while processing authorization policies. We first cover the trust management systems originally proposed by Blaze [19]. Next, we cover systems whose focus is to support the delegation of authority. Finally, we describe the most general trust management systems whose policies are based on logic.

PolicyMaker [19, 20] is a trust management system that determines whether a particular set of credentials satisfy authorization policies. PolicyMaker eliminates the boundary between an authentication process and an authorization process; PolicyMaker does not have a separate authentication process that verifies the identity of a requester. The core of the PolicyMaker trust management system is a function that takes as inputs a request, a set of local policies, and a set of credentials and that returns a boolean decision (yes or no), depending on whether the credentials constitute a proof that the request complies with the policies. Therefore, this functionality is called *compliance checking*. The enforcement of an authorization decision is left open to each application that issues the authorization query; PolicyMaker just provides an application with an answer for the authorization query.

PolicyMaker expresses both policies and credentials as an assertion that consists of a set of predicates, and one or more public keys. The predicates check the attributes of a request and an environment (including information about the current context). If a source principal that issues an assertion holds authority for the compliance checking of a request, that assertion is applied to the request to obtain principals to which the authority is delegated; that is, if the predicates in the assertion are evaluated to be true, the authority for compliance checking is delegated to the set of principals in the assertion. In PolicyMaker, any language that can be safely interpreted can be used as a language for implementing predicates. For example, a variant of AWK without file I/O operations and with program execution time limits is provided as a sample language. Initially, a local principal that handles a request has the authority for compliance checking and, therefore, a local policy is applied to the request first. Once the authority is delegated to other principals, credentials signed by those principals can be applied to the request in the same way. This process is iterated until the principal that issues the request gains the authority for the request or until the function exhausts all applicable assertions. Thus, the problem of compliance checking is to find a sequence of assertions that derive a requesting principal as the authority for

claiming the statement in the request.

Blaze [20] shows that compliance checking in the general case is undecidable, and PolicyMaker, therefore, only supports a special case that does not support negative credentials. PolicyMaker also assumes that the time complexity of each local function associated with a policy or a credential is bounded. The algorithm for the special case runs in polynomial time. The collection and verification of credentials is outside the scope of PolicyMaker; that is, an application that issues a query to PolicyMaker is responsible for collecting credentials and checking cryptographic digital signatures.

KeyNote [16, 17, 18] is a descendant of PolicyMaker. In addition to the original goals of PolicyMaker, KeyNote aims to standardize a language syntax for policies and credentials and to ease the integration of the trust management engine into applications. KeyNote requires that policies and credentials are written in a specific language so that they can be handled smoothly with KeyNote's compliance checker. KeyNote is responsible for checking cryptographic digital signatures. However, it does not automatically fetch credentials from remote hosts.

REFEREE [29] is a trust management system for web applications. REFEREE provides both a general policy-evaluation mechanism and a language for specifying trust policies. Referee's policies, which are represented as s-expressions, specify conditions on attributes of credentials fetched from remote hosts. Referee's engine provides mechanisms for fetching and verifying credentials from remote hosts.

Herberg [50] developed a Trust Policy Language (TPL) that maps public keys of principals to business roles, based on certificates issued by third parties. A role in TPL is a group of entities that represent a specific organization unit. TPL is written in XML, and the expressiveness of TPL is equivalent to a logic-based language, such as Prolog. TPL is designed to simplify the syntax of a policy that needs multiple parties to sign the same statement about the attributes of a user. There are two major differences from prior trust

management systems such as PolicyMaker [19], KeyNote [18], and REFEREE [29]. First, TPL supports negative certificates as well as positive certificates. It is, however, necessary for a policy maker to define where the policy engine searches for negative certificates. Second, the TPL policy engine provides a mechanism for collecting missing certificates from remote hosts. The TRL policy engine is a single point that collects all the policies and certificates to assign roles to users.

The Simple Public Key Infrastructure (SPKI) [36] defines a standard for authorization certificates. A SPKI certificate binds either a name or a key to authorization. SPKI defines an internal representation of certificates as well as its own certificate format in order to handle PGP certificates [92] and X.509 certificates [127]. The SPKI incorporates local names defined by a Simple Distributed Security Infrastructure (SDSI) [97] so that fully qualified linked names represent globally unique names. SPKI internal representation encodes the delegation of authorization rights as a 5-tuple that consists of an issuer, a subject, a boolean flag, authorization, and validity dates. The boolean flag specifies whether the subject is permitted by the issuer to further propagate the authorization; SPKI does not support integer control to restrict the depth of delegation. The authorization field is represented as an S-expression (a LISP-like parenthesized expression) that contains a set of primitive permissions. An authorization is granted if there is a delegation path from a principal in an access-control list that protects a resource to a principal that makes a request. Although there is a general algorithm for finding the delegation path from a set of unordered certificates in a central repository, the designers of SPKI expect that each principal who is granted some authority should receive a sequence of certificates delegating that authority and that the algorithm might be used only rarely.

The attribute-based privilege management infrastructure (PMI) [64] applies X.509 attribute certificates to enable a principal to delegate its privileges to another principal. A privilege verifier makes an authorization decision by checking whether there is a delega-

tion chain of certificates from the source of authority, which is trusted by the privilege verifier with respect to assignments of privileges to principals, to the principal that issues the request. There is no detail about the mechanism for collecting certificates and checking the delegation chain.

As part of the Digital Distributed System Security Architecture [39, 40], the logic of authentication [3, 66, 122] provides a formal way to define authorization policies that consider statements made by different principals in a distributed environment. The logic of authentication handles only propositional statements that do not contain variables. However, new operators in modal logic [56] are introduced to express different principals' belief in a distributed environment. Those operators makes it possible to define rules that derive a principal's belief based on other principals' belief. For example, we can use the operator *says* to express a principal $A$'s belief on a statement $s$. The statement $A$ says $s$ means that principal $A$ believes that the statement $s$ is true. The *speaks for* operator, which is denoted as $\Rightarrow$, allows us to define principal $A$'s belief based on principal $B$'s belief. The statement $A \Rightarrow B$ means that if principal $A$ believes that a statement $s$ is true, then principal $B$ also believes $s$ to be true; that is, this statement allows principal $B$ to delegate its authority to principal $A$. A request is granted if the system generates a proof that a requesting principal $p$ speaks for one of principals in the access-control list on the requested resource. Lampson's implementation [66] assumes that a server that makes an authorization decision collects all the certificates to construct a proof. The subsequent logic-based trust management systems [6, 9, 34, 61, 74] incorporate the modal operators in the logic of authentication into their logic-based languages using the same or similar syntax. SPKI/SDSI [36, 97] supports the notion of *speaks for* relation between two principals, but uses different syntax. Howell [55] defines the semantics of SPKI based on the logic of authentication.

Aura [7] models delegation certificates as a directed graph, where nodes represent principals' keys and edges represent access rights, and studies an efficient algorithm to find

a delegation chain for making an authorization decision. Aura's algorithm combines a depth-first forward search and a breadth-first backward search to handle threshold certificates efficiently.

Snowflake [54] is a delegation-based authorization system based on SPKI. Howell [55] extends Abadi's logic of authentication [3] to formally define the semantics of SPKI that includes restricted delegation. Howell's logic represents restricted delegation as $A \Rightarrow^T B$ where $A$ and $B$ are principals and $T$ is a set of permissions to be delegated. We read this statement as "principal $A$ speaks for principal $B$ regarding the statements in $T$." In Snowflake, a client who makes a request is responsible for constructing a proof. When the client issues the request, a server returns a principal that the client must speak for and the minimum restriction set that the delegation must allow. The client's prover object constructs a proof that derives a statement that a client speaks for the given principal by traversing search space from that principal.

Proof-carry authorization (PCA) [6] uses higher-order logic, in which we can quantify formulas, to define authorization policies. The goal of PCA is to prove application-specific rules, which appears in Abadi's logic and SPKI, as lemmas proved from a single set of inference rules. Since higher-order logic in PCA is undecidable, PCA puts the burden of constructing a proof on a client principal. Bauer [10, 11] later developed a web-based authorization system based on PCA. In Bauer's system, a client constructs a proof by responding to a challenge issued from the server. Although the server could hide some confidential policies from the client, there is no unified mechanism that protects confidential credentials that the client collects from other servers.

Ranganathan [95] proposes to use a first-order logic to model a user's context and to reason about it. To reason with context information stored in multiple hosts, each context provider on those hosts provides an interface that handles a query from a remote host. However, their scheme does not provide a way to define confidentiality and integrity policies on

rules and facts in a knowledge base of each host, and thus does not have any mechanism for protecting confidential rules and facts, either. Their system does not support any caching mechanism that stores retrieved facts from remote hosts.

Delegation Logic (DL) [74, 75] is a logic-based language for trust management systems. The major difference from earlier trust management systems, such as PolicyMaker [19] and KeyNote [18], is that the semantics of DL is defined based on a well-studied logic programming language. The monotonic version of DL, called D1LP, is based on Datalog [113], which does not support negated atoms and functions with non-zero arities. D1LP extends Datalog by introducing explicit constructs for expressing delegations with integer depths and complex principal structures such as a k-out-of-n threshold that requires agreement among $k$ out of $n$ principals. DL supports the construct *says* to express a rule or a fact stated by a remote principal, and this construct allows a policy maker to express trust on a remote principal's statement. Since DL provides a language construct for expressing trust in the integrity of a statement, it is possible to define a rule that refers to a principal's attribute to decide whether the principal's statement is trustworthy. On the other hand, our scheme does not have this flexibility, because our integrity policy is a list of principals.

DL uses the construct *delegates* to express the delegation of authority; for example,

$$\textit{Bob delegates } access(docZ)\hat{}1 \textit{ to David}.$$

The above rule states that Bob delegates the authority to access $docZ$ to David, and the depth of the delegation is 1, which means that Bob does not trust people that Dave trusts. Ninghui [75] shows that there exists a function that converts a DL program into the equivalent Datalog program by introducing the predicates $holds$ and $delegates$. The function is computable in polynomial time with respect to the size of the program and the maximum delegation depth. Since Datalog is computationally tractable, DL is proved to be tractable

as well.

An authorization engine that supports DL needs to collect all the credentials that are necessary to make an authorization decision and translate those credentials into rules in DL. DL does not provide a mechanism for retrieving credentials from remote servers while evaluating the request. Li [72] also defines a nonmonotonic version of DL, called D2LP, that supports negative facts, and reports that it is difficult to support nonmonotonic policies because complete information is inherently hard to obtain in a distributed environment.

Ninghui [76] later developed the Role-based Trust-management language (RT) that supports role-based access control (RBAC) [103] policies. RT provides a language syntax that allows each principal to define a local role and to delegate authority to another principal by including the remote principal's role in the local role. RT also provides language constructs for defining parameterized roles [42] and separation of duty (SoD). SoD policies define mutually exclusive roles that a principal cannot hold at the same time. However, it is difficult to collect complete information in a distributed environment. RT, therefore, changes the semantics of SoD so that the policy does not involve negative statements; that is, if there is a task that involves multiple roles, RT provides a language construct that ensures that each role is hold by a different principal. The semantics of RT is defined with a transformation function from RT to Datalog.

SD3 [61] is a logic-based language for a trust management system. SD3 is an extension of Datalog [113], which stands for Secure Dynamically Distributed Datalog. SD3 extends Datalog with SDSI linked local names [2, 97], which are local names paired with public keys. The expression $K\$E$ where $K$ is a key and $E$ is a statement means that a key holder of $K$ says the statement $E$. The inference engine of SD3 constructs a proof tree for a given query so that the querier can verify the correctness of the query result. Its focus is to retrieve certificates (that correspond to facts in a knowledge base) from remote hosts automatically, and the whole proof tree is constructed on a central server. Therefore, SD3 requires all the

remote hosts that provide certificates to trust the central server with the inference engine to preserve the confidentiality policies of their facts, as in the authorization systems in Section 7.1. SD3 does not support the transmission of rules in certificates.

Binder [34] is a logic-based authorization language that extends Datalog [113] with a modal operator *says*. In Binder, inference engines on different servers can exchange statements (facts or rules) as signed certificates. When a statement is imported into an inference engine, that statement is automatically quoted with the *says* operator to differentiate it with local statements. The *says* operator can be used to specify trust in the integrity of a fact maintained in a remote server. For example, the following statement expresses the trust in an authority at Dartmouth College that possesses the public key *rsa:3:clebad8d* to say who is a Dartmouth student.

$$student(P, Dartmouth) \text{ :- rsa:3:clebad8d } \textit{says } student(P, Dartmouth).$$

This statement corresponds to our integrity policy on a fact. However, Binder does not provide a way to define trust on a rule, and thus does not have the notion of integrity for a proof that consists of rules and facts. Although Binder allows multiple servers to exchange certificates that contain rules and facts, it does not have a mechanism for decomposing a proof; that is, a server that issues a query collects all the facts and rules for deriving the query result. Binder provides no mechanism for protecting confidential rules and facts.

Bauer [9] developed a distributed proving system that constructs a proof that grants access to a resource; a principal that constructs a proof could delegate a task of building a sub-proof to another principal rather than collecting all the certificates that are necessary to construct a whole proof. Bauer's authorization language is based on the logic of authentication [3], and provides the modal operators such as *says* and *speaksfor*. Bauer's scheme is similar to ours in the sense that a proof is produced by multiple principals in a distributed

environment. However, their algorithm does not address the issue of protecting confidential information in certificates, which are used to construct a proof. Like Binder [34], Bauer's language uses a construct *says*, to express trust in the integrity of a fact in a remote server. However, even if a principal that issues a query trusts the integrity of a principal that handles that query in terms of the integrity of the query result, the handler principal still returns a proof that contains multiple certificates and thus a whole proof for an initial query is constructed on the host of a principal that issues an initial query. There is no mechanism for proof decomposition like ours. Although their system caches both positive and negative facts, it does not support mechanisms for revoking cached information.

The idea of delegating the evaluation of a proof to a trusted server also appears in some protocols used to verify a certificate in a public-key infrastructure. To verify a certificate, one must construct a certificate chain from the certificate authority (CA) that issued the certificate to a CA that is trusted by a querier. The Simple Certificate Validation Protocol (SCVP) [81] allows a client with limited processing and communication capabilities to ask a trusted server about the validity of a certificate. The client can specify a list of trusted CAs in its validation policy to be observed by the server. The client can ask the server to provide additional information, such as a certification path and corresponding revocation status, depending on the trustworthiness of the server. Although it is similar to our work in the sense that the protocol uses the client's trust in the server to split the overhead of verifying a certificate between them, it is specialized in handling certificate chains, and it does not support general rules. In addition, there is no mechanism that addresses the confidentiality of rules or facts, because cross certificates (trust relations) among CAs are considered to be public knowledge.

PeerAccess [121] is a framework for reasoning about authorization in a distributed environment. PeerAccess consists of a set of peers that maintain a local knowledge base. Each peer makes an authorization decision by collecting rules and facts maintained by

other peers. Each knowledge base also contains release policies that protect the local authorization policies and facts. Release policies in PeerAccess are *sticky* in the sense that those policies are permanently attached to the information they protect; that is, PeerAccess assumes that all peers that exchange their rules and facts with each other enforce the original publishers' release policies correctly. Although PeerAccess supports mechanisms that protect confidential authorization policies, each peer needs to collect all the rules and facts in a proof to make the authorization decision.

We conclude this section with the summary of the major features of the trust management systems, comparing with our system. First, the trust management systems makes an authorization decisions by considering credentials issued by remote servers. Our system does that as well because a query result is a logical statement or a proof that is digitally signed by a remote principal. However, the query result has a more complex structure than a credential in a trust management system; that is, the query result could be embedded with other credentials issued by multiple different principals as we describe in Section 3.2 and Section 4.1 respectively. Second, the trust management systems provide an authorization language that supports logical inference. There are several systems [9, 34, 61, 74] whose authorization languages are the extension of Datalog. They all except for D2LP [72] only support monotonic reasoning and do not support the negation of statements in their language because it is essentially difficult to collect complete information in a distributed environment. Our authorization language, which is also based on Datalog, does not support nonmonotonic reasoning, either. Third, in trust management systems, all the statements converted from credentials are explicitly associated with the issuer of that statement. For example, the modal operator *says* in a logic-based language combines a statement with its issuer. It is possible to implicitly specify the trust in the correctness of a statement obtained from a remote principal by defining a rule that derives a local statement from a statement in which a trusted remote principal is quoted. That is, the trust management systems based

on logic incorporate integrity policies into their authorization language. On the other hand, our system provides a different way to define integrity policies; that is, our integrity policies are essentially a mapping table that associates a rule or a fact in authorization policies with a list of trusted principals. The approach with the modal operator allows a policy maker to specify integrity policies based on rules. For example, a principal $p_0$ can express an integrity policy that requires two principals $p_1$ and $p_2$ to state a same statement as follows.

$$loc(P, L) \leftarrow p_1 \textit{ says } loc(P, L) \wedge p_2 \textit{ says } loc(P, L)$$

In the above policy, a statement of a principal's location is trusted if principals $p_1$ and $p_2$ agree on the statement. Our system cannot express such integrity policies that involve inference. However, this approach makes it difficult to share integrity policies with other principals, because there is no explicit distinction between authorization and integrity policies in a trust management system. Integrity policies might be tightly coupled with authorization policies that contain confidential information. Therefore, it is difficult for one principal to delegate the task of constructing a subproof to another principal. Fourth, the trust management systems evaluate authorization policies in a central server, although their authorization policies encode trust relations among different principals. Our system does not have a central server for evaluating an authorization query to protect the participants' confidential policies.

In summary, the trust management systems are distributed authorization systems whose policy languages express integrity policies among principals. Existing trust management systems except for PeerAccess [121] do not provide an explicit mechanism for expressing confidentiality policies on rules and facts. Although authorization policies in trust management systems consider trust relations among principals in multiple security domains, the process of evaluating an authorization query must be performed in a centralized way;

that is, an authorization decision is made by a central server that collects all the credentials referred to by the authorization policies.

## 7.4 Automated trust negotiation systems

An automated trust negotiation (ATN) system [21, 105, 118, 119, 132, 133] is an attribute-based authorization system whose focus is to protect authorization policies on resources. In ATN, a requesting client is responsible for providing attribute certificates in order to show that the client satisfies the authorization policies for accessing the resource. However, if the server asks the client to submit a set of certificates, the server might disclose its confidential policies. To solve this problem, an ATN system treats a policy as a resource to be protected; that is, an authorization policy that protects a resource can be protected with another policy, and that meta policy is also protected with another policy recursively. When a server receives a request for a resource, the server must traverse the policies, starting from the resource, until it finds the outermost policy, which is not protected with any policy. To make an authorization decision, the server first checks the outermost unprotected policy by asking the client to provide certificates that are necessary to check that policy. If the client satisfies that policy, the server checks the policy protected by the outermost policy. This process continues until the server checks the confidentiality policy that protects the requested resource. Similarly, an ATN system considers a client's certificates as a resource and has the client define a sequence of policies on a certificate in the same way. Therefore, the process of making an authorization involves iterative disclosures of certificates between the client and server.

We can consider an ATN system as the special case of our distributed proof system; there exist only two principals (i.e., a client and a server) who participate in making an authorization decision. In this setting, if one principal, who receives a query from another

principal issues a subsequent query, back to the principal that issues the original query, the former principal reveals information about the rule that is used to handle the original query. As we discuss in Section 8.6, we avoid this inference problem by preventing each server from issuing a circular subsequent query. We believe this solution is satisfactory for systems in pervasive computing, because a mobile client, who only carries a portable device, is unlikely to be involved in the process of making a granting decision. (On the other hand, ATN systems are primarily intended for web applications [120].) If a server that receives a client's request issues a subsequent query to another server, the server does not disclose its authorization policy to the client. Therefore, we do not introduce policies that protect confidentiality policies, which corresponds to meta policies in an ATN system.

Ye [131] proposes the collaborative trust negotiation scheme for peer-to-peer systems. In that scheme, a locally trusted third party (LTTP) breaks cyclic interdependent policies that are involved in the negotiation process between a client and a server. Every server in a peer-to-peer system could be a LTTP that caches other server's credentials obtained through the process of trust negotiation. Also, each server maintains a list of LTTPs that receive its credentials; each server maintains a different list of LTTPs, depending on with which servers it succeeded in a trust negotiation. When two parties fails in a trust negotiation, they try to find a LTTP trusted by both parties. If there exists such a LTTP, that LTTP disclose the interdependent policies to each negotiating parties to enable them to continue their negotiation process. Although, Ye's scheme involves a trusted third party in a trust negotiation, credentials, which are involved in the negotiation, belong to either a client or a server; the negotiation does not require credentials from other servers.

## 7.5 Secure function evaluation

Secure function evaluation (SFE) [26, 44, 45, 130] is a technique that enables mutually untrusted principals to compute a function jointly while hiding their inputs from each other; that is, there is a general algorithm that computes a function $f(x_1, \ldots, x_n)$ where $x_i$ is principal $p_i$'s private input, while ensuring that each principal $p_i$ learns only the output of the function $f$ from the computation. The objective of SFE is similar to ours: We want mutually untrusted principals to compute an authorization decision with each principal's private rules and facts as inputs to the function. However, SFE is not directly applicable to our problem because the algorithm of the function $f$ to be computed in SFE must be public knowledge among participating principals before the computation. In our system, there is no way to know in advance which rules and facts in multiple servers are involved to make an authorization decision. Nevertheless, in this section, we cover cryptographic schemes that provide a specialized solution for a two-party SFE problem in order to address the problem of cyclic policies in automatic trust negotiation systems where a client and a server possess private attributes and private policies respectively.

Li [73] solves the problem of cyclic policies in an ATN system by applying a two-party secure function evaluation (SFE) [130]. Li assumes that the content of a certificate is public and that only the fact that a trusted party signs the certificate is sensitive, and formalizes the problem as follows: Suppose that Alice and Bob have certificates $P_A$ and $P_B$ respectively and that those certificates involve cyclic dependency. Let $P_A$ be a pair $(M_A, \rho)$ where $M_A$ and $\rho$ are the content and signature of the certificate $P_A$ respectively. Alice can disclose $M_A$ to Bob. However, Alice keeps $\rho$ private and Bob keeps $P_B$ private as well. Alice and Bob can break the cyclic policies by jointly computing the function that takes as inputs $M_A$, $\rho$, and $P_B$. SFE ensures that Alice and Bob do not learn the other party's private input (i.e., $\rho$ or $P_B$) from the computation and that Alice obtains Bob's certificate $P_B$ at the end of the

computation if Alice uses a valid signature $\rho$ for the computation. Since this computation involves the verification of the signatures of the attribute certificates, the general technique of SFE is computationally expensive. Therefore, Li developed a scheme called oblivious signature-based envelope (OSBE) to solve this problem efficiently. OSBE enables two parties to share a secret key through a technique similar to that in the Diffie-Hellman key agreement protocol [35] if Alice possesses the valid signature for her certificate. At the end of the OSBE protocol, Alice is able to decrypt Bob's certificate with the jointly computed secret key.

Holt proposes the scheme of hidden credentials [53] where a client can decrypt a requested resource if the client possesses the right credentials that satisfy the server's authorization policies. Holt's scheme makes use of identity-based encryption [22], in which an arbitrary string can be used as a public key; that is, a server encrypts a resource with a public key, which is the concatenation of a client's identity and an attribute, and the client decrypts it with the private key that corresponds to the credential specified by the server. In Holt's scheme, a client can access a resource without disclosing its credentials, and a server can protect its policies from a client that does not possess a privilege for accessing the resource. Holt's scheme supports a policy represented as a boolean formula containing multiple credentials. A conjunctive policy that requires a client to possess a set of credentials can be enforced by encrypting a resource recursively with different keys. The disjunctive policy can be enforced by providing multiple encrypted resources, each of which is encrypted with a different public key corresponding to a different credential. Holt's scheme has the drawback that an authorized client could learn the structure of a policy. Since a public-key operation in the Holt's scheme produces ciphertexts longer than the input plain text, it is possible to infer how many *AND* operators are contained in a policy. Bradshaw [23] adopts a secret splitting scheme to avoid disclosing information about the structure of a policy to an unauthorized client. Frikken [38] proposes a scheme for hidden credentials that provide

stronger security properties. In Frikken's scheme, a client does not learn anything about a server's policies even if the client has a set of credentials that satisfy the servers' policies.

Oblivious commitment-based envelope (OCBE) [69] is a scheme that enables oblivious access control. The scheme of OCBE is similar to that of hidden credentials [23, 38, 53] in that a server sends a client encrypted information that the client can decrypt if attribute certificates of the client satisfy the server's policies. OCBE is based the Pedersen commitment scheme [91], and a client gives a server an attribute certificate that contains the commitment of attribute values. The major difference from the scheme of hidden credentials is that OCBE supports policies that involve comparison predicates such as $\leq$ and $\neq$. Therefore, OCBE allows a server to define a policy that specifies not only an exact value of an attribute but also an acceptable range of that attribute. OCBE also supports the conjunction and the disjunction operators to express multiple conditions on the client's attributes. However, OCBE requires a server to disclose its policies to a client before starting the OCBE protocol; it does not protect the servers' private policies even if the client does not have the privilege to access the requested resource.

Based on OCBE [69], Li [70] developed a scheme called certified input private policy evaluation (CIPPE) for policy-hiding access control. In CIPPE, a client and a server compute an authorization decision without disclosing the client's attributes and the server's policies to the other party. CIPPE adopts Yao's scrambled circuit protocol [80, 130] to protect a server's policies from the client. The server constructs a scrambled circuit from its policy and gives it to the client. The client can obtain the input keys of the circuit, which are necessary to evaluate the circuit with its inputs, if the client's knows the attributes values that correspond to cryptographic commitments [91] shared with the server as in the scheme of OCBE. With the correct input keys, the client evaluates the scrambled circuit and obtains the scrambled output, which must be decrypted by the server. The server finally obtains the decision by decrypting that output received from the client. CIPPE also

provides a mechanism that makes the topology of a scrambled circuit obscure to avoid revealing the structure of a policy. Li [71] later proposes a general framework for automatic trust negotiation whose protocol supports cryptographic credentials such as hidden credentials [23, 38, 53] and OCBE [69].

Hengartner [48, 47] adopts a scheme of hierarchical identity-based encryption (HIBE) [41] to support oblivious access control in pervasive computing where privileges could be defined hierarchically. In pervasive computing, there are many situations where a clients privilege for accessing fine-grained information allows the client to access coarse information inferred from the former information. Location information that is defined based on containment relations is such an example. Hengartner's scheme allows a certificate holder of fine-grained information to access a resource protected with a policy that refers to coarser information obtained from the fine-grained information.

In this section, we introduce cryptographic schemes that protect a client's confidential credentials from a server. However, those schemes assume that all the credentials and the authorization policies are managed by a client and a server respectively. Since our system must deal with a situation where facts and policies are distributed across multiple parties, the schemes in this section is not applicable to our problem.

## 7.6 Caching for an inference engine

There is little published about caching for an inference engine. We conclude this chapter covering a few logic-based systems that support caching mechanisms.

Katsiri [63] built a prototype of a dual-layer knowledge base based on a first-order logic. The higher Deductive Abstract layer caches abstract context information derived from low-level knowledge in the lower layer to make the system scalable. The system consists of a single server trusted by all and does not support a revocation mechanism in a distributed

environment.

Wullems [126] developed a role-based authorization system that is similar to OASIS [8, 52]. It has a central server that contains an authorization engine and a context management service. Context-sensitive policies, which are expressed with a propositional logic, are used to activate or deactivate a user's role membership. The server maintains a set of active roles and reevaluates them when receiving specified triggered context events; that is, the server caches the privileges of each user as a set of roles. If the role membership status of a user changes, the server notifies services that make authorization decisions based on that role membership. However, since the system is centralized, they do not support a revocation mechanism that involves multiple context management services in a distributed environment like ours.

# Chapter 8

# Discussion

In this chapter, we discuss several design issues, limitations, and security properties of our system.

## 8.1 Completeness of our algorithm

The algorithm for the function GENERATEPROOF in Figure 3.10 and Figure 4.6 is not complete. That is, it is not guaranteed to find a proof that derives a granting decision even if one exists. This situation could happen when a principal who handles a query is able to produce multiple alternative proofs. Suppose that the proof that the principal produces first contains an encrypted subproof obtained from another principal and that encrypted subproof contains a FALSE value. When the principal returns the proof, a principal who receives the proof finds that proof invalid after decrypting its embedded subproof. However, since the principal that returned the proof finished handling the query, that principal is not able to find another alternative proof.

There are two possible ways to make our algorithm complete. One is to add a mechanism that enables a principal that finishes handling a query to resume processing the same

query from the previous point of the search space; an inference engine continues to maintain various data structures, such as a proof tree, for a query after a proof for that query is returned, and those data structures must be associated with a unique identifier of the query to resume the query handling process. This approach does not waste computing and network resources since a proof is constructed on demand. However, it could consume a lot of memory to maintain these data structures. Furthermore, this approach makes it possible for a principal that handles a query to infer the query result in an encrypted subproof produced by another principal; that is, if a querier principal issues a same query again, the handler principal can infer that the encrypted subproof embedded in the previous proof contains a *FALSE* value. Considering this security issue, we cannot choose the first approach.

The other is to modify an interface for handling a query so that a principal that handles a query can return multiple alternative proofs together. A querier principal specifies the number of proofs he needs as a parameter in a query, and the querier and handler principals maintain a network session until all the proofs are transmitted. This approach does not have the security risk of the first approach. However, we need to extend the representation of a proof in Section 4.1 to evaluate a set of alternative subproofs as the disjunction of the query results in those subproofs. This situation happens when a principal that handles a query issues a subsequent query and obtains multiple alternative proofs together. If the principal cannot decrypt those proofs, the principal must embed the disjunction of those proofs into a proof that the principal returns. Although we believe this approach is feasible, it requires a lot of changes in our current implementation, such as the algorithm for checking the integrity of a proof, the data structure of a proof object, and so on. Therefore, we plan to make our algorithm complete in our future work. Although this approach could consume a lot of computational and network resources to construct multiple proofs for a single query, to maintain the cached facts derived from the proof is as efficient as with our original algorithm since it is not necessary to modify our caching and revocation mechanism in

Chapter 5. Thus, the amortized performance in common cases may not be unreasonable.

## 8.2 Security assurance

Our authorization scheme ensures that each principal's confidentiality policies are preserved while participating in the evaluation of an authorization query. A malicious principal that represents an internal node of a proof subtree cannot inappropriately obtain a rule or a fact from other principals by modifying the *receivers* list in a subquery it issues, because each principal discloses its rules or facts to other principals only if they satisfy its confidentiality policies as described in Section 4.6.1.

The malicious principal could also modify the integrity policies *i_policies* in a subquery to disturb the evaluation of a query. This attack can be prevented if every principal publishes its integrity policies with its digital signature on a well-known server, and each principal can cache other principal's integrity policies. The *i_policies* in a query can then be retrieved by identifying the principal sending the query, which should also be listed last in the $receivers$ list.

We use a nonce to prevent a reply attack by a malicious party that is capable of intercepting and modifying a message. All the participating principals that evaluate an authorization query use the same nonce, generated by the original querying principal, because the receiver of a proof might be different from a querier principal. The nonce in a proof must match the nonce in the query, for the proof to be valid.

## 8.3 Timeliness of authorization decisions

As we see in Chapter 6, to process an authorization query involves latency for constructing a proof, and the authorization decision is thus derived from the proof that might contain

dynamic facts previously published at different times. Since an authorization decision is made based on information collected in the past, our system might grant a request that should have been denied if the current information was available to the system. This limitation in our system might allow a malicious user to gain access to a resource illegally by changing his context from a legitimate state (that grants his access) to an illegitimate state before the system detects the change of the the user's context. Therefore, our system should provide a policy maker with a way to define explicit timeliness constraints on authorization decisions; that is, a policy maker should be able to specify a time $T$ such that all the information in a proof was published within time $T$ prior to the current time. Although our system does not explicitly support this mechanism, experimental results in Section 6.3 imply that our system would work even if $T$ were as small as six hundred milliseconds for a large proof of depth 10.

Our system supports persistent queries that monitor users' privileges continuously. Our revocation mechanism allows the system to update the authorization decisions responding to the context change within the delay of six hundred milliseconds.

## 8.4 Expressiveness of the authorization language

Our example in Section 2.1 represents policies about the current context. Although we do not treat temporal information specially in our language, our language can express some policies about historical context by defining predicates that take a timestamp as an argument. The following is an example policy in a workflow system where an authorization decision is based on whether a requester has performed a series of actions in a specified sequence.

$$grant(P, purchase, X) \text{ :- } approved(mgr, P, X, t_1), approved(senior\_mgr, P, X, t_2), prior(t_1, t_2)$$

The above policy requires that a requester needs to obtain an approval from the manager first, and then from the senior manager. The predicate $prior$ is used to check whether timestamp $t_1$ is prior to $t_2$. For policies that depend on freshness of information, we could define the predicate $recent$ that checks the recency of a timestamp using the $prior$ predicate as follows.

$$recent(T, X) \text{ :- } prior(now - X, T).$$

The above rules says that a timestamp $T$ is within the duration $X$ from now if $T$ is later than the time $now$ - $X$. We assume that the system define a special dynamic variable "now."

Our language does not express *separation of duty* [4] in role-based access control (RBAC) model [103], because to express separation of duty with a logic language (as Jajodia [60] proposes) requires support for rules that contain the negations of atoms. A querier possibly obtains a false negative in our system due to the constraints of the security policies of each principal. Therefore, a query that is a negation of an atom causes false positive, which is not acceptable to any authorization system.

For the same reason, our system does not support a mechanism for detecting or resolving conflicting facts. That is, if our system constructs a proof that contains the fact $location(bob, sudikoff)$, our system considers that proof as valid even under the presense of a conflicting fact (e.g., $location(bob, moore)$). To reduce such conflicts in facts, we need a rule that derives the negation of a fact; however, our authorization language does not support the negation of atoms. Therefore, we cannot define the rule below to detect Bob's conflicting location information.

$$\neg location(bob, L1) \text{ :- } location(bob, L2), L1 \neq L2)$$

Support for negation and confliction detection is future work.

## 8.5 User feedback

It would be useful, in the case of a FALSE proof, to provide some feedback for the user about why the proof failed and what policies prevent them from obtaining the desired access. Although to return an incomplete proof is a plausible solution, there are two issues to be addressed. First, due to confidentiality policies the user may not be allowed to receive the incomplete proof, and, as a result, the user is not able to know which subproof failed. Second, because there could be multiple incomplete proofs for a given query, we need some mechanism that chooses a useful proof for the user from them. The KNOW system [62], which is a centralized rule-based authorization system, uses a cost function to rank proofs for a query based on the likeliness that the user is able to satisfy the conditions in the proofs. It is, however, difficult to define a reasonable cost function in a decentralized system like ours because there is no single administrator who knows all the rules and security policies that are involved in authorization decisions. We leave this complex problem for future work.

## 8.6 Information leak through inference

In our system, it is possible for a principal who receives a query to issue a subsequent query to the principal who issues the original query. This opportunity allows the querier principal to learn the handler principal's confidential rule. For example, suppose that a principal $p_0$ issues a query $?grant(bob, document)$ and that $p_1$, who maintains a rule $grant(P, document)$ :- $employee(P, IBM)$, issues a query $?employee(bob, IBM)$ to principal $p_0$. Then, principal $p_0$ learns that $p_1$'s granting policy depends on the requester's employment, although $p_0$ is not sure whether there are any other conditions to be granted access. One possible solution is to prevent each principal that handles a query from issuing

a query to a principal in the receivers list of that query. We also need to consider this issue to design the user feedback mechanism we discuss in the previous section.

# Chapter 9

# Conclusions and Future Work

This dissertation presents a secure distributed proof system for context-sensitive authorization. We first summarize our contributions, discuss limitations of and future work for the system, and finally conclude.

## 9.1 Contributions

Any context-sensitive authorization system will necessarily be distributed, because context information and rules will be produced at different places by different parties, and the confidentiality of those information must be protected in a decentralized way. A large trusted server, which is adopted by earlier context-sensitive authorization systems, is not realistic. We take a logic-based approach to address the issue of information sharing across multiple administrative domains and build a secure distributed proof system for context-sensitive authorization. Our system enables multiple hosts to evaluate an authorization query in a peer-to-peer way, while preserving the confidentiality and integrity policies of mutually untrusted principals running those hosts. The primary contributions of this thesis are

- support for fine-grained security policies that formally define trust relations among principals in terms of information confidentiality and integrity;
- a distributed algorithm that enables multiple principals to construct a proof in a peer-to-peer way, while preserving the integrity and confidentiality policies of those principals;
- a soundness proof of the algorithm;
- the design and implementation of a secure distributed proof system;
- a novel and efficient caching and revocation mechanism that considers the dependency of context information, which is frequently changing, across multiple servers; and
- a detailed performance study of our system showing that the amortized performance of our system scales to a large proof that spans across dozens of servers.

## 9.2 Limitations and future work

We discuss several limitations of our system in Chapter 8 and summarize them below. Our system is not guaranteed to find a proof that satisfies the security policies of each principal, as we discuss in Section 8.1. Therefore, we need to modify the current algorithm to ensure completeness of our algorithm. We also need to modify our algorithm to prevent the inference attack described in Section 8.6. We need a feedback mechanism in Section 8.5 for a user whose request is denied.

There are possible extensions to our current system. First, it is necessary to protect the confidentiality of a query itself in some situations, because issuing a certain query might imply the querier's interest, which he does not want to disclose. We plan to add a

mechanism for protecting each principals' queries. Second, our system still fails to construct a proof if our system cannot satisfy all the participants' security policies. We plan to improve the possibility of constructing a proof by facilitating information sharing among mutually untrusted principals with the technique of secure multi-party computation [46] and anonymization [115, 37]. Third, there are many situations where we have to deal with context information with varying uncertainty. We, therefore, plan to extend our authorization and policy languages to handle probabilistic context information. Fourth, we plan to apply our distributed proof system to different applications, such as a resource discovery system for pervasive computing, to generalize the functionality of our system. Finally, we need understand threats to integrity of raw sensor data in all context-sensitive authorization systems.

## 9.3 Conclusions

We present a secure distributed proof system for context-sensitive authorization. Our system enables multiple hosts to evaluate an authorization query in a peer-to-peer way, while preserving the confidentiality and integrity policies of mutually untrusted principals running those hosts. We show that it is possible to derive an authorization decision, even though authorization rules and context information referred to by those rules are distributed across multiple domains and are protected with different confidentiality policies.

We also developed a novel caching and revocation mechanism that improves the performance of our system. Our capability-based revocation mechanism combines an event-based push mechanism with a query-based pull mechanism where each server recursively publishes revocation messages over a network by maintaining dependencies among local and remote cached facts.

Our experimental results show that the performance overhead of public-key operations

involved in the process of a remote query were large and that our caching mechanism significantly reduced the amortized latency for handling a query. The results show that our system should be suitable to a context-aware application in which a user's privileges must be continuously monitored. Since our experiments were conducted with a wide range of parameters, the results serve as guidelines about the worst-case performance of a systems that adopts our techniques.

Although we describe our system in the context of a logic-based authorization system, we believe our scheme is general enough to support various kinds of rule-based policies in pervasive computing.

# Bibliography

[1] Summary of HIPAA privacy rule, 2004. `http://www.hhs.gov/ocr/privacysummary.pdf`.

[2] Martín Abadi. On sdsi's linked local name spaces. In *Proceedings of the 10th Computer Security Foundations Workshop (CSFW '97)*, page 98, Washington, DC, USA, 1997. IEEE Computer Society.

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

[4] Gail-Joon Ahn and Ravi Sandhu. The RSL99 language for role-based separation of duty constraints. In *Proceedings of the fourth ACM Workshop on Role-based Access Control*, pages 43–54. ACM Press, 1999.

[5] Jalal Al-Muhtadi, Anand Ranganathan, Roy Campbell, and Dennis Mickunas. Cerberus: a context-aware security scheme for smart spaces. In *Proceedings of the First IEEE International Conference on Pervasive Computing and Communications*, pages 489–496. IEEE Computer Society, March 2003.

[6] Andrew W. Appel and Edward W. Felten. Proof-carrying authentication. In *Proceedings of the 6th ACM conference on Computer and communications security*, pages 52–62. ACM Press, 1999.

[7] Tuomas Aura. Fast access control decisions from delegation certificate databases. In *Proceedings of Third Australasian Conference on Information Security and Privacy (ACISP '98)*, volume 1438 of *LNCS*, pages 284–295, Brisbane, Australia, July 1998. Springer.

[8] Jean Bacon, Ken Moody, and Walt Yao. A model of OASIS role-based access control and its support for active security. *Proceedings of the Sixth ACM Symposium on Access Control Models and Technologies*, 5(4):492–540, 2002.

[9] Lujo Bauer, Scott Garriss, and Michael K. Reiter. Distributed proving in access-control systems. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, pages 81–95, Washington, DC, USA, 2005. IEEE Computer Society.

[10] Lujo Bauer, Schneider A. Michael, and Felten W. Edward. A proof-carrying authorization system. Technical Report TR-638-01, Princeton University, April 2001.

[11] 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*, San Francisco, CA, August 2002.

[12] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying hash functions for message authentication. In *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, pages 1–15, London, UK, 1996. Springer-Verlag.

[13] Alastair R. Beresford and Frank Stajano. Location Privacy in Pervasive Computing. *IEEE Pervasive Computing*, 2(1):46–55, January-March 2003.

[14] K. Biba. Integrity considerations for secure computer systems. Technical Report 76-372, U.S. Air Force Electronic Systems Division, 1977.

[15] Andrew D. Birrell, Butler W. Lampson, Roger M. Needham, and Michael D. Schroeder. A global authentication service without global trust. In *Proceedings of the 1986 IEEE Symposium on Security and Privacy*, pages 223–230, April 1986.

[16] Matt Blaze, Joan Feigenbaum, John Ioannidis, and Angelos Keromytis. The KeyNote Trust Management System, Version 2. RFC-2704, September 1999. `http://www.crypto.com/papers/rfc2704.txt`.

[17] Matt Blaze, Joan Feigenbaum, John Ioannidis, and Angelos D. Keromytis. The role of trust management in distributed systems security. *Secure Internet programming: security issues for mobile and distributed objects*, pages 185–210, 1999.

[18] Matt Blaze, Joan Feigenbaum, and Angelos D. Keromytis. Keynote: Trust management for public-key infrastructures (position paper). In *Proceedings of the 6th International Workshop on Security Protocols*, pages 59–63, London, UK, 1999. Springer-Verlag.

[19] Matt Blaze, Joan Feigenbaum, and Jack Lacy. Decentralized trust management. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 164–173. IEEE Computer Society Press, May 1996.

[20] Matt Blaze, Joan Feigenbaum, and Martin Strauss. Compliance checking in the policymaker trust management system. In *Proceedings of the Second International Conference on Financial Cryptography*, pages 254–274, London, UK, 1998. Springer-Verlag.

[21] Piero Bonatti and Pierangela Samarati. Regulating service access and information release on the web. In *Proceedings of the 7th ACM conference on Computer and communications security*, pages 134–143, New York, NY, USA, 2000. ACM Press.

[22] Dan Boneh and Matthew K. Franklin. Identity-based encryption from the weil pairing. In *Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology*, pages 213–229, London, UK, 2001. Springer-Verlag.

[23] Robert W. Bradshaw, Jason E. Holt, and Kent E. Seamons. Concealing complex policies with hidden credentials. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 146–157, New York, NY, USA, 2004. ACM Press.

[24] Patrick Brezillon. Context-based security policies: A new modeling approach. In *Second IEEE Annual Conference on Pervasive Computing and Communications Workshops*, pages 154–158. IEEE Computer Society, March 2004.

[25] Iliano Cervesato, Aaron D. Jaggard, Andre Scedrov, and Chris Walstad. Specifying kerberos 5 cross-realm authentication. In *Proceedings of the Workshop on Issues in the Theory of Security (WITS'05)*, January 2005.

[26] David Chaum, Claude Crépeau, and Ivan Damgard. Multiparty unconditionally secure protocols. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 11–19, New York, NY, USA, 1988. ACM Press.

[27] Guanling Chen, Ming Li, and David Kotz. Design and implementation of a large-scale context fusion network. In *First Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services (Mobiquitous)*, pages 246–255, August 2004.

[28] Harry Chen, Tim Finin, and Anupam Joshi. An Ontology for Context-Aware Pervasive Computing Environments. *Special Issue on Ontologies for Distributed Systems, Knowledge Engineering Review*, 18(3):197–207, May 2004.

[29] Yang-Hua Chu, Joan Feigenbaum, Brian LaMacchia, Paul Resnick, and Martin Strauss. REFEREE: Trust management for Web applications. *Computer Networks and ISDN Systems*, 29(8–13):953–964, 1997.

[30] Michael J. Covington, Mustaque Ahamad, and Srividhya Srinivasan. A security architecture for context-aware applications. Technical Report GIT-CC-01-12, Georgia Institute of Technology, May 2001.

[31] Michael J. Covington, Prahlad Fogla, Zhiyuan Zhan, and Mustaque Ahamad. A context-aware security architecture for emerging applications. Technical Report GIT-CC-02-15, Georgia Institute of Technology, February 2002.

[32] Michael J. Covington, Wende Long, Srividhya Srinivasan, Anind K. Dey, Mustaque Ahamad, and Gregory D. Abowd. Securing context-aware applications using environment roles. In *Proceedings of the Sixth ACM Symposium on Access Control Models and Technologies*, pages 10–20. ACM Press, 2001.

[33] Data Encryption Standard (DES), October 1999. `http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf`.

[34] John DeTreville. Binder, a logic-based security language. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, page 105, Washington, DC, USA, 2002. IEEE Computer Society.

[35] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, 1976.

[36] C. M. Ellison, B. Frantz, B. Lampson, R. Rivest, B. M. Thomas, and T. Ylonen. SPKI certificate theory. Internet RFC 2693, October 1999. `http://www.ietf.org/rfc/rfc2693.txt`.

[37] Michael J. Freedman and Robert Morris. Tarzan: a peer-to-peer anonymizing network layer. In *Proceedings of the 9th ACM conference on Computer and communications security*, pages 193–206, New York, NY, USA, 2002. ACM Press.

[38] Keith Frikken, Mikhail Atallah, and Jiangtao Li. Hidden access control policies with hidden credentials. In *Proceedings of the 2004 ACM workshop on Privacy in the electronic society*, pages 27–27, New York, NY, USA, 2004. ACM Press.

[39] Morrie Gasser, Andy Goldstein, Charlie Kaufman, and Butler Lampson. The digital distributed system security architecture. In *Proc. 12th NIST-NCSC National Computer Security Conference*, pages 305–319, 1989.

[40] Morrie Gasser and Ellen McDermott. An architecture for practical delegation in a distributed system. In *Proceedings of the 1990 IEEE Symposium on Security and Privacy*, pages 20–30. IEEE Computer Society, May 1990.

[41] Craig Gentry and Alice Silverberg. Hierarchical ID-based cryptography. In *Proceedings of the 8th International Conference on the Theory and Application of Cryptology and Information Security*, pages 548–566, London, UK, 2002. Springer-Verlag.

[42] Luigi Giuri and Pietro Iglio. Role templates for content-based access control. In *Proceedings of the Second ACM Workshop on Role-based Access Control*, pages 153–159. ACM Press, 1997.

[43] Virgil D. Gligor, Shyh-Wei Luan, and Joseph N. Pato. On inter-realm authentication in large distributed systems. *Journal of Computer Security*, 2(2–3):137–158, 1993.

[44] Oded Goldreich. *Foundations of Cryptography Volume 2: Basic Applications*. Cambridge University Press, 2004.

[45] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game. In *Proceedings of the 19th annual ACM conference on Theory of computing*, pages 218–229, New York, NY, USA, 1987. ACM Press.

[46] Shafi Goldwasser. Multi party computations: past and present. In *Proceedings of the sixteenth annual ACM symposium on Principles of distributed computing*, pages 1–6, New York, NY, USA, 1997. ACM Press.

[47] Urs Hengartner and Peter Steenkiste. Exploiting hierarchical identity-based encryption for access control to pervasive computing information. Technical Report CMU-CS-04-172, Carnegie Mellon University, October 2004.

[48] Urs Hengartner and Peter Steenkiste. Exploiting hierarchical identity-based encryption for access control to pervasive computing information. In *Proceedings of First IEEE/CreateNet International Conference on Security and Privacy for Emerging Areas in Communication Networks (SecureComm)*, pages 384–393, September 2005.

[49] Karen Henricksen and Jadwiga Indulska. A software engineering framework for context-aware pervasive computing. In *Proceedings of the Second IEEE International Conference on Pervasive Computing and Communications (PerCom'04)*, pages 77–86, Washington, DC, USA, 2004. IEEE Computer Society.

[50] Amir Herzberg, Yosi Mass, Joris Michaeli, Yiftach Ravid, and Dalit Naor. Access control meets public key infrastructure, or: Assigning roles to strangers. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, pages 2–14, Washington, DC, USA, 2000. IEEE Computer Society.

[51] Jeffrey Hightower, Barry Brumitt, and Gaetano Borriello. The Location Stack: A Layered Model for Location in Ubiquitous Computing. In *Proceedings of the Fourth IEEE Workshop on Mobile Computing Systems and Applications*, pages 22–28, Callicoon, New York, June 2002. IEEE Computer Society Press.

[52] John A. Hine, Walt Yao, Jean Bacon, and Ken Moody. An architecture for distributed OASIS services. In *IFIP/ACM International Conference on Distributed Systems Platforms*, pages 104–120. Springer-Verlag New York, Inc., April 2000.

[53] Jason E. Holt, Robert W. Bradshaw, Kent E. Seamons, and Hilarie Orman. Hidden credentials. In *Proceedings of the 2003 ACM workshop on Privacy in the electronic society*, pages 1–8, New York, NY, USA, 2003. ACM Press.

[54] Jon Howell and David Kotz. End-to-end authorization. In *Proceedings of the 2000 Symposium on Operating Systems Design and Implementation*, pages 151–164. USENIX Association, October 2000.

[55] Jon Howell and David Kotz. A formal semantics for SPKI. In *Proceedings of the Sixth European Symposium on Research in Computer Security (ESORICS 2000)*, pages 140–158. Springer-Verlag, October 2000.

[56] GE Hughes and MJ Cresswell. *A New Introduction to Modal Logic*. Routledge, 1996.

[57] R. J. Hulsebosch, A. H. Salden, M. S. Bargh, P. W. G. Ebben, and J. Reitsma. Context Sensitive Access Control. In *Proceedings of the 10th ACM Symposium on Access Control Models and Technologies*, pages 111–119, Baltimore, MD, June 2005.

[58] National incident management system, March 2004. `http://www.fema.gov/pdf/nims/nims_doc_full.pdf`.

[59] Internet2. http://www.internet2.edu. http://www.internet2.edu.

[60] S. Jajodia, P. Samarati, and V. S. Subrahmanian. A logical language for expressing authorizations. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 31–42. IEEE Press, 2001.

[61] Trevor Jim. SD3: A trust management system with certified evaluation. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, pages 106–115. IEEE Computer Society, 2001.

[62] Apu Kapadia, Geetanjali Sampemane, and Roy H. Campbell. KNOW Why your access was denied: regulating feedback for usable security. In *Proceedings of the 11th ACM conference on Computer and Communications Security*, pages 52–61. ACM Press, 2004.

[63] Eleftheria Katsiri and Alan Mycroft. Knowledge representation and scalable abstract reasoning for sentient computing using first-order logic. In *Proceedings of Challenges and Novel Applications for Automatic Reasoning (CADE-19)*, pages 73–87, July 2003.

[64] Scott Knight and Chris Crandy. Scalability issues in pmi delegation. In *Proceedings of the 1st Annual PKI Research Workshop*, pages 77–88. NIST, April 2002.

[65] Hugo Krawczyk, Mihir Bellare, and Ran Canetti. Hmac: Keyed-hashing for message authentication. Internet RFC 2693, February 1997. http://www-cse.ucsd.edu/users/mihir/papers/rfc2104.txt.

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

[67] Lightweight Directory Access Protocol (v3), December 1997. `http://www.ietf.org/rfc/rfc2251.txt`.

[68] Rebekah Lepro. Cardea: Dynamic access control in distributed systems. Technical Report NAS-03-020, NASA Ames Research Center, November 2003.

[69] Jiangtao Li and Ninghui Li. Oacerts: Oblivious attribute certificates. In *Proceedings of 3rd Conference on Applied Cryptography and Network Security*, pages 301–317. Springer, June 2005.

[70] Jiangtao Li and Ninghui Li. Policy-hiding access control in open environment. In *Proceedings of the twenty-fourth annual ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 29–38, New York, NY, USA, 2005. ACM Press.

[71] Jiangtao 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*, pages 46–57, New York, NY, USA, 2005. ACM Press.

[72] Ninghui Li. *Delegation Logic: A Logic-based Approach to Distributed Authorization*. PhD thesis, New York University, September 2000.

[73] Ninghui Li, Wenliang Du, and Dan Boneh. Oblivious signature-based envelope. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 182–189, New York, NY, USA, 2003. ACM Press.

[74] Ninghui Li, Joan Feigenbaum, and Benjamin N. Grosof. A logic-based knowledge representation for authorization with delegation. In *Proceedings of the 1999 IEEE*

*Computer Security Foundations Workshop*, page 162, Washington, DC, USA, 1999. IEEE Computer Society.

[75] Ninghui Li, Benjamin N. Grosof, and Joan Feigenbaum. Delegation logic: A logic-based approach to distributed authorization. *ACM Transactions on Information and System Security*, 6(1):128–171, 2003.

[76] 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 and Privacy*, page 114, Washington, DC, USA, 2002. IEEE Computer Society.

[77] M. Lorch, D. B. Adams, D. Kafura, M. S. R. Koneni, A. Rathi, and S. Shah. The prima system for privilege management, authorization and enforcement in grid environments. In *Proceedings of the Fourth International Workshop on Grid Computing*, page 109, Washington, DC, USA, 2003. IEEE Computer Society.

[78] Markus Lorch, Seth Proctor, Rebekah Lepro, Dennis Kafura, and Sumit Shah. First experiences using xacml for access control in distributed systems. In *Proceedings of the 2003 ACM workshop on XML security*, pages 25–37, New York, NY, USA, 2003. ACM Press.

[79] MACE Middleware Architecture Committee for Education. `http://middleware.internet2.edu/MACE/`.

[80] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay — a secure two-party computation system. In *Proceedings of the 13th Usenix Security Symposium*, pages 287–302, August 2004.

[81] A. Malpani, R. Housley, and T. Freeman. Simple certificate validation protocol (SCVP). Internet Draft, draft-ietf-pkix-scvp-14.txt, April 2004. `http://www.oasis-open.org/committees/download.php/2406/oasis-xamcl-1.0.pdf`.

[82] Christopher P. Masone. Role Definition Language (RDL): A Language to Describe Context-Aware Roles. Technical Report TR2002-426, Dartmouth College, Computer Science, Hanover, NH, May 2002.

[83] Paul J. Mazzuca. Access Control in a Distributed Decentralized Network: An XML Approach to Network Security using XACML and SAML. Technical Report TR2004-506, Dartmouth College, Computer Science, Hanover, NH, Spring 2004.

[84] Silvio Micali. NOVOMODO scalable certificate validation and simplified PKI management. In *Proceedings of the 1st Annual PKI Research Workshop*, pages 15–25, April 2002.

[85] Ghita Kouadri Mostéfaoui and P. Brézillon. A generic framework for context-based distributed authorizations. In *Fourth International and Interdisciplinary Conference on Modeling and Using Context (Context'03)*, pages 204–217. Springer, June 2003.

[86] Kouadri Mostéfaoui. Context-based security policies: A new modeling approach. In *Second IEEE Annual Conference on Pervasive Computing and Communications Workshops*, pages 154–158, March 2004.

[87] M. Myers, R. Ankney, A. Malpani, S. Galperin, and C. Adams. X.509 internet public key infrastructure online certificate status protocol - ocsp, June 1999. `http://www.ietf.org/rfc/rfc2560.txt`.

[88] Ginger Myles, Adrian Friday, and Nigel Davies. Preserving privacy in environments with location-based applications. *IEEE Pervasive Computing*, 2(1):56–64, January-March 2003.

[89] Sidharth Nazareth and Sean Smith. Using SPKI/SDSI for Distributed Maintenance of Attribute Release Policies in Shibboleth. Technical Report TR2004-485, Dartmouth College, Computer Science, Hanover, NH, January 2004.

[90] OpenSSL. `http://www.openssl.org`.

[91] Torben P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *Proceedings of the 11th Annual International Cryptology Conference on Advances in Cryptology*, pages 129–140, London, UK, 1992. Springer-Verlag.

[92] OpenPGP Message Format. `http://www.ietf.org/internet-drafts/draft-ietf-openpgp-rfc2440bis-15.txt`.

[93] PKCS #1: RSA Cryptography Standard. `ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-1/pkcs-1v2-1.pdf`.

[94] G. J. Pottie and W. J. Kaiser. Wireless integrated network sensors. *Communications of the ACM*, 43(5):51–58, 2000.

[95] Anand Ranganathan and Roy H. Campbell. An infrastructure for context-awareness based on first order logic. *Personal Ubiquitous Computing*, 7(6):353–364, 2003.

[96] RFC 1423 - Privacy Enhancement for Internet Electronic Mail: Part III: Algorithms, Modes, and Identifiers, February 1993. `http://www.faqs.org/rfcs/rfc1423.html`.

[97] Ron Rivest and Butler Lampson. SDSI – a simple distributed security architecture. `http://theory.lcs.mit.edu/~cis/sdsi.html`.

[98] Ronald L. Rivest. The MD5 message-digest algorithm, April 1992. http://www.ietf.org/rfc/rfc1321.txt.

[99] Ronald L. Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 26(1):96–99, 1983.

[100] Joseph Rosen, Eliot Grigg, Jaron Lanier, Susan McGrath, Scott Lillibridge, David Sargent, and C.Everett Koop. The Future of Command and Control for Disaster Response. *IEEE Engineering in Medicine and Biology*, 21(5):56–68, September/October 2002.

[101] Security Assertion Markup Language (SAML) 2.0 Technical Overview, July 2004. http://www.oasis-open.org/committees/download.php/13786/sstc-saml-tech-overview-2.0-draft-07-diff.pdf.

[102] SAML V2.0 Executive Overview, April 2005. http://www.oasis-open.org/committees/download.php/13525/sstc-saml-exec-overview-2.0-cd-01-2col.pdf.

[103] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, Feb 1996.

[104] Bill N. Schilit, Norman Adams, and Roy Want. Context-aware computing applications. In *Proceedings of IEEE Workshop on Mobile Computing Systems and Applications*, pages 85–90, Santa Cruz, California, December 1994. IEEE Computer Society Press.

[105] Kent E. Seamons, Marianne Winslett, and Ting Yu. Limiting the disclosure of access control policies during automated trust negotiation. In *Proceedings of the Network and Distributed System Security Symposium*, February 2001.

[106] Shibboleth Project. `http://shibboleth.internet2.edu/`.

[107] Simple Object Access Protocol (SOAP) 1.1, May 2000. `http://www.w3.org/TR/2000/NOTE-SOAP-20000508/`.

[108] Geoffrey H. Stowe. A Secure Network Node Approach to the Policy Decision Point in Distributed Access Control. Technical Report TR2004-502, Dartmouth College, Computer Science, Hanover, NH, June 2004.

[109] Sun's XACML Implementation, January 2005. `http://sunxacml.sourceforge.net`.

[110] Andrew S. Tanenbaum, Robbert van Renesse, Hans van Staveren, Gregory J. Sharp, and Sape J. Mullender. Experiences with the amoeba distributed operating system. *Communications of the ACM*, 33(12):46–63, 1990.

[111] Mary R. Thompson, Abdelilah Essiari, and Srilekha Mudumbai. Certificate-based authorization policy in a pki environment. *ACM Transactions on Information and System Security*, 6(4):566–588, 2003.

[112] Anand Tripathi, Tanvir Ahmed, Devdatta Kulkarni, Richa Kumar, and Komal Kashiramka. Context-based secure resource access in pervasive computing environments. In *Proceedings of the Second IEEE Annual Conference on Pervasive Computing and Communications Workshops*, pages 159–163. IEEE Computer Society, March 2004.

[113] Jeffrey D. Ullman. *Database and Knowledge-Base Systems, Volumn 2*. Computer Science Press, 1989.

[114] Jean Vaucher. XProlog.java: the successor to Winikoff's WProlog, Feb 2003. `http://www.iro.umontreal.ca/~vaucher/XProlog/AA_README`.

[115] Luis von Ahn, Andrew Bortz, and Nicholas J. Hopper. k-anonymous message transmission. In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 122–130, New York, NY, USA, 2003. ACM Press.

[116] Roy Want, Gaetano Borriello, Trevor Pering, and Keith I. Farkas. Disappearing hardware. *IEEE Pervasive Computing*, 1(1):36–47, January-March 2002.

[117] Mark Weiser. The computer for the 21st century. *Scientific American*, 265(3):66–75, January 1991.

[118] W. Winsborough and N. Li. Towards practical automated trust negotiation. In *Proceedings of the 3rd International Workshop on Policies for Distributed Systems and Networks (POLICY'02)*, page 92, Washington, DC, USA, 2002. IEEE Computer Society.

[119] William H. Winsborough and Ninghui Li. Safety in automated trust negotiation. In *Proceedings of the 2004 IEEE Symposium on Security and Privacy*, pages 147–160. IEEE Computer Society, May 2004.

[120] Marianne Winslett, Ting Yu, Kent E. Seamons, Adam Hess, Jared Jacobson, Ryan Jarvis, Bryan Smith, and Lina Yu. Negotiating trust on the web. *IEEE Internet Computing*, 6(6):30–37, 2002.

[121] 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*, pages 168–179, New York, NY, USA, 2005. ACM Press.

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

[123] Thomas Y. C. Woo and Simon S. Lam. Authorization in distributed systems: A formal approach. In *Proceedings of the 1992 IEEE Symposium on Security and Privacy*, page 33, Washington, DC, USA, 1992. IEEE Computer Society.

[124] Thomas Y. C. Woo and Simon S. Lam. Authorization in distributed systems: A new approach. *Journal of Computer Security*, 2(2-3):107–136, 1993.

[125] Thomas Y. C. Woo and Simon S. Lam. Designing a distributed authorization service. In *Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies*, pages 419–429, March 1998.

[126] Chris Wullems, Mark Looi, and Andrew Clark. Towards context-aware security: An authorization architecture for intranet environments. In *Second IEEE Annual Conference on Pervasive Computing and Communications Workshops*. IEEE Computer Society, March 2004.

[127] Internet X.509 Public Key Infrastructure Certificates and CRL Profile, January 1999. `http://www.ietf.org/rfc/rfc2459.txt`.

[128] Security frameworks for open systems: Access control framework. ITU-T Recommendation X.812, 1995. `http://www.itu.int/rec/recommendation.asp?type=folders&lang=e&parent=T-REC-X.812`.

[129] OASIS eXtensible Access Control Markup Language (XACML) Version 2.0, September 2004. http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-core-spec-os.pdf.

[130] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *Proceedings of the 27th Annual Symposium on Foundations of Computer Science*, pages 162–167. IEEE Computer Society Press, October 1986.

[131] Song Ye, Fillia Makedon, and James Ford. Collaborative automated trust negotiation in peer-to-peer systems. In *Fourth International Conference on Peer-to-Peer Computing (P2P'04)*, pages 108–115. IEEE Computer Society, August 2004.

[132] Ting Yu and Marianne Winslett. A Unified Scheme for Resource Protection in Automated Trust Negotiation. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, pages 110–122. IEEE Computer Society, May 2003.

[133] Ting Yu, Marianne Winslett, and Kent E. Seamons. Interoperable strategies in automated trust negotiation. In *Proceedings of the 8th ACM conference on Computer and Communications Security*, pages 146–155, New York, NY, USA, 2001. ACM Press.

[134] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Untrusted hosts and confidentiality: Secure program partitioning. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*. ACM, October 2001.

[135] Peifang Zheng. Tradeoffs in certificate revocation schemes. *ACM SIGCOMM Computer Communication Review*, 33(2):103–112, 2003.