Available online at www.sciencedirect.com

Pervasive and Mobile Computing 1 (2005) 123–156

pervasive and mobile computing

www.elsevier.com/locate/pmc

Secure context-sensitive authorization

Kazuhiro Minami*, David Kotz

Department of Computer Science, Dartmouth College, Hanover, NH 03755, USA

Abstract

There is a recent trend toward rule-based authorization systems to achieve flexible security policies. Also, new sensing technologies in pervasive computing make it possible to define context-sensitive rules, such as "allow database access only to staff who are currently located in the main office". However, these rules, or the facts that are needed to verify authority, often involve sensitive context information. This paper presents a secure context-sensitive authorization system that protects confidential information in facts or rules. Furthermore, our system allows multiple hosts in a distributed environment to perform the evaluation of an authorization query in a collaborative way; we do not need a universally trusted central host that maintains all the context information. The core of our approach is to decompose a proof for making an authorization decision into a set of sub-proofs produced on multiple different hosts, while preserving the integrity and confidentiality policies of the mutually untrusted principals operating these hosts. We prove the correctness of our algorithm.
© 2005 Elsevier B.V. All rights reserved.

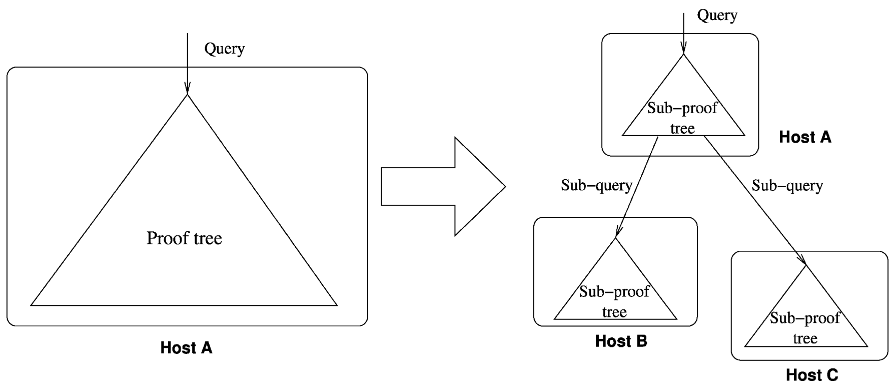
Keywords: Mobile computing; Pervasive computing; Ubicomp; Context-aware; Security; Privacy

1. Introduction

Pervasive computing leads to an increased integration between the real world and the computational world. Many such applications adapt to the user's context, that is, the user's situation and environment. 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

* Corresponding author.
E-mail addresses: minami@cs.dartmouth.edu (K. Minami), dfk@cs.dartmouth.edu (D. Kotz).

1574-1192/$ - see front matter © 2005 Elsevier B.V. All rights reserved.
doi:10.1016/j.pmcj.2005.01.004



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

Fig. 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, Hosts A, B, and C produce only a subtree of the proof.

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.

There are several rule-based authorization systems [3,4,9,18] that allow a resource owner or a manager to define authorization rules that refer to the context of the requester. These 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. An authorization rule that refers to user location, for example, may raise concerns about location privacy [5,10, 11,18], because the information might allow others to infer their activities (e.g., a secret business meeting). A policy that depends on the medical condition of patients must respect HIPAA rules about the confidentiality of medical records [1]. To the best of our knowledge, no previous work addresses the issue of information confidentiality in authorization rules.

We propose a secure, distributed, context-sensitive rule-based authorization system. When a client requests access to a resource, the resource owner constructs a logical statement (query) that, if proven TRUE, indicates that access may be granted; otherwise access is denied. Although the resource's host has a knowledge base containing rules that represent authorization policies and facts about the users, it may not have all of the necessary information and thus collaborates with other hosts to attempt to construct a proof for the query. Thus, rather than depending on a central trusted server (Fig. 1(a)), we decompose a proof into sub-proofs produced by multiple hosts (Fig. 1(b)). 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. We assume that these trust relationships are defined by *principals*, each of which represents a specific user or organization, and that each host is associated with one principal (e.g., the owner of a PDA, or the manager of a server).

Our approach provides several benefits:

Confidentiality: Information used for making an authorization decision is protected according to confidentiality policies defined by the owner of that information.
Integrity: Proofs are evaluated by principals (hosts) that are trusted by the queriers.
Scalability: By distributing the knowledge base and proof construction we off-load work from a resource that may have limited processing or communication capability.

In the following sections, we introduce our authorization rule language and how this language can define integrity and confidentiality policies. Section 4 describes our authorization system for the simpler case, where policies apply only to facts. We describe the architecture of our system and introduce the concept of distributed processing for an authorization query. We next describe our enforcement mechanism for confidentiality policies and give some key algorithms for handling queries in a distributed way. We give an example application at the end of the section. In Section 5, we describe the general case that supports policies on rules as well, following the structure of the preceding section. We describe the representation of a proof and the algorithm that can verify the integrity of the proof. Section 6 proves that our algorithm ensures the integrity and confidentiality policies of the principals constructing an arbitrary proof tree. We discuss related work in Section 7. Section 8 covers some design issues and security properties in our system and Section 9 concludes.

2. Background

In this section, we describe our language for defining authorization policies and introduce the concept of a proof tree, which is constructed when evaluating an authorization query.

2.1. Authorization rule language

In rule-based authorization systems, authorization policies are represented as logical expressions. We express access-control policies with Horn clauses since they are expressive enough to support the rules in existing rule-based authorization systems [3,4,9]. We do not use a general first-order logic, which is not decidable in general. The syntax of a Horn clause is $b \leftarrow a_1 \wedge a_2 \cdots \wedge a_n$, 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 usually used to state a fact. An atom is formed from a predicate symbol followed by a parenthesized list of variables and constants. We can express the fact “Bob is in Hanover” as *location(Bob, Hanover)*, for example.

Rules:

$$grant(P) \leftarrow role(P, operation\_chief) \tag{1}$$
$$role(P, operation\_chief) \leftarrow roleIn(P, police\_chief, police\_dept) \wedge location(P, airport) \tag{2}$$
$$location(P, L) \leftarrow owner(P, D) \wedge location(D, L) \tag{3}$$
$$location(D, L) \leftarrow wifi(D, A) \wedge in(A, L) \tag{4}$$
$$location(D, L) \leftarrow gps(D, X, Y) \wedge closeTo(X, Y, L) \tag{5}$$

Facts:

$roleIn(bob, police\_chief, police\_dept)$.	Bob is chief of the local police department.	(6)
$owner(bob, pda15)$.	Bob owns device pda15.	(7)
$wifi(pda15, ap39)$.	pda15 is associated with access point ap39.	(8)
$in(ap39, airport)$.	Access point ap39 is at the airport.	(9)

Fig. 2. Sample set of rules. We use uppercase for variables and lowercase for constants and names.

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 [13] defines clear roles for the many participants in a large-scale response, so role-based access control (RBAC) [19] 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.

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. Fig. 2 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 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 Fig. 2, we can construct the proof tree shown in Fig. 3 by unifying the query with the first four rules, substituting variables as needed. We return

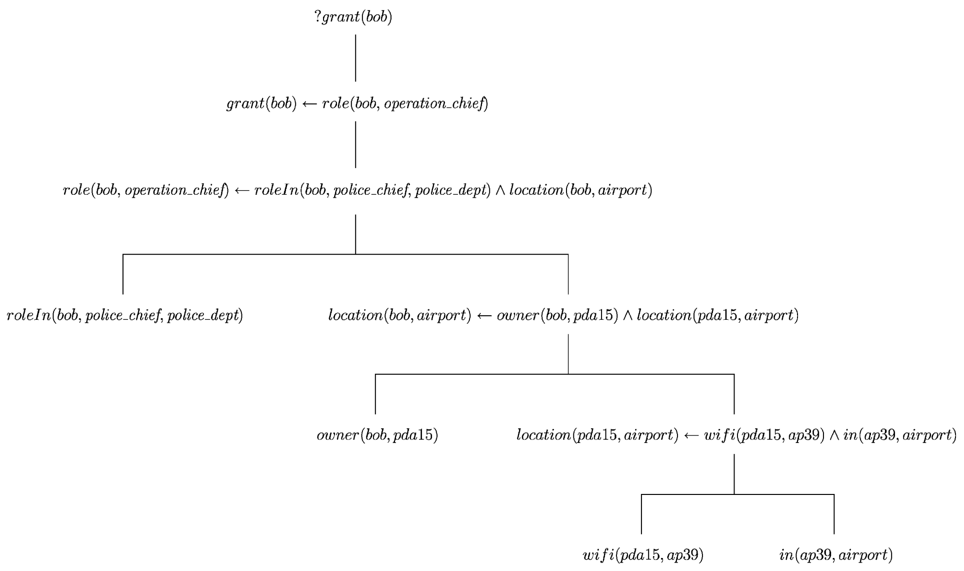

?grant(bob)
grant(bob) ← role(bob, operation_chief)
role(bob, operation_chief) ← roleIn(bob, police_chief, police_dept) ∧ location(bob, airport)
roleIn(bob, police_chief, police_dept)
location(bob, airport) ← owner(bob, pda15) ∧ location(pda15, airport)
owner(bob, pda15)
location(pda15, airport) ← wifi(pda15, ap39) ∧ in(ap39, airport)
wifi(pda15, ap39)
in(ap39, airport)

Fig. 3. Example proof tree based on the rules in Fig. 2.

to this example in Sections 4.6 and 5.6 to explain how we construct this proof in a distributed fashion.

3. Security policies

Each principal defines *confidentiality policies* to protect information in its knowledge base. It also defines *integrity policies* to specify whether it believes that evaluation results or rules received from other principals are correct.

3.1. Rule patterns

We first introduce the notion of *rule patterns*, which are mechanisms for expressing these security policies in our security model. 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. A rule pattern is associated with a set of rules or facts that match it through *unification*, a pattern-matching 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 $role(P, physician) \leftarrow occupation(P, physician) \wedge location(P, hospital)$ by instantiating X to P and Y to *physician*.

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.

3.2. Integrity policies

Integrity policies express trust in the correctness of rules and facts. Our definition is based on the information flow theory [6] whose focus is confidence on the accuracy of information rather than alternation of information. When a principal p_i defines the integrity policy (rp, t) it means that p_i trusts those principals in t, which we often 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 the query $loc(bob, hanover)$ is replaced 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 proof as we describe in Section 5.1.

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

3.3. 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 $location(bob, hanover)$.

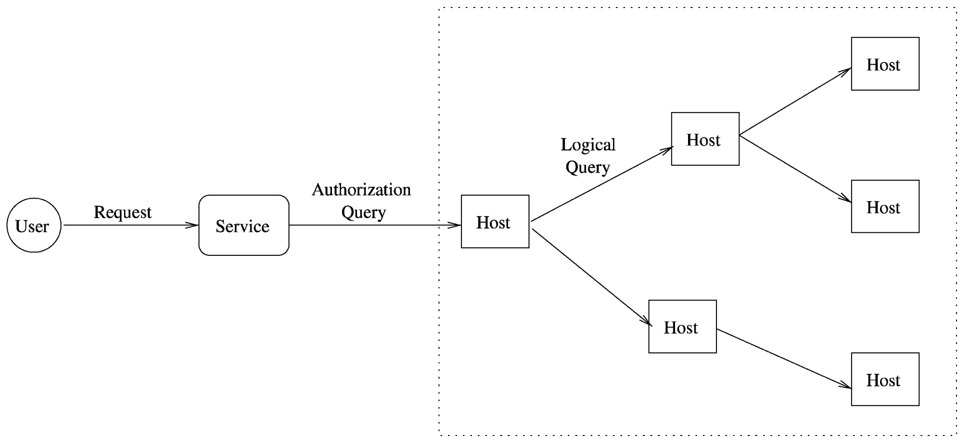


Fig. 4. Architectural overview. The hosts enclosed in the dotted lines make an authorization decision in a collaborative way.

3.4. Assumptions

In this paper we make a few assumptions to maintain our focus on the confidentiality and integrity issues in distributed context-sensitive authorization systems. First, the integrity and confidentiality policies of each principal are public knowledge. Second, 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. Third, we assume that there is a directory service that knows which principal handles what types of queries.

For purposes of simplifying our explanation, we consider the basic case that supports security policies only on facts first in Section 4, and then the general case that supports security policies on facts and rules in Section 5.

4. Authorization for the basic case

In this section, we describe our authorization system for the basic case that supports security policies only on facts.

4.1. 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 Fig. 4, 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.

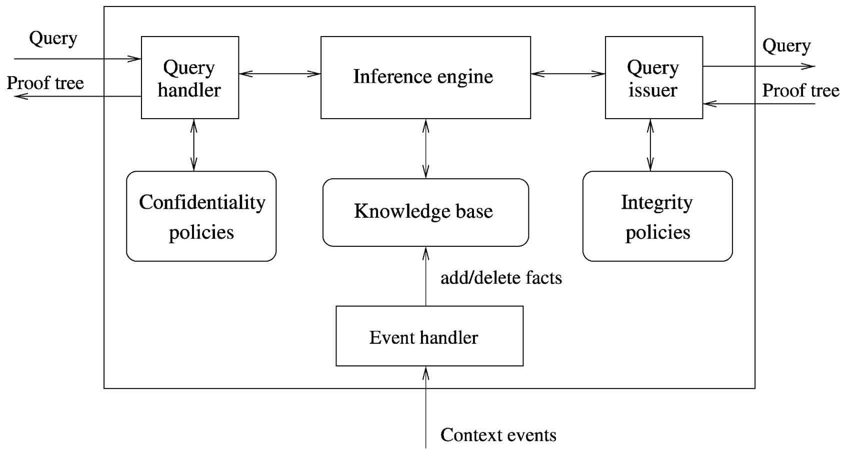


Fig. 5. Structure of a host.

The structure of a host is shown in Fig. 5. 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 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 response and checks its integrity based on the integrity policies. The event handler 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 [7].

4.2. *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. 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. We omit the field of a nonce n in the proof object for brevity in the following discussion. 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 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 5, 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. 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. The digital signature of a proof signed by a handler principal ensures *nonreputability* 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 $pf_{r(i)}$ (for which $r(i) = 0$) is *FALSE*, p_0 returns a simple proof $(p_r, (FALSE)_{K_r})$. Otherwise, if there are some subproofs that p_0 cannot decrypt (because $r(i) \neq 0$), then principal p_0 returns the proof $(p_r, (\Pi_i(p_{r(i)}, (value_i)_{K_{r(i)}}))_{K_r})$ for all $r(i) \neq 0$, as a response to query q_0. 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*.

4.3. Decomposition of a proof tree

When a querier issues a query to a principal that the querier trusts with the integrity of evaluating the 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.

Fig. 6 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

Fig. 7 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 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

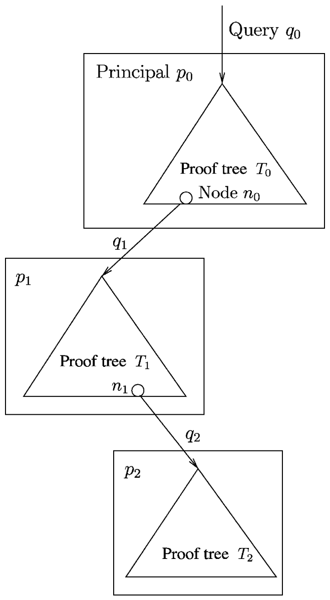


Fig. 6. Decomposed proof tree. Principals p_0, p_1, and p_2 construct a proof tree for query q_0 in a distributed way. Nodes n_0 and n_1 are leaf nodes of proof trees T_0 and T_1 respectively. Principal p_0 that handles query q_0 issues query q_1 to principal p_1 to obtain the fact in node n_0, and principal p_1 similarly issues query q_2 to principal p_2.

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).

4.4. 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. 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 $\mathit{rep}(n)$, and

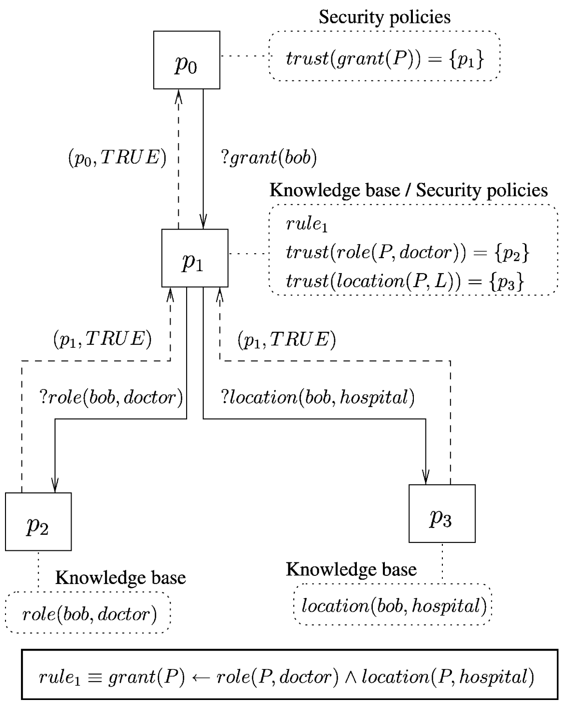

Security policies
p_0
$trust(grant(P)) = \{p_1\}$
$(p_0, TRUE)$
$?grant(bob)$
Knowledge base / Security policies
p_1
$rule_1$
$trust(role(P, doctor)) = \{p_2\}$
$trust(location(P, L)) = \{p_3\}$
$(p_1, TRUE)$
$(p_1, TRUE)$
$?role(bob, doctor)$
$?location(bob, hospital)$
p_2
p_3
Knowledge base
$role(bob, doctor)$
Knowledge base
$location(bob, hospital)$
$rule_1 \equiv grant(P) \leftarrow role(P, doctor) \wedge location(P, hospital)$

Fig. 7. Example of distributed query processing. 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. The definition of $rule_1$ is enclosed in the rectangle at the bottom of the figure.

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 $receivers(p)$. In Fig. 8, 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 $\langle p_0, p_1, p_2\rangle$, for example.

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

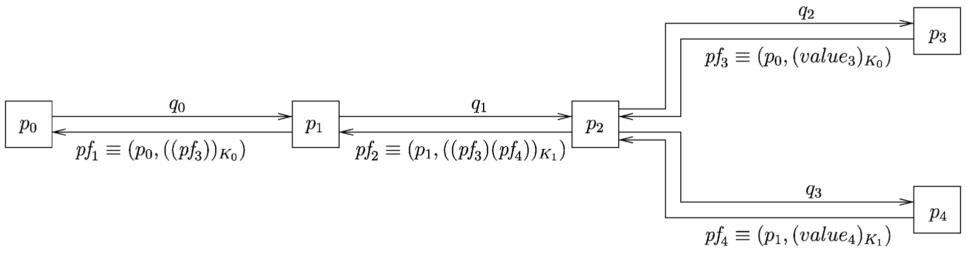

q_0
q_1
q_2
q_3
p_0
p_1
p_2
p_3
p_4
$pf_1 \equiv (p_0, ((pf_3))_{K_0})$
$pf_2 \equiv (p_1, ((pf_3)(pf_4))_{K_1})$
$pf_3 \equiv (p_0, (value_3)_{K_0})$
$pf_4 \equiv (p_1, (value_4)_{K_1})$

Fig. 8. Enforcement of confidentiality policies. 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.

This recursive encryption is necessary to prevent a 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 encryptions 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 Fig. 8, principal p_3 chooses p_0 as the receiver of proof $pf_3 \equiv (p_0, (value_3)_{K_0})$ 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 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. 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. Because principal p_1 decrypts and evaluates the proof pf_4, p_1 only embeds pf_3 into proof pf_1, which is decrypted by principal p_0, if the evaluation of pf_4 is *TRUE*. (Otherwise, p_2 drops the proof pf_3 and returns a proof that contains a *FALSE* value.) If principal p_2 chooses p_0 as the receiver of proof pf_2 instead, 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 $\langle p_0, \ldots, p_{i-1} \rangle$, 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 important 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.

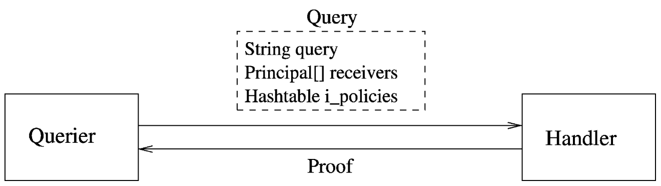


Fig. 9. Query interface.

4.5. Algorithms

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 4.4, and a querier principal's integrity policies *i_policies*, as shown in Fig. 9. The function HANDLEREMOTEQUERY calls the function GENERATEPROOF to obtain a proof.

Fig. 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. The algorithm handles the simpler case that a proof from a remote principal contains a query result (not concatenated subproofs). We describe how to handle concatenated subproofs in Section 5.5. The function takes several parameters: principal p_0 that issues a query, principal p_1 that handles a query, a query string q, 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. If p_0 is an initial querier, it includes itself in 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 where 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 $minIndex(s, receivers)$ in line 5.

Line 7 checks whether the handler principal p_1 satisfies the querier p_0's integrity policies (we use the symbol '|' to denote "such as" in our algorithm for brevity). 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 where query q matches the head of rule r in p_1's knowledge base. Line 12 unifies query q and rule $r \equiv A \leftarrow B_1, \ldots, B_n$, resulting in the instantiated rule $A' \leftarrow B'_1, \ldots, B'_n$. Lines 13–14 obtain subproofs for the subqueries $B'_1, \ldots, B'_n$ iteratively. If principal p_1 can decrypt all the values in the subproofs, and all the subproofs contain a *TRUE* value, then line 16 returns a proof with a *TRUE* value to principal p_r. Line 17 checks whether the subproofs decrypted by p_0 contain a *TRUE* value, and if so,

```
GENERATEPROOF($p_0, p_1, q, receivers, i\_policies_0, i\_policies_1, c\_policies_1, KB_1$)
1   ▷ 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 $s = \emptyset$ ▷ if set $s$ is empty.
4      then return $(p_0, (REJECT)_{K_0})$
5   $p_r \leftarrow minIndex(s, receivers)$
6   ▷ 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 $(p_r, (FALSE)_{K_r})$
9   if $\exists$ fact $f \mid ((f \in KB_1) \wedge (f$ matches $q))$
10     then return $(p_r, (TRUE)_{K_r})$
11  elseif $\exists$ rule $r \equiv A \leftarrow B_1, \ldots, B_n \mid ((r \in KB_1) \wedge (A$ matches $q))$
12     then unify $q$ and $A \leftarrow B_1, \ldots, B_n$, resulting in $A' \leftarrow B'_1, \ldots, B'_n$
13          for $i \leftarrow 1$ to $n$
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 a receiver principal of $pf_i$
15          if $\forall i\ ((pf_i = (p_1, (value_i)_{K_1})) \wedge (value_i = TRUE))$
16             then return $(p_r, (TRUE)_{K_r})$
17          elseif $\forall i\ ((pf_i = (p_{r(i)}, (value_i)_{K_{r(i)}})) \wedge (((r(i) \neq 1) \vee ((r(i) = 1) \wedge (value_i = TRUE))))$
18             then if $\exists\ p_{r'} \mid (\forall i\ (((p_{r'} \in s) \wedge (index(p_{r(i)}, receivers) \leq index(p_{r'}, receivers)) \wedge (r(i) \neq 1))$
                          $\vee (r(i) = 1)))$
19                    then return $(p_{r'}, (\Pi_i\ pf_i)_{K_{r'}})$
                             for all $i$ where $pf_i = (p_{r(i)}, (value_i)_{K_{r(i)}}) \wedge (r(i) \neq 1)$
20  ▷ If we fail to construct a proof that derives the query locally, we try to obtain a proof from a remote principal.
21  if $\exists$ principal $p_l$ ($\exists$ policy $p = (rp, t)$ $((p \in i\_policies_1) \wedge (rp$ matches $q) \wedge (p_l \in t)))$
22     then append $p_1$ to $receivers$
23          $proof \leftarrow$ ISSUEREMOTEQUERY($p_l, q, receivers, i\_policies_1$)
24          return $proof$
25     else return $(p_r, (FALSE)_{K_r})$
```

Fig. 10. Algorithm for generating a proof.

line 18 checks whether there is some principal $p_{r'}$ that satisfies the constraint due to the recursive encryption we describe in Section 4.4; 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 19 returns a proof containing the subproofs whose values could not be decrypted by p_1 with principal $p_{r'}$ as the recipient.

When lines 7–19 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 21–25. Line 21 checks whether there is any principal p_l that satisfies p_1's integrity policies for query q. If that holds true, line 22 appends p_1 into the ordered list *receivers*, and line 23 calls the function ISSUEREMOTEQUERY. Line 24 returns the returned proof. If line 21 fails to find such a principal p_l, then line 25 returns a proof with a *FALSE* value.

4.6. Example application

Consider again our initial example of an incident management system (IMS) shown in Fig. 2; a centralized server would produce the proof tree in Fig. 3. Fig. 11 shows how user *bob* (principal p_0) requests images from the surveillance camera image server managed by

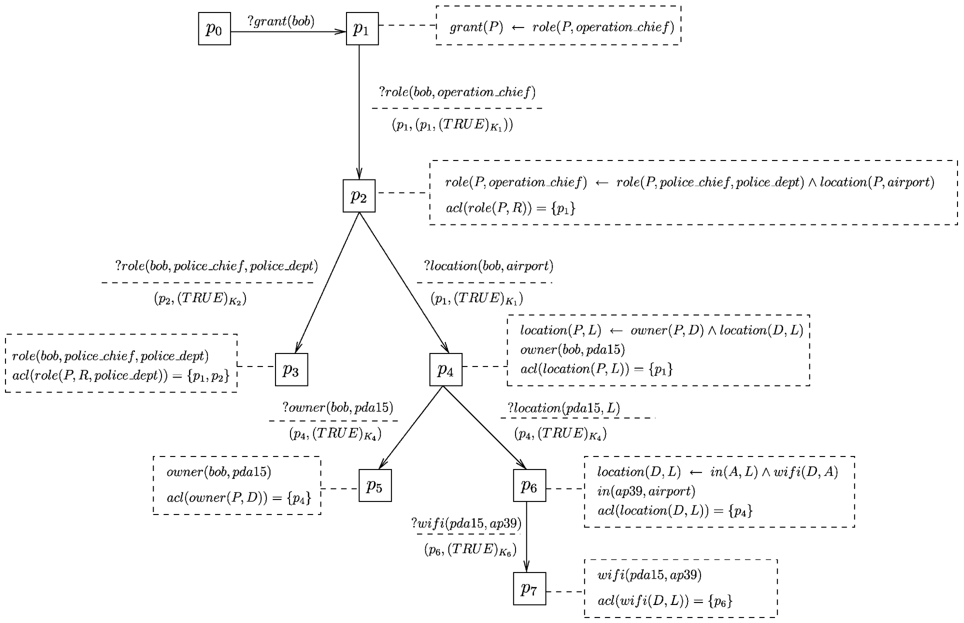

p_0
$?grant(bob)$
p_1
$grant(P) \leftarrow role(P, operation\_chief)$
$?role(bob, operation\_chief)$
$(p_1, (p_1, (TRUE)_{K_1}))$
p_2
$role(P, operation\_chief) \leftarrow role(P, police\_chief, police\_dept) \wedge location(P, airport)$
$acl(role(P, R)) = \{p_1\}$
$?role(bob, police\_chief, police\_dept)$
$(p_2, (TRUE)_{K_2})$
$?location(bob, airport)$
$(p_1, (TRUE)_{K_1})$
$role(bob, police\_chief, police\_dept)$
$acl(role(P, R, police\_dept)) = \{p_1, p_2\}$
p_3
p_4
$location(P, L) \leftarrow owner(P, D) \wedge location(D, L)$
$owner(bob, pda15)$
$acl(location(P, L)) = \{p_1\}$
$?owner(bob, pda15)$
$(p_4, (TRUE)_{K_4})$
$?location(pda15, L)$
$(p_4, (TRUE)_{K_4})$
$owner(bob, pda15)$
$acl(owner(P, D)) = \{p_4\}$
p_5
p_6
$location(D, L) \leftarrow in(A, L) \wedge wifi(D, A)$
$in(ap39, airport)$
$acl(location(D, L)) = \{p_4\}$
$?wifi(pda15, ap39)$
$(p_6, (TRUE)_{K_6})$
p_7
$wifi(pda15, ap39)$
$acl(wifi(D, L)) = \{p_6\}$

Fig. 11. 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.

the airport (principal p_1). Bob's request is handled by multiple principals $p_1, p_2, \ldots, p_7$. In Fig. 11, 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.

5. Authorization for the general case

In this section, we extend our authorization scheme so that it supports security 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 the handler's rule that is used to decompose the query into subqueries. We describe the integrity of a proof tree, the representation of the proof that contains a proof tree, and the enforcement mechanisms for confidentiality and integrity policies respectively.

5.1. *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.

5.2. *Representation of a proof*

We represent a *proof* using nested parentheses based on the grammar in Fig. 12. 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.

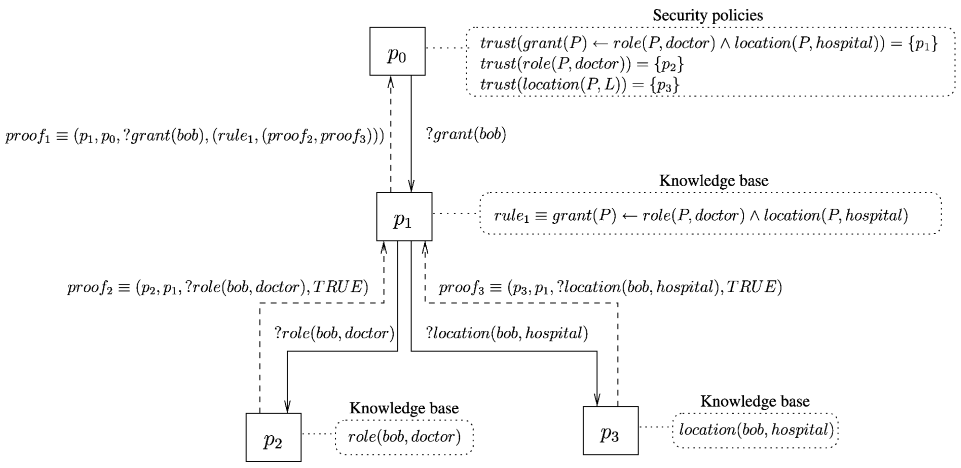
The hierarchical structure of a proof tree is built by embedding subproofs into a proof recursively. That is, the proof contains a proof tree that consists of a root node (representing a rule) of the proof tree and the subproofs that contain the subproof trees under the root node. Therefore, each node 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 4.4. The digital signature of a proof is attached with the proof so that a receiver principal can check its authenticity. It also ensures *non-reputability* of the sender principal. When a proof tree is a single-node one, the field for a proof tree contains a query result (value). 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 4.2.

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

Fig. 12. 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 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 Fig. 12 for brevity. The receiver's identity is necessary when a proof tree is encrypted by the receiver's public key as we discuss in Section 4.4. 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.



Security policies
$trust(grant(P) \leftarrow role(P, doctor) \wedge location(P, hospital)) = \{p_1\}$
$trust(role(P, doctor)) = \{p_2\}$
$trust(location(P, L)) = \{p_3\}$
p_0
$proof_1 \equiv (p_1, p_0, ?grant(bob), (rule_1, (proof_2, proof_3)))$
$?grant(bob)$
Knowledge base
$rule_1 \equiv grant(P) \leftarrow role(P, doctor) \wedge location(P, hospital)$
p_1
$proof_2 \equiv (p_2, p_1, ?role(bob, doctor), TRUE)$
$proof_3 \equiv (p_3, p_1, ?location(bob, hospital), TRUE)$
$?role(bob, doctor)$
$?location(bob, hospital)$
p_2
Knowledge base
$role(bob, doctor)$
p_3
Knowledge base
$location(bob, hospital)$

Fig. 13. 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.

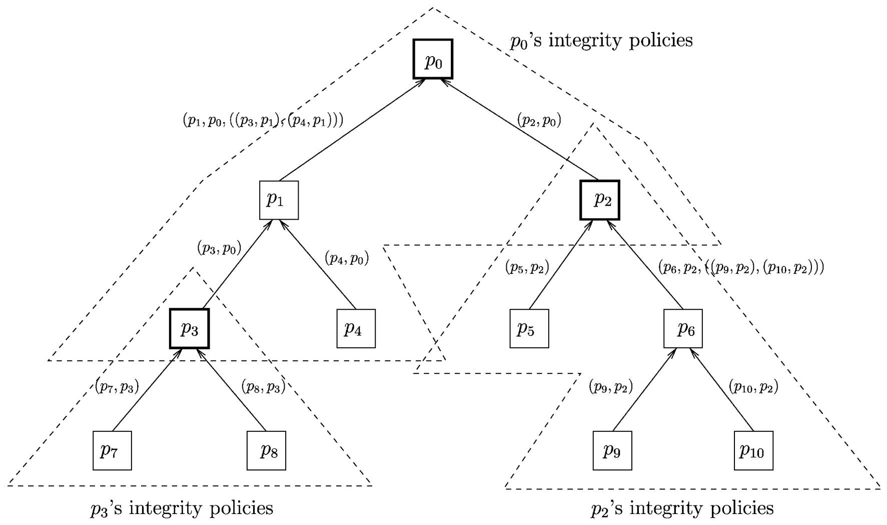
Example

The example in Fig. 13 is a modification of Fig. 7. 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.

5.3. *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 Fig. 14, 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 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



p_0's integrity policies
p_0
$(p_1, p_0, ((p_3, p_1), (p_4, p_1)))$
(p_2, p_0)
p_1
p_2
(p_3, p_0)
(p_4, p_0)
(p_5, p_2)
$(p_6, p_2, ((p_9, p_2), (p_{10}, p_2)))$
p_3
p_4
p_5
p_6
(p_7, p_3)
(p_8, p_3)
(p_9, p_2)
(p_{10}, p_2)
p_7
p_8
p_9
p_{10}
p_3's integrity policies
p_2's integrity policies

Fig. 14. 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.

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.

5.4. *Enforcement of confidentiality policies*

We apply the same mechanism for enforcing confidentiality policies in Section 4.4. 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.
- Principal p_l 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, and there is no other principal p_j (where $l < j < i$), that satisfies this condition.

Notice that our new definition does not change the definition of *receivers*(p) in Section 4.4, 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).

5.5. 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 Fig. 15, we extend the algorithm in Fig. 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 5.2. We use the parameter name *rcvrs* instead of *receivers* 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 n obtained by calling the function GENERETEPROOF 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 n, 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 where principal p_1 cannot decrypt all the query results in the proofs pf_i for $i = 1$ to n 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.

Third, we handle the case where 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 21–27. Line 21 checks whether there is a rule $R \equiv A \leftarrow B_1 \wedge \cdots \wedge B_n$ 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 a rule R, line 22 checks whether querier p_0 has an integrity policy $p = (rp', t')$ that trusts the integrity of p_1's rule R. Line 23 unifies query q and rule R resulting $R' \equiv A' \leftarrow B'_1, \ldots, B'_n$. Lines 24–25 obtain the proofs for the atoms $B'_1, \ldots, B'_n$ iteratively. Line 26 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 4.4. If that holds true, we return the proof that contains rule R' as the root node of the proof tree, and the proofs for $B'_1, \ldots, B'_n$ as the subproofs under the root

```
GENERATEPROOF($p_0, p_1, q, n, rcvrs, i\_policies_0, i\_policies_1, c\_policies_1, KB_1$)
1  ▷ 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 $(rp_i, t_i) \in c\_policies_1$ where $rp_i$ matches $q$
3  if $s = \emptyset$ ▷ if set $s$ is empty.
4      then return $(p_1, p_0, q, n, (REJECT)_{K_0})$
5  $p_r \leftarrow minIndex(s, rcvrs)$
6  ▷ Check whether principal $p_1$ satisfies querier $p_0$'s integrity policies.
7  if $\exists$ policy $p = (rp, t) \mid ((p \in i\_policies_0) \wedge (rp$ matches $q) \wedge (p_1 \in t))$
8      then append $p_1$ to rcvrs
9           if $\exists$ fact $f \mid ((f \in KB_1) \wedge (f$ matches $q))$
10             then return $(p_1, p_r, q, n, (TRUE)_{K_r})$
11          elseif $\exists$ rule $r \equiv A \leftarrow B_1, \ldots, B_n \mid ((r \in KB_1) \wedge (A$ matches $q))$
12             then unify $q$ and $A \leftarrow B_1, \ldots, B_n$, resulting in $A' \leftarrow B'_1, \ldots, B'_n$
13                  for $i \leftarrow 1$ to $n$
14                      do $pf_i \leftarrow$ GENERATEPROOF($p_1, p_1, B'_i, n, rcvrs, i\_policies_1, i\_policies_1, c\_policies_1, KB_1$)
                            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 $\forall i\ ((r(i) = 1) \wedge (((pt_i = value_i) \wedge (value_i = TRUE))$
                      $\vee((pt_i = \Pi_j(p_{r(i,j)}, (value_{ij})_{K_{r(i,j)}})) \wedge \forall j\ ((r(i,j) = 1) \wedge (value_{ij} = TRUE)))))$
16                    then return $(p_1, p_r, q, n, (TRUE)_{K_r})$
17                  elseif $\forall i\ ((r(i) \neq 1) \vee ((r(i) = 1) \wedge (((pt_i = value_i) \wedge (value_i = TRUE))$
                      $\vee ((pt_i = \Pi_j(p_{r(i,j)}, (value_{ij})_{K_{r(i,j)}})) \wedge (\forall j\ ((r(i,j) \neq 1) \vee ((r(i,j) = 1) \wedge (value_{ij} = TRUE)))))))$
18                    then if $\exists\ p_{r'}\ ((p_{r'} \in s) \wedge (\forall i\ ((r(i) = 1) \vee\ (((pt_i = value_i)$
                            $\wedge(index(p_{r(i)}, rcvrs) \leq index(p_{r'}, rcvrs))) \vee ((pt_i = \Pi_j(p_{r(i,j)}, (value_{ij})_{K_{r(i,j)}}))$
                            $\wedge(\forall j\ ((r(i,j) = 1) \vee (index(p_{r(i,j)}, rcvrs) \leq index(p_{r'}, rcvrs))))))))$
19                          then return $(p_1, p_{r'}, q, n, ((\Pi_i\ (p_{r(i)}, pt_i))(\Pi_{ij}\ (p_{r(i,j)}, (pt_{ij})_{K_{r(i,j)}})))_{K_{r'}})$
                                where $((pf_i = (p_{s(i)}, p_{r(i)}, B'_i, n, (pt_i)_{K_{r(i)}}) \wedge (r(i) \neq 1))$
                                $\vee((r(i) = 0) \wedge (pt_i = \Pi_j(p_{r(i,j)}, (value_{ij})_{K_{r(i,j)}})) \wedge r(i,j) \neq 1))$
20 ▷ Construct a proof with a rule that satisfies principal $p_0$'s integrity policies and $p_1$'s confidentiality policies.
21 if $(\exists$ rule $R \mid ((R \in KB_1) \wedge (R \equiv A \leftarrow B_1 \wedge \ldots \wedge B_n) \wedge(A$ matches $q)))$
     $\wedge(\exists$ policy $p \mid ((p \in c\_policies_1) \wedge (p = (rp, t)) \wedge rp$ matches rule $R)))$
22     then if $\exists$ policy $p' = (rp', t') \mid ((p' \in i\_policies_0) \wedge (rp'$ matches $R) \wedge (p_1 \in t'))$
23          then unify $q$ and rule $R$ resulting $R' \equiv A' \leftarrow B'_1, \ldots, B'_n$
24               for $i \leftarrow 1$ to $n$
25                   do $pf_i \leftarrow$ GENERATEPROOF($p_1, p_1, B'_i, n, rcvrs, i\_policies_0, i\_policies_1, c\_policies_1, KB_1$)
                         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 $pf_i$
26               if $\exists p_{r'}\ ((p_{r'} \in s) \wedge (\forall i\ (index(p_{r(i)}, rcvrs) \leq index(p_{r'}, rcvrs))))$
27                 then return $(p_1, p_{r'}, q, n, ((R', p_c), \Pi_i\ pf_i)_{K_{r'}})$ where $p_c$ is a signer principal of rule $R$
28 ▷ If we fail to construct a proof that derives the query locally, we try to obtain a proof from a remote principal.
29 if $\exists$ principal $p_l$ that is capable of handling query $q$
30     then if $p_1 \in rcvrs$
31          then proof ← ISSUEREMOTEQUERY($p_l, q, rcvrs, i\_policies_1$)
32               $(trusted, proof')$ ← CHECKPROOFINTEGRITY($p_1, q, n, proof, i\_policies_1$)
33               if trusted
34                 then return proof'
35          else proof ← ISSUEREMOTEQUERY($p_l, q, rcvrs, i\_policies_0$)
36               return proof
37 return $(p_1, p_r, q, n, (FALSE)_{K_r})$
```

Fig. 15. Algorithm for generating a proof.

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

```
CHECKPROOFINTEGRITY($p_c, q, n_c, pf, i\_policies_c$)
1  if $\neg((pf = (p_s, p_r, q, n, (pt)_{K_r})) \wedge (n_c = n))$
2     then return ($false, NULL$)
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 ($true, pf$)
5  elseif $((r = c) \wedge (pt = ((R, p_d), (\Pi_{i=1}^{n}\ pf_i))$
      where $R$ is a rule, $p_d$ is the signer principal of $R$, and $pf_i$ 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 $R \equiv A \leftarrow B_1 \wedge \ldots \wedge B_n$
7             then for $i \leftarrow 1$ to $n$
8                   do $(trust, pf'_i)$ = CHECKPROOFINTEGRITY($p_c, B_i, pf_i, i\_policies_c$)
9                      if $\neg trust$
10                        then return ($false, NULL$)
11                 return ($true, (p_c, p_c, q, n, (\Pi_i\ pf'_i)_{K_c})$)
12          else return ($false, NULL$)
13 else return ($false, NULL$)
```

Fig. 16. Algorithm for checking proof integrity.

handler principal p_1. Line 30 checks that condition by checking whether p_1 belongs to *rcvrs*, because line 8 appends p_1 to *rcvrs* if the condition holds. If that holds true, line 32 issues a remote query with p_1's integrity policies $i\_policies_1$. That is, p_1's integrity policies are applied to the succeeding queries. Line 32 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 32 returns the proof returned by the function CHECKPROOFINTEGRITY. If p_1 does not belong to *rcvrs*, line 35 issues a remote query with principal p_0's integrity policies; that is, the querier principal's integrity policies are applied to the succeeding queries. Line 36 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.

Algorithm for checking the integrity of a proof

The function CHECKPROOFINTEGRITY in Fig. 16 checks whether a proof satisfies given integrity policies, based on the definition given in Section 5.1. 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 nonce n for the query. 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

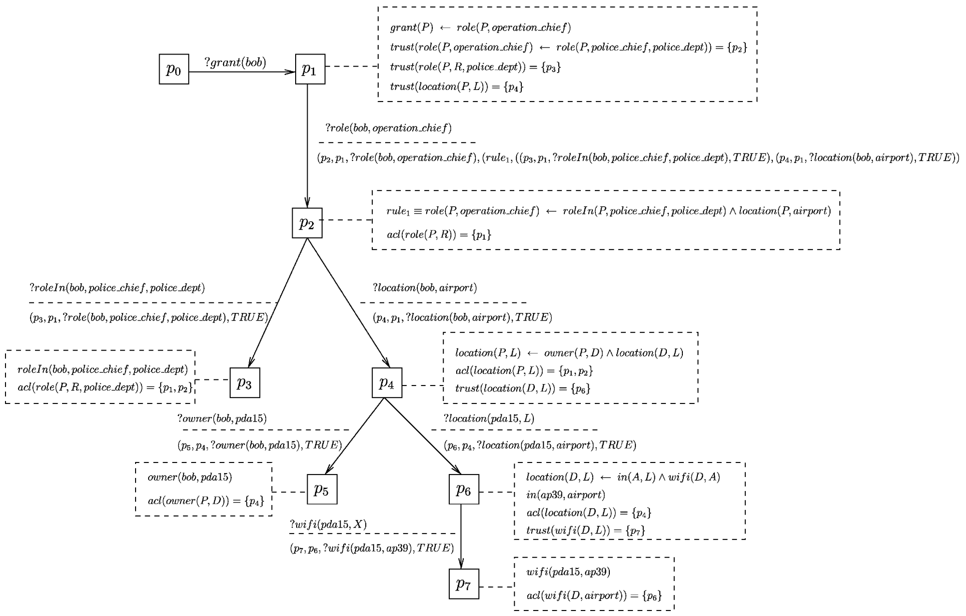

p_0
$?grant(bob)$
p_1
$grant(P) \leftarrow role(P, operation\_chief)$
$trust(role(P, operation\_chief) \leftarrow role(P, police\_chief, police\_dept)) = \{p_2\}$
$trust(role(P, R, police\_dept)) = \{p_3\}$
$trust(location(P, L)) = \{p_4\}$
$?role(bob, operation\_chief)$
$(p_2, p_1, ?role(bob, operation\_chief), (rule_1, ((p_3, p_1, ?roleIn(bob, police\_chief, police\_dept), TRUE), (p_4, p_1, ?location(bob, airport), TRUE))$
p_2
$rule_1 \equiv role(P, operation\_chief) \leftarrow roleIn(P, police\_chief, police\_dept) \wedge location(P, airport)$
$acl(role(P, R)) = \{p_1\}$
$?roleIn(bob, police\_chief, police\_dept)$
$(p_3, p_1, ?role(bob, police\_chief, police\_dept), TRUE)$
$?location(bob, airport)$
$(p_4, p_1, ?location(bob, airport), TRUE)$
$roleIn(bob, police\_chief, police\_dept)$
$acl(role(P, R, police\_dept)) = \{p_1, p_2\}$
p_3
p_4
$location(P, L) \leftarrow owner(P, D) \wedge location(D, L)$
$acl(location(P, L)) = \{p_1, p_2\}$
$trust(location(D, L)) = \{p_6\}$
$?owner(bob, pda15)$
$(p_5, p_4, ?owner(bob, pda15), TRUE)$
$?location(pda15, L)$
$(p_6, p_4, ?location(pda15, airport), TRUE)$
$owner(bob, pda15)$
$acl(owner(P, D)) = \{p_4\}$
p_5
p_6
$location(D, L) \leftarrow in(A, L) \wedge wifi(D, A)$
$in(ap39, airport)$
$acl(location(D, L)) = \{p_4\}$
$trust(wifi(D, L)) = \{p_7\}$
$?wifi(pda15, X)$
$(p_7, p_6, ?wifi(pda15, ap39), TRUE)$
p_7
$wifi(pda15, ap39)$
$acl(wifi(D, airport)) = \{p_6\}$

Fig. 17. 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.

(i.e., principal p_r is a receiver principal of the proof pf) and reads rule R at the root of the proof tree. 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 satisfy 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.

5.6. Example application

We revisit the example of an incident management system (IMS); in Fig. 11, 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.

Fig. 17 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, $role(P, operation\_chief) \leftarrow$

$role(P, police\_chief, police\_dept) \wedge in(P, airport)$ is correct, and 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.

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 in Section 5, which covers the basic case in Section 4 as its special case. We separate the proof into two parts: the proof on confidentiality policies, and the proof on integrity policies.

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.

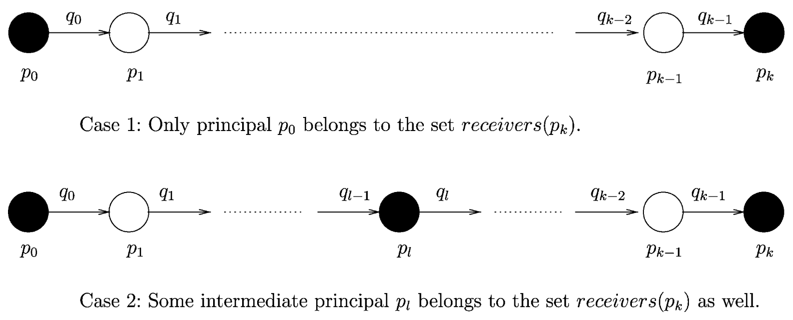
Base case

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 Fig. 15 ensures this condition in line 3. Therefore, principals p_0 and p_1 construct a proof only if their confidentiality 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. (The base case above considers a tree of depth 0.) Without loss of generality, we consider the case where a proof tree is linear. Because our algorithm for enforcing confidentiality policies on each node depends only on the nodes

[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.



q_0
q_1
q_{k-2}
q_{k-1}
p_0
p_1
p_{k-1}
p_k
Case 1: Only principal p_0 belongs to the set $receivers(p_k)$.
q_0
q_1
q_{l-1}
q_l
q_{k-2}
q_{k-1}
p_0
p_1
p_l
p_{k-1}
p_k
Case 2: Some intermediate principal p_l belongs to the set $receivers(p_k)$ as well.

Fig. 18. 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.

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 Fig. 18. In case 1, only principal p_0 belongs to a set of principals *receivers*(p_k) defined in Section 5.4. 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), 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; 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 Fig. 15 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 $p_0, \dots, p_{l-1}$ 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 $p_0, \dots, p_{l-1}$. 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 where 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 where 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 $n_1, \dots, n_{k-1}$ in the same way as the case where 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 $p_1, \dots, p_{k-1}$ for choosing a receiver principal of their proofs from the set $\textit{receivers} = \{p_0, p_l\}$ or $\{p_0\}$. The depth of the tree with nodes $n_1, \dots, n_{k-1}$ is $k - 1$. Therefore, by the induction hypothesis, our algorithm ensures that a proof tree is constructed only when the confidentiality policies of principals $p_1, \dots, p_{k-1}$ 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 where 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 are satisfied. We cover all the cases in terms of confidentiality policies and conclude the proof.

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 policies 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 Fig. 15 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 polices 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 where a proof tree is linear as we do in Section 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 given integrity policies 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 5.1, 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 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 Fig. 18.

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 Fig. 15. The proof from p_l does not contain any other encrypted nodes because p_l needs to read the nodes $n_{l+1}, \dots, n_{k-1}$ to check the integrity of the proof from p_{l+1}. Principal p_l checks whether the rules in nodes $n_{l+1}, \dots, n_{k-1}$ satisfies p_l's integrity policies, which is done in line 6 of the function CHECKPROOFINTEGRITY in Fig. 16. If principal p_l cannot decrypt all the nodes $n_{l+1}, \dots, n_{k-1}$, 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 $p_0, \dots, p_{l-1}$ 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 $p_0, \dots, p_{l-1}$ 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 $p_l, \dots, p_k$. 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 $n_{l+1}, \dots, n_{k-1}$ in that proof. Therefore, principals $p_{l+1}, \dots, p_{k-1}$ must choose p_l as the receiver principal of their returning proofs. Principal $p_{l+1}, \dots, p_{k-1}$ work in the same way as the case where 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.

7. Related work

Although others have developed context-sensitive authorization systems, they all use a trusted central context server that collects context information, and they do not address the protection of context information used in authorization rules or facts. Cerberus [3] 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. Generalized RBAC (GRBAC) [8,9] 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. OASIS [4,12] 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 centralized object-relational database that stores context predicates. Myles [18] provides an XML-based authorization language for defining privacy policies that protect users' location information. Users must trust a set of validators that collect context information and make authorization decisions.

SD3 [15] is an inference engine for a trust management system that 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 a whole proof tree is constructed on a central server. Therefore, all the remote hosts must trust the central server to preserve the confidentiality policies of their facts.

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) [17] 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.

8. Discussion

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

8.1. Completeness of our algorithm

The algorithm of the function GENERATEPROOF in Figs. 10 and 15 is not complete. That is, it does not guarantee to find a proof that derives a granting decision, because when the function finds a proof that contains encrypted subproofs from other principals, it stops searching other proofs. If the returned proof turns out to be invalid because some encrypted subproof derives *false*, or because the evaluation is impossible due to tight integrity or confidentiality policies, our algorithm fails to find a possibly existing proof with some other combination of rules and facts. To address this problem, we need to modify our algorithm so that it continues to search for another proof from the point of the search space where a previous proof is found.

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 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 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 specified by the last index of the *receivers* list.

We use a nonce to prevent a reply attack by a malicious principal that is capable of intercepting and modifying a message. All the participating principals that evaluate an authorization query use the same nonce 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. Complexity of policy definition

Although it seems difficult for each principal to define confidentiality and integrity policies for rules and facts, it is possible for a principal to refer to the policies of other principals to reduce the administrative work for defining policies. For example, principal p_0 could define a meta-rule that says "if principal p_1 trusts the integrity of the evaluation of a query q by principal p_2, then p_0 trusts the integrity of q in the same way". This meta-rule would allow most users to defer on many policies to a trusted administrator, for example.

When principals consider trust relations in terms of the confidentiality and integrity policies *transitive*, it is possible for each principal to expand its confidentiality and integrity policies automatically while collaborating with other principals to construct proof trees. Here we assume that the integrity and confidentiality policies of each principal are public knowledge as stated in Section 3.4.

We first describe how each principal expands its integrity policies by issuing a query to another principal. Suppose that principal p_0 issues a query q_0 ($?grant(p)$) to principal p_1 and p_1 issues a subsequent query q_1 ($?a(x)$) to p_2. If principal p_0 trusts p_1's integrity for evaluating query q_0 (i.e., $p_1 \in trust_0(grant(p))$), p_0 also trusts p_2's integrity for evaluating query q_1 implicitly. Therefore, p_0 should update its integrity policy for query q_1 such that $trust_0(a(x)) = trust_0(a(x)) \cup trust_1(a(x))$. Principal p_0 could obtain p_1's integrity policy $trust_1(a(x))$ with query q_0's result from p_1. The handler principal p_1 actually returns the integrity policy on the rule pattern that is matched with the query q_0; that is, if principal p_1 handling query *grant*(*bob*) unifies query q_0 with rule $grant(P) \leftarrow a(P)$ and issues a remote query $?a(bob)$, p_1 returns $trust_1(a(P))$ so that a querier can update its policies on rule pattern $a(P)$ rather than on its instance $a(bob)$. If a handler principal obtains integrity policies from its downstream principals in the same way, it forwards those integrity policies to its querier so that they are shared among the principals involved in constructing a proof tree.

We can apply the same idea to update confidentiality policies of each principal. For example, suppose that principal p_1 issues a query q_1 ($?a(x)$) to principal p_2. Principal p_2 returns a query result if p_1 satisfies p_2's confidentiality policies (i.e., $p_1 \in acl_2(a(x))$). If p_1 allows another principal p_0 to disclose q_1's result based on its policy ($p_0 \in acl_1(a(x))$),

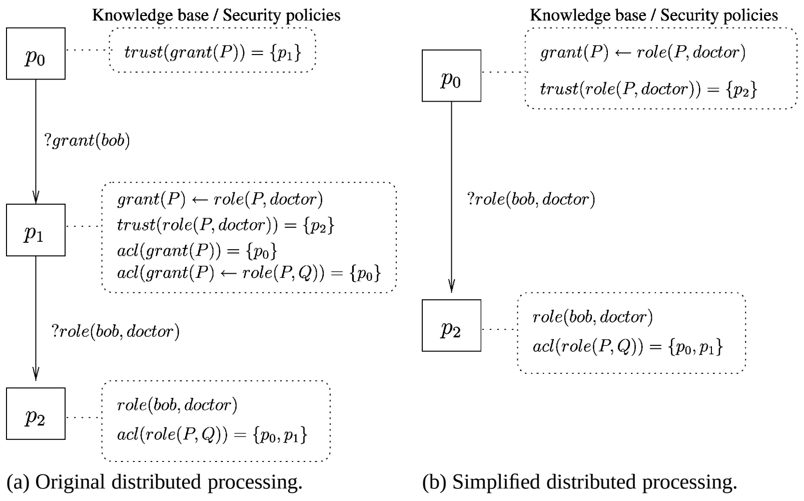

Knowledge base / Security policies
p_0
$trust(grant(P)) = \{p_1\}$
$?grant(bob)$
p_1
$grant(P) \leftarrow role(P, doctor)$
$trust(role(P, doctor)) = \{p_2\}$
$acl(grant(P)) = \{p_0\}$
$acl(grant(P) \leftarrow role(P, Q)) = \{p_0\}$
$?role(bob, doctor)$
p_2
$role(bob, doctor)$
$acl(role(P, Q)) = \{p_0, p_1\}$
(a) Original distributed processing.
Knowledge base / Security policies
p_0
$grant(P) \leftarrow role(P, doctor)$
$trust(role(P, doctor)) = \{p_2\}$
$?role(bob, doctor)$
p_2
$role(bob, doctor)$
$acl(role(P, Q)) = \{p_0, p_1\}$
(b) Simplified distributed processing.

Fig. 19. Simplified distributed processing of a proof tree. Principal p_0 replicates rule $grant(P) \leftarrow role(P, doctor)$ from p_1 and integrity policy $trust(role(P, doctor)) = \{p_2\}$ from principal p_1 into its repository, and can in the future make queries directly to p_2.

then p_2 should also allow p_0 to disclose it; principal p_2 updates its confidentiality policies such that $acl_2(a(x)) = acl_2(a(x)) \cup acl_1(a(x))$.

8.4. Scalability

When many principals are involved in constructing a proof tree that contains a lot of rules and facts, the communication overhead (including work for security operations such as the verification of digital signatures) could cause long latency. Although we leave the experiments to evaluate the scalability of our system as our future work, we discuss several possible solutions to address this issue. First, each querier could set a timeout period to cancel a query request, and the querier interprets the occurrence of a timeout event as that the query result is *false*.

Second, although our algorithm described in Section 5.5 chooses a querier in a depth-first manner, we could modify our algorithm so that a querier principal can choose the handler principal that is most likely to reply with the minimum latency from a list of principals capable of handling the query. Because we believe that most authorization granting services and other query services are long running, it may be possible for each principal to choose a good handler principal based on the logs of the latency of the past queries. It may also be helpful to issues the same query to multiple principals in parallel.

Third, there are some situations where we can reduce the number of principals involved in constructing a proof tree by replicating rules, facts, and security policies aggressively while preserving each principal's security policies. We give a small example in Fig. 19 that shows how the distributed processing is simplified. In Fig. 19, principal p_0 issues query $?grant(bob)$ to p_1, which issues subsequent query $role(bob, doctor)$ to p_2. The three principals p_0, p_1, and p_2 are involved in the original query processing in Fig. 19(a).

Because principal p_0 trusts the integrity of p_1's query result for ?*grant*(*bob*), it can also trust the integrity of rule $grant(P) \leftarrow role(P, doctor)$, which is unified with the query. In addition, when principal p_0 considers trust relations in integrity policies transitive, p_0 trusts the integrity of fact *role*(*bob*, *doctor*) maintained by principal p_2. Therefore, p_0 replicates p_1's rule and integrity policy $trust(role(P, doctor)) = \{p_2\}$ into its repository, and, as a result, the query processing is simplified as in Fig. 19(b) if principal p_2 allows p_0 to read the query result of ?$role(P, Q)$.

8.5. 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.

$$\begin{aligned} &grant(P, purchase, X) \leftarrow approved(mgr, P, X, t_1),\\ &\qquad approved(senior\_mgr, P, X, t_2), prior(t_1, t_2). \end{aligned}$$

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.

Our language does not express *separation of duty* [2] in role-based access control (RBAC) model [19], because to express separation of duty with a logic language (as Jajodia [14] 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 a false positive, which is not acceptable to any authorization system.

8.6. 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, the user might not have sufficient privileges to receive the incomplete proof, and, as a result, the user is not able to know that the 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 [16], which is a centralized rule-based authorization system, proposes to use 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.

9. Summary and future work

We describe a secure context-sensitive authorization system that supports the decentralized construction and evaluation of authorization decisions, involving multiple principals from different administrative domains, and respects the confidentiality and integrity policies of each principal involved.

We define our security model based on the notion of *rule patterns* that allow each principal to define confidentiality and integrity policies on the rules and facts in its knowledge base. Because our system evaluates an authorization query on multiple evaluation nodes in a distributed way, it is possible for each principal to choose to which principal it is willing to disclose the information needed to evaluate the authorization query. We describe our key algorithms and prove that our algorithms guarantee that the proof for an authorization query is constructed only if the security policies of each participating principals are satisfied.

Our current prototype system is implemented in Java, by extending XProlog [20] with a feature to construct a proof for a query instead of simply evaluating the query and returning a result. We plan to deploy our current implementation in realistic large-scale applications and to evaluate the performance and scalability of our system. We also plan to explore various optimization techniques such as caching and parallel search for a proof to improve the performance and the scalability of the system. Another possible extension of our system is to add some mechanism for giving user feedback as we discuss in Section 8.6.

Acknowledgments

We thank the anonymous reviewers whose valuable comments and suggestions helped improve the quality of this paper. We are also grateful to Guanling Chen, for providing helpful feedback on earlier drafts of this paper. This project of the Center for Mobile Computing and the Institute for Security Technology Studies at Dartmouth College was supported under Award No. 2000-DT-CX-K001 from the Office for Domestic Preparedness, U.S. Department of Homeland Security. Points of view in this document are those of the author(s) and do not necessarily represent the official position of the U.S. Department of Homeland Security.

References

[1] Summary of HIPAA privacy rule, 2004. http://www.hhs.gov/ocr/privacysummary.pdf.
[2] G.-J. Ahn, R. Sandhu, The RSL99 language for role-based separation of duty constraints, in: RBAC'99: Proceedings of the fourth ACM Workshop on Role-based Access Control, ACM Press, 1999, pp. 43–54.
[3] J. Al-Muhtadi, A. Ranganathan, R. Campbell, D. Mickunas, Cerberus: a context-aware security scheme for smart spaces, in: Proceedings of the First IEEE International Conference on Pervasive Computing and Communications, March, IEEE Computer Society, 2003, pp. 489–496.
[4] J. Bacon, K. Moody, W. 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) (2002) 492–540.
[5] A.R. Beresford, F. Stajano, Location privacy in pervasive computing, IEEE Pervasive Computing 2 (1) (2003) 46–55.
[6] K. Biba, Integrity considerations for secure computer systems, Technical report 76-372, U.S. Air Force Electronic Systems Division, 1977.

[7] G. Chen, M. Li, D. 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), August 2004, pp. 246–255.

[8] M.J. Covington, M. Ahamad, S. Srinivasan, A security architecture for context-aware applications, Technical report GIT-CC-01-12, Georgia Institute of Technology, May 2001.

[9] M.J. Covington, W. Long, S. Srinivasan, A.K. Dey, M. Ahamad, G.D. Abowd, Securing context-aware applications using environment roles, in: Proceedings of the Sixth ACM Symposium on Access Control Models and Technologies, ACM Press, 2001, pp. 10–20.

[10] M. Gruteser, D. Grunwald, Anonymous usage of location-based services through spatial and temporal cloaking, in: Proceedings of Mobisys 2003: The First International Conference on Mobile Systems, Applications, and Services, May, USENIX Associations, San Francisco, CA, 2003.

[11] U. Hengartner, P. Steenkiste, Access control to information in pervasive computing environments, in: Proc. of 9th Workshop on Hot Topics in Operating Systems, HotOS IX, May 2003, pp. 157–162.

[12] J.A. Hine, W. Yao, J. Bacon, K. Moody, An architecture for distributed OASIS services, in: IFIP/ACM International Conference on Distributed Systems Platforms, April, Springer-Verlag, New York, 2000, pp. 104–120.

[13] National incident management system (coordination draft), 2004. http://www.dhs.gov/dhspublic/interweb/assetlibrary/NIMS-90-web.pdf.

[14] S. Jajodia, P. Samarati, V.S. Subrahmanian, A logical language for expressing authorizations, in: Proceedings of the 1997 IEEE Symposium on Security and Privacy, IEEE Press, 2001, pp. 31–42.

[15] T. Jim, SD3: A trust management system with certified evaluation, in: Proceedings of the IEEE Symposium on Security and Privacy, IEEE Computer Society, 2001, pp. 106–115.

[16] A. Kapadia, G. Sampemane, R.H. Campbell, KNOW Why your access was denied: regulating feedback for usable security, in: CCS'04: Proceedings of the 11th ACM Conference on Computer and Communications Security, ACM Press, 2004, pp. 52–61.

[17] A. Malpani, R. Housley, 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.

[18] G. Myles, A. Friday, N. Davies, Preserving privacy in environments with location-based applications, IEEE Pervasive Computing 2 (1) (2003) 56–64.

[19] R.S. Sandhu, E.J. Coyne, H.L. Feinstein, C.E. Youman, Role-based access control models, IEEE Computer 29 (2) (1996) 38–47.

[20] J. Vaucher, XProlog.java: the successor to Winikoff's WProlog, February 2003. http://www.iro.umontreal.ca/~vaucher/XProlog/AA_README.