Secure Context-sensitive Authorization

Kazuhiro Minami and David Kotz
Department of Computer Science, Dartmouth College
{minami, dfk}@cs.dartmouth.edu

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.

1 Introduction

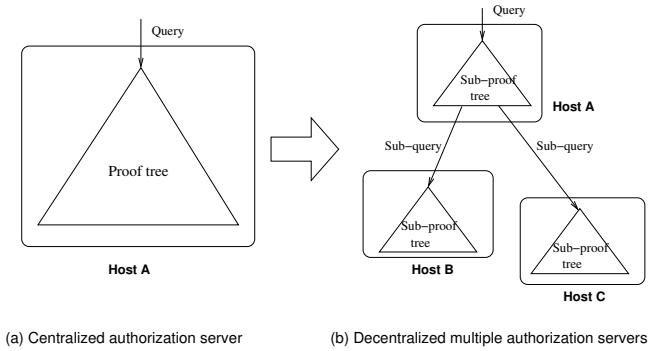
Pervasive computing leads to an increased integration between the real world and the computational world. Many such applications adapt to the user's context, that is, the user's situation and environment. We consider a class of applications that wish to consider a user's context when deciding whether to authorize a user's access to important physical or 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 [1, 2, 5, 11] 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.

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 (Figure 1a), we decompose a proof into sub-proofs produced by multiple hosts (Figure 1b). This collaboration is only possible if the querier can trust the integrity of other hosts (to provide correct facts and to properly evaluate rules) and if the other hosts can trust the querier with confidential facts. 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 access-control policies defined by the owner of that information.



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

Figure 1. Decentralized evaluation of an authorization query. The proof of a query is decomposed into sub-proofs and produced on distributed multiple hosts. On the left, Host A generates a whole proof on a centralized server. On the right, Host A, B, and C produce only a subtree of the proof.

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 our system architecture and introduce the concept of distributed processing for an authorization query. We next describe the enforcement mechanism for confidentiality policies and the algorithm for constructing a proof in a distributed way. In Section 5, we extend our system to allow policies about rules. We discuss related work in Section 6. Section 7 covers the security assurance and policy issues in our system. Section 8 describes our current status and future work, 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 [1, 2, 5]. We do not use a general first-order logic, which is not decidable in general. The syntax of a Horn clause is $b \leftarrow a_1 \wedge a_2 \ldots \wedge a_n$, which says that simple statements called *atoms* a_1 through a_n, if all true, imply b. The atom b is called the *head* 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.

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 [7] defines clear roles for the many participants in a large-scale response, so role-based access control (RBAC) [12] 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. Figure 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 or not with a given set of rules and facts. The

Rules:

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

Facts:

$roleIn(bob, \textsf{police\_chief}, \textsf{police\_dept}).$ 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)

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

proof tree consists of nodes that represent rules (or facts) and edges that represent the unification of the atom in the body of the rule in a parent node with the head of the rule in a child node. Every leaf node contains a fact that has no atom in its body.

Given the facts listed in Figure 2, we can construct the proof tree shown in Figure 3 by unifying the query with the first four rules, substituting variables as needed. We return to this example in Section 4.6 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. A rule pattern is used to define a policy for any 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 chooses. 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. 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, prin-

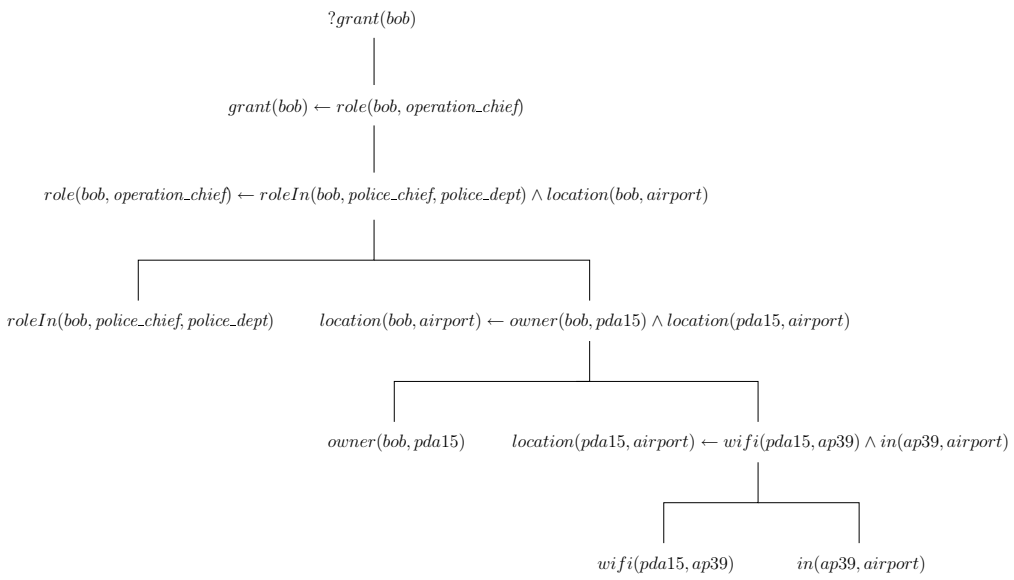


Figure 3. Example proof tree based on the rules in Figure 2.

cipal 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 tree as we describe in Section 2.2.

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

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 $grant(P) \leftarrow loc(bob, \textit{sudikoff})$ 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 $location(bob, hanover)$.

4 Authorization on the basic security model

In this section, we describes our authorization system for the simpler case, where policies apply only to facts. We make a few assumptions to maintain our focus on the confidentiality and integrity issues in distributed context-sensitive authorization systems. First, the integrity 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.

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 Figure 4,

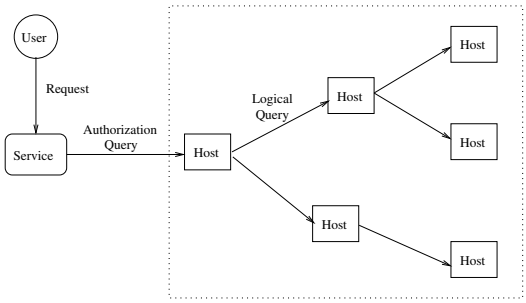


Figure 4. Architectural overview. The hosts enclosed in the dotted lines make an authorization decision in a collaborative way.

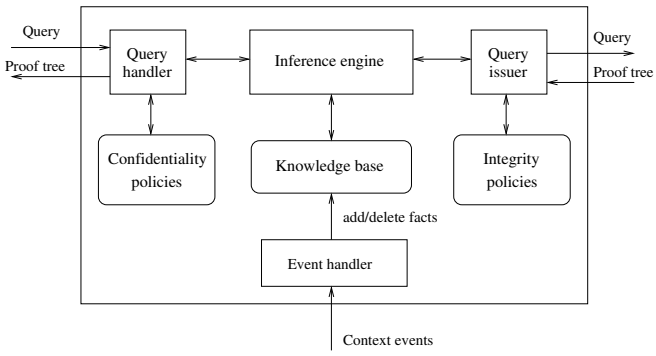


Figure 5. Structure of a host.

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.

The structure of a host is shown in Figure 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 [3].

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 extended model, 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.

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,

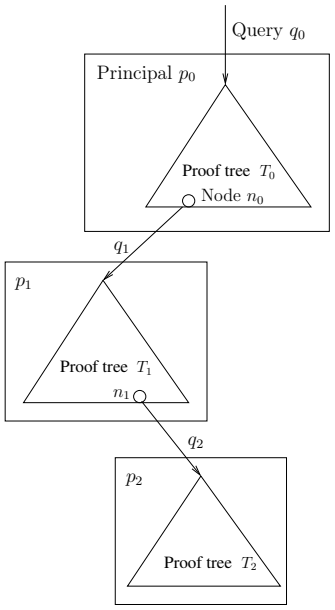


Figure 6. Decomposed proof tree. Principals p_0, p_1, and p_2 construct a proof tree for query q_0 in a distributed way. Nodes n_0 and n_1 are leaf nodes of proof trees T_0 and T_1 respectively. Principal p_0 that handles query q_0 issues query q_1 to principal p_1 to obtain the fact in node n_0, and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

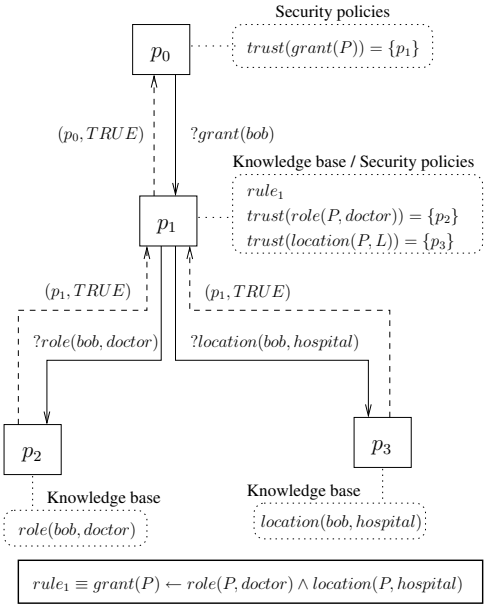


Figure 7. Example of distributed query processing. The solid arrows are labeled with queries and the dashed arrows are labeled with returned proofs. The rounded rectangles with dotted lines represent the knowledge bases and security policies of those principals respectively. The definition of $rule_1$ is enclosed in the rectangle at the bottom of the figure.

the 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.

Figure 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. Figure 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 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 proof/response up the tree; eventually the receiver principal may decrypt the result and compute the conjunction to see whether the tree is *true*.

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

The list of upstream principals for p is defined as $rep(upstream\_nodes(n))$, which we denote as $receivers(p)$. In Figure 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 $< p_0, p_1, p_2 >$, 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. 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 encryption on embedded subproofs are decrypted when the proof reaches the receiving principals of the subproofs. Otherwise, the embedded subproofs pass the receiving principals without being decrypted, and the proof fails.

In Figure 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 return a proof that contains a *FALSE* value.) If principal p_2 chooses p_0 as the receiver of proof pf_2 instead, the proof pf_4, which is embedded in proof pf_2, is forwarded to p_0 without being decrypted by p_1 and the proof is not usable by p_0.

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

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

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.

4.5 Algorithm

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 $i\_policies$. The function HANDLEREMOTEQUERY calls the function GENERATEPROOF to obtain a proof.

Figure 9 shows the algorithm for the function GENERATEPROOF, run on p_1's host to build a proof while

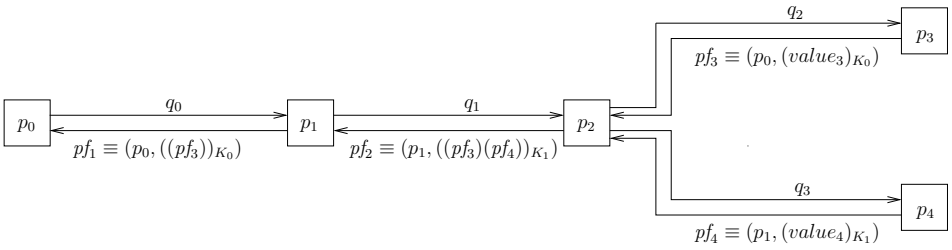


Figure 8. Enforcement of confidentiality policies. Principal p_0's query q_0 is handled by principals p_1, p_2, p_3, and p_4 in a distributed way. Principal p_i handles query q_{i-1}, and returns the proof pf_i, for $i = 1$ to 4.

enforcing confidentiality policies of the handler principal. The algorithm covers the simpler case that a proof from a remote principal contains a query result (not concatenated subproofs) due to the lack of space. 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.

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

Line 5 sets the receiver principal of a proof in the case that the query result in the proof is obtained locally. The chosen receiver is that 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 that query q matches the head of rule r in p_1's knowledge base. Line 12 unifies query q and rule $r \equiv A \leftarrow B_1, \ldots, B_n$, 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, 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. Notice that principal p_1's integrity policies $i\_policies_1$ is given as the parameter. 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 Figure 2; a centralized server would produce the proof tree in Figure 3. Figure 10 shows how user *bob* (principal p_0) requests images from the surveillance camera image server managed by the airport (principal p_1). Bob's request is handled by multiple principals $p_1, p_2, \ldots, p_7$. In Figure 10, 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

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** $(p_0, (\textit{REJECT})_{K_0})$
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** $(p_r, (\textit{FALSE})_{K_r})$
9 **if** $\exists$ fact $f \mid ((f \in KB_1) \wedge (f$ matches $q))$
10 **then return** $(p_r, (\textit{TRUE})_{K_r})$
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** $\forall i\ ((pf_i = (p_1, (value_i)_{K_1})) \wedge\ (value_i = \textit{TRUE}))$
16 **then return** $(p_r, (\textit{TRUE})_{K_r})$
17 **elseif** $\forall i\ ((pf_i = (p_{r(i)}, (value_i)_{K_{r(i)}})) \wedge (((r(i) \neq 1) \vee ((r(i) = 1) \wedge (value_i = \textit{TRUE}))))$
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** $(p_r, (\textit{FALSE})_{K_r})$

Figure 9. Algorithm for generating a proof.

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, and 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 on the extended security model

In this section, we describe how we extend the authorization system for the basic model in Section 4 to support security policies on rules. Due to the page limitation, we cover the major features to be added briefly. See our technical report [10] for its complete description.

Even if the result of evaluating a local proof tree is *true*, the result returned to a proof that contains a proof tree instead of simply the result *TRUE*. This would happen 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. When the querier receives a proof object, it checks the integrity of its proof tree by checking the integrity of all the nodes published by different principals.

The evaluation of a proof tree is performed by the principals whose query results are trusted by their queriers. If there are multiple such principals participating in evaluating an authorization query, the whole proof tree is decomposed into several subtrees and is evaluated by those principals in a distributed way. We enforce confidentiality policies as we describe in Section 4.4, except that a receiver principal must be an upstream principal that evaluates a proof subtree. Again, our report [10] has the details.

6 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 [1] 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) [4, 5] 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 [2, 6] 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 [11] provides a 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 [8] 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) [9] 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.

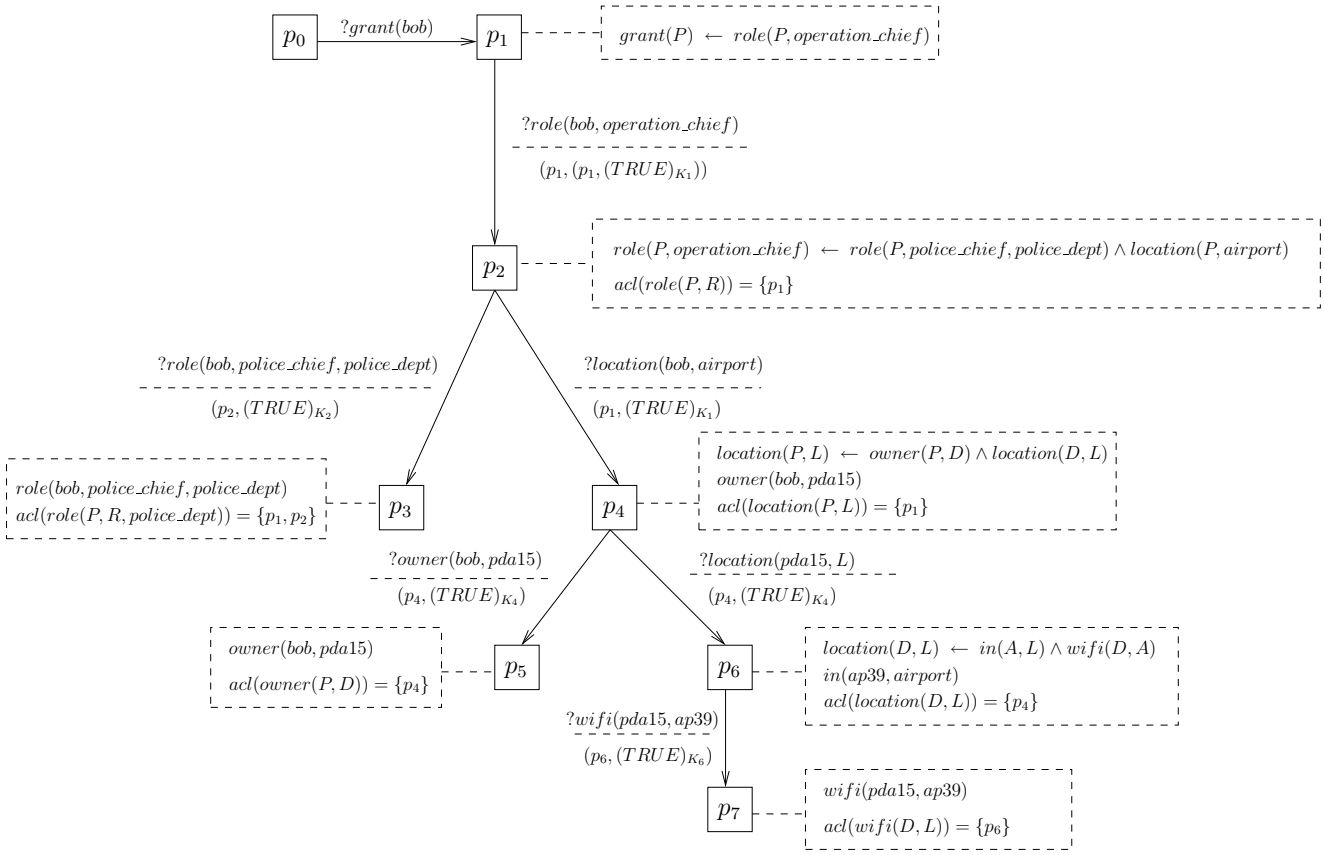


Figure 10. Example of an emergency response system. Principal p_0 is a first responder whose role is "operation_chief". Principal p_1 represents a surveillance camera image server. Principal p_2 is the role membership server of an incident management system (IMS). Principal p_3 is the role membership 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.

7 Discussion

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 4.

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.

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 q in the same way."

8 Current status and future work

Our current prototype system is implemented in Java, by extending XProlog [13] 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.

9 Summary

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 hosts run by different principals 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.

References

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

[2] J. Bacon, K. Moody, and 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):492–540, 2002.

[3] G. Chen, M. Li, and 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)*, pages 246–255, Aug. 2004.

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

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

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

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

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

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

[10] K. Minami and D. Kotz. Secure context-sensitive authorization. Technical Report TR2004-529, Dept. of Computer Science, Dartmouth College, December 2004.

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

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

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