Scalability in a Secure Distributed Proof System

Kazuhiro Minami and David Kotz

Department of Computer Science, Dartmouth College,
Hanover, NH, USA 03755
{minami, dfk}@cs.dartmouth.edu

Abstract. A logic-based language is often adopted in systems for pervasive computing, because it provides a convenient way to define rules that change the behavior of the systems dynamically. Those systems might define rules that refer to the users' context information to provide context-aware services. For example, a smart-home application could define rules referring to the location of a user to control the light of a house automatically. In general, the context information is maintained in different administrative domains, and it is, therefore, desirable to construct a proof in a distributed way while preserving each domain's confidentiality policies. In this paper, we introduce such a system, a secure distributed proof system for context-sensitive authorization and show that our novel caching and revocation mechanism improves the performance of the system, which depends on public key cryptographic operations to protect confidential information in rules and facts. Our revocation mechanism maintains dependencies among facts and recursively revokes across multiple hosts all the cached facts that depend on a fact that has become invalid. Our initial experimental results show that our caching mechanism, which maintains both positive and negative facts, significantly reduces the latency for handling a logical query.

1 Introduction

One of the major goals of pervasive computing is to meet a user's continuously changing requirements without taking explicit input from the users. Therefore, a system in pervasive computing needs to consider the user's context and change its behavior dynamically based on a set of rules. Many systems [7, 10, 19, 22] in pervasive computing apply a logic-based language to express those rules, since it also makes it possible to define a context model where a contextual fact is expressed with a boolean predicate. Besides defining triggering actions [22] of pervasive applications (e.g., a smart meeting room), a logical language provides a way to infer high-level context information [13, 19], such as a user's activity, from raw sensor data. One promising application of the logic-based approach is a context-sensitive authorization system [1, 2, 6, 8, 11, 18, 24] that considers a requester's context as well as his identity to make a granting decision; the system derives the granting decision (true or false) with a set of rules encoding policies and facts encoding context information.

Those logic-based systems assume a central server that maintains global knowledge of all the context information. However, in many realistic applications of pervasive computing, sources of context information are inherently distributed among many administrative domains that have different security policies. For example, imagine a large

K.P. Fishkin et al. (Eds.): PERVASIVE 2006, LNCS 3968, pp. 220–237, 2006.

office building where there are sensors managed by the city, the building owner, the companies leasing space, and the individual employees. An active-map application that displays the current location of an employee in that building might need to access multiple indoor location tracking systems in different organizations.

To achieve such information sharing among organizations, we must address two trust issues. First, each administrative domain (organization) defines confidentiality policies to protect information in that domain. It is necessary for an administrator of a location tracking system to protect users' location privacy [5, 18], for example. Therefore, a requester must satisfy the confidentiality policies of an information provider to access the requested information. Second, each administrative domain defines integrity policies that specify whether to trust information from other domains in terms of the integrity (correctness) of that information. Because context information is computed from raw sensor data, it inherently involves uncertainty. It is, therefore, important for each domain to choose reliable sources of information to derive correct context information. 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 previous work on a secure context-sensitive authorization system [16, 17] enables mutually untrusted principals, which have partial knowledge about rules and context information, to evaluate a logical query without a universally trusted principal or a centralized knowledge base. The core of the 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 confidentiality and integrity policies of the principals operating those hosts. Our scheme relies on public-key operations to enforce security policies of the principals, and those public-key operations might cause long latency during the process of making an authorization decision. However, we had not previously reported the performance of the system.

In this paper, we present the design and implementation of a novel caching and revocation scheme that significantly improves the performance of the original system. Our current target application is an emergency-response system [12] that provides an information dissemination infrastructure for responders in a disastrous incident. Since the responders who belong to different state or local agencies share information across the agencies on a need-to-know basis, we adopt context-sensitive authorization policies that consider a responder's location and medical condition to grant access to information about the incident. Our system should, therefore, scale to support tens of different administrative domains, meeting three key goals:

Speed: the average latency for handling a query should be comparable to that of a local query in a centralized system; we, therefore, aggressively cache query results from remote hosts to avoid issuing remote queries.

Freshness: a query result must be derived only from context information that satisfies a timeliness condition; all the context information in the proof must be generated within a given interval between the current time and a recent past time.

Fault tolerance: a query result, if produced, must be guaranteed to be correct under the presence of host failures or adversaries that intercept messages between hosts.

To achieve those goals, our caching mechanism enables each host to maintain both positive and negative query results to avoid issuing remote queries. To ensure the freshness of cached results, we develop an efficient capability-based technique for revoking cached query results. Unlike existing revocation methods [26] in which only an issuer of a certificate can revoke it, our scheme must allow multiple hosts to revoke a given cached result because the result might depend on (contextual) facts maintained by different hosts. Every principal that handles a query returns a query result with a randomly generated capability so that it can revoke the result by sending that capability to the receiver of the result. Each host maintains dependencies among cached facts, and the revocation process is recursively iterated across multiple hosts until all the cached facts that depend on the fact that has initially become invalid are revoked. Each host maintains the freshness of each cached result by exchanging messages that update the timestamp associated with the result and discards obsolete results periodically.

To demonstrate the effectiveness of our caching scheme, we measured the performance of our system with and without our caching mechanism. The results show that our caching mechanism significantly improved the amortized cost for handling queries, and the performance with the caching mechanism was comparable to that of a centralized system where all the rules and facts are stored in its local knowledge base. We also measured the latency for revoking a query result to show how our scheme can meet the timeliness condition on cached results.

The rest of the paper is organized as follows. We introduce our secure context-sensitive authorization system in Section 2, and cover the design of our caching and revocation mechanism in Section 3. Next, we describe a mechanism for keeping cached information updated in Section 4. We show the results of our experiments in Section 5 and discuss some limitations of our scheme in Section 6. We cover related work in Section 7 and conclude in Section 8.

2 Overview of a Secure Context-Sensitive Authorization System

In this section, we provide an overview of our secure context-sensitive authorization system [16, 17]. The system consists of multiple identical servers, which collaborate peer-to-peer to construct a proof for an authorization query in a distributed way. We first describe the structure of each host and then show how a set of servers, each of which only maintains partial knowledge about policies and (contextual) facts, make an authorization decision in a distributed environment.

2.1 Structure of the Authorization Server

Figure 1 shows the structure of an authorization server that consists of a knowledge base and an inference engine. The knowledge base stores both authorization policies and facts including context information. The context server publishes context events and updates facts in the knowledge base dynamically. The inference engine receives an authorization (or a logical) query from a remote server, such as a resource server, that receives a user's request and returns a proof that derives the fact in the query by retrieving information in the knowledge base. If the engine cannot construct a proof, it returns

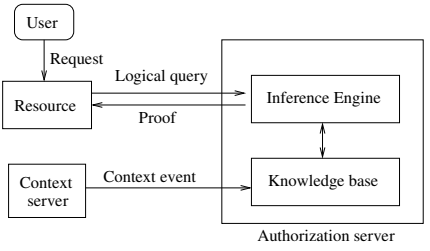


Fig. 1. Structure of an authorization server

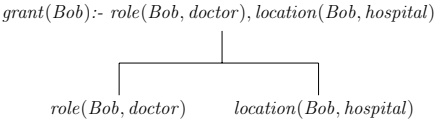


Fig. 2. Sample proof tree

a proof that contains a false value. In an open environment of pervasive computing, each server could belong to a different administrative domain.

Rules and facts in a knowledge base are represented as a set of Horn clauses in Prolog. For example, a medical database may define an authorization policy that requires a requester P to hold a role membership "doctor" and to be physically located at the "hospital" as follows.

$$grant(P) \text{ :- } role(P, doctor), location(P, hospital)$$

The atoms $role(P, doctor)$ and $location(P, hospital)$ on the right side of the clause are the conditions that must be satisfied to derive the granting decision $grant(P)$ on the left. If a user *Bob* issues a request to read a medical database, the proof tree in Figure 2 could be constructed based on the above rule. The root node in the tree represents the rule and the two leaf nodes represent the facts respectively. Notice that variable P in the rule is replaced with a constant *Bob*. A user's location, which is expressed with the $location$ predicate, is a dynamic fact; i.e., the second variable of the predicate $location$ should be updated dynamically as Bob changes his location.

2.2 Proof Decomposition in Distributed Query Processing

Multiple servers in different administrative domains handle an authorization query in a peer-to-peer way, since there need not be any single server that maintains all the rules and context information; a server must issue a remote query to another server when it does not have necessary information in its local knowledge base. However, the principals running those servers must preserve their confidentiality and integrity policies. The key idea for this goal is that when a principal who issues a query trusts a principal who handles a query in terms of the integrity of the query result, the handler principal does not disclose all the information in the proof. It might be sufficient to return a proof that simply states the fact in the query is true, and a proof thus is decomposed into multiple sub-proofs produced by different hosts.

Figure 3 describes such collaboration between a querier and a handler hosts. Suppose that host A run by principal Alice, who owns a projector, receives an authorization query $?grant(Dave, projector)$ that asks whether Dave is granted access to that projector. Since Alice's authorization policy in her knowledge base refers to a requester's location (i.e., $location(P, room112)$), Alice issues a query $?location(Dave, room112)$

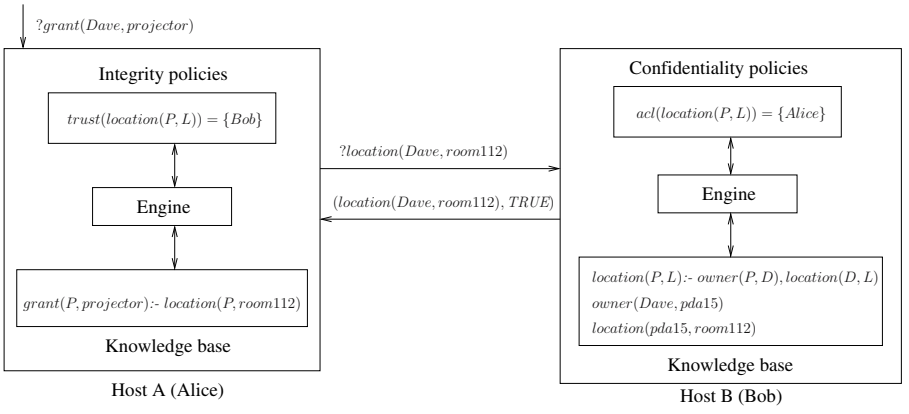


Fig. 3. Remote query between two principals. Alice is a principal who owns a projector, and Bob is a principal who runs a location server.

to host B run by Bob. Alice chooses Bob, because Bob satisfies Alice's integrity policies for queries of the type $location(P, L)$ (i.e., $trust(location(P, L)) = \{Bob\}$). Each principal decides to which principal a query should be sent by looking up his integrity policies. Bob processes the query from Alice, because Alice satisfies Bob's confidentiality policies for queries of the type $location(P, L)$ as defined in Bob's policy $acl(location(P, L)) = \{Alice\}$. Bob derives that Dave is in *room112* from the location of his device using the facts $location(pda15, room112)$ and $owner(Bob, pda15)$. However, he only needs to return a proof that contains a single root node that states that $location(Dave, room112)$ is true, because Alice believes Bob's statement about people's location (i.e., $location(P, L)$) according to her integrity policies. The proof of the query is thus decomposed into two subproofs maintained by Alice and Bob. In general, Bob could return a proof tree that contains multiple nodes. If Alice only trusts Bob's rule that derives Bob's location instead of Bob's fact, he would need to submit a larger proof tree to satisfy Alice's integrity policies.

2.3 Enforcement of Confidentiality Policies

Each principal who participates in constructing a proof enforces his confidentiality policies by encrypting a query result with a receiver principal's public key. A principal who returns a query result is allowed to choose a receiver principal from a list of upstream principals in a proof tree; a query is appended with a list of upstream principals that could receive the query result. Therefore, it is possible to obtain an answer for a query even when a querier principal does not satisfy the handler principal's confidentiality policies. Figure 4 shows the collaboration among principals p_0, p_1, p_2, and p_3. When principal p_0 issues an authorization query q_0 to principal p_1, p_1 issues a subsequent query q_1, which causes principal p_2's queries q_2 and q_3. Since a receiver principal of a proof might not be a principal who issues a query, a reply for a query is a tuple $(p_i, (pf)_{K_i})$ where p_i is an identity of a receiver principal and $(pf)_{K_i}$ is an encrypted proof with the receiver's public key. We assume that, in this example, each principal

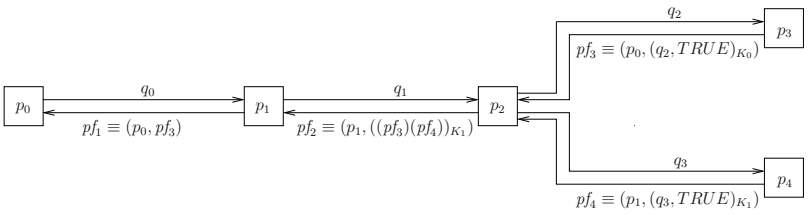


Fig. 4. Enforcement of confidentiality policies. The first item in a proof tuple is a receiver principal, and the second item is a proof tree encrypted with the receiver's public key.

who issues a query trusts the integrity of the principal who receives that query in terms of the correctness of whether the fact in the query is true or not. For example, p_0's integrity policies contains a policy $trust(q_0) = \{p_1\}$.

Suppose that query q_1's result (i.e., true or false) depends on the results of queries q_2 and q_3, which are handled by principals p_3 and p_4 respectively and that p_3 and p_4 choose principal p_0 and p_1 as a receiver respectively since p_2 does not satisfy their confidentiality policies. Because principal p_2 cannot decrypt the results from principals p_3 and p_4, p_2 encrypts those results with the public key of principal p_1[1] , which p_2 chose as a receiver. A principal p_2 forwards the encrypted results from p_3 and p_4 because the query result of q_1 is the conjunction of those results. Principal p_1 decrypts the encrypted result from p_2 and obtains the encrypted results originally sent from principals p_3 and p_4. Since p_1 is a receiver of the proof from p_4, p_1 decrypts the proof that contains a true value. Since a query result for q_0 depends on the encrypted proof from p_3, principal p_1 forwards it in the same way. The principal p_0 finally decrypts it and obtains an answer for query q_0. Notice that principal p_0 is not aware of the fact that the query result is originally produced by principal p_3.

Each proof must be signed with a sender principal's public key so that a principal who receives a proof that contains sub-proofs produced multiple principals can check its integrity. Our system applies public-key operations only to a randomly generated symmetric key to reduce the performance overhead and use the symmetric key to encrypt and decrypt a proof; that is, a proof consists of a new symmetric key encrypted with a receiver's public key and a proof encrypted with that symmetric key. In addition to the public-key encryption, the querier and handler principals use another shared symmetric key to protect other data fields (e.g., a receiver identity) in a proof and a query from eavesdroppers. We assume that the two principals share the symmetric key via a protocol using public-key operations when the querier and handler principal authenticate with each other for the first time.

3 Caching and Revocation Mechanism

In this section, we describe a caching and revocation mechanism that improves the performance of our system. Our caching mechanism supports both positive and negative

[1] This recursive encryption is necessary to prevent an attack by malicious upstream principals of the message flow. The malicious colluding principals could read principal p_2's query result illegally by modifying the list of upstream principals given to p_2 along with a query q_1.

query results and avoids issuing remote queries, which otherwise cause long latency due to cryptographic operations and the transmission of data over a network. Our capability-based revocation mechanism allows any principal who contributes to producing a proof to revoke the cached result derived from that proof.

3.1 Capability-Based Revocation

A proof for a query contains (context) information provided by multiple different principals, and the derived fact from the proof must be revoked if any information in the proof becomes invalid; that is, there might be multiple principals that are eligible to revoke a given cached fact. We, therefore, developed a revocation mechanism based on capabilities [23] so that all the principals involved in constructing a proof may revoke the derived result from the proof.

Each node in a proof tree is associated with a capability (a large random number). The capability is created by a principal who provides the information (i.e., a fact or a rule) in the node. Since a principal who publishes a proof encrypts the query result and capability together with a receiver principal's public key, the capability is a shared secret between the publisher and the receiver of the proof node. Therefore, the principal who sent the proof can later revoke the fact or rule in the proof by sending the capability to the receiver principal. The sender principal of the revocation message does not need to authenticate itself to the receiver principal who maintains the cached information.

Figure 5 describes our revocation scheme in a distributed environment. A principal p_0 issues a query $?location(Bob, hospital)$, and a principal p_1 returns a proof that consists of a rule node produced by p_1 and two leaf nodes produced by p_2 and p_3 respectively. A principal p_0 caches the fact $location(Bob, hospital)$ derived from the received proof. Since principals p_1, p_2, and p_3 contribute to constructing the proof tree, they all should be eligible to revoke p_0's cached fact. Therefore, each principal p_i for $i = 1, 2, 3$ includes a capability c_i into his produced node so that p_i can revoke the proof later. A principal p_0 who caches the fact $location(Bob, hospital)$ associates it with the capabilities c_1, c_2, and c_3 obtained from the proof. Since principal p_3 chose p_0, not p_1, as a receiver of his proof pf_3, p_3 revokes his proof by sending a capability

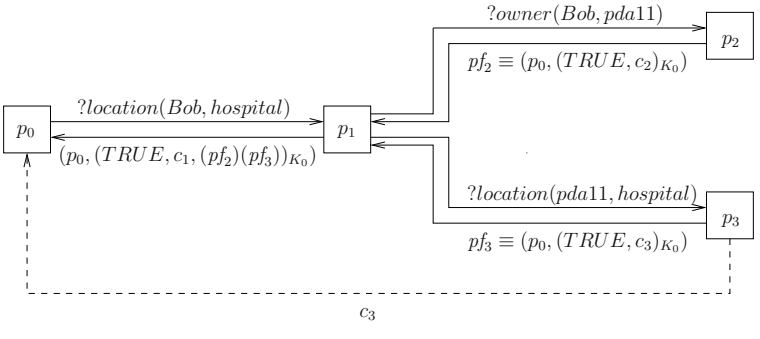


Fig. 5. Capability-based revocation. The dashed line represents a revocation message sent by principal p_3.

c_3 directly to principal p_0. Receiving that revocation message, a principal p_0 removes the cached fact associated with the capability c_3. Principals p_1 and p_2 could revoke the same cached fact in the same way. Our capability-based revocation does not involve any public-key operations, which are computationally expensive, because a revocation message can be directly sent to a principal who maintains a cached fact. When our system constructs a proof tree responding to an authorization query, public-key encryptions are necessary to prevent intermediate principals between a sender and a receiver principal from reading the sender's query result. Furthermore, a revocation message does not need to be signed by a sender principal, because it is not necessary for a sender to authenticate himself to a receiver principal of the revocation message. When we extend the revocation scheme to support negative caching, however, we do require encryption (see Section 3.4).

3.2 Structural Overview

Our revocation mechanism is based on a publisher-subscriber model; that is, a querier principal subscribes to a handler principal who handles his query, and the handler principal sends a revocation message when the query result becomes invalid. This process might occur recursively until all the cached facts that depend on the invalidated fact are revoked across the network. Figure 6 shows the structure of our caching and revocation mechanism and the message flow among the components when a cached fact is revoked. Each server consists of two components (an inference engine and a revocation handler) and four data structures (a subscribers list, a dependencies list, a subscription list, and a knowledge base). The inference engine is responsible for constructing a proof and caching query results obtained from other principals with the knowledge base and the subscription list. The engine also maintains information on other principals who issue a query to the engine with the subscribers and dependencies list so that the engine can revoke cached results in remote hosts. When principal p_1 receives a query q_0 from

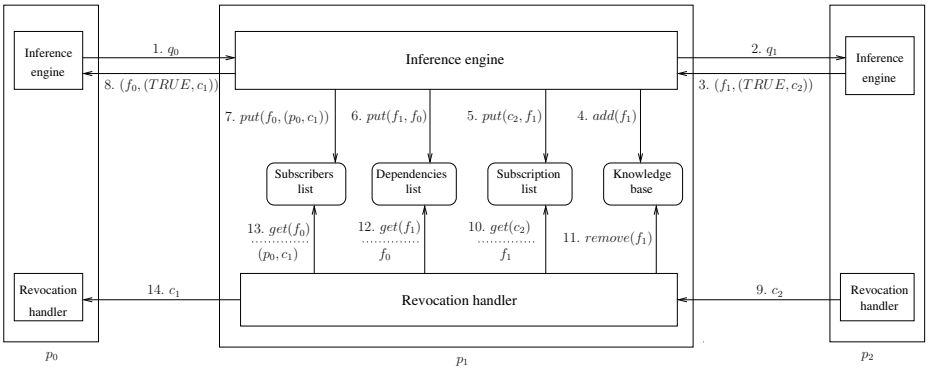


Fig. 6. Structure of a caching and revocation mechanism. We omit the data structures from the servers of p_0 and p_2 for brevity. The number at the beginning of each message represents the sequence of the entire revocation process. The return value of a message is shown under a dotted line in the messages 10, 12, and 13.

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

The revocation process occurs when principal p_2 sends a revocation message that contains a capability c_2. Principal p_1's revocation handler receives the message, obtains a fact to be revoked with capability c_2 from the subscription list, and removes fact f_1 from the knowledge base. Next, the revocation handler obtains fact f_0, which depends on f_1, from the dependencies list and then accesses the subscribers list to obtain a capability c_1 for revoking principal p_0's cached fact f_0, and sends c_1 to p_0's revocation handler. The same process is repeated on p_0's server.

If a capability is a shared secret that is only used once, a capability does not need to be encrypted as we explain in this section. In Section 3.4 below, though, we add support for caching negative results and in that case we do need an encrypted channel for this message.

3.3 Synchronization Mechanism

There is a race condition to be resolved between the inference engine and the revocation handler, because both modules access the four data structures in Figure 6. For example, it is possible that the revocation handler accesses the subscription list with a capability c that revokes a fact f before the inference engine writes the subscription information (i.e., (c, f)) to that list.

However, we cannot use a coarse mutual exclusion mechanism that allows the thread of the inference engine to block other threads' access to the data structure while processing a query, since a deadlock occurs when the engine issues a remote query that causes a closed cycle of subsequent queries by remote servers. For example, if a downstream server that receives a subsequent query issues a remote query back to the server of the inference engine, a new thread that is created to handle that query blocks because the inference engine on that server already obtains a lock on the data structures, which the new thread needs to access. Thus, the inference engine would wait for a reply for the remote query forever. We, therefore, built a fine-grained synchronization mechanism that ensures that the engine that receives a proof-tree node with capability c updates the data structures before the revocation handler that receives a capability c accesses them.

3.4 Negative Caching

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

To support negative caching, each principal maintains the same set of data structures in Figure 6; that is, each server maintains another knowledge base that stores negative facts. The semantics of a negative revocation is different from that of positive caching; that is, when a cached negative fact is revoked, that fact can be cached as a positive fact. (On the other hand, to revoke a positive fact does not necessary mean that the revoked fact is no longer true; there might be another proof that derives the fact. Note that if a host that maintains a positive fact has first-hand knowledge about the validity of that fact without checking with other hosts, that host could convert the revoked positive fact into a negative cached fact.)

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

Therefore, a revocation message for a negative cached result needs to contain a new capability to revoke the switched positive cached result. Since the new capability must be a shared secret between a sender and a receiver of the revocation message, we need to encrypt the message with a shared key between those two parties. However, to establish a symmetric secure channel for all the pairs of two principals that participate in our system requires n^2 symmetric keys, where n is the number of participating principals. To avoid this key-management problem, our system encrypts a revocation message with the same randomly generated symmetric key that is used to encrypt the proof node that contains the cached result as we describe in Section 2.3; that is, each server records the capabilities in a received proof with a symmetric key that is used to decrypt that proof. Suppose that the proof contains a node with a capability c_n and was encrypted with a symmetric key K when the server receives it. A server stores a (c_n, K) pair in a hash table to handle a revocation message $(c_n, (c_n, c_p)_K)$ where c_n is a capability that revokes a current negative result, c_p is a capability that revokes the switched positive result in the future, and K is the symmetric key associated with c_n. When a server

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

4 Timeliness of Cached Information

Our system must ensure that all the cached facts meet a given timeliness condition; all the timestamps associated with cached facts must be within a given interval between the current time and a recent past time. To simply keep the latest messages does not guarantee the freshness of the cached facts because some hosts might crash or an adversary might intercept revocation messages so a server would make an incorrect decision based on obsolete cached information.

We, therefore, develop a mechanism that ensures the freshness of cached positive and negative facts obtained from remote servers. The updater thread on each server periodically sends each subscriber in a subscribers list a message that updates the timestamp of a cached fact by sending the capability with a new timestamp. We assume that all the server clocks are approximately synchronized. Since the server sends the same capability to refresh the same cached fact repeatedly, the updater thread encrypts the message with the same symmetric key that would be used to send a revocation message for revoking that cached fact. The watcher thread on another server receives that message and updates the timestamp of the fact in a subscription list. The watcher thread must synchronize with the inference engine using the same synchronization method we describe in Section 3.3. If the watcher thread finds any subscription with an old timestamp (possibly because an adversary intercepts revocation messages), it discards that subscription and initiates the revocation process described in Section 3.2.

5 Experiments and Evaluation

We set out to measure the performance of our system. Since many context-aware applications, such as an emergency-response system in which responders continuously access information on an incident over a duration of a few hours to several days, need to keep track of a user's privileges continuously, our focus is to show that our caching mechanism significantly improves amortized performance of our system.

We used a 27 node cluster connected with a Gigabit Ethernet. Each node had two 2.8GHz Intel XEONs and 4GB RAM, and runs RedHat Linux 9 and Sun Microsystem's Java runtime (v1.5.0-hotspot). Our system has approximately 12,000 lines of Java code, extending a Prolog engine XProlog [25]. We used the Java Cryptographic Extension (JCE) framework to implement RSA and Triple-DES (TDES) cryptographic operations. We used a 1024-bit public key whose public exponent is fixed to 65537 in our experiments. The RSA signing operation uses MD5 [21] to compute the hash value of a message. We used Outer-CBC TDES in EDE mode [9] to perform symmetric key

operations. The length of our DES keys was 192 bits, and the padding operation in TDES operations conforms to RFC 1423 [20].

5.1 Analysis of Performance Overhead

We first show the latency of the system with two hosts that did not use the caching mechanism. One host maintains a rule $a0(P) \leftarrow a00(P)$, and the other host maintains a fact $a00(bob)$. When the former host receives a query $?a0(bob)$, it issues a remote query $?a00(bob)$ to the other host. We measured the wall-clock latency for handling a query and also the latency of each cryptographic operation in that process. The measurement is iterated one hundred times, and we report the average of the measurements. Table 1 shows the results.

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

Table 1. Average latency of processing a query with two hosts without caching capability. The latency is measured in milliseconds. The *ratio* columns shows the ratio of the latency of each primitive operations compared with the total local processing time.

	host 0		host 1	
	latency	ratio	latency	ratio
Total latency	138.1		85.2	
Issue remote queries	87.9		0.0	
Local computation	50.2	1.00	85.2	1.00
TDES decryption on a received query	0.0	0.00	2.2	0.03
TDES encryption on a returning proof	0.0	0.00	10.9	0.12
TDES decryption on a received proof	10.9	0.22	0.0	0.00
TDES encryption on an issued query	1.2	0.02	0.0	0.00
RSA decryption on DES keys	26.6	0.53	0.0	0.00
RSA encryption on DES keys	0.0	0.00	18.7	0.22
Create a RSA signature for a proof	0.0	0.00	14.4	0.17
Verify a RSA signature for a proof	2.2	0.04	0.0	0.00

5.2 Latency for Handling Queries

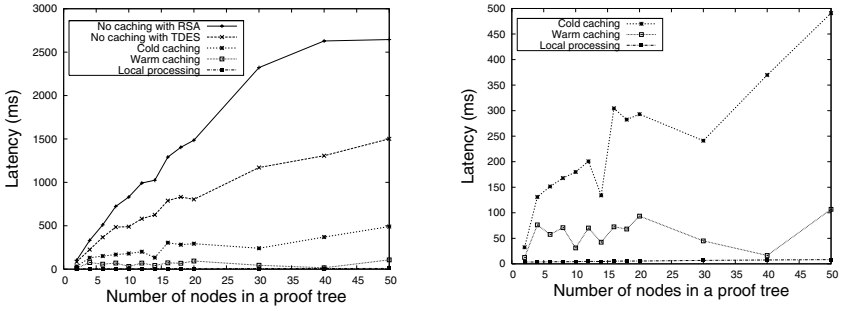
We next measured the latency of handling a query with different size proof trees to evaluate the scalability of our caching scheme. We performed our experiments with 27 servers run by different principals; those servers could correspond to 27 different agencies in the emergency-response system. Our test program generated authorization, confidentiality, and integrity policies of those principals such that our system constructs a proof tree of a given size (i.e., the number of nodes in the proof tree) for the given query. Each query takes the form of $?grant(P, R)$ where P is a principal and R is a

resource. The body of each rule takes the form of $a_0(c_0), \ldots, a_{n-1}(c_{n-1})$ where a_i for $i = 0$ to $n - 1$ is a predicate symbol and c_i for $i = 0$ to $n - 1$ is a constant. The size of the domain of predicate symbols is 1,000, and the size of the domain of constants is 20. There are possibly 20,000 different atoms in authorization policies that our test program generates, and it is, therefore, unlikely that a cache hit occurs when a query is evaluated for the first time. Those policies are independent of any particular application in a sense that the test program chose the topology of a proof tree randomly. However, we conducted our experiment up to a proof tree with 50 nodes, which we believe is significantly larger than that in most applications, and, therefore, our results should provide guidelines about the worst-case latency of those applications. We prepared the facts and rules to allow ten different proof trees of the same size, and in the experiment a given host issued a sequence of ten different queries of that size.

Our latency measurements also include the performance overhead for handling revocation messages. While measuring latency for handling queries 100 times, our test driver program updates all the facts in the knowledge bases dynamically. We assumed the extreme case that all the facts in each proof tree are dynamic contextual facts, and updated every fact 20 times per second during the experiments. We believe that this update frequency is much faster than most context-aware applications need.

Figure 7(a) compares query-handling latency under five different conditions; each data point is an average of 100 runs. In the *No caching, with RSA* case, each server did not cache any results obtained from other servers and used public-key operations for encrypting DES keys and signing proof trees. *No caching, with TDES* is same as the first case, except that every pair of the principals shared a secret DES key and used it to attach a message authentication code (MAC) using Keyed-Hashing for Message Authentication (HMAC-MD5) hashing algorithm [4, 14] to authenticate an encrypted proof. We included this case to show that to use symmetric key operations instead of public-key operations (assuming that all the pair of the principals share a secret key) does not solve the problem of the long latency for handling a query. In the *Cold caching* case, every server cached results from other servers and all the latency data including that of the initial round of queries were used to compute average latency. In the *Warm caching* case, every server cached results from other servers and we used only the latency data after the first round of ten different queries to compute average latency. In the *Local processing* case, all the rules and facts were stored in a single server. Therefore, there was no remote query involved, and no encryption.

The two cases without caching show significantly longer latency than the other three cases, although using MD5 and DES operations rather than RSA reduced the latency 15 – 50%. The latency grew longer than 500 ms when a proof tree contained more than ten nodes. Figure 7(b) shows the same latency results for the other three cases, omitting the case of no caching. The latency of the cold caching case is 5 to 20 times longer than that of the warm caching, because the initial queries require the whole process of constructing a proof tree as in the case of no caching. The latency of the warm caching case were 2 to 15 times higher than that of the local processing case. The reason for the longer latency is a cache miss due to a revocation of a positive cached fact. However, the latency of the warm caching case was 3 to 23 times faster than the cold caching case, and thus we could improve the performance by prefetching query results in advance.



(a) Five cases including no-caching cases. (b) Three cases without no-caching cases.

Fig. 7. Latency for handling queries

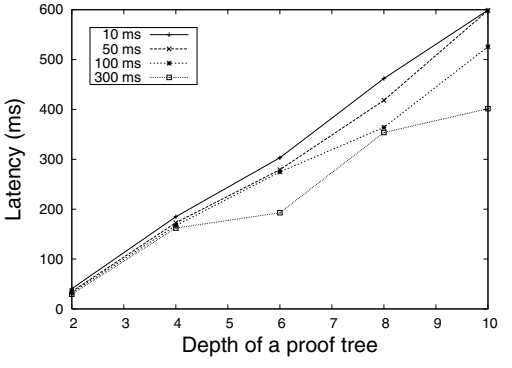


Fig. 8. Latency for revoking cached facts. Each curve represents a different period between fact updates in the knowledge bases, in milliseconds.

5.3 Latency for Revoking Cached Facts

We measured the latency for revoking cached facts with another experiment. We used linear proof trees of various depths to measure the latency between the moment the test driver sent an event that updates a fact in the knowledge base and the moment that the test driver received the notification of a revoked cached fact from the root server that handles queries from the test driver. We conducted the same experiment 100 times and report the average of the measurements. Figure 8 shows the latency for revoking cached facts with four different frequencies for updating the knowledge bases. The results show that the latency increased linearly as the depth of a proof tree grows. The latency slightly increased as the period for publishing an event decreases. The system handled 100 events per second with the latency less than 600 ms and a proof tree of depth 10.

6 Discussion

Although, in Section 5, we conducted the experiments in a cluster with low-latency connections, our implementation, to some extent, simulates a low-latency network by

encoding a proof as a set of Java objects, which is much larger than the corresponding string representation. For example, in the experiment in Section 5.1, the sizes of a query object and a proof object were 723 bytes and 34184 bytes respectively, and the corresponding strings for the query and the proof were less than 124 bytes. Also, our caching mechanism could improve the performance of the system even more drastically in a wireless environment with low bandwidth and high data-loss ratio, because to handle a query with local cache is a common case for a long-running continuous query. The mechanism in Section 4 refreshes cached information periodically, and thus prevents false positive decisions due to a disconnected wireless network.

To process an authorization query involves latency for constructing a proof, and the authorization decision is thus derived from the proof that might contain dynamic facts previously published at different times. Since an authorization decision is made based on information collected in the past, our system might grant a request that should have been denied if the current information was available to the system. This limitation in our system might allow a malicious user to gain access to a resource illegally by changing his context from a legitimate state (that grants his access) to an illegitimate state before the system detects the change of the the user's context. Therefore, our system should provide a policy maker with a way to define explicit timeliness constraints on authorization decisions; that is, a policy maker should be able to specify a time T such that all the information in a proof was published within time T prior to the current time. Although our system does not explicitly support this mechanism, the experimental results in Section 5.3 imply that our system would work even if T were as small as six hundreds milliseconds for a large proof of depth 10.

7 Related Work

In this section, we cover systems that support caching mechanisms for an inference engine. See our technical report [15] for a comprehensive survey on distributed authorization.

We developed our caching and revocation mechanisms based on our previous research on the secure context-sensitive authorization system [17]. We measured the performance of our original scheme in detail, which is not included in our previous paper.

Ranganathan [19] proposes to use a first-order logic to model a user's context and reason about it. To reason with context information stored in multiple hosts, each context provider on those hosts provides an interface that handles a query from a remote host. However, their scheme does not support any caching mechanism across the hosts. Bauer [3] developed a distributed proving system that constructs a proof that grants access to a resource in a distributed way; a principal who constructs a proof could delegate a task of building a sub-proof to another principal rather than collecting all the certificates that are necessary to construct a whole proof. Bauer's scheme is similar to ours in a sense that a proof is produced by multiple principals in a distributed environment. However, the algorithm does not address the issue of protecting confidential information in certificates, which are used to construct a proof. Although their system caches both positive and negative facts, there is no detail about mechanisms for revoking cached information. Katsiri [13] built a prototype of a dual-layer knowledge base based on a first-order logic. The higher Deductive Abstract layer caches abstract context

information derived from low-level knowledge in the lower to make the system scalable. The system consists of a single server and does not support a revocation mechanism in a distributed environment.

8 Summary

We describe a novel caching and revocation mechanism that improves the performance of a secure context-sensitive authorization system. Our major contribution is to show that we could build a secure distributed proof system whose amortized performance scales to a large proof that spans across tens of servers. Our capability-based revocation mechanism combines an event-based push mechanism with a query-based pull mechanism where each server publishes a revocation message over a network recursively by maintaining dependencies among local and remote cached facts. Our revocation mechanism supports both positive and negative caching and is capable of converting a revoked negative fact into a valid positive cached fact to reduce the number of cache misses, while ensuring the secrecy of a new capability without having n^2 secret keys among n principals. We also incorporate a mechanism that ensures the freshness of cached information under the presence of an adversary that is capable of intercepting revocation messages.

Our experimental results show that the performance overhead of public-key operations involved in the process of a remote query were large and that our caching mechanism significantly reduced the amortized latency for handling a query. Therefore, our system is suitable to a context-aware application in which a user's privileges must be continuously monitored. Since our experiments were conducted with a wide range of parameters, the results should serve as guidelines about the worse-case performance of many systems in pervasive computing.

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

Acknowledgments

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

References

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

2. Jean Bacon, Ken Moody, and Walt Yao. A model of OASIS role-based access control and its support for active security. *Proceedings of the Sixth ACM Symposium on Access Control Models and Technologies*, 5(4):492–540, 2002.
3. Lujo Bauer, Scott Garriss, and Michael K. Reiter. Distributed proving in access-control systems. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, pages 81–95, Washington, DC, USA, 2005. IEEE Computer Society.
4. Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying hash functions for message authentication. In *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, pages 1–15, London, UK, 1996. Springer-Verlag.
5. Alastair R. Beresford and Frank Stajano. Location Privacy in Pervasive Computing. *IEEE Pervasive Computing*, 2(1):46–55, January-March 2003.
6. Patrick Brezillon. Context-based security policies: A new modeling approach. In *Second IEEE Annual Conference on Pervasive Computing and Communications Workshops*, pages 154–158. IEEE Computer Society, March 2004.
7. Harry Chen, Tim Finin, and Anupam Joshi. An Ontology for Context-Aware Pervasive Computing Environments. *Special Issue on Ontologies for Distributed Systems, Knowledge Engineering Review*, 18(3):197–207, May 2004.
8. Michael J. Covington, Wende Long, Srividhya Srinivasan, Anind K. Dey, Mustaque Ahamad, and Gregory D. Abowd. Securing context-aware applications using environment roles. In *Proceedings of the Sixth ACM Symposium on Access Control Models and Technologies*, pages 10–20. ACM Press, 2001.
9. Data Encryption Standard (DES), October 1999. `http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf`.
10. Karen Henricksen and Jadwiga Indulska. A software engineering framework for context-aware pervasive computing. In *Proceedings of the Second IEEE International Conference on Pervasive Computing and Communications (PerCom'04)*, pages 77–86, Washington, DC, USA, 2004. IEEE Computer Society.
11. R. J. Hulsebosch, A. H. Salden, M. S. Bargh, P. W. G. Ebben, and J. Reitsma. Context Sensitive Access Control. In *Proceedings of the 10th ACM Symposium on Access Control Models and Technologies*, pages 111–119, Baltimore, MD, June 2005.
12. National incident management system, March 2004. `http://www.fema.gov/pdf/nims/nims_doc_full.pdf`.
13. Eleftheria Katsiri and Alan Mycroft. Knowledge representation and scalable abstract reasoning for sentient computing using first-order logic. In *Proceedings of Challenges and Novel Applications for Automatic Reasoning (CADE-19)*, pages 73–87, July 2003.
14. Hugo Krawczyk, Mihir Bellare, and Ran Canetti. Hmac: Keyed-hashing for message authentication. Internet RFC 2693, February 1997. `http://www-cse.ucsd.edu/users/mihir/papers/rfc2104.txt`.
15. Kazuhiro Minami. Secure context-sensitive authorization. Technical Report TR2006-571, Dept. of Computer Science, Dartmouth College, February 2006.
16. Kazuhiro Minami and David Kotz. Secure context-sensitive authorization. In *Proceedings of the Third IEEE International Conference on Pervasive Computing and Communications (PerCom)*, pages 257–268, Kauai, Hawaii, March 2005.
17. Kazuhiro Minami and David Kotz. Secure context-sensitive authorization. *Journal of Pervasive and Mobile Computing*, 1(1):123–156, March 2005.
18. Ginger Myles, Adrian Friday, and Nigel Davies. Preserving privacy in environments with location-based applications. *IEEE Pervasive Computing*, 2(1):56–64, January-March 2003.
19. Anand Ranganathan and Roy H. Campbell. An infrastructure for context-awareness based on first order logic. *Personal Ubiquitous Computing*, 7(6):353–364, 2003.
20. RFC 1423 - Privacy Enhancement for Internet Electronic Mail: Part III: Algorithms, Modes, and Identifiers, February 1993. `http://www.faqs.org/rfcs/rfc1423.html`.

21. Ronald L. Rivest. The MD5 message-digest algorithm, April 1992. http://www.ietf.org/rfc/rfc1321.txt.
22. Bill N. Schilit, Norman Adams, and Roy Want. Context-aware computing applications. In *Proceedings of IEEE Workshop on Mobile Computing Systems and Applications*, pages 85–90, Santa Cruz, California, December 1994. IEEE Computer Society Press.
23. Andrew S. Tanenbaum, Robbert van Renesse, Hans van Staveren, Gregory J. Sharp, and Sape J. Mullender. Experiences with the amoeba distributed operating system. *Communications of the ACM*, 33(12):46–63, 1990.
24. Anand Tripathi, Tanvir Ahmed, Devdatta Kulkarni, Richa Kumar, and Komal Kashiramka. Context-based secure resource access in pervasive computing environments. In *Proceedings of the Second IEEE Annual Conference on Pervasive Computing and Communications Workshops*, pages 159–163. IEEE Computer Society, March 2004.
25. Jean Vaucher. XProlog.java: the successor to Winikoff's WProlog, Feb 2003. http://www.iro.umontreal.ca/~vaucher/XProlog/AA_README.
26. Peifang Zheng. Tradeoffs in certificate revocation schemes. *ACM SIGCOMM Computer Communication Review*, 33(2):103–112, 2003.