Available online at www.sciencedirect.com
ScienceDirect

Pervasive and Mobile Computing 2 (2006) 427–452

pervasive and mobile computing

www.elsevier.com/locate/pmc

Avoiding privacy violations caused by context-sensitive services

Urs Hengartner[a,*], Peter Steenkiste[b]

[a] *David R. Cheriton School of Computer Science, University of Waterloo, Waterloo, ON, Canada*
[b] *Departments of Computer Science and Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA, USA*

Received 31 January 2006; received in revised form 2 June 2006; accepted 27 July 2006
Available online 1 September 2006

Abstract

The increasing availability of information about people's context makes it possible to deploy context-sensitive services, where access to resources provided or managed by a service is limited depending on a person's context. For example, a location-based service can require Alice to be at a particular location in order to let her use a printer or learn her friends' location. However, constraining access to a resource based on confidential information about a person's context can result in privacy violations. For instance, if access is constrained based on Bob's location, granting or rejecting access will provide information about Bob's location and can violate Bob's privacy. We introduce an access-control algorithm that avoids privacy violations caused by context-sensitive services. Our algorithm exploits the concept of access-rights graphs, which represent all the information that needs to be collected in order to make a context-sensitive access decision. Moreover, we introduce hidden constraints, which keep some of this information secret and thus allow for more flexible access control. We present a distributed, certificate-based access-control architecture for context-sensitive services that avoids privacy violations, two sample implementations, and a performance evaluation.

Keywords: Context awareness; Distributed access control; Location-based services

* Corresponding author.
E-mail addresses: uhengart@cs.uwaterloo.ca (U. Hengartner), prs@cs.cmu.edu (P. Steenkiste).

1574-1192/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.pmcj.2006.07.007

1. Introduction

The increasing numbers of networked devices (e.g., cellphones or handhelds) that individuals are carrying and of networked sensors (e.g., cameras) makes more context-sensitive information about people electronically available. This trend enables the deployment of context-sensitive services, where access to resources provided or managed by a service depends on a person's context. For instance, many pervasive computing projects provide location-based services, where a resource is available to an individual only if the individual is at a particular location [1–6]. For example, a user of a buddy service could allow her friends to learn her location only if they are nearby. Similarly, the administrator of a service managing devices (e.g., a projector or a printer) in a meeting room could decide to let only people in the room access these devices. However, the deployment of context-sensitive services poses serious privacy challenges. Namely, we must ensure that these services do not leak confidential information about an individual's context to unauthorized entities. In this paper, we show how to avoid such privacy violations.

Let us demonstrate how a naïve implementation of context-sensitive access decisions to resources can lead to privacy violations. In our first example, confidential information leaks to a service that provides information. Assume that Alice lets people see her current calendar entry only if they stand in front of her office, that is, she imposes a context-sensitive *constraint*. A cellphone service provides people's location information, and a calendar service offers Alice's calendar information. Given this setup, when Bob asks the calendar service for Alice's calendar entry, the calendar service could learn Bob's location while making an access decision, either by querying the location service directly or by being told by a third entity that the constraint imposed by Alice is fulfilled. Therefore, Bob's location information could leak to the calendar service (i.e., to the organization running this service), and his privacy could be violated.

In the second example, confidential information leaks to a person who is granted access to some other information. Assume that Alice allows people to access her calendar entry if she is in her office. Therefore, if somebody can retrieve this entry, he will also learn that Alice is in her office. A person planning on breaking into Alice's house would happily take advantage of this information leak.

In our third example, confidential information leaks to a person who grants other people access to her information. Assume that Alice grants herself access to her calendar entry constrained to Bob being at a particular location. When the calendar system grants Alice access to her entry, she will learn Bob's location, which could be an information leak.

Related work has largely ignored privacy violations caused by context-sensitive services. Avoiding these violations is a complex problem, especially when constraints are recursive (e.g., "Alice says that Bob can access her calendar when she is in her office." and "Alice says that Bob can access her location when she is not busy."). As a result, in our first contribution, we present a systematic investigation of information leaks caused by context-sensitive services so that we understand all the opportunities for information leaks. Our second contribution is a set of algorithms to avoid information leaks caused by context-sensitive services. In particular, our algorithms include:

Access-rights graphs. We introduce algorithms for building and resolving access-rights graphs. These graphs represent all the information that will have to be collected in order to ensure the satisfaction of constraints associated with a resource. Furthermore, we present an access-control algorithm that, based on access-rights graphs, resolves constraints and avoids information leaks.

Hidden constraints. We propose hidden constraints, which make it possible to implement more flexible constraints by keeping constraint specifications secret. Furthermore, hidden constraints lead to a simpler access-control algorithm.

Finally, we present a distributed, certificate-based access-control architecture that exploits these algorithms in order to provide context-sensitive services that do not leak confidential information, two example implementations of this architecture, and a performance evaluation.

This paper expands on our PerCom 2006 paper [7] in that we discuss the case where multiple services offer the same information (Section 2.2), address staleness of information used for constraining access (Section 2.3), present scenarios where information leaks cannot be avoided (Section 3.1), compare our main access-control approach to alternative approaches (Sections 3.3 and 5.3), introduce an algorithm for building access-rights graphs (Section 4.1), present sample certificates (Section 6.1), discuss and evaluate another implementation of hidden constraints (Sections 6.2.2 and 7, respectively), and elaborate on additional related work (Section 8).

We first introduce our system model (Section 2). Next, we focus on a restricted set of constraints and discuss information leaks caused by these constraints and how to avoid these leaks (Section 3). Based on this discussion, we then drop the restrictions on constraints and introduce access-rights graphs (Section 4) and hidden constraints (Section 5). Finally, we present our access-control architecture (Section 6) and its performance (Section 7).

2. System model

In this section, we describe the system model that we will use for studying privacy violations caused by context-sensitive access control. In particular, we introduce (constrained) access rights and client-based access control, discuss staleness of information, and present our security and threat model.

2.1. Access rights and constraints

For simplicity reasons, we assume that the resources offered or managed by a context-sensitive service consist of confidential information (e.g., the location of an individual's friends or a person's calendar entry). It is straightforward to apply our algorithms to a service that manages physical devices, such as a printer or a projector.

For an entity to be granted access to confidential information, there must be an *access right* authorizing this access. An access right consists of four parts: An *issuer* issuing the access right, a *subject* being given access, *information* to which access is granted, and a tuple of *constraints* that must be satisfied for the subject to get access to the information.

Either the subject or the tuple of constraints can be omitted from the access right. Each piece of information has an owner, who is responsible for issuing access rights to this information. For example, Alice issues access rights to her calendar information.

We assume that a constraint consists of information and a set of permitted values. The constraint is satisfied if the current value of its information equals one of the values in the set. A tuple of constraints attached to an access right is satisfied if each constraint in the tuple is satisfied. We observe that many sensible constraints in pervasive computing involve information about the context of a person. A person's context can include, for example, the current time, her current activity, or her current location. In addition, a constraint is typically about the person that either is granted an access right (e.g., "Alice grants Bob access to her calendar if he is in his office.") or grants an access right (e.g., "Alice grants Bob access to her calendar if she is in her office."), but not about third entities. Therefore, we are mainly interested in constraints that deal with context-sensitive information about the first two entities (though our presented solution is powerful enough to support constraints involving third entities). We focus on context-sensitive constraints that are confidential (e.g., a person's location or activity, but not the current time) and that have dynamic values, which makes it infeasible to check the satisfaction of a constraint upon the specification of an access right.

2.2. Client-based access control

We will study information leaks for the following scenario: A client wants to retrieve information provided by a service. We use the terms *primary information* and *primary service* for referring to this information and service, respectively. The client's access right to the primary information has a tuple of constraints. We call the information listed in the constraints *constraint information* and the services offering it *constraint services*. Let us look at an example. Alice's access right to Carol's calendar is constrained to Alice being in her office. Carol's calendar and Alice's location information are offered by a calendar and a location service, respectively. Here, Carol's calendar corresponds to the primary information and Alice's location to constraint information. The calendar service is the primary service and the location service is a constraint service. Note that for different requests, the same information can be either primary or constraint information, and a service can be either the primary or a constraint service.

There are multiple approaches to deploy access control in this scenario. We concentrate on *client-based access control* [8,9], where the client needs to prove to the primary service that it is authorized to access the primary information. The service makes the final access decision by validating this proof of access. The proof contains the client's access right to the primary information and confirms that each of the constraints in the access right is satisfied. We use the term *assurance* for such a confirmation. Therefore, before the client can contact the primary service, the client needs to retrieve assurances. In particular, for each constraint in the client's access right to the primary information, the client has to build a proof of access for the constraint information, contact the corresponding constraint service, and have it issue an assurance. We illustrate client-based access control in Fig. 1. Both access rights and assurances can be represented as digital certificates. If the client's

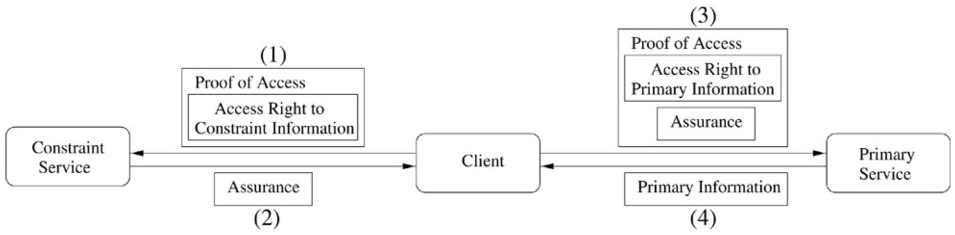

Fig. 1. Client-based access control. The client sends a proof of access to the constraint service to retrieve an assurance for the constraint information. Next, the client sends a proof of access, including the assurance, to the primary service to access the primary information. (Numbers indicate order of events.)

access rights to the constraint information were also constrained, the client recursively would have to retrieve assurances for the constraints in these access rights beforehand.

The advantage of client-based access control is the lack of a centralized entity making access decisions (i.e., a single point of failure). Furthermore, by assigning parts of the access-control load (i.e., constraint resolution) to the client, the approach reduces the load on the primary service. Alternatively, we could employ a centralized entity or a service for resolving constraints. We study these alternative approaches and their privacy implications in Section 3.3.

Multiple services can provide the same type of information. For example, there are multiple ways to locate a person. For simplicity reasons, we assume that there is only one service that offers the primary information in our problem setting. Extending our algorithms to support multiple such services is straightforward. However, picking a constraint service requires some thought. We could let the owner of constraint information choose a service. However, a malicious owner can pick a service that always guarantees satisfaction of a constraint. In the calendar example above, Alice, as the owner of her location information, could designate a fake location service that always returns Alice's office as her location. Instead, it is the issuer (Carol in our example) of a constrained access right who should pick the constraint service(s) providing the constraint information in the access right.

2.3. Staleness of constraint information

Since both the primary and the constraint information will typically be dynamic, we need to address the question of *when* the information is collected. The most natural way to interpret an access right is that all the information that it lists should be collected at the same time, thus guaranteeing that the primary information returned to the client is collected at a time when the constraint is satisfied. The access-control algorithm introduced in Section 2.2 however does not meet this requirement. (The same problem also occurs in the alternative access-control approaches.) Here, the client retrieves an assurance for a constraint from a constraint service, submits the assurance to the primary service, and gets the primary information from the primary service. This algorithm will not guarantee that the constraint is still satisfied at the exact time when the primary information is accessed.

We can guarantee that all information is collected at the same time by assigning the same timestamp to all queries sent to the (time-synchronized) constraint and primary services.

A service returns the value of the information at the time indicated by the timestamp. A timestamp can correspond either to a time in the past or to a time in the future.

For times in the past, a service needs to remember past values of the information that it offers, which can be difficult to implement. For example, a location service could periodically poll the location of all the locatable individuals. However, the polled values do not necessarily let the service correctly deduce the location of an individual for a random timestamp in the past. We could restrict the granularity of the permitted timestamps to the polling frequency. However, for low frequencies, clients might be given stale information and high frequencies cause more overhead. Moreover, being able to impose the same frequency on all services seems unrealistic. Finally, requiring services to keep historical logs raises privacy problems.

Timestamps can also correspond to a time in the future. Here, a notification step precedes the submission of a query to a service. In this step, both the primary service and the constraint services are notified of the time for which the upcoming query will expect the value of the information provided by the service. In the example given above, this approach allows the location service to poll the location of a particular user (and of no other users) at the notified time. Later, when the actual query arrives, the service returns the polled value. The announced timestamp needs to correspond to a time in the future that is far away enough to guarantee that all the involved services can be contacted beforehand. This process could take hundreds of milliseconds and negatively affect responsiveness. Furthermore, in a pervasive computing environment, services might not always be connected.

In summary, guaranteeing that primary and constraint information are collected at exactly the same time raises implementation challenges and can reduce responsiveness of the system. However, we do not expect that this guarantee is critical for most pervasive computing applications, as opposed to other environments (e.g., military or financial services). Namely, in pervasive computing, constraints involve context-sensitive information. Typically, this information changes little between the time that it is retrieved and the time that the information requested by a client is accessed. For example, since individuals move at a finite speed, the change in an individual's location between retrieving her location from a constraint service and accessing the primary service is limited. To prevent attackers from artificially prolonging this period, a service offering context-sensitive information should indicate for how long the service expects the information to keep its current value. The primary service should return primary information to the client only within the indicated time frame. We take this approach in our implementation.

2.4. Security model

In our security model, services that provide confidential information implement the access-control algorithms described in this paper. The goal of an attacker is to learn confidential information that the attacker is not authorized to access. In order to achieve this goal, an attacker can choose between the following actions: An attacker can send requests to a service and observe their fate. A request is either denied or granted access. In the latter case, the attacker will see the requested information. Alternatively, an attacker can set up services and observe requests reaching such a service. An attacker can also issue (constrained) access rights to information owned by the attacker and snoop network traffic. Attackers can collude. We do not examine other attacks, such as traffic-analysis or statistical-inference attacks or attacks based on the physical observation of a person.

3. Constraints and information leaks

In this section, we define information leaks, as studied in this paper, and discuss how they can occur in client-based access control and how to avoid them.

3.1. Definition

When a single entity or multiple, colluding entities are familiar both with a constraint specification in an access right and with the outcome of a request exploiting this access right, they can infer some knowledge about the constraint information listed in the specification. If the single entity and all of the colluding entities, respectively, are not authorized to access this knowledge, there will be an information leak. (If any of the colluding entities is authorized, there will not be a leak, since the authorized entity could always proxy for the unauthorized entities.) In particular, the leaked knowledge reveals either that the current value of the constraint information is in the set of values listed in the constraint specification or that the current value is not in this set. In inference-control research [10], these two cases are called *positive* and *negative compromises* [11], respectively. We assume that the range of values that constraint information can have is publicly known. Therefore, both for positive and negative compromises, a set of possible current values leaks. If this set contains only one element, the current value leaks. In inference-control research, this scenario is called an *exact compromise* [12]. If this set contains more than one element, there is still a *partial compromise* [12], since the set is smaller than the range of values that the constraint information can have.

To avoid an information leak, the client should be able to gather confidential knowledge about constraint information, such as its current value, only if the client had an access right to this constraint information. However, it is not always possible to enforce this requirement. For example, if the client had an access right to information constrained to particular values of this information (e.g., "Carol grants Alice access to Carol's location if Carol is in her office.") and if the client was denied access to the information, the client could infer a set of possible current values for the information, where this set is disjunct from the set of values that the client is permitted to learn. In the worst case, where the client is permitted to access all but one of possible values, a denied request leaks the current value of the primary information to the client, and there is an exact compromise.

We can generalize this case to the case where multiple access rights depend on each other. For instance, assume that there are two possible values for both Alice's location information and her activity information. Alice grants Bob access to her location information constrained to a particular value of her activity information. She also grants him access to her activity information constrained to a particular value of her location information. If Bob issues a query for Alice's location and is denied access, the probability that Alice is performing the activity not listed in Bob's access right to the location information will be twice as high as the probability that she is performing the listed activity, and there is a partial compromise. It is possible for such a loop to involve more than two access rights.

In this paper, we want to avoid information leaks for the enforceable cases, that is, cases where the information that an access right grants access to is different from the constraint information in the access right and where access rights do not depend on each other.

3.2. Client-based access control

For information leaks to occur, an entity needs to know the constraint specifications in an access right. The following entities know the constraint specifications in the client's access right to the primary information: the issuer of the access right, the client, and the primary service. Let us discuss for each entity how to prevent information from leaking to the entity. For now, we assume that access rights to constraint information are not constrained.

Client: Client-based access control makes the client build proofs of access for the constraint information in the client's access right, as shown in Fig. 1. Without these proofs, the client will not be able to retrieve assurances from the constraint services. Therefore, confidential knowledge about constraint information cannot leak to the client.

Primary service: The primary service could learn confidential knowledge about constraint information from a proof of access received from the client because the proof lists assurances, in addition to the client's access right. Therefore, the client must validate that the primary service has access to the constraint information before sending the proof. Note that since the client's access to the primary information depends on the primary service having access to the constraint information, there is no incentive for the client to perform this validation. For this reason, some form of punitive action (e.g., revoke the client's access right) should be taken for misbehaving clients. As mentioned above, we assume for now that access rights to constraint information are not constrained, so the client does not need to validate access rights of a constraint service when sending a proof to the service.

Issuer: The issuer of the client's access right to the primary information could collude with the client or the primary service to learn confidential knowledge about constraint information in the access right. However, since we ensure that both candidates have access to this knowledge, this is not an information leak, as defined in Section 3.1.

Access rights to constraint information can recursively be constrained, which makes avoiding information leaks more difficult. For simplicity reasons, let us next assume that there is only one level of recursion, that is, if an entity has a constrained access right to constraint information, the entity's access rights to the constraint information in that access right are not constrained. (We discuss the more general case in Section 4.) As discussed above, the client needs to ensure that the primary service has access rights to the constraint information in the client's access right to the primary information. If the service's access rights are constrained, the client has to validate these constraints. Namely, the client has to retrieve constraint information from a constraint service, using access rights issued to the client. If such an access right was constrained and its issuer colluded with the primary service, the issuer would know that whenever the primary service is contacted, the constraints in this access right are satisfied and the issuer could derive confidential knowledge about constraint information in the access right. We can avoid this leak by requiring the client to ensure that the issuer of an access right has access to the constraint information listed in the constraints of the access right before using the access right in a proof of access.

In client-based access control, access rights are typically represented as digital certificates. We have not discussed where an entity that is granted an access right stores the corresponding certificate. The entity could store access rights in a publicly accessible database and retrieve them from this database when building proofs of access. However, if access rights were stored in such a database, the primary service could exploit the information leak just described without having to collude with the issuer of an access right. This observation suggests not storing access rights in a publicly accessible database.

To ensure that constraint information does not leak to an issuer of an access right or to a service, as mentioned above, the client needs to know the issuer's and the service's access rights to this information, respectively. However, if access rights are not publicly available, the client will not easily be able to learn about these access rights and thus might not be able to ask the primary service for the primary information. We can solve this conflict by keeping the types of constraints listed in an access right restricted. A *restricted constraint* in an access right is a constraint whose information is restricted to information about the subject or the issuer of the access right. (As mentioned in Section 2.1, we expect this to be the most useful case in pervasive computing anyway.) Here, if a constraint in an access right granted to the client involves the client, the client itself can decide whether it wants the issuer of the access right or a service to have access to the constraint information. If a constraint involves the issuer of the access right, the issuer automatically has access to the constraint information. In terms of services having access to this information, the issuer could inform the client of these services when issuing the access right to the client. Besides keeping constraints restricted, another option are hidden constraints, which prevent a service from learning the constraint specification in the first place (see Section 5).

In summary, the client must ensure that the primary service has access to the constraint information in the client's access right for the primary information and that the issuer of an access right has access to the constraint information in the access right. Furthermore, access rights should not be publicly available and constraints should be restricted else the client's chances for successfully completing the outlined steps to avoid information leaks (and thus accessing the primary information) decrease.

3.3. Alternative access-control approaches

As mentioned in Section 2.1, apart from client-based access control, where a client resolves constraints, it is also possible to have a centralized entity or a service resolve constraints. Here, we summarize the privacy implications of these approaches, a more detailed discussion is in the first author's Ph.D. thesis [13, Chapter 5].

3.3.1. Centralized access control

In centralized access control, the client sends its request for the primary information to a centralized entity, which runs access control on behalf of individual services. If there is an access right that grants access to the client, the centralized entity will ensure that all the constraints in the access right are satisfied. (We assume that access rights are stored with the centralized entity.) Therefore, the centralized entity has to retrieve the current values of the constraint information from the corresponding constraint services. If all the constraints are satisfied, the centralized entity will retrieve the primary information from the primary service and return the information to the client.

Here, the centralized entity must validate that the client has access to the constraint information in the client's access right to the primary information. Else the client could derive confidential knowledge about the constraint information by observing whether its request for the primary information is granted or denied access. Due to similar reasons as in the case of client-based access control, we can also conclude that the centralized entity must ensure that the issuer of an access right has access to the constraint information in the access right before exploiting this access right in an access decision and that access rights should be kept confidential.

3.3.2. Service-based access control

In service-based access control, the client asks the primary service directly for the primary information. The primary service has to ensure that there is an access right for the client. For a constrained access right, the service has to contact the corresponding constraint services and check satisfaction of the constraints. The constraint services also run access control.

Here, a service must ensure that the client (or another service) has access to the constraint information in the client's (or service's) access right to the requested information. Else the client (or service) could derive confidential knowledge about the constraint information by observing whether its request is granted or denied access. Similar to client-based and centralized access control, a service must also validate that the issuer of an access right has access to the constraint information in the access right and access rights should be kept confidential. Finally, similar to client-based access control, access rights should be kept simple, else it becomes difficult for a service to ensure that a client (or another service) has access to constraint information, assuming access rights are confidential.

4. Access-rights graphs

In the previous section, we have seen that even if we require some access rights to be unconstrained, access control is already difficult. Let us now discuss the general case, where any access right can be constrained. To increase the client's chances to complete access control, we require that access rights are not publicly available (i.e., only the subject and issuer of an access right initially know the contents of the access right).

4.1. Design

Our access-control algorithm for the general case exploits *access-rights graphs*. Such a graph captures relationships between access rights and constraints on them, allows for easy detection of potential problems, like information leaks, loops, or conflicting constraints, and simplifies resolution of constraints.

An access-rights graph is built for particular information in terms of an entity's access rights. The graph represents the conditions under which this entity has access to the

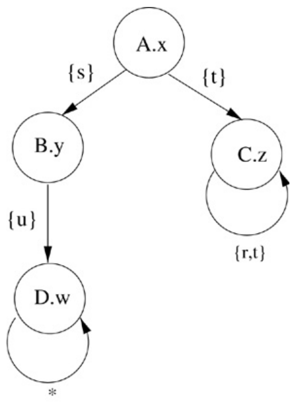

Fig. 2. Access-rights graph. The graph is for information $A.x$ in terms of an entity's access rights. In particular, the entity has access rights to $A.x$ constrained to $B.y = s$ and $C.z = t$, to $B.y$ constrained to $D.w = u$, to $C.z$ constrained to $C.z \in \{r, t\}$, and to $D.w$ in an unconstrained way.

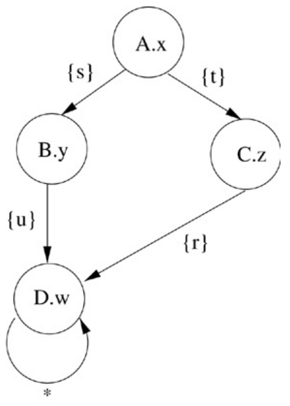

Fig. 3. Access rights graph with conflict. The graph has conflicting constraints on $D.w$.

information. The edges and nodes of the graph are derived from the entity's access rights. In particular, a node in the graph represents information, and the edges outgoing from a node denote the constraints on an entity's access right to the information in the node. An edge has a set of values attached to it, meaning that the information in the node that the edge is pointing to is constrained to the values in the set. If an access right to information is unconstrained, the corresponding node has an outgoing edge that goes back to the node and that is marked with "$*$"; such a node cannot have more than one outgoing edge. We call the node representing the information for which the graph is built the *root node*. Fig. 2 shows an example of an access-rights graph. We use the scheme "Alice.location" for representing information in the graph. The first part (i.e., "Alice") denotes the owner of the information. The second part (i.e., "location") corresponds to the type of information.

We call an access-rights graph *conflict-free* if for nodes with multiple incoming edges, the intersection of the sets of values attached to these edges is not empty. Fig. 3 shows an example of a graph with a conflict. There could be multiple graphs for the same information in terms of an entity's access rights if the entity had multiple access rights to this information, but with different constraints on them.

The entity whose access rights are used for building an access-rights graph has access to the information in the root node if (1) each node has at least one outgoing edge (i.e., there is an access right to the information in the node), (2) the graph is conflict-free, and (3) the current value of the information in each node is listed in each of the node's incoming edges.

Assuming that an entity's access rights are locally available, building a conflict-free access rights graph out of these access rights is a completely local step and straightforward. We present the pseudocode in Fig. 4. Ensuring that each constraint is satisfied requires traversal of the graph and contacting remote services that offer the information in a node. This graph traversal is called *resolution*. Resolution is bottom up; we discuss resolution for client-based access control in Section 4.2. For the other access-control approaches, we refer to the first author's Ph.D. thesis [13, Chapter 5].

Access-rights graphs can become arbitrarily complex. At present, we expect them to be rather simple for practical scenarios, such as the graph in Fig. 2. This expectation is based on two observations: First, people typically specify access rights in a manual way, which tends to lead to simple access rights, having no or only a few, broad constraints. Second, the amount of context information that is currently available about people and that can be used to constrain access is still limited. However, both observations probably will no longer hold in the future. If users let electronic agents manage access rights on their behalf, access rights will become more complicated and can involve more and narrower constraints. Also, the amount of information available about people keeps increasing.

4.2. *Client-based access control*

Let us now discuss how we employ access-rights graphs in client-based access control. Here, the client must build proofs of access for the primary and constraint information. In particular, the client builds a conflict-free access-rights graph for the primary information in terms of its access rights and assembles proofs for the nodes in the graph while resolving this graph. In addition, the client must ensure that no confidential knowledge about constraint information leaks to a service receiving a proof of access or to an issuer of an access right. Therefore, for each node in the graph, the client builds additional access-rights graphs for the information in the nodes pointed to by that node. These graphs are either in terms of the access rights of the service offering the information in that node or in terms of the access rights of the issuer of the access right associated with that node. In particular, the client implements the algorithm shown in Fig. 5. To make it easier for the client to build access-rights graphs in terms of a service's or issuer's access rights, constraints in an access right should be restricted, as defined in Section 3.2.

While resolving an access-rights graph, the client needs to build proofs of access. When contacting a constraint service, the client might receive an assurance stating that a constraint is satisfied. Once it has received assurances for all the nodes that a node is pointing to, it can build a proof of access for the information in this node and contact the corresponding constraint (or primary) service. For example, in the graph shown in Fig. 2,

```
// The algorithm assumes that nodes contains the current set of nodes in the access-
// rights graph. To start with, nodes contains the root node of the graph and the
// algorithm is called for this node. In addition, rights contains the access rights of the
// entity in terms of which the graph is built. The algorithm also assumes the existence
// of a routine boolean has_conflict(Node node, Values values) that returns
// true if the intersection of the values attached to any incoming edge of node
// and values is empty.

// Return true if there is conflict-free access-rights graph for the information in the node.
boolean build_graph(Node current_node)
{
  // Retrieve access rights to information in current_node.
  // (There could be multiple access rights (and thereby graphs).)
  Set rights = rights.retrieve(current_node.get_information());
  while (rights.notEmpty()) {
    AccessRight right = rights.remove();
    Constraints constraints = right.get_constraints();
    while (constraints.notEmpty()) {
      Constraint constraint = constraint.remove();
      // If there already is a node for the information in the constraint, retrieve it.
      Node node = nodes.retrieve(constraint.get_information);
      if (node != null) {
        // Go through node's incoming edges and check for conflict with values
        // permitted by constraint.
        if (has_conflict(node, constraint.get_values()) break ;
        // Establish an edge from current_node to existing node listing values in constraint.
        nodes.establish_edge(current_node, node, constraint.get_values());
      } else {
        // Establish new node for constraint information and edge to it from current_node.
        Node new_node = nodes.establish_node_and_edge(constraint,
          current_node);
        // Try to build subgraph for information in this node.
        if (!(build_graph(new_node))) {
          nodes.remove_node_and_edge(new_node);
          break ;
        }
      }
    }
    return true;
  }
  return false;
}
```

Fig. 4. Building of access-rights graphs. The pseudocode gives the algorithm for building a conflict-free access-rights graph.

the client first retrieves an assurance for $D.w = u$ from the constraint service offering $D.w$, using its access right as a proof of access. The client then uses this assurance and its access right to $B.y$ to build a proof of access for getting an assurance for $B.y = s$. Similarly, it gets an assurance for $C.z = t$. These two assurances and the access right to $A.x$ allow the client to build a proof of access for $A.x$. The service offering $A.x$ validates the proof and returns the current value of $A.x$.

```
// Return true if entity has access to information at given value.
boolean can_access(Entity entity, Information information, Value value) {
  Set graphs = conflict-free access-rights graphs with at least one outgoing edge per node
    for information in terms of entity's access rights. If value != null and information
    in root node of a graph is constrained to particular values of this information, value
    must be contained in these values.
  while (graphs.notEmpty()) {
    Graph graph = graphs.remove();
    if (is_resolvable(graph)) return true;
  }
  return false;
}

// Return true if the constraints in the access-rights graph are satisfied and if there are
// no information leaks.
boolean is_resolvable(Graph graph) {
  // Gather nodes that can be resolved.
  Set readySet = all nodes in graph with no outgoing edges other than an edge to itself;
  while (readySet.notEmpty()) {
    Node node = readySet.remove();
    Information information = node.get_information();
    Value value = retrieve signed statement containing current value of information
      from constraint service, using access right associated with node and previously
      gathered assurances;
    if (value is not listed in all incoming edges of node) return false;
    // We now have an assurance for the node. Next, ensure that issuer of an access right
    // and services receiving access right in proof of access can access constraint
    // information in the access right.
    parents = nodes with an outgoing edge to node;
    while (parents.notEmpty()) {
      Node parent = parents.remove();
      Entity owner = parent.get_information().get_owner();
      if (!(can_access(owner, information, value))) return false;
      Entity service = service offering parent.get_information();
      if (!(can_access(service, information, value))) return false; (*)
      if (all nodes with incoming edge from parent have been removed from readySet)
        readySet.add(parent);
    }
  }
  return true;
}
```

Fig. 5. Access-control algorithm. Access control consist of building a conflict-free access-rights graph and of resolving this graph. In addition, access control must recursively ensure that issuers of access rights and services receiving proofs of access can access constraint information.

Proof building becomes difficult for conflict-free access-rights graphs with loops involving more than one node, since there is no obvious node at which a client can start resolution. There are multiple ways to deal with such cases. If the information of all the nodes in the loop was offered by the same service, a client could have this service resolve the loop. If different services offered this information, a client can contact some of these services and ask them to resolve the constraints on its behalf. However, this option requires trust relationships between the services before exchanging constraint information. None of this information must leak to the client unless all the constraints are satisfied.

5. Hidden constraints

In this section, we introduce the concept of hidden constraints and apply it to client-based access control.

5.1. Design

In our scenario, the client can access the primary information only if both the client and the primary service have access to the constraint information in the client's access right to the primary information. In practice, this requirement can lead to owners of constraint information granting the primary service access to the information to ensure that the client can access the primary information. This approach is problematic since intruders into the service can exploit the service's access rights. Alternatively, if an owner of information is not willing to grant the primary service access, the client will not be able to access the primary information. For example, assume that Alice uses a service for providing important information about her and that Bob has no trust relationship with this service. Alice grants Bob an access right to the information, given that he is at a particular location. Bob is now in a dilemma: Either he releases his location to the untrusted service in his proof of access or he cannot learn Alice's important information.

We now propose a solution that increases the number of cases where the client can access the primary information and that does not require owners of constraint information to issue access rights to the primary service. Our solution exploits *hidden constraints*. According to our definition of an information leak in Section 3.1, an entity must know the constraint specification in an access right in order to be able to derive confidential knowledge when observing the outcome of requests exploiting this access right. However, if a constraint specification is hidden from the entity, observing requests will not allow the entity to infer this confidential knowledge. In our example above, Alice can issue the access right to her important information such that the constraint in the access right remains hidden from the service providing the important information. Therefore, the service cannot learn the specification from the proof of access and will not be able to learn Bob's location.

Note that hidden constraints do not hide the existence of a constraint in an access right from an entity, they hide only its specification. Furthermore, hiding a constraint specification from an entity does not mean that the entity can never learn the specification. If the entity had access to the constraint information in the specification, it could learn the specification by observing the system. However, this is not an information leak, since the entity has access to the constraint information.

5.2. Client-based access control

A constraint specification consists of constraint information, a set of permitted values, and the identity of the constraint service responsible for acknowledging constraint satisfaction. Let us now explore which parts of a constraint specification we can hide from which entity in client-based access control.

Obviously, we cannot keep the constraint specifications in an access right secret from the issuer of the access right. We can hide a constraint specification entirely from a service. Namely, a service is not interested in this specification; it wants to know only whether the corresponding constraint is satisfied. To support this feature, the issuer of an access right needs to associate a constraint with the access right such that a service cannot learn the constraint specification when looking at the access right or at an assurance in a proof of access. However, the client building this proof remains able to gather assurances for the constraint. We present implementations of this concept based on digital certificates and based on one-way chains in Section 6.2. Such a hidden constraint prevents confidential knowledge from leaking to a service. Namely, hidden constraints eliminate the check marked with ($*$) in the access-control algorithm in Fig. 5.

It is not possible to hide a constraint specification entirely from the client since the client must know the identity of the constraint service responsible for resolving the constraint. We can hide only the constraint information and the set of permitted values from the client. (For example, the issuer of an access right encrypts the two items with the public key of the responsible constraint service.) However, depending on the type of constraint information or service, knowing the constraint service might allow the client to deduce the type of constraint information (e.g., when a constraint service provides only one type of information), the owner of the constraint information (e.g., when a constraint service provides only one individual's information), or even the value of the constraint information (e.g., when a constraint service has limited coverage, such as a location service covering only one building). Due to these reasons, our access-control architecture presented in the next section supports hiding constraints only from a service, but not from the client.

5.3. Alternative access-control approaches

Hidden constraints are also applicable to centralized and service-based access control, as introduced in Section 3.3. In these two approaches, hidden constraints allow us to hide a constraint specification from a client. We provide a more detailed discussion in the first author's Ph.D. thesis [13, Chapter 5].

Let us summarize what information leaks we have to avoid for the different access-control approaches. We present this summary in Table 1. For an entity running access control (or building a proof of access in the case of client-based access control) depending on a constrained access right, this entity needs to ensure that the principals listed in the table have access to the different types of constraint information in the access right.

6. Architecture

We now present a client-based access-control architecture that focuses on access rights with context-sensitive constraints. We give an overview of our architecture and take a closer look at the implementation of hidden constraints.

Table 1
Information leaks

Approach	Constraints	
	Non-hidden	Hidden
Client-based	Service, issuer	Issuer
Centralized	Client, issuer	Issuer
Service-based	Client, issuer	Issuer

For each approach, we show the potential information leaks due to deduction of constraint information. For client-based access control, constraints are hidden from a service. For centralized and service-based access control, they are hidden from a client.

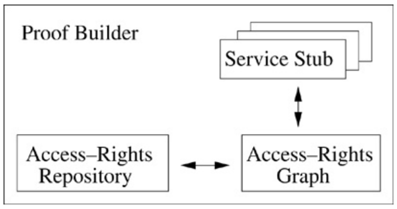

Fig. 6. Proof-building architecture. The access-rights-graph component interacts with the access-rights repository for building a graph and with service stubs for resolution of the graph.

6.1. Overview

Fig. 6 gives an overview of the client's components involved in proof building. The access-rights-graph component is responsible for building and resolving access-rights graphs. This component retrieves required access rights from the access-rights repository and implements the algorithm given in Fig. 5. While resolving a graph, the component asks service stubs to get an assurance (for nodes other than the root node) or the value of the information (for the root node). A service stub knows how to interact with a service. We use extended SPKI/SDSI digital certificates [14] for expressing access rights and assurances. We give three example statements in Fig. 7. In the first example, Carol grants Alice access to her calendar information if Alice is in Wean Hall 4103 and if an additional, hidden constraint is fulfilled. (We explain the implementation of hidden constraints in the next section.) As explained in Section 2.2, Carol, as the issuer of the access right, must indicate the identity of the constraint service responsible for resolving a constraint when defining this constraint. The second and third example show an assurance for the non-hidden and hidden constraint, respectively.

6.2. Hidden constraints

We now discuss how we hide constraints from services. Here, the issuer of a constrained access right includes only a reference to the constraint specification in the access right, but not the actual specification. There are multiple ways to implement such a scheme. We discuss an approach based on digital certificates and another one based on one-way chains.

```
(cert
    (issuer (public_key:carol))
    (subject (public_key:alice))
    (permission (information (public_key:carol) calendar))
    (tag
        (nonhidden-constraint
            (information (public_key:alice) location)
            (``Wean Hall 4103'')
            (public_key:service))
        (hidden-constraint
            (public_key:validator))))

(assurance
    (issuer (public_key:service))
    (nonhidden-constraint
        (information (public_key:alice) location)
        (``Wean Hall 4103'')
        (public_key:service)))

(assurance
    (issuer (public_key:validator)))
```

Fig. 7. Extended SPKI/SDSI certificates. We show an access right with constraints and two assurances. Signatures and lifetimes are omitted.

6.2.1. Digital certificates

In the first approach, a constraint service signs a statement declaring that a constraint is satisfied. The signature has a lifetime corresponding to the time frame during which the constraint service believes the constraint to remain satisfied, as explained in Section 2.3.

Namely, the issuer of an access right includes a public key, H, in the access right, where H serves as a reference to a constraint specification. This public key will also be used for validating assurances signed with the corresponding private key, H^{-1}. The issuer of the access right should generate H and H^{-1}. To avoid information gathering based on correlation, the issuer should not re-use H in different access rights. The constraint specification referred to by H is also defined by the issuer and consists of the following parts:

Constraint definition. This part lists the constraint information and a set of permitted values.

Signing key. The signing key corresponds to private key H^{-1}. It is encrypted with the public key of a constraint service, S.[1] By choosing this encryption key, the issuer of the access right and of the constraint specification picks the constraint service that provides the constraint information.

Validation key. The validation key corresponds to public key H.

Public key of service. This part lists the public key of the constraint service, S.

Integrity data. This data ensures the integrity of the constraint specification. We use a cryptographic hash of the constraint specification (excluding signing key and integrity data) and encrypt this hash together with the signing key.

[1] We use an AES-based hybrid encryption scheme and HMAC for integrity checking.

This constraint specification and the access right containing reference H to it are used as follows: Their issuer gives both of them to the client. When building a proof of access, the client retrieves the identity of the constraint service, S, from the specification and gives the constraint specification to S. This service ensures that the current value of the constraint information corresponds to one of the permitted values. It then decrypts the ciphertext in the specification to get H^{-1} and to ensure that the specification has not been tampered with. Next, it uses H^{-1} to issue an assurance in the form of a digital certificate. The assurance consists of the validation key, H, signed with the signing key, H^{-1}. The signature has a lifetime corresponding to the time frame during which the constraint service expects the constraint to remain satisfied. Next, the client sends the access right, together with the assurance, to the primary service, which validates the signature of the access right. For reference H included in the access right, the service ensures that there is an assurance covering H and signed with H^{-1}. Note that the service never sees the actual constraint specification.

This approach has the advantage that a constraint service can freely choose the lifetime of a signature covering an assurance, depending on the service's domain-specific knowledge. The drawback of the approach is that issuing a digital signature can be an expensive operation. (We present some measurements in Section 7.)

6.2.2. One-way chains

Our second option does not require the generation of a potentially expensive digital signature when issuing an assurance. Instead, it is based on one-way chains. A one-way chain consists of a seed value a_n and repeated applications of a public one-way function, f, to this seed, or $a_{i-1} = f(a_i)$ for $0 < i \leq n$. a_0 is publicly known, whereas a_n is available only to a constraint service. The idea is to have the constraint service gradually release values in the chain, starting with a_1, and to have the primary service ensure that a released value is part of the chain, based on a_0. One-way chains guarantee that a released value, a_i, cannot be used for computing values to be released in the future, a_j $(j > i)$.

In more detail, our approach looks as follows: For each hidden constraint, the issuer of an access right chooses n and a_n and computes a_0. The issuer also picks a time in the future, T_0, and a time interval, ΔT. In our formal model, the tuple $\langle n, a_0, T_0, \Delta T\rangle$ represents $\tilde{H}$ and a_n represents $\tilde{H}^{-1}$. Given a tuple, it is possible to compute T_i with $T_i = T_{i-1} + \Delta T$ for $0 < i < n$. We associate a_i with time frame $[T_{i-1}, T_i]$ (see Fig. 8). If a constraint is fulfilled in time frame $[T_{i-1}, T_i]$, a constraint service releases a_i. If the primary service manages to recompute a_0 based on a_i, it will know that the constraint is satisfied. For one-way chains, the constraint specification looks as follows:

Constraint definition. This part is identical to the certificate-based case.

Seed. This is the seed of the one-way chain, a_n. It is encrypted with the public key of the constraint service, S, that provides information $B.x$.

Validation tuple. The validation tuple corresponds to the tuple $\langle n, a_0, T_0, \Delta T\rangle$.

Public key of service. This part lists the public key of the constraint service, S.

Integrity data. This part is identical to the certificate-based case.

When receiving a constraint specification from a client, a constraint service ensures that the constraint is satisfied and decrypts the ciphertext to get a_n and to detect tampering

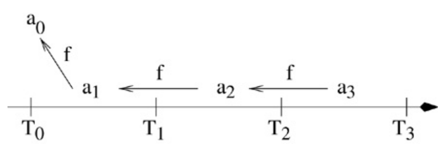

Fig. 8. One-way chain. Each value in the chain, a_i, is associated with a time frame. Value a_0 validates values in the chain.

attacks. Based on the validation tuple, it determines the number of applications of f to a_n that are required to get a_i for the current time frame. Value a_i serves as assurance and is given to the client. The client forwards the value to the primary service. From the tuple $\langle n, a_0, T_0, \Delta T \rangle$ and the current time, this service figures out the number of applications of f to a_i required to get a_0. If the service computes a wrong value, either the client is cheating or the current time frame has moved since the constraint service computed a_i. (We assume reasonably synchronized clocks, depending on the granularity of ΔT.) In the latter case, the client will issue another request. When the primary service manages to recompute a_0 from a_i, it can cache a_i and use the cached value for validating future requests.

This approach has the advantage that it is less expensive for a constraint service. In particular, computationally cheap cryptographic hash functions (e.g., SHA-1) can be used for implementing f. Another advantage is that the approach allows trading off performance vs. convenience when deciding about the length of a one-way chain. A one-way chain covers only a limited time period. After this time period, a new chain must be created and the access right using the chain needs to be re-issued. One the one hand, longer chains require fewer updates of access rights. On the other hand, longer chains make it more expensive for a constraint service or a primary service to compute values in the chain. The approach has the drawback that the issuer of an access right chooses the length of the time interval, ΔT, during which the issuer expects a constraint to remain satisfied. A constraint service cannot incorporate its own domain-specific knowledge. However, for some information, this limitation is not a problem. For example, for access rights constrained to location information, an issuer can choose an interval of a few minutes. Since the speed at which individuals move is finite, the error in location remains limited.

Both the approach based on one-way chains and the one based on digital certificates require a constraint service to perform an asymmetric decryption operation, which can be expensive. However, it is possible for a constraint service to cache decrypted values. In this way, when a service is asked to issue an assurance for the same constraint multiple times, it needs to perform a decryption operation only for the first request.

7. Performance analysis

We present a performance analysis of our access-control architecture. Our implementation is in Java and based on an existing access-control framework for Web environments [9]. We deploy it in the Aura pervasive computing environment [15]. SSL provides peer authentication and confidentiality and integrity of transmitted messages. We run our measurements on a Pentium IV/2.5 GHz with 1.5 GB of memory, Linux 2.4.20, and Java 1.4.2. Our asymmetric cryptographic operations employ 1024 bit RSA keys. We use SHA-1 for building one-way chains and authenticating constraint specifications. (Our machine can compute about 330,000 Java-based hashes/s) An experiment is run 100 times. We report the mean and standard deviation (in parentheses).

We study the cost of access control when Alice grants Bob access to her calendar information under different constraints. In the first experiment, Alice grants access only if she is currently in her office. Alice does not hide this constraint. In the second experiment, Alice grants access only if Bob is currently in his office. Alice hides this constraint. If Alice did not hide the constraint, Bob would have to reveal his location to the calendar service, which he might not be willing to do and thus would not be able to access Alice's calendar. The third experiment is identical to the second one, but the constraint service caches decrypted signing keys. Our location service fingers a person's desktop computer and determines her location based on her activity. Our calendar service is based on Oracle CorporateTime. In the fourth and fifth experiment, we repeat the second and third experiment, but the hidden constraint is based on a one-way chain, instead of certificates.

The implementation builds an access-rights graph for Carol's calendar information in terms of Alice's access rights. In the first experiment, the implementation learns that it must retrieves an assurance for Carol's location before accessing the calendar information. It exploits Alice's access right to retrieve this assurance from the location service. (The location service fingers Carol's desktop computer and determines Carol's location based on her activity.) Next, the implementation ensures that the calendar service is authorized to access Carol's location information using the access right granted to the service. Similarly, Carol must have access to her location information, which is straightforward to validate. Finally, the implementation combines the assurance received from the location service and Alice's access right to the calendar information in a proof of access and sends the proof to the calendar service, which is a centralized calendar system running Oracle CorporateTime.

In the first experiment, the mean response time experienced by the client is 463 ms (26 ms). Detailed results are in Table 2. About 25% of the cost is due to retrieving an assurance from the constraint service. Most of this overhead is caused by setting up an SSL connection, which requires two costly RSA decryption/signing operations (about 16 ms each), and by acquiring the location information. The cost for issuing an assurance corresponds to the cost of generating a digital signature. Access control takes only a few milliseconds, the main cost is checking a signature. Constraint processing has only limited influence on the primary service. In addition to checking the signature of the client's access right, it now also needs to validate the signature of the assurance. The main cost is due to setting up SSL and the retrieval of the requested information. This SSL setup is more expensive than the first one. Closer inspection reveals that the additional delay is due to Java's garbage collection triggering during the setup.[2]

[2] Choosing different amounts of memory allocations or using incremental garbage collection has no or negative influence on the results.

Table 2
Client-response time

Entity	Step	Non-hidden		Hidden		Hidden, w/caching	
		μ	(σ)	μ	(σ)	μ	(σ)
Client/constraint service	**SSL socket creation**	50	(3)	50	(3)	50	(3)
Constraint service	Deserialization	13	(2)	18	(2)	18	(3)
Constraint service	**Access control**	3	(1)	4	(2)	3	(2)
Constraint service	Retrieve location	37	(3)	38	(3)	38	(3)
Constraint service	**Issue assurance**	17	(1)	35	(1)	17	(1)
Client/primary service	**SSL socket creation**	92	(12)	96	(16)	96	(16)
Primary service	Deserialization	23	(4)	21	(7)	20	(2)
Primary service	**Access control**	5	(2)	5	(2)	5	(2)
Primary service	Retrieve calendar entry	202	(23)	204	(16)	201	(11)
	Total	463	(26)	485	(14)	469	(15)

Mean and standard deviation of elapsed time for security operations (in bold) and for other, expensive operations using either non-hidden or hidden, certificate-based constraints [ms].

In the other experiments, the implementation must retrieve an assurance for Alice's location instead of Carol's location. Furthermore, the implementation does not have to verify that the calendar service has access to Alice's location, since the constraint is hidden from the service. The implementation still needs to ensure that Carol as the issuer of the access right with Alice's location information as constraint information can access this information. The implementation can use an access right previously issued by Alice for this purpose, or it could ask Alice whether she wants to release her location to Carol. Finally, the proof of access is built in the same way as in the first experiment.

In the second experiment, we use a hidden constraint based on digital certificates. The mean response time increases to 485 ms (14 ms). As shown in Table 2, the main cause for this increase is the larger cost for creating an assurance. Namely, the constraint service needs to decrypt the ciphertext in the constraint specification before it can issue an assurance. Deserialization cost also becomes larger since the constraint specification is now separate from the access right. The third experiment is identical to the second one, but the constraint service caches decrypted ciphertexts. Therefore, the cost for issuing an assurance is reduced to the cost for generating a signature, as presented in Table 2.

For the next experiments, we use a hidden constraint based on one-way chains in Carol's access right. The chain has 2016 steps and $\Delta T = 5$ min, that is, the chain covers a one-week period. During the lifetime of the one-way chain, the number of hash operations to be computed changes both for the constraint service and for the primary service. For example, at the beginning, the constraint service needs to compute 2015 steps, after $1/2$ week 1008 steps, and after one week zero. This observation is confirmed by the first two curves in Fig. 9, which show (a) the constraint service's cost for issuing an assurance and (b) the cumulative cost for issuing this assurance and for the primary service's cost for running access control. The cumulative cost remains constant. The results can be explained with the help of Fig. 8: The number of hash operations required for computing the value associated with a time frame decreases over time, whereas the number of hash operations required for

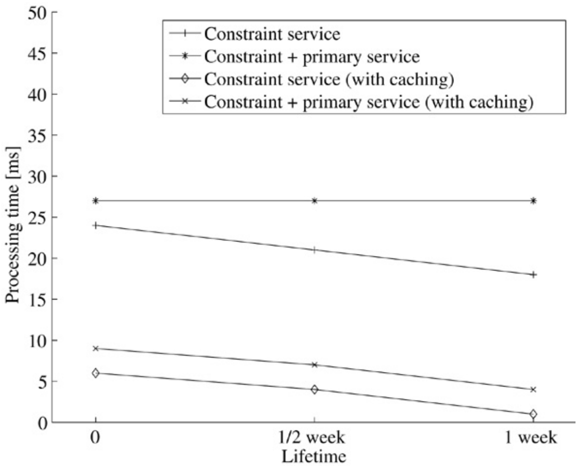

Fig. 9. Hidden, one-way chain-based constraints. The curves show the processing time it takes the constraint service to issue an assurance and the cumulative cost of issuing this assurance and of the primary service running access control, varying over the lifetime of the hash chain.

this value's validation increases, so the total number of operations remains constant. For this experiment, the mean client response time is 468 ms (23 ms), which is less than in the case with hidden, certificate-based constraints (without caching).

Fig. 9 also shows the processing time for the case where the constraint service caches decrypted ciphertexts and where the primary service caches chain values that it receives from the client and that it successfully validates. The cumulative cost is minimal after one week, when the constraint service does not need to compute any hash operations and just returns the seed of the chain. Here, the mean client-response time is 439 ms (14 ms).

8. Related work

Multiple pervasive computing environments support context-sensitive access control to confidential information [1,2,4,5]. Al-Muhtadi et al. [1], Chen et al. [2], and Gandon and Sadeh [4] each employ centralized rule engines for running access control. None of them discusses whether and how they address information leaks caused by constraints. Minami and Kotz [5] present an access-control architecture where services resolve constraints. Access rights are publicly available in their architecture. To be able to ensure satisfaction of constraints, the primary service needs to have access rights to the constraint information listed in the client's access right to the primary information. The authors assume that those access rights are never constrained. This limitation avoids information leaks where the client exploits publicly available access rights to derive confidential knowledge about constraint information in the service's access rights.

Covington et al. [3,16], Neumann and Strembeck [17], and Bacon et al. [18] add context awareness to role-based access control. None of the approaches considers information leaks caused by context-sensitive constraints.

Classic access-control models, such as mandatory access control, discretionary access control, or role-based access control, have no or very limited support for context-sensitive access rights to information. This limitation has been addressed in newer models, such as UCON_{ABC} [19] or GAA API [17]. Both models support context-sensitive constraints, but the authors do not discuss information leaks caused by context-sensitive constraints.

McDaniel [20] discusses various evaluation issues for constraints in a distributed environment, lists desired security properties (e.g., non-repudiation), and reviews different implementation approaches. He does not discuss information leaks caused by constraints.

Hidden constraints keep constraint specifications secret from clients or services. In automated trust negotiation [21], a client or a service keeps its credentials or access policy initially secret from a service or client, respectively. Here, after successfully completing negotiations between the client and the service, a party will typically know the other party's policy or credentials. For hidden constraints, the constraint specification must remain secret from a client or service throughout the processing of a query. A client or service might eventually deduce a constraint specification if it was authorized to access the listed constraint information, which is not an information leak.

9. Conclusions and future work

We showed that context-sensitive constraints on access rights can lead to privacy violations and discussed how to avoid these violations. We also introduced the concepts of access-rights graphs and hidden constraints. Access-rights graphs represent the conditions under which access should be granted. Hidden constraints avoid information leaks by keeping constraint specifications secret. We presented a distributed, context-sensitive access-control architecture that avoids privacy violations. Our implementation and its evaluation demonstrate the feasibility of our approach.

Our discussion revealed that access rights should not be publicly available and that constraints should be kept restricted, otherwise running the access-control algorithm can become complex. In particular, constraints should involve either a subject being granted an access right or an entity issuing an access right.

We are deploying our access-control infrastructure in additional services to investigate what kind of access rights and constraints users define on them. Another area of future work is investigating how users actually specify access rights, either directly or through agents. Finally, there is need for a method that eases the debugging or justification of access-control decisions, particularly when hidden constraints are used.

Acknowledgments

We thank the anonymous reviewers for their comments. This research was supported by the Army Research Office through grant number DAAD19-02-1-0389 and by the NSF under award number CNS-0411116.

References

[1] J. Al-Muhtadi, A. Ranganathan, R. Campbell, M.D. Mickunas, Cerberus: A context-aware security scheme for smart spaces, in: Proceedings of IEEE International Conference on Pervasive Computing and Communications, PerCom 2003, 2003, pp. 489–496.

[2] H. Chen, T. Finin, A. Joshi, Semantic web in the context broker architecture, in: Proceedings of 2nd IEEE International Conference on Pervasive Computing and Communications, PerCom 2004, 2004, pp. 277–286.
[3] M.J. Covington, P. Fogla, Z. Zhan, M. Ahamad, A context-aware security architecture for emerging applications, in: Proceedings of 18th Annual Computer Security Applications Conference, ACSAC 2002, 2002.
[4] F. Gandon, N. Sadeh, A semantic Ewallet to reconcile privacy and context awareness, in: Proceedings of 2nd International Semantic Web Conference, ISWC2003, 2003.
[5] K. Minami, D. Kotz, Secure context-sensitive authorization, Journal of Pervasive and Mobile Computing (PMC) 1 (1) (2005).
[6] G. Myles, A. Friday, N. Davies, Preserving privacy in environments with location-based applications, Pervasive Computing 2 (1) (2003) 56–64.
[7] U. Hengartner, P. Steenkiste, Avoiding privacy violations caused by context-sensitive services, in: Proceedings of 4th IEEE International Conference on Pervasive Computing and Communications, PerCom 2006, 2006, pp. 222–231.
[8] L. Bauer, M.A. Schneider, E.W. Felten, A general and flexible access-control system for the web, in: Proceedings of 11th Usenix Security Symposium, 2002, pp. 93–108.
[9] J. Howell, D. Kotz, End-to-end authorization, in: Proceedings of 4th Symposium on Operating System Design & Implementation, OSDI 2000, 2000, pp. 151–164.
[10] D.E. Denning, Cryptography and Data Security, Addison Wesley, ISBN: 0-201-10150-5, 1982.
[11] S. Castano, M. Fugini, G. Martella, P. Samarati, Database Security, Addison Wesley Professional, ISBN: 0-201-59375-0, 1994.
[12] N.R. Adam, J.C. Wortmann, Security-control methods for statistical databases: A comparative study, ACM Computing Surveys (CSUR) 21 (4) (1989) 515–556.
[13] U. Hengartner, Access control to information in pervasive computing environments, Ph.D. Thesis, Computer Science Department, Carnegie Mellon University, available as Technical Report CMU-CS-05-160, August 2005.
[14] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, T. Ylonen, SPKI certificate theory, RFC 2693, September 1999.
[15] D. Garlan, D. Siewiorek, A. Smailagic, P. Steenkiste, Project aura: Towards distraction-free pervasive computing, IEEE Pervasive Computing 1 (2) (2002) 22–31.
[16] M.J. Covington, W. Long, S. Srinivasan, A. Dey, M. Ahamad, G. Abowd, Securing context-aware applications using environment roles, in: Proceedings of 6th ACM Symposium on Access Control Models and Technologies, SACMAT '01, 2001, pp. 10–20.
[17] G. Neumann, M. Strembeck, An approach to engineer and enforce context constraints in an RBAC environment, in: Proceedings of 8th ACM Symposium on Access Control Models and Technologies, SACMAT 2003, 2003, pp. 65–79.
[18] J. Bacon, K. Moody, W. Yao, A model of OASIS role-based access control and its support for active security, ACM Transactions on Information and System Security (TISSEC) 5 (4) (2002) 492–540.
[19] J. Park, R. Sandhu, The UCON_{ABC} usage control model, ACM Transactions on Information and System Security (TISSEC) 7 (1) (2004) 128–174.
[20] P. McDaniel, On context in authorization policy, in: Proceedings of 8th ACM Symposium on Access Control Models and Technologies, SACMAT 2003, 2003, pp. 80–89.
[21] W.H. Winsborough, N. Li, Safety in automated trust negotiation, in: Proceedings of 2004 IEEE Symposium on Security and Privacy, 2004, pp. 147–160.

Urs Hengartner is an assistant professor in the David R. Cheriton School of Computer Science at the University of Waterloo, Canada. His research interests are in information privacy and in computer and networks security. His current research focuses on security and privacy issues that affect future computing environments, such as pervasive computing or location-based services, and on applied cryptography. He has a degree in computer science from ETH Zürich and an MS and Ph.D. in computer science from Carnegie Mellon. His web page is http://www.cs.uwaterloo.ca/~uhengart.

Peter Steenkiste is a professor of Computer Science and of Electrical and Computer Engineering at Carnegie Mellon University. His research interests are in the areas of networking and distributed systems. He has done research in high-performance networking and distributed computing, network quality of service, overlay networking, and autonomic computing. His current research is the areas of wireless networking, pervasive computing, and self-management network services. He has an engineering degree from the University of Gent, Belgium, and an MS and Ph.D. in Electrical Engineering from Stanford University. He is a senior member of the IEEE and a member of the ACM. His web page is http://www.cs.cmu.edu/˜prs.