Avoiding Privacy Violations Caused by Context-Sensitive Services

Urs Hengartner
University of Waterloo
uhengart@cs.uwaterloo.ca

Peter Steenkiste
Carnegie Mellon University
prs@cs.cmu.edu

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 an individual to be at a particular location in order to let the individual use a printer or learn her friends' location. However, constraining access to a resource based on confidential information about a person's context could result in privacy violations. For instance, if access is constrained based on a person's location, granting or rejecting access will provide information about this person's location and could violate the person's privacy. We introduce an access-control algorithm that avoids privacy violations caused by context-sensitive services. Our algorithm exploits the concepts 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, a sample implementation, and a performance evaluation.

1. Introduction

The increasing numbers of networked devices (e.g., cellphones or handhelds) that individuals are carrying and of networked sensors (e.g., cameras) let more context-sensitive information about people become 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, 4, 5, 8, 13, 14]. 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 centralized calendar system 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

Proceedings of the Fourth Annual IEEE International Conference on Pervasive Computing and Communications (PERCOM'06)
0-7695-2518-0/06 $20.00 © 2006 IEEE

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 more simplified 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, an example implementation of this architecture, and a performance evaluation.

We start by introducing our system model (Section 2). We then focus on a restricted set of constraints and discuss information leaks that these constraints could cause 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 measure 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 and present our security 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 activity information.

We assume that a constraint consists of information and of 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*. 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. Here, we assume that, for a particular type of information, there is only one service that provides this type of information. We discuss the more general case in the extended version of this paper [10, Chapter 5].

There are multiple approaches to deploy access control in this scenario. We concentrate on *client-based access con-*

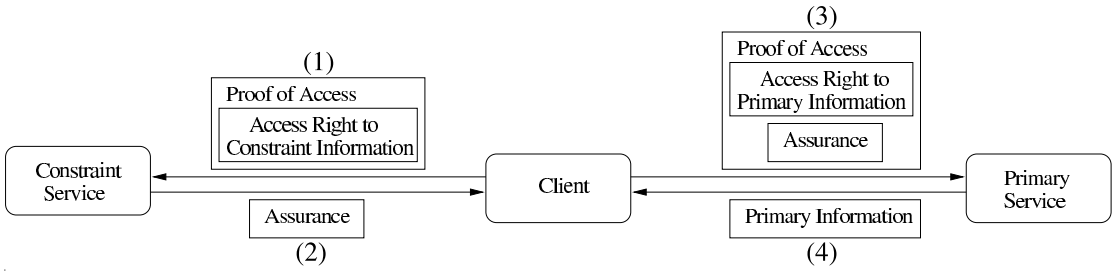

Figure 1. Client-based access control. The client sends a proof of access to the constraint service to retrieve an assurance for the constraint information. Next, the client sends a proof of access, including the assurance, to the primary service to access the primary information. (Numbers indicate order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

trol [3, 11], where the client needs to prove to the primary service that the client 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 Figure 1. Both access rights and assurances can be represented as digital certificates. If the client's 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. We study approaches that employ a centralized entity or that have a service resolve constraints in the extended version [10, Chapter 5], where we observe that similar privacy violations can occur in all approaches.

A constraint service returning an assurance indicates for how long it expects the corresponding constraint to remain satisfied. The primary service should return primary information only within this time window. (We assume reasonably synchronized clocks.) While it is possible that a constraint service errs and that a constraint does become invalid within the indicated window, we believe that our approach is sufficient for context-sensitive constraints. For example, individuals move at a finite speed, which limits the possible change in their location within a (short) time window. Instead of a window-based approach, it is also possible to timestamp requests. However, such an approach raises several implementation challenges [10, Chapter 5].

2.3. 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. We assume that the range of values that constraint information can have is publicly known. Therefore, both cases leak a set of possible current values. If the leaked set contains only one element, the current value leaks, and there is an *exact compromise*. If the leaked set contains more than one element, there is still a *partial compromise*, since the set is smaller than the range of values that the constraint information can have.

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 Figure 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 since the proof also 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 to the service. (Since the client's access rights to the constraint information are not constrained, the client does not need to perform this check when sending a proof of access to a constraint service.) In case the client is not willing to perform this validation, its access rights should be revoked.

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, as mentioned above, this is not an information leak.

Access rights to constraint information can recursively be constrained, which makes avoiding information leaks more difficult. For simplicity reasons, let us 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 constraint information in the access right before using the access right in a proof of access.

In client-based access control, access rights are 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 to store 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. Apart from 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 ser-

vice 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.

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 and where access control thus becomes even more complex. 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 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 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 containing the information for which the graph is built *root node*. Figure 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. Figure 3 shows an example of a graph with a conflict. There could

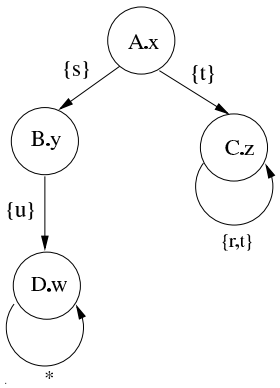

Figure 2. Access-rights graph. The graph is for information $A.x$ in terms of an entity's access rights. In particular, the entity has access rights to $A.x$ constrained to $B.y = s$ and $C.z = t$, to $B.y$ constrained to $D.w = u$, to $C.z$ constrained to $C.z \in \{r, t\}$, and to $D.w$ in an unconstrained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

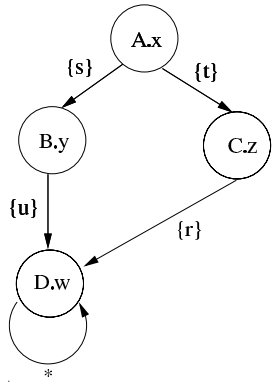

Figure 3. Access rights graphs with conflict. The graph has conflicting constraints on $D.w$.

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 the extended version [10, Chapter 5]. Ensuring that each constraint is satisfied requires traversal of the graph and contacting remote services that offer the information in a node. We call this graph

traversal *resolution*.

Access-rights graphs can become arbitrarily complex. At present, we expect them to be rather simple for practical scenarios, such as the graph in Figure 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 rather limited. However, both observations probably will no longer hold in the future. For example, 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 is steadily 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 Figure 4. 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 Figure 2, 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 $C.x = t$. 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 $A.x$.

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 multiple services offered this information, a client could contact some of these services and ask them to resolve the constraints on its behalf. This option requires trust relationships between the services so that they can exchange constraint information. None of this constraint 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 could 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 could 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 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 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 such that the constraint in the access right remains hidden from the location service. Therefore, the service cannot learn the specification from the proof of access and will not be able to learn Bob's location.

```
// 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;
}
```

Figure 4. 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.

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 an implementation of this concept based on digital

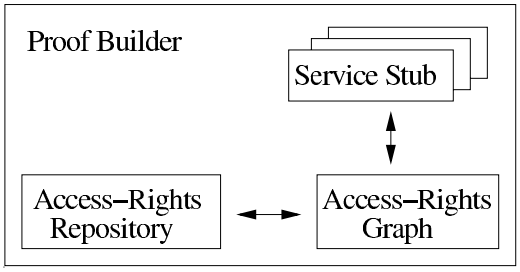

Figure 5. Proof-building architecture. The access-rights–graph component interacts with the access-rights repository for building a graph and with service stubs for resolution of the graph.

certificates 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 Figure 4.

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.

6. Architecture

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

6.1. Architecture

Figure 5 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 Figure 4. 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 [7] for expressing access rights and assurances. We give some example statements in the extended version of this paper [10, Chapter 5].

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 in this paper and another one based on one-way chains in the extended version [10, Chapter 5].

The issuer includes a public key, H, in an 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 an 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.

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. The 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

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

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.

A constraint service needs to perform an asymmetric decryption operation, which can be expensive. However, it is possible for the service to cache decrypted signing keys. In this way, when the 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 [11]. We deploy it in the Aura pervasive computing environment [9]. SSL [17] 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 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.

The results for the three experiments are in Table 1. Overall, the cost caused by access control and issuing assurances is small. For the second experiment, issuing an assurance becomes more expensive since the constraint service needs to decrypt the ciphertext. However, this additional cost gives us more flexibility when running access control. We can reduce this cost by caching decrypted ciphertexts, as shown in the third experiment.

8. Related Work

Multiple pervasive computing environments support context-sensitive access control to confidential information [1, 4, 8, 13]. Al-Muhtadi et al. [1], Chen et al. [4], and Gandon and Sadeh [8] 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 [13] 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. [5, 6], Neumann and Strembeck [15], and Bacon et al. [2] add context awareness to role-based access control. The first two approaches make the assignment of a permission to a role conditional on the current context; the third one conditions role activations on the current context. 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} [16] or GAA API [15]. Both models support context-sensitive constraints, but there is no discussion of how information leaks caused by context-sensitive constraints are avoided.

McDaniel [12] 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.

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

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)

Table 1. Client-response time. Mean and standard deviation of elapsed time for security operations (in bold) and for other, expensive operations using either non-hidden or hidden constraints (100 runs each) [ms].

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 in order to investigate what kind of access rights and constraints on them users define.

References

[1] J. Al-Muhtadi, A. Ranganathan, R. Campbell, and 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)*, pages 489–496, March 2003.

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

[3] L. Bauer, M. A. Schneider, and E. W. Felten. A General and Flexible Access-Control System for the Web. In *Proceedings of 11th Usenix Security Symposium*, pages 93–108, August 2002.

[4] H. Chen, T. Finin, and A. Joshi. Semantic Web in the Context Broker Architecture. In *Proceedings of 2nd IEEE International Conference on Pervasive Computing and Communications (PerCom 2004)*, pages 277–286, March 2004.

[5] M. J. Covington, P. Fogla, Z. Zhan, and M. Ahamad. A Context-Aware Security Architecture for Emerging Applications. In *Proceedings of 18th Annual Computer Security Applications Conference (ACSAC 2002)*, December 2002.

[6] M. J. Covington, W. Long, S. Srinivasan, A. Dey, M. Ahamad, and G. Abowd. Securing Context-Aware Applications Using Environment Roles. In *Proceedings of 6th ACM Symposium on Access Control Models and Technologies (SACMAT '01)*, pages 10–20, May 2001.

[7] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. SPKI Certificate Theory. RFC 2693, September 1999.

[8] F. Gandon and N. Sadeh. A Semantic eWallet to Reconcile Privacy and Context Awareness. In *Proceedings of 2nd International Semantic Web Conference (ISWC2003)*, October 2003.

[9] D. Garlan, D. Siewiorek, A. Smailagic, and P. Steenkiste. Project Aura: Towards Distraction-Free Pervasive Computing. *IEEE Pervasive Computing*, 1(2):22–31, April-June 2002.

[10] U. Hengartner. *Access Control to Information in Pervasive Computing Environments*. PhD thesis, Computer Science Department, Carnegie Mellon University, August 2005. Available as Technical Report CMU-CS-05-160.

[11] J. Howell and D. Kotz. End-to-end authorization. In *Proceedings of 4th Symposium on Operating System Design & Implementation (OSDI 2000)*, pages 151–164, October 2000.

[12] P. McDaniel. On Context in Authorization Policy. In *Proceedings of 8th ACM Symposium on Access Control Models and Technologies (SACMAT 2003)*, pages 80–89, June 2003.

[13] K. Minami and D. Kotz. Secure Context-sensitive Authorization. *Journal of Pervasive and Mobile Computing (PMC)*, 1(1), March 2005.

[14] G. Myles, A. Friday, and N. Davies. Preserving Privacy in Environments with Location-Based Applications. *Pervasive Computing*, 2(1):56–64, January-March 2003.

[15] G. Neumann and 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)*, pages 65–79, June 2003.

[16] J. Park and R. Sandhu. The UCON_{ABC} Usage Control Model. *ACM Transactions on Information and System Security (TISSEC)*, 7(1):128–174, February 2004.

[17] Claymore Systems. PureTLS. `http://www.rtfm.com/puretls/`.