Controlling access to pervasive information in the “Solar” system

Kazuhiro Minami and David Kotz

Dept. of Computer Science, Dartmouth College
Hanover, NH, USA 03755
{minami, dfk}@cs.dartmouth.edu
http://www.cs.dartmouth.edu/~solar/

Dartmouth Computer Science Technical Report TR2002-422
February 28, 2002

Abstract. Pervasive-computing infrastructures necessarily collect a lot of context information to disseminate to their context-aware applications. Due to the personal or proprietary nature of much of this context information, however, the infrastructure must limit access to context information to authorized persons. In this paper we propose a new access-control mechanism for event-based context-distribution infrastructures. The core of our approach is based on a conservative information-flow model of access control, but users may express discretionary relaxation of the resulting access-control list (ACL) by specifying *relaxation functions*. This combination of automatic ACL derivation and user-specified ACL relaxation allows access control to be determined and enforced in a decentralized, distributed system with no central administrator or central policy maker. It also allows users to express their personal balance between functionality and privacy. Finally, our infrastructure allows access-control policies to depend on context-sensitive roles, allowing great flexibility.

We describe our approach in terms of a specific context-dissemination framework, the Solar system, although the same principles would apply to systems with similar properties.

1 Introduction

Many pervasive-computing applications automatically adapt to the changing conditions in which they execute. These *context-aware* applications take account of information about the context, such as the location of the user and relevant devices, the presence of other people, light or sound conditions, or available network bandwidth. While the necessary sensors are increasingly available, particularly for location [9], we note that most *sensor data* must be processed into higher-level *context information* before use by applications. It is unreasonable to expect every application to work with the raw sensor data, and it is unscalable to expect a single “context server” to support the diversity of transformations and the scale of numerous applications and users [2].

We gratefully acknowledge the support of the Cisco Systems University Research Program, IBM, Microsoft Research, DoD MURI contract F49620-97-1-03821, and the Dartmouth Center for Mobile Computing.

We therefore need an infrastructure to aggregate and transform low-level sensor data for context-aware applications, while remaining flexible and scalable [5]. Many such systems use an event-flow model, in which sensor data are represented as events, and a graph of operators transform these event streams into the event streams desired by the applications. Our Solar system, described in a companion paper [4], is one example. Gryphon [1] is another.

Any such infrastructure must allow many applications and many users to share sensor data and context information. On the other hand, the context often includes information considered "private" by many users, such as their location, or "proprietary" by organizations, such as the marks written on a shared whiteboard or the calendar of meetings for product teams. A context-information infrastructure must, therefore, control access to the information it disseminates. Although researchers often note the importance of privacy and security in context-aware computing [7,13,14,10], there is surprisingly little literature about access control in context-information systems.

In this paper we propose an access-control mechanism for protecting context information in an infrastructure for collecting, processing, and disseminating context information. The mechanism must support a variety of, and large number of, sources, policies, and applications. In an infrastructure like Solar, context information is derived from numerous sources through a wide variety of operations. Applications and users can dynamically request new derivative information, and supply new operations. In such a dynamic environment it is unreasonable to expect a system administrator, or the users, to manually specify an access-control list for each event or event stream.

In our approach, each event is tagged with an access control list (ACL). Each operator in the event-flow graph automatically tags the events it produces based on the ACLs of its input events, on an optional restriction rule provided by the operator programmer, and on optional relaxation rules provided by users. In this way, operators that derive information from sensor data also derive the necessary ACL.

Our approach is discretionary, in that users may use their discretion to explicitly relax ACLs to grant access to other users. Where there is no explicit relaxation, however, we derive a conservative default access-control policy using principles borrowed from information-flow theory [6]. By convention, sources publish information with narrow ACLs, so by default personal information remains private until users explicitly release it in an explicit and structured fashion.

Our system also allows ACLs to include *context-sensitive roles*; for example, a thermostat application may allow access to anyone currently in the room, by specifying a role whose membership changes as people come and go. Thus, the access-control policies are themselves context-aware.

In the next section, we briefly describe the event-distribution infrastructure in the Solar system. Section 3 describes our design objectives, and Section 4 explains the semantics of our access-control mechanism. In Section 5, we explain some key implementation issues in our system. We discuss related work in Section 6 and future work in Section 7, and then summarize in Section 8.

2 Solar

Our access-control mechanism is designed for the Solar system, which is under development at Dartmouth College, and for systems like Solar. In this section we summarize the features of Solar needed to understand this paper. For more information on Solar see the companion paper [4] or our earlier paper [3].

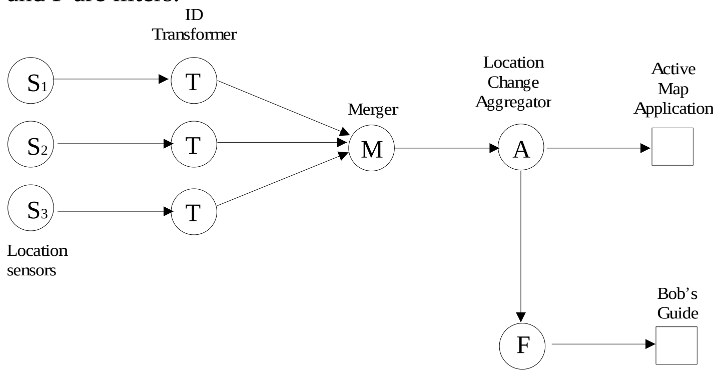
Solar is middleware that supports the collection, processing, and dissemination of context information for context-aware applications. In Solar, sensor data and context information are represented as *events* flowing from *sources* through *operators* (which filter, transform, or aggregate events into new events) to applications. Solar uses a publish/subscribe model of event flow. Sources and operators each publish a single stream of events; operators and applications may subscribe to many streams of events. Unlike some systems, each Solar publisher produces one event stream and each event stream has only one publisher. An operator subscribing to multiple event streams receives one event at a time, as if the streams were (arbitrarily) interleaved: events from a given publisher are delivered in the order they were published, but no ordering is defined between events from different publishers.

Operator graph. An operator is an object that subscribes to and processes one or more input event streams, and publishes another event stream. Since the inputs and output of an operator are all event streams, the operators can be connected recursively. When an application wants a flow of context information, it asks Solar to instantiate a *subscription tree* that describes the flow of events from a set of sources (leaves of the tree) through a set of operators to the application (the root of the tree). Solar re-uses existing sub-trees where possible, so that applications and users may (transparently) share operators and their event streams. The overlapping subscription trees form a directed acyclic graph we call the *operator graph*.

Example. Consider the example in Figure 1, which might be used in an office building with a location-tracking system. Each room has a location sensor that periodically reports the identification number for the badge(s) it detects in that room. *Transform* operators map the badge ID to the name of the person or device attached to that badge, and the *merge* operator combines these streams into one event stream. An *aggregator* operator uses internal state to detect location changes, emitting an event only when a person or device changes location. The Active Map application uses the resulting stream to display the current location of tracked objects. Another application is a "tour guide" for Bob, which uses a *filter* to obtain only events about Bob's movements. Note that both applications can share many of the operators and streams.

This example demonstrates the four varieties of Solar operators: Filter, Transform, and Merge operators are stateless; Aggregation operators may have internal state. Ultimately, though, operators are simply objects implementing an Operator interface. Solar allows applications to use operators instantiated from classes found in a library of common operators or from classes provided by the application programmer. As a result, the functionality of an operator is opaque to Solar. Solar asks the operator to handle a new event, and the result is that the operator publishes zero or more events.

Fig. 1. A sample operator graph. T are transformers, M are mergers, A are aggregators, and F are filters.



Architecture. The Solar system is distributed across many hosts, although the only abstraction presented to operators and applications is the operator graph [3,4]. Applications run outside the Solar system, on any platform, using a small Solar library that allows them request subscriptions and to receive events over standard network protocols.

Summary. Although we describe our access-control mechanism in terms of the Solar infrastructure, most of its features are not dependent on the specifics of Solar. Ultimately, our mechanism suits any event-based context-dissemination infrastructure in which events are produced by sources, are processed by a shared collection of operators that subscribe, process, and publish derived events, and are consumed by applications. In particular our approach does not depend on Solar's graph structure or model of event streams.

3 Design goals

We have four objectives in our design of system for controlling access to context information in Solar-like systems.

Our primary objective is to limit application access to events, because events often contain information considered private by certain individuals or proprietary to certain organizations. We assume that applications run on behalf of a specific principal,[1] so this objective means that applications must only receive events to which their principal is allowed access.

In accomplishing this first objective, consider several properties of the operator graph. First, each node in the operator graph may be shared by many applications and users. Second, there are a variety of information sources and of applications, so there many operators are needed to derive the desired event streams from the sources. It is

[1] We imagine a special principal *anonymous* that could represent applications running on a public display, kiosk, or other unauthenticated device.

infeasible to manually specify an access-control list for each event stream. Third, the operator graph changes as applications and sensors come and go; we cannot know the topology of the operator graph in advance. Fourth, for scalable performance the operator graph is likely to be deployed across many servers.

The second objective is avoid the use of any central authority that defines the access-control policies. By generating access-control lists automatically using principles from information flow, then giving users discretion to relax ACLs where they deem appropriate, we empower users and reduce management overhead. We accommodate personal differences in privacy policies, and encourage users to contribute new sources, operators, and applications, and to define their access-control policies, without requiring the blessing of any central authority.

The third objective is to allow users to specify some *context-sensitive* access-control policies. For example, Bob may not release his current location to anyone, but lets others receive events about his location if they are currently in the same room. Since presumably they can see Bob, physically, these events do not significantly reduce Bob's privacy, but may allow the other user to implement many context-aware applications mentioned in the literature (e.g., an application that reminds you to "mention the party the next time I see Bob.").

The fourth objective (specific to Solar) is to maintain the transparency of operator sharing. Solar may overlap subscription trees to produce a graph with shared operators, transparently to the operators and applications.

Finally, there are certain features that are explicitly *not* an objective of our current research.

First, we do not want to enforce a strict information-flow policy, because it is insufficiently flexible. It is also impractical: we cannot assume that all computations will occur inside a trusted environment that maintains information-flow principles. Once an application (outside Solar) has an event, we cannot stop the application from giving away the information. We prefer to directly support limited forms of ACL relaxation, to allow most reasonable information sharing to occur explicitly within the framework.

Second, we do not address covert channels.

Third, we do not protect against any data-mining effort to infer sensitive information from information legitimately obtained.

Finally, this paper is about access control, not about authentication or trust. That said, we do assume that the Solar system authenticates application users sufficiently to attach a principal to each running application. We also assume that communications with Solar, and among the hosts constituting the Solar infrastructure, use encrypted channels. Thus our focus is on limiting the set of events receivable by legitimate principals, rather than on protection against eavesdroppers. Finally, we assume that applications, users, and operators trust the Solar infrastructure, although Solar does not trust applications or operators.

4 Access-control semantics

In this section, we describe our approach to access control, which is based on principals and access-control lists. Each user is represented by a named *principal*. A principal may be in one or more named *roles*. A role contains a list of principals and other roles. An *access-control list (ACL)* is a list of principal and role names. In our notation, p is a principal, r is a role, and U is the universal set of all principals and roles.

We must first decide whether to attach an ACL to each *event* or to each *event stream*. Note that some operators, such as most of those in Figure 1, may publish events that reasonably should have different ACLs. In that example, an event about Bob's location should have ACL $\{\text{Bob}\}$, while an event about Alice's location should have ACL $\{\text{Alice}\}$. We thus choose to attach a separate ACL to each event. Every event e_i has two parts:

$$e_i = (d_i, a_i), \text{where } d_i \text{ is the data field and } a_i \text{ is the ACL field.}$$

An application executes on behalf of one principal. An application may subscribe to any event stream, but may receive an event if, and only if, the application's principal is a member of the event's ACL or a holder of a role mentioned in the ACL. Specifically, for principal p and an event e_i with ACL a_i, we allow access iff $p \in^* a_i$, defined as follows:

$$p \in^* a_i \Leftrightarrow [(p \in a_i) \text{ or } (r \in a_i \text{ and } p \in^* r)].$$

Notice that this definition is recursive, when roles are defined hierarchically. (An implementation must take care, in recursion, to avoid any cycles in role definitions.)

4.1 ACL derivation

An operator may receive events from many different publishers, and produce many different events. It is often inconvenient or impossible for the operator programmer to determine the appropriate ACL for each output event (consider a generic operator that filters room-temperature events). And, since an operator may be shared by many applications and users, it is not possible for any one of them to define the ACL of each published event. So, we wish to derive the ACL for each output event as a function of the ACL of incoming events, the desires of the operator programmer, and the desires of any interested user. We call our approach ACL propagation, because access-control information propagates through the operator graph.

In the discussion that follows, we focus on a single operator. Over its lifetime it has received a series of events $e_1, e_2, ..., e_i$. Events from a given subscription arrive in the order they were published, but events from multiple subscriptions are arbitrarily interleaved into the sequence. The operator is allowed to execute its handler once for each event e_i. The handler's result is to publish zero or more new events $\{e_{ij}\}$, although the handler only produces the data field:

$$\{d_{ij}\} = \text{handleEvent}(d_i).$$

We now show how to derive the ACLs $\{a_{ij}\}$ from a_i in three stages. First, Solar computes a default ACL using a conservative information-flow approach. Second, the operator programmer may further restrict the ACL. Third, any principal in that ACL may relax the ACL.

Default ACL. After receiving the set $\{d_{ij}\}$, Solar computes a default ACL DEF_{ij} for each event. The computation depends on whether the operator's handler read or wrote any internal state in computing the event.

For stateless operators, there is no concern about whether sensitive information in earlier events may "leak" into this event. The default output ACL is the same as the input ACL.

$$DEF_{ij} = a_i, \text{ if no state read}$$

For operators with internal state, Solar computes a default ACL using principles from information-flow control systems [6]. Since operators are opaque, we cannot apply information-flow principles inside the operator, and we conservatively derive each output event's ACL as the intersection of the ACLs of every event ever received by this operator. Thus, $DEF_{ij} = \cap a_k$ for $k = 0$ to i. This policy is too restrictive in some cases. Consider the Merge operator in Figure 1, which receives an event about Bob's location tagged $\{$Bob$\}$, then an event about Alice's location tagged $\{$Alice$\}$. Because of the intersection rule, every event published after the arrival of Alice's event has the empty ACL $\{\}$.

Thus, Solar forces operators to represent any internal state as a set of state objects, each associated with a simple key k (e.g., a string). The key's meaning is determined by the operator programmer; for example, the key may be a principal's name. Think of the set of states as a hash table with methods `s=get(k)` and `put(k,s)` and the following rules apply within the handling of a given event: 1) the operator may call `get` at most once and `put` at most once. 2) The state retrieved by `get` is unchanged unless followed by a `put`; that is, `get` retrieves a copy of the state. 3) If it calls both `get` and `put`, both calls must be for the same key. 4) A call to `put(k,s)` creates state s_k if it does not exist. These rules prevent information from "leaking" across states.

Solar maintains an accumulated ACL for each state. Informally, since the state may contain information gathered from all prior events, its accumulated ACL must be the least upper bound (intersection) of the ACLs on those prior events. After each input event e_i, we define the accumulated ACL for each state s to be

$$ACC_{is} = \begin{cases} ACC_{(i-1)s} & \text{if no } \texttt{put} \\ a_i & \text{if } \texttt{put} \text{ only, or } \texttt{put} \text{ followed by } \texttt{get} \\ ACC_{(i-1)s} \cap a_i & \text{if } \texttt{get} \text{ followed by } \texttt{put} \end{cases}$$

where $ACC_{0s} = U$ for all s.

Now we update our definition of DEF_{ij}

$$DEF_{ij} = \begin{cases} ACC_{is} \cap a_i & \text{if the handler read state } s \\ a_i & \text{otherwise} \end{cases}$$

An intuitive reading of this rule is that the output event inherits its ACL from the input event, unless its computation was "polluted" by state information that may contain information that should remain private.

Restriction rule. The operator programmer may wish to restrict the ACL of output events, perhaps because the operator's parameters may contain sensitive information that affects the output results, or because the operator contains an algorithm that computes valuable information from less-valuable input data. Therefore, the developer has the option to include a function Restrict(e_i, d_{ij}) that computes the list of principals to be removed from DEF_{ij}. The default Restrict(e_i, d_{ij}) returns U. We use this function to compute the "output ACL suggested by the operator:"

$$OP_{ij} = DEF_{ij} \cap \text{Restrict}(e_i, d_{ij})$$

Although we anticipate few situations where it would be helpful, we allow the Restrict() function to `get` any state, because it cannot change the data d_{ij} and it can only narrow the eventual ACL a_{ij}.

Relaxation rule. We allow users to attach their own *relaxation function* to any operator, to relax the ACL for events output by that operator. Each principal p may specify their own access-control policy by defining and attaching their own relaxation function, Relax$^p(e_i, d_{ij})$. We expect that in many cases the user may supply a constant rather than a function, that is, Relax$^p(e_i, d_{ij}) = RLX^p$, but the function allows more flexibility. If neither is supplied, the default Relax$^p(e_i, d_{ij}) = \{\}$.

Solar computes only those relaxation functions contributed by principals who already have access to the event, according to OP_{ij}. The rationale is that any of those principals may subscribe directly to this operator, receive the events, and pass them to their friends anyway. The relaxation function allows a more structured solution.

We compute the set of principals added by all eligible principals,

$$USR_{ij} = \bigcup \{\text{Relax}^p(e_i, d_{ij}) \,|\, p \in^* OP_{ij}\}.$$

Note that we use $\in^*$, but it is computationally expensive in the presence of context-sensitive roles (see Section 5.3), so we expect many implementations to use $\in$.

Finally, we can define the ACL of the output event:

$$a_{ij} = OP_{ij} \cup USR_{ij}$$

Sources. Although we discuss operators above, these rules apply to sources as well. The difference is that sources have no input events. All of the above equations hold, then, if we apply them for $i = 0$ and $a_0 = U$, as the source publishes a sequence of events e_{0j}. By definition, $ACC_{0s} = U$ for all s, so $DEF_{0j} = U$. So

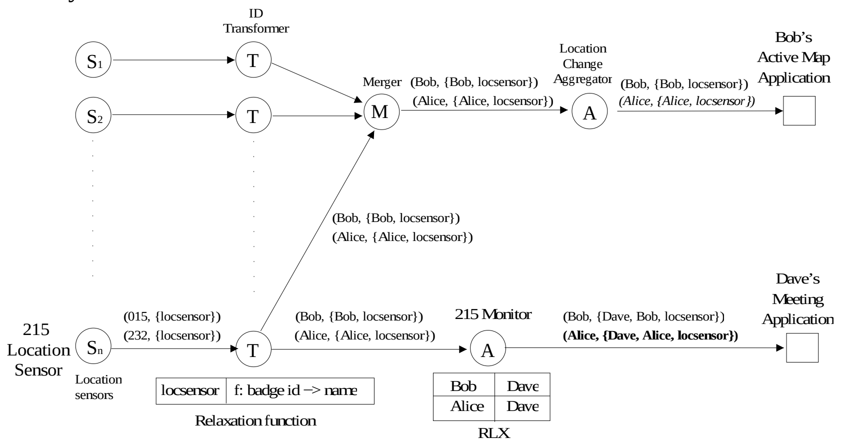
$$OP_{0j} = \text{Restrict}(e_0, d_{0j})$$

(though e_0 is null), and a_{0j} is computed as above from OP and USR. Thus, the programmer of the source can use Restrict() to define any output ACL, and that ACL may be relaxed by users according to the same rules as for operators.

4.2 Example

In Figure 2, the location sensors use Restrict() to define the ACL on their raw sensor data to be {locsensor}, a principal representing the administrator of the location-sensing system. On the "ID Transformer" operator, which translates badge IDs into person names, the locsensor principal adds a relaxation function that adds principal p to any location event about person p (assuming a clear mapping between people and principals). This approach is conservative: the raw sensor data remains private to the administrator, and the named sensor data is private to the individual.

Fig. 2. An example of ACL propagation. The arrows are labeled with the event represented by a tuple in which the first item denotes the principal name in the data field, and the second item denotes the ACL field. The data field also contains the location, but for brevity we omit that information here.



Notice that events pass through the stateless Merge operator with unchanged ACLs. The Location-Change Aggregator takes care to use a separate state object s_p for each principal p, so that $ACC_{is_p} = \{p\}$ and those location events can pass through with unchanged ACLs. Although the aggregator publishes events about Alice, tagged with ACL $\{$Alice$\}$, Solar does not deliver those events to the Active Map application run by Bob (see Section 5.2).

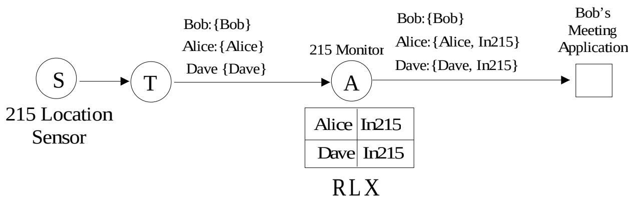
It is left to individuals to decide when to relax the ACL on events about them. Assume that Alice does not want to be visible to most applications, but she is willing to allow Dave's Meeting application (run by Dave) to see that she is in the meeting room 215. She adds Dave to her relaxation list on the aggregator "215 monitor," which outputs arrival and departure events about people in room 215.

4.3 Context-Sensitive Roles

There are many instances where the policy itself should be context-sensitive, that is, where the access list produced by a relaxation function depends on the current context beyond the information available in the event. Furthermore, it is difficult for most users to write relaxation functions. For both reasons, we allow ACLs to contain the name of *roles*. In Role-Based Access Control (RBAC) [12], permissions are associated with roles, not directly with principals. Roles usually have semantics that are meaningful to ordinary users, such as “people in room 215” or “chairperson of the meeting”. If a role is defined elsewhere to be a context-sensitive set of principals, then any ACL including that role will itself be context-sensitive.

For example, in Figure 3, suppose Alice allows people in the meeting room (215) to receive her location information only when she is there, too. So, she adds a relaxation function (in this case, a constant) that outputs the role name “In215.” The role “In215” is defined elsewhere to contain the set of principals corresponding to people currently in room 215; if Bob and Dave are in room 215, their applications can receive the event about Alice (see Section 5.2).

Fig. 3. An example context-sensitive policy.



5 Implementation issues

In this section, we describe several important implementation issues.

5.1 Operator state

As we discuss in Section 4, it is necessary to insist that operators have no internal state other than that managed by Solar, through the `get(k)` and `put(k,s)` functions. The implementation, which in our case is in Java, must provide four properties. First, the operator can have no other state: each operator class defines `handleEvent()` and `Restrict()` as static methods, and is not allowed to use static fields, to access files, to open network connections, and the like.[2] Second, we ensure that the operator can read Solar-managed state only with `get`, by hiding that state in a Solar object that offers only `get` and `put` as public methods. Third, the state can be created or updated only by calling `put`, because `get` returns a copy of the state, rather than a reference to Solar’s copy of the state. Fourth, the `get/put` rules described in Section 4.1 are enforced by the implementations of `get` and `put`.

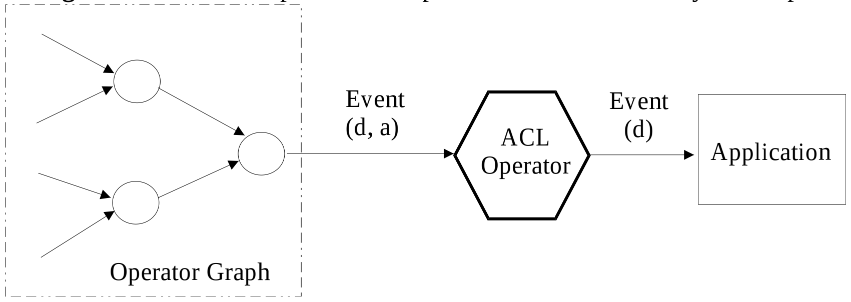
5.2 Access-control enforcement

Ultimately, the point of computing an event’s ACL is to limit the set of applications that may receive the event. For this purpose, Solar adds a special *ACL operator* at the root of

[2] We can enforce these properties through byte-code analysis.

every subscription tree, as shown in Figure 4. For an event $e_i = \{d_i, a_i\}$, this operator's special event-handling method `accessAllowed(`a_i`)` returns a boolean indicating whether to pass d_i on to the application. The ACL operator, constructed with parameter p where p is the principal running the application, returns true iff $p \in^* a_i$. To evaluate that expression, an ACL operator must recursively check p's membership in any roles named in a_i, as we describe below.

Fig. 4. Solar inserts a special ACL operator at the root of every subscription tree.



5.3 Evaluation of role names

Our approach allows *roles* to be named in ACLs, and for those roles to be defined in a context-sensitive way. Thus, we need efficient mechanisms a) to define context-sensitive roles, and b) to test role membership.

To define a new context-sensitive role, any user may ask Solar to deploy (and name) a subscription tree whose root operator publishes events listing the current set of member principals.[3] Anyone may keep up-to-date on the role membership, as it changes, simply by subscribing to the role's root operator, by name. (Solar's name space is beyond the scope of this paper; see [3].)

There are two instances in which Solar needs to evaluate role membership: when filtering events in an ACL operator and when computing relaxation functions in every publisher. Consider the ACL operator, which needs to test $p \in^* a_i$. It needs to evaluate, for a single principal p, membership in any role that is attached to an incoming event. The simplest approach is to query each role's operator about p's membership. Operators do not have a query interface, however, and this approach requires a round-trip message for each role for each event, so we prefer subscription-based approaches that allow the ACL operator to maintain a list of p's current roles. For example, for any principal p with active applications, Solar deploys a trusted aggregator that subscribes to all roles, publishing an event whenever p joins or leaves a role. ACL operators for p

[3] For efficiency, most implementations would allow three types of events: `set` *`list`*, which announces the full membership, `add` *`list`*, which announces additions to the membership, and `del` *`list`*, which announces deletions from the membership. Any subscribers would maintain the current list in their state.

subscribe to this operator. (This approach explains why we chose to implement access-control enforcement as an operator; its normal `handleEvent(`d`)` method processes the incoming role updates, while its special `accessAllowed(`a`)` method processes incoming data events.)

Consider an ordinary publisher, which needs to test $p \in^* OP_{ij}$ for all p that have registered relaxation functions. The simplest approach is to subscribe to the same per-principal aggregator mentioned above, for each principal that registers a relaxation functions. If there are many, it may be more efficient to monitor all the roles directly. To avoid the overhead of monitoring role membership, we expect that many implementations will narrow the test to $p \in OP_{ij}$.

6 Related work

There is little published about access control in event-distribution systems for pervasive computing. Spreitzer and Theimer [14] discuss privacy issues involved in location-aware applications, and developed a mechanism to encode a user's privacy policy in the "user agent" that collected and disseminated the user's personal context information. Their approach is limited to location context and does not generalize to derived context information. Ebling [7] also uses role-based access control to protect the privacy of users in context-aware applications, but provides no details. Our system introduces context-sensitive roles, whose membership is computed from context information, and the use of context-sensitive roles allows us specify flexible access-control rules for many applications.

Our ideas are substantially influenced by recent work on information-flow models of access control. Since the traditional information-flow policy is too restrictive, several projects provide ways to "declassify" information. Ferrari [8] is an object-oriented system that controls access to objects using information-flow principles, but provides trusted functions that can relax the strict information-flow policy. That is, a principal with no rights to access some object directly can access it through the trusted function. They, like us, use ACLs to define the security level in their flow policy. Although their trusted function does allow access to objects by principals not in the ACL, it is not clear how their mechanism could be used in our situation, in which we relax an event's ACL so it can be accessed, later, by applications downstream. Furthermore, in their paper it is not clear whether ordinary users would have the power to implement trusted functions or contribute to the relaxation lists $RW(m)$ and $IW(m)$ for the method m.

Myers et al. [11] extend the Java programming language to allow variables to be tagged with labels. Labels contain an access-control policy (essentially, an ACL) for each "owner" of that variable. Variables computed from others are given a label computed from the labels of operands in the computation, according to information-flow principles. An owner principal may declassify a variable by extending its own ACL in the label, but other owner's ACLs are unchanged. A principal may only obtain the value of the variable if it is in the *intersection* of the owners' ACLs. Our relaxation semantics are more liberal than their declassification semantics, as we allow anyone with current access to add any other principal to the ACL. Although their approach is intended for compile-time analysis of fine-grained information-flow in Java programs, it may be possible to apply the same concepts to events and operators, at run-time.

7 Future work

We plan to prototype our proposed access-control mechanism within the Solar system, and apply it to some realistic applications. Our goal is to to evaluate the efficiency of the mechanism, its flexibility in expressing realistic policies, and the amount of administrative overhead involved in management. We need to investigate alternatives for testing membership in context-sensitive roles.

Our approach attaches an ACL to each event, to control application access to event data. We wonder whether it may be useful to attach trust or integrity information to each event, so that subscribers can interpret the event's data in terms of their willingness to trust the source or intermediate operators.

This paper focuses on controlling access to context information. We believe there is potential to use context information to control access to a wide variety of services, whether computational (such as access to a database) or physical (such as access to a locked room). Although our framework allows context-sensitive access-control decisions through the use of context-sensitive roles, we plan to study a wider variety of uses for context-sensitive authorization under a more general model.

8 Summary

Security is a critical component in any realistic deployment of pervasive computing. A pervasive-computing infrastructure necessarily collects a lot of context information to disseminate to its context-aware applications, but due to the personal or proprietary nature of much of this context data, the infrastructure must limit access to context information to authorized persons. We propose a new access-control mechanism for event-based context-distribution infrastructures. Our approach is based on ACL propagation, in which each event e is tagged with an ACL derived automatically from the ACL on events that contributed to the production of e. Our approach is based on a conservative information-flow model of access control, but we allow users to express their access-control policies through relaxation functions. This combination of automatic ACL derivation and user-specified ACL relaxation allows access control to be determined and enforced in a decentralized, distributed system with no central administrator or central policy maker. It also allows users to express their personal balance between functionality and privacy. Finally, our infrastructure allows access-control policies to depend on context-sensitive roles, allowing great flexibility.

We met our four objectives: 1) applications may only receive events to which their principal has access, 2) access-control policies are determined by information-flow principles and relaxed where appropriate by request of users, 3) the access-control policies can themselves be context-aware, through the use of context-sensitive role definitions, and 4) operators and their event streams may be shared by multiple users and applications, while this sharing remains transparent to the users and applications. The result is an approach that is secure, flexible, distributed, and (we believe) scalable.

Finally, although this paper describes our work in the context of the Solar system, we believe that our ACL-propagation technique would also apply to many other event-based context-dissemination infrastructures.

Acknowledgements

We are grateful to Guanling Chen for his design and development of the Solar system, to Sean Smith and Chris Hawblitzel for many helpful discussions about security, and to Chris Masone and Chris Hawblitzel for their feedback on drafts of this paper.

References

1. Guruduth Banavar, Marc Kaplan, Kelly Shaw, Robert E. Strom, Daniel C. Sturman, and Wei Tao. Information flow based event distribution middleware. In *Proceedings of the Middleware Workshop at the 19th IEEE International Conference on Distributed Computing Systems*, pages 114–121, Austin, Texas, May 1999. IEEE Computer Society Press.
2. Guanling Chen and David Kotz. Context aggregation and dissemination in ubiquitous computing systems. Technical Report TR2002-420, Dept. of Computer Science, Dartmouth College, December 2001. Submitted to *WMCSA 2002*.
3. Guanling Chen and David Kotz. Supporting adaptive ubiquitous applications with the SOLAR system. Technical Report TR2001-397, Dept. of Computer Science, Dartmouth College, May 2001.
4. Guanling Chen and David Kotz. Solar: A pervasive-computing infrastructure for context-aware mobile applications. Technical Report TR2002-421, Dept. of Computer Science, Dartmouth College, February 2002. Submitted to *Pervasive 2002*.
5. Norman H. Cohen, Apratim Purakayastha, John Turek, Luke Wong, and Danny Yeh. Challenges in flexible aggregation of pervasive data. Technical Report RC21942, IBM Research Division, Thomas J. Watson Research Center, P.O.Box 704, Yorktown Heights, NY 10598, January 2001.
6. Dorothy Elizabeth Robling Denning. *Cryptography and Data Security*. Addison-Wesley Publishing Company, 1982.
7. Maria R. Ebling, Guerney D. H. Hunt, and Hui Lei. Issues for context services for pervasive computing. In *Proceedings of the Workshop on Middleware for Mobile Computing 2001*, Heidelberg, Germany, November 2001.
8. Elena Ferrari, Pierangela Samarati, Elisa Bertino, and Sushil Jajodia. Providing flexibility in information flow control for object-oriented systems. In *Proc. IEEE Symposium on Security and Privacy*, pages 130–140, May 1997.
9. Jeffrey Hightower and Gaetano Borriello. Location systems for ubiquitous computing. *IEEE Computer*, 34(8):57–66, August 2001.
10. Marc Langheinrich. Privacy by design— principles of privacy-aware ubiquitous systems. In *Proceedings of UbiComp 2001: International Conference on Ubiquitous Computing*, volume 2201 of *Lecture Notes in Computer Science*, pages 273–291. Springer-Verlag, 2001.
11. Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, 2000.
12. Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, Feb 1996.
13. M. Satyanarayanan. Pervasive computing: Vision and challenges. *IEEE Personal Communications*, 8(4):10–17, August 2001.
14. Mike Spreitzer and Marvin Theimer. Providing location information in a ubiquitous computing environment. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles(SOSP'93)*, pages 270–283, Ashville, NC, 1993. ACM Press.