Know Why Your Access Was Denied: Regulating Feedback for Usable Security*

Apu Kapadia† Geetanjali Sampemane Roy H. Campbell

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801

{akapadia,geta,rhc}@cs.uiuc.edu

ABSTRACT

We examine the problem of providing useful feedback about access control decisions to users while controlling the disclosure of the system's security policies. Relevant feedback enhances system usability, especially in systems where permissions change in unpredictable ways depending on contextual information. However, providing feedback indiscriminately can violate the confidentiality of system policy. To achieve a balance between system usability and the protection of security policies, we present *Know*, a framework that uses Ordered Binary Decision Diagrams (OBDDs) and cost functions to provide feedback to users about access control decisions. *Know* honors the policy protection requirements, which are represented as a meta-policy, and generates permissible and relevant feedback to users on how to obtain access to a resource. To the best of our knowledge, our work is the first to address the need of useful access control feedback while honoring the privacy and confidentiality requirements of a system's security policy.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Access Controls*; H.5.m [**Information Interfaces and Presentation**]: User Interfaces

General Terms

Security, Human factors

*This research is supported in part by the National Science Foundation NSF CCR 00-86094 and CISE EIA 99-72884 EQ

†Apu Kapadia is funded by the U.S. Dept. of Energy's High-Performance Computer Science Fellowship through Los Alamos National Laboratory, Lawrence Livermore National Laboratory, and Sandia National Laboratory.

CCS'04, October 25-29, 2004, Washington, DC, USA.

Keywords

Security, access control, policy protection, privacy, feedback, usability

1. INTRODUCTION

When a user is denied access to a resource, what level of feedback should the system provide? At one extreme, the system can conceal all policy information and respond with a simple "Access denied." Many security systems use this policy, on the grounds that an unauthorized user must not be given any further information. However, such a system is less user-friendly to legitimate users, because they are not given enough feedback to reason about, and correct, errors. This problem is exacerbated when access also depends on contextual information, and permissions change based on factors like room activities and time-of-day—without feedback, a user has no way of discerning why some attempts succeed and others fail. At the other extreme, the system can be completely open about its policies and make them public knowledge. Users can then reason about their access to resources, making such a system more usable. While feedback to legitimate users is often a desirable feature, unrestricted feedback can be harmful, for example, by assisting intruders probing system security in finding out where to direct their attacks. Another problem with open policies is information leakage—policies contain enough information to allow legitimate users to deduce what other system users can access, thus violating the privacy of those other users.

As systems get more complex, access policies get more complicated too, and it is unreasonable to expect users to memorize all the conditions that affect access. It thus becomes important to provide useful feedback to legitimate users, while also addressing privacy and security concerns of the system's policies (i.e., providing suitable *policy protection*). Any such system makes a trade-off between usability and policy protection. While this issue has not been adequately researched, common operating systems do provide primitive forms of policy protection. For example, UNIX allows users to look at a file's permissions if they have (execute) access to that directory. However, for two files in the same directory, there is no way to hide the policy for one file and not the other. Users generally protect access to files or hide their existence (by denying a directory listing). While this coarse-grained policy protection has been adequate in

conventional systems, we believe that new ubiquitous computing [18] environments accentuate the problem of policy protection and feedback.

Users in ubiquitous computing environments typically interact with a plethora of computing, communication or I/O devices in their vicinity in many ways—voice, gestures, and traditional keyboard-and-mouse input being some of them. Different sets of users are allowed access to different subsets of resources, and these permissions may change depending on contextual information such as the time of day, the current activity, or the set of people involved. In such an environment, it may not be clear to a user why he or she was denied access to certain resources. Thus, informative feedback about why access was denied becomes very important if the system is to avoid annoying users with apparently-arbitrary restrictions. However, as mentioned earlier, unrestricted feedback about who is allowed to do what in the system could itself compromise system security and privacy; therefore, policies need to be protected against inadvertent disclosure. As a first step in this direction, we present a feedback model called *Know*, which uses *meta-policies* for policy protection and *cost functions* to compute useful feedback.

Know examines the meta-policy that protects a policy and determines the level of feedback that can be provided to a particular user. For example, a student trying to access an audio device in a conference room may be told to return after the ongoing meeting has finished. However, a meeting participant may be informed that the meeting chair has access to the audio device. The basic challenge in providing informative feedback is to identify the conditions required for the requested operation to be allowed, i.e., find a way to "satisfy" the access control rule guarding that operation. This may involve the user activating a different role (or presenting a different credential) or waiting for the context to change (e.g., the current activity configuration of the space). Searching all the rules that guard a particular action will determine all the situations in which this operation is permitted. *Know* can use this information to suggest viable alternatives. A cost function is used to represent the relative difficulty of changing an attribute to satisfy an access condition—it may be easier for a Student to wait for the end of a meeting to be allowed access to a printer, rather than to become a Room Administrator to print the document immediately.

Satisfiability is an NP-complete problem in general. However, in our experience with smart spaces, policies for individual devices involve a small number of variables, resulting in tractable state space exploration. For this purpose, we use ordered binary decision diagrams (OBDDs) [14] as an efficient and compact representation of policies (or rules), and search for conditions that satisfy the access rules. OBDDs are graph structures, which makes it easy to apply cost functions and shortest path algorithms for providing useful feedback. While certain degenerate cases can result in exponentially large OBDDs, there are several heuristics to reduce the size of OBDDs in general. To improve performance, the OBDDs are computed in advance by *Know*. *Know* provides feedback to the user only if it can do so within reasonable bounds of time and space.

This work is an initial attempt to address the problem of providing useful feedback about security decisions in ubiquitous computing systems while honoring protected policies. Our main argument is that system feedback is more useful if it is tailored to the intended recipient and based on his or her current permissions rather than a generic "Access Denied" message, and that it is possible to provide such feedback while still protecting policy information appropriately.

The rest of the paper is organized as follows—Section 2 discusses some necessary background and Section 3 describes the system architecture including the use of OBDDs and cost functions to efficiently generate feedback. In Section 4, we describe our implementation of *Know* and show how it presents useful feedback for a realistic policy. We then discuss some related work and issues raised by our approach in Section 5 before concluding in Section 6.

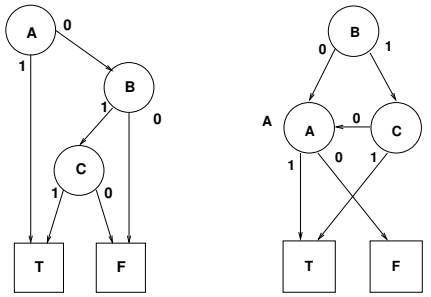
2. BACKGROUND

While our method of providing feedback is generally applicable to security systems, we focus on its use in ubiquitous computing systems. Good security feedback is particularly important for these systems for a variety of reasons:

- Ubiquitous computing is an area of active research [15, 9]; new systems and modes of applications are being developed. While these systems are still being used in experimental ways by researchers, application developers and early adopters, good security feedback will help direct secure application design.
- Ubiquitous computing environments, currently most prevalent in academic and research environments, are envisioned to percolate into everyday use, with a majority of non-technical users. In both these situations, feedback is important for usability, since users either disable or work around security mechanisms that seem incomprehensibly obstructive.
- Ubiquitous computing systems can be more confusing to users than traditional distributed systems due to the inherent dynamism (mobile users and devices may enter and leave the system), the large number of devices and the context-sensitive environment—without adequate feedback, it can be difficult to tell whether access was denied due to a bug in the system (especially in experimental systems) or due to user permissions changing in response to non-obvious changes in the context.

We believe that these features make ubiquitous computing systems an ideal test-bed for *Know*.

Test-bed: The Active Spaces project at the University of Illinois takes an operating system approach to ubiquitous computing environments. A "space OS" called Gaia [15] interacts with all the devices in the space and provides a uniform programming interface to application developers. The Gaia OS provides infrastructure services such as naming and context that applications can use, as well as security services for authentication and access control. The Gaia access control system [17] uses an extension of the role-based access control system to assign roles to users within an Active Space. The "space role" assigned to a user decides the permissions available to the user at that time. The user's space role may change depending on contextual information such as the time of day, the current activity being undertaken in the space or even the set of people currently in the space. Thus, permissions available to a user at any point in time depend on a variety of factors, and good feedback about access control decisions is very important.



(a) Variable ordering: a, b, c (b) Variable ordering b, c, a

Figure 1: Example OBDDs for $a \vee (b \wedge c)$

Policy Protection: UniPro [21] provides a scheme to model protection of resources, including policies, in trust negotiation. It allows policies to be treated as resources in the system, and allows the specification of policies to protect them. *Know* uses the UniPro notation for writing meta-policies that protect the policies to decide what feedback can be provided to a user.

Representation: Ordered binary decision diagrams (OBDDs) [14] are a canonical-form representation for boolean formulas where two restrictions are placed on binary decision diagrams: the variables should appear in the same order on every path from the root to a terminal, and there should be no isomorphic subtrees or redundant vertices in the diagram. A binary decision diagram is a rooted, directed acyclic graph with two types of vertices: terminal and nonterminal. Each nonterminal vertex v is labeled by a variable $var(v)$ and has two successors, $low(v)$ and $high(v)$. We call the edge connecting v to $low(v)$ the 0-edge of v (since it is the edge taken if $v = 0$) and the edge connecting v to $high(v)$ the 1-edge of v. A single formula may be represented by multiple different OBDDs based on the order that variables in the formula are tested; however, given a particular variable-ordering, the OBDD structure is fixed (canonical form for that variable-ordering). Figure 1 gives an example of two OBDDs that each represent the simple boolean formula $a \vee (b \wedge c)$. The first is the canonical-form OBDD for the variable-ordering a, b, c and the second is the canonical-form OBDD for the variable-ordering b, c, a. To test for satisfiability, we start at the root node and test whether the variable at the root is *true* or *false*. If it is false, we follow the 0-edge, and if true, the 1-edge, and repeat this process. Eventually we reach either the T-node or the F-node (also called the 1-node and 0-node, respectively). If we reach the T-node, then the given assignment satisfies the formula; if we reach the F-node, it does not. For example, applying the assignment $\langle a = false, b = true, c = false \rangle$ to either of the OBDDs in Figure 1 tells us that the formula is not satisfied. We use OBDDs because they are a compact and graphical representation of boolean formulas. This allows us to use cost functions and shortest path algorithms to find conditions of satisfiability that are of "least cost" to the user.

Know stores access control rules as OBDDs, and can efficiently search these OBDDs for paths that satisfy the rules. When access is denied, the OBDD can provide information about alternate paths that would allow access. *Know* provides information to the user about such paths as feedback. Since satisfiability is an NP-complete problem in general, the number of nodes in an OBDD can be exponential in the size of the boolean expression. However, there are several heuristics to find an ordering that reduces the size of the OBDD. In our experience with policies for smart spaces, the access rules for devices involve a small number of variables, resulting in compact OBDDs in general. *Know* provides feedback only if it can be done with acceptable overhead. Example policies in the Gaia system are presented in Sections 3 and 4.

With this background, we proceed to describe the *Know* architecture.

3. ARCHITECTURE

We augment the Gaia access control mechanism with our feedback component (*Know*). The Gaia access control mechanism intercepts all requests for service, and checks them against the system policy. If disallowed, the access request is forwarded to *Know*, which prepares a feedback message for the user. The message contains a list of alternative conditions under which access to the given service is permitted. Since Gaia is highly context-driven, the feedback may suggest changes in context. For example, if a printer is inaccessible due to the current context (e.g., a meeting in the room disallowing the use of noisy printers), feedback may be of the form, "if you return after the meeting, then you will have access to Printer X."

When providing feedback to a user, a system must not compromise sensitive components of the system policy. For example, feedback of the form, "if you are a Motorola or IBM employee, then you will have access to this room" reveals sensitive information that IBM and Motorola may be collaborating on a project. To protect such information, *Know* augments the policies with meta-policies. A meta-policy governs access to a policy, thereby treating policies as objects themselves. When determining feedback for a user (Alice), *Know* first checks the meta-policy to see what parts of the policy Alice is allowed to read, and constructs feedback using only those parts. UniPro [21] provides a generalized framework to protect parts of the policy with a policy, which in turn can be protected by another policy, and so on. Here, in the interest of simplicity, we focus on policies and their meta-policies (and not multiple levels of meta-policies). Henceforth our notation will resemble that of UniPro, since we use a subset of its functionality. We now provide some concrete examples of access policies, and their associated meta-policies.

Example 1 The access policy of an electronic door lock might allow access only to Computer Science professors or members of the CIA. When a person is denied access to the room, feedback of the form, "if you are a CS professor or a member of the CIA, then you will have access to this room" is potentially dangerous. Collaboration between the Computer Science department and the CIA could be sensitive information. Outsiders may also glean intelligence information about where CIA members meet. Clearly, we may not want to reveal parts of this access rule. Feedback of the form, "if you are a professor in Computer Science, then you will have access to this room" may be acceptable. A meta-policy would control this flow of information to denied users. Formally, we can represent the policy and meta-policy as shown

below:

Policy:

$$\begin{aligned} R &: P \\ P &\leftrightarrow P1 \vee P2 \\ P1 &\leftrightarrow User.role = Professor \\ &\quad \wedge User.department = CS \\ P2 &\leftrightarrow User.role = CIA \end{aligned}$$

Meta-Policy:

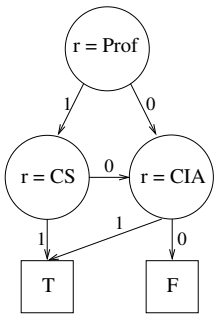
$$\begin{aligned} P1 &: User.department = CS \\ P2 &: false \end{aligned}$$

A policy definition includes two types of expressions. An expression of the form $O : P$ means that an object O is protected by policy P, where policies themselves can be objects (since policies may be protected by meta-policies). An expression of the form $P \leftrightarrow E$ means that the policy P is defined by expression E. Expressions can contain both atomic propositions (e.g., $User.department = CS$) and references to sub-policies (e.g., $P \leftrightarrow P1 \vee P2$, where $P1$ and $P2$ are defined subsequently). The access policy for the room is $R : P$, which means that access to the room R is protected by policy P. P is defined as the disjunction of policies $P1$ and $P2$. $P1$ is the policy, "User must be a professor in Computer Science." $P2$ is the policy, "User must be a member of the CIA." Hence the policy P to access the room is "User must be a professor in Computer Science or the user must be a member of the CIA." Figure 2(a) shows the associated OBDD for this policy. The meta-policy $P1 : User.department = CS$ indicates that the policy $P1$ may be revealed only to subjects in the Computer Science department, while the meta-policy $P2 : false$ does not reveal $P2$ under any circumstances[1]. For example, a denied student in Computer Science would receive the feedback "if you are a professor in Computer Science, then you will have access to this room," while a student in Civil Engineering will be informed, "access is denied." In either case, no policy information involving the CIA is revealed. We discuss how to apply meta-policies to OBDDs through cost functions in Section 3.1.

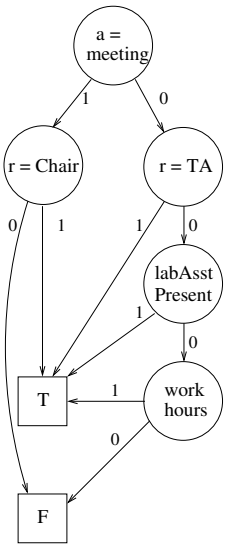
We make two assumptions here. First, we assume that any logical dependencies between atomic propositions are captured within the policy. For example, a policy may contain atomic propositions $User.isAdult$ and $User.isMinor$. We know that $User.isAdult \Leftrightarrow \neg User.isMinor$, and hence feedback of the form "if you are an adult and a minor, you will have access" would be absurd. Such inconsistencies are avoided by either replacing occurrences of $User.isMinor$ by $\neg User.isAdult$ or by adding the logical rule $User.isAdult \Leftrightarrow \neg User.isMinor$ to the policy. This will avoid any inconsistencies in feedback. The second assumption we make is that all references to an atomic proposition a are protected by the same meta-policy. We elaborate on this in Section 3.2.

It is important to note that in all our examples we are careful to provide feedback as "if...then" clauses. This is important for policy protection. Feedback of the form, "only professors may access this room" gives more information than "if you are a professor, then you will have access to this room." If both types of feedback were allowed, the user may infer from the latter feedback that there is a protected policy not being revealed. Hence, if feedback is consistent in its use of "if...then" clauses, users will not gain any extra information about protected policies.

[1]In our examples we omit meta-policies of the form $P : true$ for clarity. In practice however, all meta-policies are assumed to be of the form $P : false$ unless a meta-policy is explicitly specified.



(a) Example 1, with $r \equiv User.role$



(b) Example 2, with $r \equiv User.role$, $a \equiv Context.Activity$, $t \equiv Context.time$

Figure 2: OBDDs for the examples

.

Example 2 Since we are interested in providing useful feedback for complex policies, we provide a more complex example. We first present the policy P for a printer A, and then augment it with a meta-policy. The policy $A : P$ may be as follows:

Policy:

$$\begin{aligned} A &: P \\ P &\leftrightarrow P1 \vee P2 \\ P1 &\leftrightarrow Context.activity \neq meeting \wedge (P3 \vee P4 \vee P5) \\ P2 &\leftrightarrow Context.activity = meeting \wedge P6 \\ P3 &\leftrightarrow User.role = TeachingAssistant \\ P4 &\leftrightarrow Context.labAssistantPresent = true \\ P5 &\leftrightarrow Context.workingHours = true \\ P6 &\leftrightarrow User.currentRole = MeetingChair \end{aligned}$$

To understand this policy, first note that printer A is noisy,

and so, disruptive to meetings being held in the room. During meetings, printer access is restricted to the person in charge of the meeting. $P2$ and $P6$ ensure that nobody will disturb the meeting by using the printer, but the meeting chair may use the printer if needed. When there is no meeting in effect, we would like to grant access to the printer to anybody during normal business hours ($P1, P5$). At other times, users are only allowed to access this printer while no meeting is in progress, and then only in the presence of a Lab Assistant ($P1, P4$). Teaching Assistants have 24-hour access ($P1, P3$) to the printer to perform their duties (again, as long as there is no ongoing meeting in the room). Figure 2(b) shows the associated OBDD for this policy.

Consider the case when a Student is denied access to the printer. An "Access Denied" message may be confusing to the Student who was able to access the printer the previous day. In the spirit of offering the user a consistent view of the system's policies, we would like to inform the user why the access was denied. Was it because a meeting was in effect? Should the user come back during regular business hours when there is no meeting? Should the user be informed that there is no lab-assistant in the room and that it is past business hours? Clearly there are several useful options available to the user. Now consider some other options that may not be of much help to the user. Let us say that access was denied outside of working hours, and no meeting was taking place in the room. Feedback of the form "Sorry, access is denied, but if you are a Teaching or Lab Assistant, then you will have access to this printer" is clearly less useful to the user, since becoming a Teaching or Lab Assistant is a non-trivial task.

Consider a final scenario when a person is denied access to the printer because of an ongoing meeting. If the user is not a member of the meeting, feedback of the form "if you are the meeting chair, then you will have access to this printer" may suggest to users that they request the chair to print documents. This is clearly disruptive, so we would like to disclose this fact only to people who are participating in the meeting. Consider the following meta-policy for the policy provided above:

Meta-Policy:

$$P3 \;:\; false$$
$$P6 \;:\; User \in Context.activityMembers$$

Using this meta-policy, we restrict feedback provided to users. Users who are denied access are not informed that Teaching Assistants have 24-hour access ($P3 : false$), since this may result in several students accosting their Teaching Assistants for their personal printing needs. Further, users who are denied access to the printer during an ongoing meeting, are only informed about the meeting chair's printing capability if they are a member of the current meeting ($P6 : User \in Context.activityMembers$).

We have presented examples of feedback that a user may, or may not, find useful. Furthermore, there were some examples of feedback that were restricted or eliminated by the meta-policy due to privacy concerns. Providing useful feedback in the face of restrictions by a meta-policy, and the relative usefulness of the various options available to the user, suggests the use of a *cost function*. This cost function can evaluate each feedback option, and generate an ordering that says one option is better than another. For example, the system may provide the user with the three most useful options as determined by the cost function. We now describe cost functions in more detail and formalize the notion of "feedback."

3.1 Cost Functions

When the access policy for a resource is not satisfied, we can try to compute all the paths from the root of the OBDD of the policy to the *true* node. Essentially, a set of such paths will be presented as feedback to the user since they represent assignments that satisfy the policy. Some of the edges followed in the path will correspond to conditions that do not currently hold. Consider Example 2. Suppose a meeting is in progress, and a TA tries to access the printer. One possible path from the root to *true* is $Context.activity \neq meeting, r = TA$. This requires one change since there is a meeting in progress. Other paths may require more changes. The number of changes that must be made for each path, and the relative difficulty in making certain changes over others, suggests the use of cost functions to rank the options. We first introduce some notation and a formal definition of *feedback*.

We use the notation $S \models P$ to indicate that a policy P is satisfied under the atomic propositions specified by S. For example we could have $S = \{Context.meeting = \text{false}, Context.workingHours = \text{true}\}$. In the notation $S[a] \models P$, $S[a]$ is the set of atomic propositions in S along with any update provided by a. In our example above, $S[Context.meeting = \text{true}] = \{Context.meeting = true, Context.workingHours = true\ \}$. This notation can be naturally extended for a set of updates, e.g., $S[A]$, where A is a set of atomic propositions. Let C be the set of atomic propositions relating to the context of the system and U be the set of atomic propositions specific to the user (identity, role, etc.). Given a policy P and a user U, the user is granted access when $C \cup U \models P$, and denied access when $C \cup U \not\models P$. In essence, if $C \cup U \not\models P$, then a set of updates X such that $(C \cup U)[X] \models P$ constitutes a feedback option to the user.

To formalize the notion of feedback, let $\Pi = \{\pi_1, \ldots, \pi_n\}$ be the set of paths from the root node to the *true* node in the OBDD of P. Let π_i' be the set of atomic propositions that appear in $\pi_i \in \Pi$ and not in $C \cup U$, i.e., the set of propositions that must be changed (or a set of updates to the state) for the policy to be satisfied. Let $\mathcal{F} = \{\pi_1', \ldots, \pi_n'\}$. Note that $(C \cup U)[\pi_i'] \models P$ for all $\pi_i' \in \mathcal{F}$. We define any subset F of $\mathcal{F}$ to be the $feedback$ offered to the user. In other words, each *feedback option* f_i in the *feedback* F corresponds to a set of atomic propositions the user must change to be granted access. $\mathcal{F}$ is the set of all possible feedback options available to the user. Since $\mathcal{F}$ can be very large, our primary goal is to find a way to offer the user only a few relevant feedback options in $\mathcal{F}$. We do this through the use of cost functions. The cost function assigns a cost to each $f \in \mathcal{F}$, and returns the k lowest-cost feedback options, where k is a tunable parameter.

A naïve cost function could assign the same cost to each change, in which case the user would be given feedback with the least number of changes that need to be made to access a resource. For example, we could sort the elements f_i of $\mathcal{F}$ in ascending order based on $|f_i|$ (number of atomic propositions in f_i) and return the first k choices. However, changing roles might be more difficult than changing context. For example,

a Student may be able to come back at a later time, but it would be extremely difficult to acquire a Professor role. This suggests the use of more sophisticated cost functions.

We need to define an appropriate cost function that is applied to edges in the OBDD as edge weights. Using these weights we can use shortest path algorithms from the root to *true* to provide feedback with lowest total cost. Running Dijkstra's algorithm gives us a path with lowest total cost in polynomial time. There are several proposed algorithms for k shortest paths for graphs. Eppstein [5] presents an algorithm that computes k shortest paths in time $O(m + n\log n + k)$, where n is the number of vertices, and m is the number of edges in the graph. This is the best known bound for k shortest paths in directed acyclic graphs. Since an OBDD with n nodes has $2n-4$ edges (two children for each node, except the *true* and *false* nodes), the complexity for computing the k shortest paths in an OBDD is $O(n\log n+k)$.

Let A be the set of atomic propositions (this corresponds to the condition being tested within a node of the OBDD) for the policy P. We define a cost function $c : A \times \{0,1\} \rightarrow \mathbb{R}^+ \cup \{\infty\}$, where $\mathbb{R}^+$ is the set of non-negative real numbers. An infinite cost disallows any changes to the current value of the proposition. This function tells us the cost to follow a 0-edge or a 1-edge for a node in the OBDD. When a request for access is denied, let $T \subseteq A$ be the set of propositions that evaluate to true, and $F \subseteq A$ be those that evaluate to false (note that T, F form a partition of A). We define $c(t,1) = 0$ for all $t \in T$ and $c(f,0) = 0$ for all $f \in F$ since there is no cost to follow an edge that is satisfied under the current conditions ($C \cup U$), and we would like to assign non-zero cost when a user must *change* some atomic proposition. Cost functions will differ according to their assignments to $c(t,0)$ for all $t \in T$ and $c(f,1)$ for all $f \in F$. Now, for all $a \in A$, assign the weight $c(a,0)$ to the 0-edge of a, and $c(a,1)$ to the 1-edge of a. What results is a directed acyclic graph with weights assigned to each edge. We can now apply k-shortest path algorithms to this graph to get the k lowest-cost paths. For small k the running time for such algorithms is dominated by the structure of the OBDD and not k. Specifically, since we expect to have $k < n$ (for example $k = 3$ might be sufficient), then the running time is $O(n\log n)$. Our naïve cost function that considers all changes to be equally expensive would set $c(t,0) = 1$ for all $t \in T$ and $c(f,1) = 1$ for all $f \in F$. Hence the total cost of any path is equal to the number of propositions that need to be changed under the given conditions.

3.2 Meta-Policies

Now that we have described the basic algorithm for computing feedback using OBDDs and shortest path algorithms, we must modify the algorithm to honor the meta-policies. Each meta-policy determines whether a user can read certain nodes in the policy's OBDD. Let $D \subset A$ be the set of nodes forbidden by the meta-policy. For each $d \in D$, we assign infinite cost to the edge that effects a change in the current value of d. This does two things: first, it prevents shortest path algorithms from exploring a change in d and hence does not return any feedback options that require a change in d. Second, since this proposition d cannot be changed, it will not appear within a feedback option, which includes only those propositions that must be changed. Since no atomic proposition that is precluded by the meta-policy appears in any feedback option, the feedback given to the user honors the meta-policy. We assume that all nodes corresponding to a particular atomic proposition a are protected by the same meta-policy, allowing us to perform such a transformation. Finding efficient ways of computing consistent feedback where references to the same atomic proposition are protected by different meta-policies is left to future work.

3.3 A Useful Cost Function

We now present a useful cost function that improves on the results of the naïve cost function in the context of ubiquitous computing environments. We improve upon the naïve cost function by forbidding the exploration of certain, obviously undesirable, options. Paths to the *true* nodes will still be graded according to the number of changes, but certain changes are forbidden.

Activities: Gaia policies for a smart space depend on the current activity in that space. Example 2 showed the policy for the activities *meeting* and *no meeting*. Consider a space with the possible activities of *meeting, conference, reception, presentation* and *no activity*. If a user is denied access to a resource during a *meeting*, feedback of the form "During a *conference* if you are the Chair, you will have access" is not very useful. Hence we only provide the user feedback for the current activity, and the absence of any real activities (*no activity*). Hence we apply an infinite cost to all 1-edges for the activities that are not current (the 0-edges will have 0 cost) and apply a cost of 1 to the 0-edge for the current activity. Hence *Know* will not explore feedback for other activities, but will explore feedback for both, the current activity and no activity.

Roles: This cost function assumes that it is difficult for a user to obtain a new role. Let N be the set of nodes in the OBDD that tests for a role that the user has not activated. For each node in N, assign infinite cost to the 1-edges (the 0-edges will have 0 cost). Hence the system will not provide any feedback that requires a user to obtain a new role. A user may choose to activate only certain roles in the system. If the user is not satisfied with the feedback options obtained, he or she may activate another role and get better feedback.

Meta-policy: As described in Section 3.2, we assign infinite costs to all edges that require a change to variable assignments forbidden by the meta-policy.

For a given feedback option (path from root to the *true*-node in the OBDD), the cost is the number of changes to be made. Since certain edges are forbidden due to infinite cost, feedback provided to the user will not require any changes in the current role, and will only be for the current activity, or no activity. We believe that this cost function is useful in the context of current policies and activities in Gaia. In our analysis, we provide a detailed example policy and show how this cost function provides more useful feedback to the user than the uniform cost function.

4. IMPLEMENTATION

We have built a prototype of the *Know* system. In this section we describe the implementation and results from a preliminary evaluation. We present results of *Know* running with an example access control policy for videoconferencing equipment located in a kiosk in a multi-purpose business center.

As currently implemented, the system access control pol-

icy is represented as an OBDD, which is then transformed into a weighted graph using an appropriate cost function to assign weights to the edges. The system meta-policy also affects these weights. Finding the k shortest paths to the 1-node of the OBDD gives us k sets of assignments to the variables that will satisfy the access control rules, and thus, describe k situations under which the particular operation is allowed. We describe the process in more detail below.

The first step is to generate an OBDD from the system access control policy. Each proposition in an access control rule forms a node of the OBDD. The 1-edge represents the situation when this proposition is satisfied, and the 0-edge represents the proposition being false. We use the BuDDy [11] library which can use some heuristics for optimizing the generated OBDDs. The end result of this is an OBDD that represents all allowable ways to perform a particular action (or access a particular resource). Given a user credential and the current context, a path from the root to either the 1-leaf or the 0-leaf of the OBDD is determined by the current values of all propositions. If the path reaches the 1-leaf, the action is permitted and *Know* is not needed. If the path reaches the 0-leaf, it means that the current credential and context do not permit this action. *Know* now attempts to find alternative paths in the OBDD that would permit the operation.

Alternative paths are found by using the Eppstein [5, 7] algorithm to find the k shortest paths from the root to the 1-node in this OBDD. Weights are assigned to the edges of the OBDD graph based on the cost function and the current values of the user roles and context variables. We provide results from the two cost functions described earlier—the naïve cost function (which just counts the number of changes required) and the "useful" cost function (which treats role changes as more difficult to achieve than context changes). Selecting a suitable cost function is site-specific—the weights assigned to the different changes will depend on the nature of tasks that are normally performed by users of the system.

Know then outputs the necessary changes that must occur to satisfy the alternative paths. It is up to the user to choose between these suggestions, and to retry the request after following the suggestion.

4.1 Evaluation

We illustrate this entire process with its application to a sample policy that governs the access to devices in the business center of a hotel. In addition to computers, the business center also contains devices such as printers, fax machines, cameras for videoconferencing and so on. The business center is located in the conference hall, and hotel guests and other members who have signed up are normally allowed to use the devices as per the security policy. The conference hall is also rented out for activities such as meetings, conferences or receptions, during which time use is restricted to participants of this activity, as per the policy configuration by the organizers. Users present their credentials to enter the business center, in the form of a smartcard (a conference badge or a hotel room key) and the system uses this information to restrict access and provide useful feedback. We present here the rules that affect access control to the camera for the videoconferencing system.

The basic policy is as follows:

- When no activity is scheduled for the room, supervisors, hotel guests or other registered users can use the videoconferencing equipment during the business day. Visitors are also allowed to use the facilities if an *operator* is present. Hotel guests may also use the system during non-business hours, but others may not.

- When an activity (such as a videoconference) is scheduled, only registered activity participants and supervisors are allowed to use the system.

- Use of the videocamera is disallowed for regular participants if the videoconferencing activity being undertaken in the conference center is labeled as confidential. However, the meeting supervisor may still turn on the videocamera if all participants have the required security clearance.

- Each activity being conducted in the center may define its own policies for its users for its duration.

- Maintenance activities are performed by designated personnel.

- Finally ambient temperature above 30C indicates some problem with the air-conditioning/cooling system, and camera use is prohibited until temperature reaches the allowed range. Similarly, overcrowding the room will violate the fire safety codes and cause access to the camera to be denied.

The meta-policy that governs feedback contains the following rules:

- Information about confidential activities is only provided to the meeting supervisor. Thus an unauthorized user trying to access the videocamera during a confidential activity will not be informed that a confidential activity is going on, but just that access is denied at that time. Similarly, feedback about the presence of uncleared users is only given to the meeting supervisor.

- Information about maintenance activities is not provided to other users.

The access control rules for the policy above are presented below. In our implementation, access to the camera C is protected by policy P. Policies $P1, \ldots, P10$ describe the various rules presented above, where $P7$ and $P8$ are rules pertaining to the Video Conference activity. In the interest of brevity, we only present the rules relevant to the Video

Conference activity.

Policy:

$$
\begin{array}{rcl}
C & : & P \\
P & \leftrightarrow & P1 \vee \ldots \vee P10 \\
\ldots & & \\
VC & \leftrightarrow & activity = VideoConference \\
 & & \wedge\neg(A_1 \vee \ldots \vee A_n) \\
CA & \leftrightarrow & Context.isConfidential = true \\
RS & \leftrightarrow & User.role = Supervisor \\
NU & \leftrightarrow & Context.UnclearedUsersPresent \\
 & & = false \\
NH & \leftrightarrow & Context.cameraOverheated \\
 & & = false \\
NF & \leftrightarrow & Context.roomFull = false \\
P7 & \leftrightarrow & VC \wedge CA \wedge RS \wedge NU \wedge NH \\
 & & \wedge NF \\
RP & \leftrightarrow & User.role = Participant \\
P8 & \leftrightarrow & VC \wedge \neg CA \wedge (RP \vee RS) \wedge NH \\
 & & \wedge NF
\end{array}
$$

Meta-Policy:

$$
\begin{array}{rcl}
P & : & true \\
CA & : & User.role = supervisor
\end{array}
$$

This policy states that during a confidential Video Conference, only a Supervisor can access the camera as long as there are no uncleared users present. During a non-confidential Video Conference, any Participant or Supervisor can access the camera. The room must never be Overheated or Full during a Video Conference. The second rule in the meta-policy states that only Supervisors will be made aware of confidential activities. Hence if an ordinary user is denied access to a camera, the user will not be told that there is a confidential conference in progress (this information itself is deemed sensitive). Only Supervisors can receive feedback about confidential activities. Since there can be only one activity at any given time, the policy specifies $VC \leftrightarrow VideoConference \wedge \neg(A_1 \vee \ldots \vee A_n)$, where $A_1 \ldots A_n$ are the remaining activities.

The OBDD generated by the above policy has 17 variables and 35 nodes (in contrast, a binary decision tree would have at least 2^{17} nodes).

To evaluate *Know*, we try to access the videocamera under a variety of situations, and present the suggestions provided by *Know* using each of the two cost functions described earlier, which we designate as the "naïve" and the "useful" cost function. Since the useful cost function just restricts information about role and activity change, feedback from the useful cost function will just be a (more useful) subset of the feedback from the naïve cost function. We describe some of the experiments below for $k = 4$. The run-time overhead for *Know* to find these suggestions was negligible—in the order of milliseconds. Since OBDDs are just a representation of the access control policy, they can be constructed ahead of time and only need to be re-computed if the policy changes. Assigning weights to the edges of the OBDD has to be performed each time a request arrives, since the weights depend on the current values of the context variables and user credentials. Since *Know* only runs when access is denied, it has no performance overhead on successful requests.

We now describe the situations and results in detail:

- A *visitor* tries to use the camera during business hours, but no *operator* is present. There is no activity in session. With the naïve cost function, *Know* suggests that the user come back a) as a *hotel guest* b) as a *registered room user* c) when an *operator* is present, or d) as a *supervisor*

  The useful cost function suppresses the suggestions involving a role change, and only advises the user to come back when an *operator* is present. This simple example illustrates the basic functionality of *Know*.

- If a *hotel guest* tries to use the equipment during working hours when the room is too hot and there is no activity in session, *Know* correctly suggests that the user try again when the temperature is within the allowed range. Specifically, with the naïve cost function, the three options suggested by *Know* are: come back a) when room is not *overHeated* b) when room is not *overHeated* and it is out of business hours and c) when room is not *overHeated* and as a *registered room user* instead of a *hotel guest*, or d) when room is not *overHeated*, as a *supervisor*, instead of a *hotel guest*. The useful cost function only offers the first two suggestions because it does not recommend role changes. Clearly, the only change *required* is for the temperature to be reduced, but *Know* does not presently try to recognize if some of its suggestions are subsets of others. This may be a useful test in some situations.

  Maintenance operations are allowed even in overheated conditions, and a straightforward search through the policy might have offered the suggestion to try coming back as a *maintenance worker*. However, the system meta-policy forbids disclosure of information about maintenance permissions, so this option is correctly ignored by *Know*.

- During a *confidential* videoconferencing activity and regular working hours, if a *participant* tries to access the videocamera when users without the required security clearance are present, the naïve cost function also suggests the user come back a) as a *hotel guest* when no activity is in progress, b) as a *room user* when no activity is in progress, c) as a *visitor* when no activity is in progress, or d) as a *supervisor* when no activity is in progress. The useful cost function does not offer any feedback, because there is no useful option for the *participant*.

  One possible suggestion is to inform the user that this operation is not permitted during a *confidential* activity and to suggest re-trying when no confidential activity is being undertaken, but the system meta-policy precludes any information about confidential activities from being revealed, so this suggestion is not offered.

- If a *supervisor* tries to use the camera when the room is reserved for a confidential *videoconference* and uncleared users are present, the uniform cost function suggests that the user come back a) after changing the activity type to be non-confidential b) when no uncleared users are present, c) when there is no activity scheduled, or d) as a participant after changing the activity type to be non-confidential. The useful cost

function suggests the first three options. Note how the *supervisor* is given feedback regarding confidential activities, as opposed to a *participant* in the previous scenario.

While the above examples are fairly simple, they validate our hypothesis that *Know* can provide useful information about alternatives when access is denied, that it can do so without compromising privacy or confidentiality requirements of the security policies, and that this can be achieved with negligible performance overheads. We are in the process of integrating *Know* fully with the Gaia system, after which we can perform larger-scale studies.

5. DISCUSSION AND RELATED WORK

Policy hashing [10] has been proposed to protect the policies for a firewall from less trusted enforcement points. This prevents intruders from reading sensitive policies on compromised enforcement points. Feedback to end-users is not a consideration. Access control systems for Web publishing [2] provide more information about the policy if conditions needs to be changed for access. However, policy protection is not addressed.

Trust negotiation protocols [19, 3, 22] address the problem of protecting the confidentiality of credentials of both parties involved in a session. At each stage, one of the parties must provide feedback to the other party as to what credentials are needed to proceed with the negotiation. *Know* can augment these systems at each stage in trust negotiation by providing useful feedback. With respect to suppressing feedback options, Bonatti et al. [3] protect the server's *state* by filtering policy feedback. Such techniques can also be applied to *Know*, which protects the server's *policies*. Policy protection in [3] is achieved by progressively revealing more requirements depending on credentials revealed by the user.

Usability has been recognized as an important concern for security systems since the early days [16] of research in computer protection systems; however, in practice, usability issues have not been a primary consideration for security designers. Usability concerns are especially important in ubiquitous computing environments, since the objective of ubiquitous computing is to blend into the background and allow the user to perform his tasks without having to pay attention to the computing environment. Work on the human-computer interface aspects of security [24, 20] have identified consistent feedback as an important aspect of usability. We posit that providing useful feedback about access control decisions is a step in the right direction. Users can then obtain a better picture of the security policies and can access resources accordingly.

While policy feedback improves system usability, one of the major concerns is information leakage. Meta-policies allow system administrators to treat the policy just like any other system resource, and configure access to it accordingly. Thus, providing feedback does not leak any unauthorized information. We recognize that writing effective meta-policies that are robust against statistical inferencing remains a challenge, and further research in writing effective meta-policies is needed.

Computing feedback options is equivalent to computing variable assignments to satisfy a boolean formula. In general, computing assignments for satisfiability (SAT) of a boolean formula is NP-complete, and finding least-cost assignments (Weighted SAT) is an NP-hard optimization problem [12]. Therefore, we cannot expect to compute "least cost" feedback in all cases. We propose that feedback should be provided if it can be done with acceptable overhead. Administrative decisions can bound the computational and storage resources usable for computing feedback and degenerate OBDDs may not be computed or stored.

OBDDs are an efficient and compact representation of boolean formulas, and tests for satisfiability are efficient. It is well known that the size of OBDDs depends on the variable ordering (the order in which variables are tested in an OBDD), and in certain cases it is not possible to reduce the exponential state space of decision trees. In fact, determining a suitable variable ordering (yielding a minimum sized OBDD) has been shown to be NP-complete [1]. Given these challenges, it is comforting to know that commonly encountered functions have reasonably sized OBDDs and there are several heuristics (e.g., group-sifting [13] is one of the popular methods, also see [4, 6]) to determine adequate variable orderings for small (non-exponentially sized) OBDDs. Degenerate cases usually involve functions that behave differently for all possible assignments (e.g., output of an integer multiplier [14] and integer division [8]), and we do not expect such state space explosion in practice.

An interesting avenue for future work is the study of suitable cost functions. More nuanced cost functions could better reflect the relative difficulty of changing propositions. Selection of an appropriate cost function will be influenced by various factors, such as user preferences and system usage patterns. Another option could be to allow users to specify their own cost function to tailor the *Know* feedback. Finally, learning algorithms [23] could be used to improve feedback over time by allowing users to rate the feedback received. Inference rules may be added. For example, if Alice is a student in a particular course, it is unlikely that she is also the TA for that course. Such cost functions will improve the quality of feedback. Bounded searches can be more efficient. For example, the user can also specify the maximum number of changes, v, that he or she is willing to accept. A naïve brute-force algorithm for computing feedback would take time $O(n^v)$, which might be acceptable overhead for small v.

6. CONCLUSIONS

We have presented *Know*, a system for providing feedback to users about access control policy decisions. When the system denies a user access to a resource, *Know* suggests alternatives for the user to gain access. While a list of all possible conditions under which a particular resource may be accessed is likely to be large and not very useful, we try to restrict the options presented to the user to a smaller set of useful options by the use of appropriate cost functions. An important consideration is that this process should not leak any information that would compromise system security or confidentiality. This is achieved by using a meta-policy to represent the required protection for the policies themselves. We present qualitative performance results from a prototype implementation.

We are currently extending our prototype for integration with Gaia, our operating environment for ubiquitous computing environments. This will allow us to evaluate *Know* with more extensive, and real, policies with actual users. Future work in the area of selecting appropriate cost functions

is also indicated. Apart from the system-selected cost function that our prototype implementation uses, user-selected cost functions may be useful since different users may assign different levels of difficulty to the same task. Artificial intelligence techniques such as learning or planning algorithms [23] might also be helpful to learn a user's preferences. Another question of interest is the feedback mechanism, since ubiquitous computing environments use a variety of mechanisms for users to interact with them, and text messages on a display monitor may not always be available or appropriate for system feedback. Audible feedback may reveal one user's feedback to other nearby users, which may not be appropriate.

Lastly, we believe that research in the area of providing useful feedback to denied users has been vastly neglected, and to the best of our knowledge this is the first attempt at integrating policy feedback with policy protection.

7. ACKNOWLEDGMENTS

We thank Sariel Har-Peled and Mahesh Viswanathan for helpful comments and the anonymous reviewers for their useful feedback.

8. REFERENCES

[1] Beate Bollig and Ingo Wegener. Improving the variable ordering of OBDDs is NP-complete. *IEEE Trans. on Computers*, 45(9):993–1001, Sept. 1996.

[2] Piero Bonatti, Ernesto Damiani, and Pierangela Samarati. A component-based architecture for secure data publication. In *Proceedings of 17th Annual Computer Security Applications Conference (ACSAC)*, New Orleans, LA, December 2001.

[3] Piero A. Bonatti and Pierangela Samarati. A uniform framework for regulating service access and information release on the Web. *Journal of Computer Security*, 10(3):241–271, 2002.

[4] Kenneth M. Butler, Don E. Ross, Rohit Kapur, and M.Ray Mercer. Heuristics to compute variable orderings for efficient manipulation of ordered binary decision diagrams. In *Proceedings of the 28th conference on ACM/IEEE Design Automation*, pages 417–420, San Francisco, CA, June 1991.

[5] David Eppstein. Finding the k shortest paths. In *Proc. 35th Symp. Foundations of Computer Science*, pages 154–165. IEEE, November 1994.

[6] Masahiro Fujita, Yusuke Matsunaga, and Taeko Kakuda. On variable ordering of binary decision diagrams for the application of multi-level logic synthesis. In *Proceedings of the conference on European Design Automation*, pages 50–54, Amsterdam, The Netherlands, February 1991. IEEE Computer Society Press.

[7] Jonathan Graehl. `kbest`, a C++ library for efficiently finding the k shortest paths in a graph. Available from `http://jonathan.graehl.org/kbest.zip`.

[8] Takashi Horiyama and Shuzo Yajima. Exponential lower bounds on the size of OBDDs representing integer divistion. In *Proceedings ISAAC*, pages 163–172, 1997.

[9] Brad Johanson, Armando Fox, and Terry Winograd. The Interactive Workspaces project: Experiences with ubiquitous computing environments. *IEEE Pervasive Computing magazine*, 1(2):67–74, Apr–Jun 2002.

[10] Håkan Kvarnström, Hans Hedbom, and Erland Jonsson. Protecting security policies in ubiquitous environments using one-way functions. In D.Hutter et al., editors, *Security in Pervasive Computing 2003*, volume 2802 of *LNCS*, pages 71–85. Springer-Verlag, Heidelberg, 2003.

[11] J. Lind-Nielsen. BuDDy – a binary decision diagram package. Technical Report IT-TR: 1999-028, Technical University of Denmark, 1999.

[12] P. Orponen and H. Mannila. On approximation preserving reductions: Complete problems and robust measures. Technical Report C-1987-28, University of Helsinki, Dept. of Computer Science, 1987.

[13] Shipra Panda and Fabio Somenzi. Who are the variables in your neighborhood. In *Proc. International Conference on Computer-Aided Design (ICCAD '95)*, San Jose, CA, November 1995.

[14] R.E.Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[15] Manuel Román, Christopher K. Hess, Renato Cerqueira, Anand Ranganathan, Roy H. Campbell, and Klara Nahrstedt. GaiaOS: A middleware infrastructure to enable Active Spaces. *IEEE Pervasive Computing*, pages 74–83, Oct–Dec 2002.

[16] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. In *Proceedings of the IEEE*, volume 63, pages 1278–1308, September 1975.

[17] Geetanjali Sampemane, Prasad Naldurg, and Roy H. Campbell. Access control for Active Spaces. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, pages 343–352, Las Vegas, NV, December 2002.

[18] Mark Weiser. The computer for the 21st century. *Scientific American*, pages 94–104, September 1991.

[19] William H. Winsborough and Ninghui Li. Safety in automated trust negotiation. In *Proceedings of the 2004 IEEE Symposium on Security and Privacy*, pages 147–160, Oakland, CA, May 2004. IEEE Press.

[20] Ka-Ping Yee. User interaction design for secure systems. In *Proceedings of the 4th International Conference on Information and Communications Security*, pages 278–290. Springer-Verlag, 2002.

[21] Ting Yu and Marianne Winslett. A unified scheme for resource protection in automated trust negotiation. In *Proceedings of the IEEE Symposium on Security and privacy*, pages 110–122, May 2003.

[22] Ting Yu, Marianne Winslett, and Kent E. Seamons. Supporting structured credentials and sensitive policies through interoperable strategies for automated trust negotiation. *ACM Trans. Inf. Syst. Secur.*, 6(1):1–42, 2003.

[23] Terry Zimmerman and Subbarao Kambhampati. Learning-assisted automated planning. *AI Magazine*, 24(2):73–96, 2003.

[24] Mary Ellen Zurko and Richard T. Simon. User-centered security. In *Proceedings of the Workshop on New Security Paradigms (NSPW)*, pages 27–33, Lake Arrowhead, CA, September 1996.