Failure Feedback for User Obligation Systems

[Technical Report : CS-TR-2010-006]

Murillo Pontual
The University of Texas at San Antonio
mpontual@cs.utsa.edu

Keith Irwin
Winston-Salem State University
irwinke@wssu.edu

Omar Chowdhury
The University of Texas at San Antonio
ochowdhu@cs.utsa.edu

William H. Winsborough
The University of Texas at San Antonio
wwinsborough@acm.org

Ting Yu
North Carolina State University
tyu@ncsu.edu

ABSTRACT

In recent years, several researchers have proposed techniques for providing users with assistance in understanding and overcoming authorization denials. The incorporation of environmental factors into authorization decisions has made this particularly important and challenging. An environmental factor that has not previously been considered in this effort to provide such assistance to users arises in systems where obligations can depend on and affect authorizations. In these systems, it is desirable to ensure that users will have the authorizations they require to fulfill their obligations, and prior work has proposed denying requests to perform non-obligatory actions that would cause this property to become violated, whether the violation is a direct result of the requested action or due to obligations that would be incurred as a result of it. Because of privacy concerns, as well as the intricate interactions between actions and pending obligations, the current work focuses on helping users find means of overcoming their denials, rather than focusing on explanation of the cause for denial. We show that in general this problem is PSPACE-hard. We then develop an approach based on an AI-planning tool and evaluate its effectiveness empirically. We find that this tool can often be quite helpful in medium sized problem instances, particularly when the number of steps that must be taken to enable the desired action is relatively small.

Categories and Subject Descriptors

K.6.5 [**Management of Computing and Information Systems**]: Security and Protection

General Terms

Security, Theory

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
All rights reserved to the department of Computer Science, The University of Texas at San Antonio.

Keywords

Obligations, RBAC, Policy, Authorization Systems, Accountability

1. INTRODUCTION

Computer systems exist to serve the needs of their users. Security policies are enforced with the aim of preventing users from interfering with one another's system usage or with the correct functioning of the system as a whole. However, it is widely recognized that due to their complexity, security policies and mechanisms can themselves hinder the ability of users to achieve their aims.

Sometimes the security policy, current authorization state, or environmental factors disallow an action that a user needs to perform. This does not necessarily mean that taking the action is impossible. For instance, when authorization depends only on privileges granted to the user, it is relatively simple for users to understand why actions they attempt are denied. Furthermore, the steps the user can take to remedy the situation are relatively straightforward: they can negotiate with the security administrator to be granted the necessary permissions based on the need they have to accomplish their aims.

When security policies are more complex, access denial may indicate only that the desired action cannot be performed under the current circumstances. Recently, researchers have studied various situations in which understanding and remedying a denial is more difficult. For instance, Kapadia et al. [10] have studied the problem of explaining access denial that is based on environmental factors, such as time and location of access attempt. Bauer et al. [2] have studied problems that arise in managing authorization state in large scale systems. Cranor and Garfinkel [6] have studied the relationship of security and usability in many practical problems (*e.g.*, phishing, password generation), and how the complexity of correctly configuring security features such as authorization state can lead to user errors or to users turning off security mechanisms altogether. Cranor et al. [7] have addressed the problem of privacy issues when considering usability. Other authors [1] [5] have studied how user actions can affect security mechanisms.

One environmental factor that can affect authorization that has not to our knowledge been studied arises in sys-

tems that support obligations, where those obligations can require permissions and affect authorization state. Obligations here refer to actions a principal is obliged to perform in some future time (*e.g.*, the 45 CFR part 164 HIPAA [8] states that an individual has the right to request a Covered Entity (CE) (*e.g.*, a hospital) to amend his protected health information (*PHI*). After this request, the CE is obliged no later than 60 days to correct the individual's PHI (for doing this, some CE's employee must have the appropriate permissions) or to provide the individual the reasons for not amending his PHI). In previous work [9] we introduced a property called strong accountability that ensures that obligatory actions are always authorized on their appointed time interval[1]. In the model we consider, obligations are introduced as a side-effect of performing nonobligatory (*discretionary*) actions. When either a discretionary action or the obligations it would incur would have the effect of violating the accountability of the system based on the current obligation pool, the request to perform the discretionary action is denied. Thus, the reason an action is denied can depend on complex interactions of the action with the obligations already in the system's obligation pool.

Clearly users need support from the system in responding to such denials, particularly when the discretionary action is essential to the user's aims. One possible form of support would be to attempt to explain to the user the interactions of their desired action with the current obligation pool and leave it in their hands to find corrective actions that could be performed by themselves or others. However, particularly with regular users (as opposed to security administrators), this approach is likely to be inadequate for three reasons. First, providing an explanation that is sufficiently concise to be usable may be impossible. Second, the explanation may reveal confidential information about system policy, authorization state, and the obligations of other users. Third, it may be quite difficult for the user to identify corrective actions, given the complexity of inter-dependencies among obligations.

Given the above reasons, we propose our approach, called *action failure feedback plan*. In this approach, instead of explaining the user the reasons for rejecting her actions, we try to present her with possible plans of actions that will enable her to perform her desired target action. Here, we consider the user's final goal as a set of additional actions that would enable her to execute her denied desired actions.

The **first contribution** of this paper is a precise description of the action failure feedback problem (*AFFP*). To the best of our knowledge, we are the first to study usability of obligation systems in which obligations depend on and affect authorizations. We consider that this approach can be particularly useful in project management environments or workflows, when we want to ensure that a user is going to be able to fulfill his main task, even when he has some particular actions denied. Our **second contribution** is a result showing that the AFFP is PSPACE-hard for a user obligation system that uses mini-RBAC and mini-ARBAC as the authorization model.

We also study the question whether it is possible to have an action failure feedback component in practice, even if the problem is theoretically intractable. To answer this question we present an approach based on an AI planning tool. Design, specification, implementation, and empirical evaluation of this technique form the **third contribution** of this paper. The empirical evaluation of our technique indicates that it is effective in many cases for obligation sets and policies of moderate size.

The remainder of this paper is organized as follows. Section 2 provides background necessary to understand our contributions. Section 3 presents the definition of the action failure feedback problem, our complexity result, as well an AI technique for solving it. Section 4 presents the empirical evaluations of our technique. Section 5 discusses related work. Section 6 discusses future work and concludes.

2. BACKGROUND

The systems we consider are finite, but unbounded. At any given point in time, the set of users $U \subseteq \mathcal{U}$ in the system is finite, though $\mathcal{U}$, the universe of users, is countably infinite to permit systems of unbounded size. The same is true of the set of objects in the system and in the universe: $O \subseteq \mathcal{O}$. Note that we assume $\mathcal{U} \subseteq \mathcal{O}$. The finite set of actions supported is given by $\mathcal{A}$. Actions are parameterized by the user requesting the action and zero or more objects to which the action will be applied. The formal type of $a \in \mathcal{A}$ is presented below. Times are denoted by $t \in \mathcal{T}$. For our purposes, time is used principally for the purpose of determining valid schedules according to which obligations can be fulfilled. Schedules themselves are simply sequences of actions and do not indicate the precise time at which obligations are fulfilled. At any given point in time, the state of the system is given by $s = \langle U, O, t, \gamma, B \rangle$, in which t is the current time, $\gamma \in \Gamma$ is a mini-RBAC authorization state (defined just below), and B is an obligation pool. An obligation has the form $b = \langle u, a, \vec{o}, \text{start}, \text{end} \rangle \in \mathcal{B}$ and $B \subset \mathcal{B}$. We use record field-selection notation such as $b.u$ to select the elements of b and require that the times defining when the obligation must be fulfilled satisfy $b.\text{start} < b.\text{end}$. Note that $b.\vec{o}$ is a tuple of objects to which the action $b.a$ must be applied by $b.u$.

2.1 mini-RBAC and mini-ARBAC

The widely studied ARBAC97 model [15] has been simplified somewhat by Sasturkar *et al.* for the purpose of studying policy analysis, forming a family of languages called *mini-RBAC* and *mini-ARBAC* [16]. The member of the family that we use supports administrative actions that modify user-role assignments, but does not consider role hierarchies, sessions, changes to permission-role assignments, or role administration operations.

DEFINITION 1 (MINI-RBAC MODEL). *A mini-RBAC model is a tuple $\gamma = \langle U, R, P, UA, PA \rangle$ in which:*

- R and P are the finite sets of roles and permissions respectively. While P remains abstract in most RBAC models, for simplicity we assume $P \subseteq \mathcal{A} \times \mathcal{O}^*$.
- $UA \subseteq U \times R$ indicates users' role memberships.
- $PA \subseteq R \times P$ indicates permissions assigned to each role.

DEFINITION 2 (MINI-ARBAC POLICY). *Amini-ARBAC policy is a tuple $\psi = \langle CA, CR, SMER \rangle$ in which:*

[1]We do not consider the property of weak accountability [9] because deciding that property is co-NP-Complete. In this paper we use "accountability" to refer to strong accountability.

- $CA \subseteq R \times \mathcal{C} \times R$ is a set of *can_assign* rules, in which $\mathcal{C}$ is the set of preconditions. Each $\langle r_a, c, r_t\rangle \in CA$ indicates that users in role r_a are authorized to assign a user to the *target role* r_t, provided the target user's current role memberships satisfy precondition c. A precondition is a conjunction of positive and negative role memberships.

- $CR \subseteq R \times \mathcal{C} \times R$ is a set of *can_revoke* rules, in which $\mathcal{C}$ is the set of preconditions. Each $\langle r_a, c, r_t\rangle \in CR$ indicates that a user belonging to the role r_a has the capability to revoke the role r_t from any target user. Provided the target user's current role memberships satisfy precondition c.

- $SMER \subseteq R \times R$ (Static Mutually Exclusive Roles) rules are unordered pairs of roles such that no user is allowed to have both roles simultaneously. They take the form $SMER(role_1, role_2)$.

2.2 Obligations

In [9] and [14], we have presented an abstract meta-model that incorporates the basic construct of an authorization system supporting obligations. However, in this section, we present an instance of the abstract meta-model that is specialized to mini-RBAC and mini-ARBAC.

User-initiated actions are events from the point of view of our system. We denote the universe of events that correspond to nonobligatory, discretionary actions by $\mathcal{D} = \mathcal{U} \times \mathcal{A} \times \mathcal{O}^*$. We denote the universe of all events, obligatory and discretionary, by $\mathcal{E} = \mathcal{D} \cup \mathcal{B}$.

Obligations are introduced in our system when users perform nonobligatory actions. This is done according to a fixed set of policy rules $\mathcal{P}$. A policy rule $p \in \mathcal{P}$ has the form $p = a(u, \vec{o}) \leftarrow cond(u, \vec{o}, a) : F_{obl}(s, u, \vec{o})$, in which $a \in \mathcal{A}$ (which means $\langle u, a, \vec{o}\rangle \in \mathcal{E}$) and *cond* is a predicate that must be satisfied by $(u, \vec{o}, a)$ (denoted $\gamma \models cond(u, \vec{o}, a)$) in the current authorization state γ when the rule is used to authorize the action. F_{obl} is an *obligation function*, which returns a finite set $B \subset \mathcal{B}$ of obligations incurred (by u or by others) when the action is performed under this rule. Note that we assume that when the actions required by obligations $b \in B$ are performed, they do not cause any new obligations to be incurred. Thus "cascading" obligations are disallowed. (Generalization to support cascading obligations is a subject of future work.)

Let us now consider the structure of actions. When, in state s, user $u \in s.U$ performs action a on the objects $\vec{o} \in s.O^*$, $a(u, \vec{o})(s.U, s.O, s.\gamma)$ returns $\langle s'.U, s'.O, s'.\gamma\rangle$ for the new state s'. Thus, actions can introduce new users and objects, have side effects, and change the authorization state. Note that in general performing an action also introduces new obligations; these depend on the policy rule used, as well as on the action, and are handled in Definition 3 below.

Note that for a given action $e \in \mathcal{E}$ and a given policy rule $p \in \mathcal{P}$, the transition relation is deterministic. The following definition formalizes this relation.[2]

[2]Notation: for $j \in \mathbb{N}$, we use $s_{0..j}$ to denote the sequence $s_0, s_1, \ldots, s_j$, and for $\ell \in \mathbb{N}$, $\ell \leq j$, $s_{0..\ell}$ denotes the prefix of $s_{0..j}$ and when $\ell < j$ the prefix is *proper*. Similarly, $\langle e, p\rangle_{0..j}$ denotes $\langle e_0, p_0\rangle, \langle e_1, p_1\rangle, \ldots, \langle e_j, p_j\rangle$.

Definition 3 (Transition relation). *Given any sequence of event/policy-rule pairs, $\langle e, p\rangle_{0..k}$, and any sequence of system states $s_{0..k+1}$, the relation $\longrightarrow \subseteq \mathcal{S} \times (\mathcal{E} \times \mathcal{P})^+ \times \mathcal{S}$ is defined inductively on $k \in \mathbb{N}$ as follows:*

(1) We have $s_k \xrightarrow{\langle e,p\rangle_k} s_{k+1}$ if and only if the policy rule $p_k = a(u, \vec{o}) \leftarrow cond(u, \vec{o}, a) : F_{obl}(s, u, \vec{o}) \in \mathcal{P}$ satisfies $a = e_k.a$, and $s_{k+1} = \langle U'', O'', t'', \gamma'', B''\rangle$ satisfies

$$
\begin{array}{l}
s_k.\gamma \models cond(u, \vec{o}, a) \equiv (\exists r).(((u, r) \in s_k.\gamma.UA) \wedge \\
((\exists \neg r').(u, r') \in s_k.\gamma.UA \wedge SMER(r, r')) \wedge \\
(a \notin \text{administrative} \rightarrow (\langle r, \langle a, \vec{o}\rangle\rangle \in s_k.\gamma.PA)) \wedge \\
(\forall u_t, r_t).((a \in \text{administrative} \wedge \vec{o} = \langle u_t, r_t\rangle) \rightarrow (\\
a = grant \rightarrow ((\exists c).((\langle r, c, r_t\rangle \in \psi.CA) \wedge (u_t \models_\gamma c)))) \wedge \\
a = revoke \rightarrow ((\exists c).((\langle r, c, r_t\rangle \in \psi.CR) \wedge (u_t \models_\gamma c))))))) \\
e_k.u \in s'.U \text{ and } e_k.\vec{o} \in s_k.O^* \\
\langle U'', O'', \gamma''\rangle = a(u, \vec{o})(s_k.U, s'.O, s_k.\gamma)
\end{array}
$$

$$
B'' = \begin{cases} (s_k.B - \{e\}), \text{ if } e_k \in \mathcal{B} \\ s_k.B \cup F_{obl}(s_k, e_k.u, e_k.\vec{o}), \text{ otherwise} \end{cases}
$$

(2) $s_0 \xrightarrow{\langle e,p\rangle_{0..k}} s_{k+1}$ if and only if there exists $s_k \in \mathcal{S}$ such that $s_0 \xrightarrow{\langle e,p\rangle_{0..k-1}} s_k$ and $s_k \xrightarrow{\langle e,p\rangle_k} s_{k+1}$.

2.3 Strong Accountability Property

In this section, we present the definition of strong accountability [9] [14]. Accountability properties are defined in terms of hypothetical schedules according to which the given pool of obligations could be executed, starting in the given state. Strong accountability requires that each obligation be authorized throughout its entire time interval, no matter when during their intervals the other obligations are scheduled, and no matter which policy rules are used to authorize them.

Given a set of obligations B, a *schedule* of B is a sequence $b_{0..|B|-1}$ that enumerates B, for $n = |B| - 1$. A schedule of B is *valid* if for all i and j, if $0 \leq i < j \leq |B| - 1$, then b_i.start $\leq b_j$.end). This prevents scheduling b_i before b_j if b_j.end $< b_i$.start. Given a system state s_0, and a policy $\mathcal{P}$, a proper prefix $b_{0..j}$ of a schedule $b_{0..|B|-1}$ for B is *authorized by* policy-rule sequence $p_{0..j} \subseteq \mathcal{P}^*$ if there exists s_{j+1} such that $s_0 \xrightarrow{\langle b,p\rangle_{0..j}} s_{j+1}$.

Definition 4 (Strong accountability). *Given a state $s_0 \in \mathcal{S}$ and a policy $\mathcal{P}$, we say that s_0 is* strongly accountable *if for every valid schedule, $b_{0..n}$, every proper prefix of it, $b_{0..k}$, for every policy-rule sequence $p_{0..k} \subseteq \mathcal{P}^*$ and every state s_{k+1} such that $s_0 \xrightarrow{\langle b,p\rangle_{0..k}} s_{k+1}$, there exists a policy rule p_{k+1} and a state s_{k+2} such that $s_{k+1} \xrightarrow{\langle b,p\rangle_{k+1}} s_{k+2}$.*

Example 5 (Accountability). *Let us assume that Joan has all the necessary permissions to grant developer role. On the other hand, Carl does not yet have permissions to develop sourceCode. If we have two pending obligations b_1 and b_2 defined as follows:*

- b_1 : *Joan must grant developer role to Carl in time* $[7, 9]$
- b_2 : *Carl must develop sourceCode in time* $[12, 20]$

The example obligation system above is strongly accountable, because obligation b_1 can be fulfilled any time in its time interval, since Joan has all the necessaries permissions to grant developer role to another user, and no matter when Joan fulfills her obligation, obligation b_2 can be fulfilled in any time of its time interval.

2.4 Partial Order Planner

Partial-order planners are plan-space search engines. As it runs, a partial-order planner keeps track of a set of candidate plans. A candidate plan has a set of actions, a set of ordering constraints, and a set of constraints on variables. It also has a list of flaws, such as unsatisfied goals or conflicts between existing actions. The unsatisfied goals are either clauses from the end goal or preconditions from the actions in the plan.

The planner chooses a flaw and attempts to remedy it by adding new steps, links between existing steps, ordering constraints, or variable constraints. This creates zero or more new candidate plans. If a plan has flaws which cannot be fixed, then the candidate plan is discarded and other candidate plans are examined instead. Through this process, the planner will either eventually find a plan with no flaws, which is necessarily a solution to the initial problem, or fail to find any such plan.

We choose to use the UCPOP [13][3] planner, in particular, as our base for demonstrating that this technique can be effective, and because it is a partial-order planner which is well established, and therefore likely to be stable. It supports an expressive input language and it can also be modified to support our needs.

3. ACTION FAILURE FEEDBACK PROBLEM

In this section, we present a precise formulation of the action failure feedback problem for user obligation systems where obligations can depend and effect the authorization state. We use mini-RBAC and mini-ARBAC as our choice of authorization model.

In such a system, a user may submit a schedule of desired actions along with their time periods, which he intends to carry out. If the system accepts this schedule of actions, then these actions and their associated time periods become obligations for that user. If the system rejects these actions then it will attempt to formulate an alternate plan. The user has the option of accepting the alternate plan provided by the system. If accepted, the actions in the plan become obligations with the given time periods. The plan can involve actions for the user himself and other parties as well. In the event that other users are given actions to do in the plan, then those users must also agree to the plan before the system will convert the plan into user obligations. The situation in which a user attempts an immediate action is just a special form of this same process where a user attempts to schedule an action for the current time, and so we do not treat this any differently.

Let us consider an example that illustrates how an action failure feedback module ($AFFM$) can be useful. In this instance, Bob attempts to read a report, but due to lack of privileges his action is denied. He then consults the AFFM and finds out a series of actions that will enable him to read the report. Let us assume, he needs roles R_1 and R_2 to read the report. Then someone needs to grant him both the roles. Moreover, according to the policy both roles can have several other pre-condition roles and so on. Thus the number of actions that can enable Bob to perform his intended action can be large making it difficult to be deduced by human beings without any tool support. The AFFM also hides the knowledge and the complexity of the security policy from the users.

The plans we generate must have several properties:

Property 1 *They must include the same set of actions which the user had in his schedule.* Although this will not guarantee that a plan will meet the user's goals, we feel that it will make it likely to.

Property 2 *The plans must, if accepted, leave the system in an accountable state.* Obviously, plans which do not result in accountability should not be generated by a system which is trying to preserve accountability.

Property 3 *Plans should not modify existing obligations.* Modifying obligations is a special task which should only be available to administrators and not ordinary users. As such, we consider plans that do not change the current obligations.

In summary, we wish to create a failure feedback module such that when a user submits a set of actions and time frames for those actions the module will return one or more plans which contain both the user's desired actions and the existing obligations and if accepted will result in the system being accountable. Our new plan can have additional actions and the users actions in it may be scheduled to happen at different times or in a different order.

3.1 Formalization of the Action Failure Feedback Problem

An *instance of the action failure feedback problem* is given by a tuple $\langle \gamma_0, \psi, U_0, u_t, A, B \rangle$ in which:

- $\gamma_0 = \langle R, UA_0, PA \rangle$ is a mini-RBAC authorization state.
- $\psi = \langle CA, CR, SMER \rangle$ is a mini-ARBAC policy.
- U_0 represents the set of users in the system.
- u_t is the user that intends to perform the actions in A.
- B is a set of strongly accountable pending obligations.
- A is a set of desired actions that we want to add to B while preserving accountability. These actions must be performed by u_t. The actions are defined as a set of *desired_action* that can have one of the following formats:

$$\begin{cases} \langle u_t, a, o \rangle, \text{if the action is non-administrative} \\ \langle u_t, grant, r, u \rangle, \text{if the action is a grant} \\ \langle u_t, revoke, r, u \rangle, \text{if the action is a revoke} \end{cases}$$

Definition 6 (Action Failure Feedback Problem). *Does there exist a set of actions A_1 and an assignment of time periods to them, where $A \subseteq A_1$ and $B \cup A_1$ is accountable?*

[3]Though RePoP [12] is faster than UCPOP due to using improved heuristics, it is not clear how to modify RePOP to fit our requirements.

3.2 Complexity of the Problem

In this section, we discuss the complexity of the action failure feedback problem. We show that the problem is PSPACE-hard. To show that the problem is PSPACE-hard, we reduce a well known problem called the unrestricted role reachability problem [17] to the action failure feedback problem.

A user-role reachability problem instance is a tuple $Rea = \langle \gamma_0^R, \psi^R, U_0^R, u_t^R, \text{goal}^R \rangle$. In the tuple, $\gamma_0^R = \langle R^R, UA_0^R \rangle$ is an initial mini-RBAC authorization state, $\psi^R = \langle CA^R, CR^R, SMER^R \rangle$ is a mini-ARBAC policy, U_0^R is a set of users, $u_t^R \in U_0^R$ is a target user, and goal^R represents a set of roles in the system, $\text{goal}^R \subset \gamma_0^R.R$.

DEFINITION 7 (ROLE REACHABILITY PROBLEM). *Is it possible for the administrative users in U_0^R to grant u_t^R all the roles defined in the goal by just applying the rules on ψ^R?*

3.2.1 Reduction

We want to reduce the *Rea* problem into the *AFFP* problem. So, the *AFFP* instance will be:

- $u_t = u_t^R$ and $\psi = \psi^R$
- $B = \emptyset$ and $U_0 = U_0^R \cup \{u_t^R\}$
- $UA_0 = UA_0^R$
- $\gamma.PA = \{\langle r, \langle a, o \rangle \rangle | (r \in \text{goal}^R) \wedge (a \in nact()^4) \wedge (o \in nobj())\}$ (*i.e.*, for each role presented in the goal, we create a new entry in PA, where this new entry is a unique permission).
- $A = \{\langle u_t, a, o \rangle | \exists r \in R. \langle r, a, o \rangle \in \gamma.PA\}$. We create one desired action for each new PA entry that was added in the previous step.

The intuition behind the reduction is that for each role $r_i (0 \leq i \leq |\text{goal}|)$ in goal we create a new unique desired action A_i in *AFFP*. We also add the associated *PA* rules which permit a user in role r_i to perform action A_i. In addition, we assume no pending obligations. The reduction can clearly be done in polynomial time. Thus, if we find a solution for the *AFFP*, we also find a solution for the *Rea* problem.

3.3 Approach

As indicated above, we use a modified version of the UCPOP to look for solutions to AFFP instances. This section discusses how to convert an AFFP instance into the format that the planner requires, as well as some modifications we did to UCPOP to adapt it to our purpose. Before we present that material, however, we first explain why certain other classes of planners are not well suited to the AFFP problem.

Other than partial-order planners, several other types of artificial intelligence planners also exist. So, an obvious question is why use a partial-order planner in particular? The use of a partial-order planner is somewhat unusual since partial-order planners are no longer considered the state-of-the-art in the artificial intelligence community. Most planning research currently focusses on GraphPlan-based planners [4] and SatPlan-based planners [11]. Although GraphPlan and SatPlan are much faster than partial-order planners, there are several subtle reasons behind our choice. *First*, as described by Nguyen and Kambhampati in [12], both GraphPlan and SatPlan tend to produce plans with more actions. Since actions in our system will have to be carried out by users, it is certainly desirable that the generated plan be short. *Second*, the plan produced by a partial-order planner is a partial order of actions. Specifically, they aim to find the least amount of ordering constraints necessary, which maps well to our requirements. *Lastly*, it is unclear how to modify the basic algorithms of GraphPlan or SatPlan to guarantee that certain actions will be included. If we cannot generate plans which include existing obligations, then our plans will not be useful for user obligation management systems. By contrast, in a partial-order planner, modifying the algorithm to generate plans which contain the obligations and desired actions is very straightforward. Partial-order planners also support the creation of custom heuristics.

[4]The methods nact(), nobj() return a unique new action and a unique new object respectively.

3.3.1 Planner Input and Modifications

The first requirement that must be satisfied to use UCPOP for our purpose is to translate the AFFP problem instance into the input language taken by the UCPOP, namely ADL (Action Description Language). Fortunately, ADL is quite expressive. Actions in the access control system become actions in the planning domain, carrying appropriate preconditions and effects.

Generally, partial-order planners start with empty plans; however, for the *AFFP* problem we need to start with an initial plan that contains the set of pending obligations and the desired actions. Because partial-order planners do not remove steps from plans under consideration, the planner will potentially add some new pending obligations that ensure the desired action will be authorized, and the initial plan will be guaranteed to be contained in the output plan. It is straightforward to show that due to this modification the output plan will satisfy the desired properties 1 and 3 presented in section 3.

After translating the system to ADL and adding the set of pending obligations, and desired actions into the initial plan, we must make certain changes to the planning process in order to ensure that the plan will have the properties that we desire.

3.3.2 Preconditions and Flaws

Permissibility of an action depends on the authorization requirement in the policy. In the planner, the authorization requirement of an action is represented as preconditions of that action. As discussed above, in its standard usage a partial-order planner starts with an empty plan and incrementally tries to add actions in the plan. If the preconditions of the added action are not satisfied currently then the planner adds them to a list of flaws that the planner eventually tries to resolve. Because we modify the planner to take an initial plan that includes both desired actions and obligations, we must explicitly construct the list of flaws for the planner that includes the preconditions for all the desired actions. As the the preconditions for our desired actions are initially unsatisfied, we leave it up to the planner to resolve the flaws and to create appropriate linkages.

3.3.3 Variable Constraints

In a partial-order planner, arguments taken by actions are represented by variables. As in an arbitrary planning

domain, not all actions initially have all of their arguments specified. But, we need to specify specific bindings for the variables that are arguments to each obligation in the pool and for the new obligation that we wish to add. These binding are represented by constraints on the values of those variables. The variables occurring in obligations in the current pool and the new desired actions should be constrained so that their values are uniquely determined.

3.3.4 Timing

The more difficult problem lies in translating between the models of time used by user obligation management systems and partial-order planners. In a user obligation management system, any obligation would have start and stop times that are distinct from each other. By contrast, in a partial-order planner, there is only a partial order which describes the relationship in time between any two actions. In order to use a partial-order planner to make plans for a user obligation management system, we must translate the discrete times into a partial order which is used to form the initial plan. Then, when the planner returns a plan, we must translate the partial order back into a set of specific time intervals to assign to the new obligations. Existing obligations should retain their time intervals unchanged.

In a partial-order planner, the partial order represents the relationship "must happen before", denoted by $\prec$. It follows from the definition of strong accountability that if an obligation is dependent on another then it is necessary that their time periods do not overlap. As such, there exists a $\prec$ relationship between any two obligations if and only if their time frames do not overlap. That is, for any two obligations $o_1 \prec o_2 \Leftrightarrow o_1.\text{end} < o_2.\text{start}$.

Thus, let us examine how we would create the initial plan in a strongly accountable system. As we stated earlier, our set of actions in the initial plan is the union of the set of obligations and the set of desired actions. However, the obligations should not be allowed to happen in any unconstrained order. Since we are looking at a strongly accountable system, we can assume that any two obligations which are in a "must happen before" relationship must also have time frames which do not overlap.

Thus we can also determine an initial partial order to give as part of our initial plan. If two obligations do not overlap then the earlier one should be in the $\prec$ relationship with the later one. If two obligations do overlap, then they can happen in either order. As such, they should be incomparable in our $\prec$ relationship.

Because we are attempting to find a plan in which the desired actions can happen, we do not constrain their timing at all in the input, leaving them incomparable with all of the obligations and with each other. It is up to the partial-order planner to add new ordering constraints between existing actions as necessary. Together with the set of obligations and the set of variable constraints, this set of ordering constraints defines our initial plan. However, there is one additional modification which must be made to the planner.

3.3.5 Timing Constraint Restriction

The fact that partial-order planners add ordering constraints between actions can be potentially problematic for our purposes. Specifically, if there exist two obligations which overlap, then they are incomparable in $\prec$. Put another way, there is no ordering constraint between them. However, if the planner were to add an ordering constraint between them, adding either that $o_1 \prec o_2$ or $o_2 \prec o_1$, this would mean that they could no longer have the same time interval when we tried to translate the partial order back into time intervals.

For example, if we had two obligations, o_1 and o_2 with intervals $[1, 10]$ and $[5, 15]$ respectively, then if the planner added $o_1 \prec o_2$ to the plan, translating that order back into time intervals would require that $[5, 10]$ would now need to be divided between o_1 and o_2. Otherwise o_2 could happen at time 7 and o_1 at time 9, and the property of $\prec$ would not be respected and the resulting set of obligations would probably not be accountable. Now, if $o_2 \prec o_1$ were added, both would have to be squeezed into $[5, 10]$ without overlapping each other such that o_2 would be required to happen before o_1.

We have modified the planner so as to guarantee that no new ordering constraints are added between obligations, which required making the planner aware of which obligations represent existing or desired obligations. Being a partial order, the $\prec$ relation is transitive. When a new action, x, is added by the planner to the plan, some care is required to avoid introducing ordering constraints among existing obligations. For instance, suppose $o_1 \prec x$ and $x \prec o_2$ were added to the plan. This would imply adding $o_1 \prec o_2$, which must be prevented. We detect errors of this kind by using the following algorithm:

$S \leftarrow$ transitive closure of initial ordering constraints
$C \leftarrow$ transitive closure of current ordering constraints
$C' \leftarrow \{(a, b) \mid (a, b) \in C \wedge a \in O \wedge b \in O\}$
Return $C' = S$

If this returns *true* then no new obligation ordering constraints have been added. If it returns *false* then at least one has. As such, every time we evaluate whether or not a new link may be added to the plan, we run this algorithm and if it returns *false* we disallow the link from being added.

With this modification complete, we can now use the planner to generate plans which will be accountable under the timing model of the planner.

3.3.6 Converting from Planner Output to Obligations

The last step is to convert the plan that the modified partial-order planner has returned into a set of obligations. Obviously, the actions in the plan that correspond to the original obligations should simply map back to themselves. For the desired actions and any additional actions suggested by the planner, we must create new obligations. But, sometimes it may not be possible to convert the plan to obligations due to unawareness of the planner of how close together any two obligations are. For example, according to the planner output $o_1 \prec da_1 \prec \cdots \prec da_{100} \prec o_2$ when the time period for o_1 is $[10, 15]$ and the time period for o_2 is $[16, 20]$. Obviously we are not going to be able to reasonably squeeze one hundred actions into a single time click. The intuition behind the conversion algorithm is to use topological sort to convert the partial order of obligations provided by the planner to a possible total order of obligations.

Algorithm

To begin with, we wish to establish whether or not a transformation is possible. In order to do so, we assume that the new obligations have time periods which are as short as possible. As such, their initial time periods will have the property that $t_s = t_e$. Later in the process, we will relax

this to create more manageable time periods, but we use these minimal obligation time periods as a starting point. Also, for brevity, we use $x.t$ to represent the start/end time for the new obligation associated with x, an action from the plan.

The next step is to convert the ordering constraints which exist relative to the obligations into ordering constraints which exist relative to specific points of time. That is, if we know that $o_1 \prec x_1$, this implies that $o_1.t_e < x_1.t$, and if $x_1 \prec o_1$ then $x_1.t < o_1.t_s$. As such, we now have a set of ordering constraints over integers and unknowns.

We allow this to define a directed graph in which each vertex is either an integer or an action and an edge exists between any two varticies if a less than relationship exists between them. Specifically, an edge from v_1 to v_2 exists if and only if $v_1 < v_2$.

Next we begin filling in values for $x.t$ such that each $x.t$ is as early as possible. We are going to be assigning time values to the vertices in our graph. Integer vertices are considered to have their number as their time value. The vertices which correspond to actions will initially have no time value, but will be assigned one through the course of the algorithm.

First, we do a topographic sort of the graph. This defines the order in which we visit the vertices. If we are visiting a vertex which has in-degree zero then if it is an integer, we do nothing. If it is a vertex which corresponds to an action, we assign it a time equal to the current system time plus one. When we visit a vertex, v, which has an in-degree greater than zero, we consider the set of nodes which have edges which point to v. We will call this set I. We can know that all vertices in I have been previously visited and, as such, have been assigned a time value. If v is an integer vertex, then we check I to determine whether or not there is any member of I with a value greater than or equal to v. If there is, we report failure and exit. If not, we proceed. When we visit a vertex which corresponds to an action, we find the maximum time value of all vertices in I, which we will call m. We then assign $m + 1$ to v.

Thus we have, when possible, created an initial set of time intervals which conform with the plan, and therefore the resulting set of obligations is accountable. Having completed our forward pass, we may now relax the time periods of the obligations by going back over the obligations utilizing one or more backwards passes.

In the backwards passes, we may stretch the time intervals for the new actions backwards. At this point, we are going to treat the vertices as time periods again, perhaps stretching the obligation out such that the start and end times are no longer identical. We visit the verticies in the reverse of the order that we did previously, thus ensuring that we will visit no vertex until we have visited all of the vertices to which it points. When we visit an integer vertex, no changes are made. When we visit a time period vertex, we can increase both its start time and its end time. The only restrictions are that its start time must not exceed its end time and its end time must not exceed the start time of any vertex it points to.

There are many ways that the decision of how to relax the time period can be made. All changes which obey the restrictions outlined above will result in accountable systems. As such, this is where the possibility of system-specific choices comes into play. There are a large number of ways that the relaxation could be done, including making multiple passes over the graph. We will, however, outline one reasonable way which would be likely to produce reasonable results in many cases.

For a given time-period node, x being visited, find the integer node, i, such that $\frac{x.t-i}{pathlen(i,x)}$ is minimized where $pathlen(i, x)$ is the length of the shortest path from i to x measured in number of edges. To put it more simply, we are finding the fixed time before x such that the average time available for each obligation between them is least. Then we find the node v pointed to by x which has the lowest start time. We then relax x's time period to be $[v.t_s - \lfloor \frac{x.t-i}{pathlen(i,x)} \rfloor, v.t_s - 1]$. Note that this will always produce sensible results as $\frac{x,t-i}{pathlen(i,x)}$ is bounded below by 1. If it were less than 1, then our forward traverse would have been a failure.

This will not be an ideal algorithm for all circumstances, but in most cases, it should help ensure that obligations are given reasonably equitable time slots, while still allowing individual obligations to have longer time periods when it does not cause problems for other obligations.

Steps presented in sections 3.3.2 to 3.3.6 of designing the planner make sure that the output plan of the planner satisfies property 2 discussed in section 3.

4. EVALUATION OF THE PLANNER

In section 3.2, we have shown that the AFFP is PSPACE-hard. This has lead us to investigate approaches using AI planning techniques that are well known to solve search-space based problems using heuristics. This section presents empirical evaluation of the effectiveness of our approach.

The main goal of our empirical evaluation is to assess the limits of our approach for moderate size, but complex problem instances. To our knowledge, there are currently no deployed systems that incorporate the support of management and enforcement of obligations as part of their security policy. Consequently, it was necessary for us to devise our own policies and problem instances. We have three classes of input instances; two are generated from automated input instance generators and the third is hand crafted. We discuss our problem instance generation techniques in one of the following sections. After presenting the experimental settings, we discuss our approach to generating those problem instances, followed by our results and findings.

4.1 Experimental Environment

All the experiments here were performed using an Intel Core 2 Duo 3.0 GHz computer with 2 GB of memory running Ubuntu 9.04. One of the input instance generators is written in C++. The other input instance generator and the translator that translates a mini-RBAC and mini-ARBAC policy to ADL is written in Objective Caml. We used UCPOP 4.1.

4.2 Problem Instance Generation

In an effort to evaluate our planner-based approach as fully as possible, we used three different approaches to generating problem instances. All approaches generate problem instances that include obligation pools that are known to be accountable and desired actions for which it is known that plans that enable the desired actions to be added to the pool, along with other enabling obligations, resulting in a new pool that is again accountable. The only question is whether the planner is able to find the enabling actions to achieve this result.

We label the three test generators used G_1, G_2, and G_3, respectively. The goal of G_1 is to generate policies, obligation pools, and desired target actions randomly based on uniform probability distributions for each component of the problem instance. The goal of G_2 is to generate instances that seem likely to arise in a realistic system deployment setting withing an enterprise. The goal of G_3 is to generate instances that are particularly intricate with the goal of making the discovery of solutions particularly difficult for the planner.

The automated generation of instances by G_1 is based on certain input parameters. Specifically, these are the number of roles (r), the number of pre-conditions in the CA rules (c), the number of CA rules (ca), number of CR rules (cr), number of users (u), number of roles per user (ru), number of actions (a), the number of objects (o), the number of PA rules (pa), number of obligations (obl) in the existing obligation pool, number of maximum length of the sequence of new actions that need to be added to enable the desired target actions to be enabled (*depth*), and number of desired target actions (*dact*). The CA rules, CR rules, $\gamma.U$, $\gamma.O$, $\gamma.PA$ and $\gamma.UA$ are generated randomly based on the parameters ca, cr, u, o, a, o, and ru, respectively. For generating each obligation in the obligation pool, we randomly select a time window and try all possible obligations that make the system strongly accountable. We select one randomly from the possible obligations and add it to the pool of pending obligations (B). This process is repeated until a pool of size obl is obtained. For generating desired target actions, we first simulate the effect of administrative obligations of B initial authorization state γ to get a new authorization state γ_1. We then iteratively generate all administrative actions that are authorized in the current authorization state, select one at random and add it to a plan that is constructed to ensure the desired target action can actually be achieved. The authorization state is then updated accordingly. This process is repeated, each time selecting randomly among administrative actions that are currently authorized, but that would not have been authorized in the previous authorization state, until a plan of the desired length is reached (*depth*). The final action is then generated, again selected at random among actions that are authorized in the final authorization state, but not in the previous one. Due to the cost of this algorithm, we found it necessary to limit *depth* to at 5. In our experiments, G_1 is used with $ca = 100$, $cr = 100$, $pa = 200$ and $u = 40$.

For the second test generator (G_2), we automatically constructed a system state intended to be reflective of a realistic enterprise organization structure. We used 10 administrative roles which are filled randomly by 40 administrative users and 25 edges in the directed acyclic graph which represent the relationship between different administrative roles.

We have three levels of operational roles with 5, 10, and 20 roles in each of the levels. We have 140 users divided into levels and then assigned to roles. Roles for users are assigned from the same level as the user with probability 0.8 and from levels above with probability 0.1 and below with probability 0.1. Selected roles obviously can not exceed the top or bottom level. Each operational role has two grant rules and two revoke rules. Each rule has three clauses in its role expression. The clauses are positive with odds 0.6. Each obligation's time period and length are determined via a uniform random function.

dact	*obl*	t_1 (sec)	t_2 (sec)
10	10	4.03	1.19
10	20	7.43	2.46
10	40	14.69	6.87
10	80	55.89	24.24
10	160	304.64	152.81

Table 1: Execution time vs. *obl*

c	*dact*	*obl*	t_1 (sec)
1	10	80	56.29
3	10	80	60.59
5	10	80	66.70
8	10	80	64.61
10	10	80	116.94

(A)

dact	*obl*	t_1 (sec)
5	80	45.48
10	80	48.98
15	80	62.29
20	80	69.05
30	80	82.29

(B)

Table 2: (a) Execution time vs. c (b) Execution time vs. *dact*

For the third test generator (G_3), we use a handcrafted policy based on a hospital setting with $r = 58$, $c = 6$, $ca = 98$, $cr = 55$, $pa = 168$, $obl = 60$ and $dact \leq 20$. We design complex problem instances (*i.e.*, when desired actions interfere with other desired actions, obligations interfere with desired actions, desired actions interfere with obligations, and also considering explicitly *SMER* rules). We also generate inputs where authorizing each desired action needs introducing more than 5 additional actions in the plan.

4.3 Results

The execution times reported here are an average of 10 successful runs. In our experiments, we put a bound on the number of possible plans the planner can examine. A planner thus restricted is not complete, and may not find solutions when such exist. When it does not find a solution in the alloted time, it reports a failure. We have 18 failures in a total of 403 trials for a success rate of 95.53%. We feel that the success rate is satisfactory, but do not rule out the possibility of improvement.

Speaking generally, the failures occurred fairly evenly across the different sets of tests, although they were slightly more common on the tests that involved large plans. We use G_3 to identify the limits of our approach. We observe that our approach is adequate for problem instance with $dact = 1$ and $obl = 60$, when the optimal plan length is 7. This signifies that authorizing the desired action needs introducing 6 additional actions. Our approach does not scale well (greater value of *dact*) for such complex problem instances.

Table 1 shows the execution time of problem instances generated by G_1 (t_1) and G_2 (t_2) with varying obl values. All the other parameters are fixed. The execution time grows linearly with the number of total actions for both generators. As, G_1 generates complex problem instances, $t_1 \geq t_2$. When the total number of actions exceeds a certain threshold (170 actions), the execution time starts growing exponentially. Table 2 (a) reports execution time with varying c ($dact = 10$, $obl = 80$). The execution time increases linearly when $c < 10$ but it starts growing exponentially for $c \geq 10$. Table 2 (b) shows execution time for different *dact* values ($obl = 80$). The execution time in this experiment increases linearly with *dact*. G_1 is used to generate input instances for both the experiments.

5. RELATED WORK

Unfortunately, there is not a very large volume of related work. Although recently there have been more papers which emphasize attempting to combine security with usability, most of the work has come at this from a different angle than our paper. Namely, most papers look at increasing the usability of the software which provides the users with data security. The papers examine reasons that users might make bad security decisions, use programs in an insecure manner, or avoid using security programs. They then attempt to improve the usability of these systems or propose usability guidelines.

For example, Zurko *et al.* [19] describe a system called Adage which handles RBAC authorizations. One way in which they distinguish Adage is that they intentionally designed it to be friendly to the user and ran usability studies to evaluate the ease of use of Adage. However, in this case, the user of Adage is the person making the security decisions, which is to say the administrator, rather than an ordinary user.

Most papers about usable security focus on the person making the security decisions. In many cases, this is an administrator, but some papers examine the usability of programs which are run by ordinary users. One such is Whitten and Tygar [18], which analyzes the user interface for PGP 5.0. Still, this type of approach focusses on aiding the user in making appropriate security decisions. We, instead, focus on attempting to ensure usability when the security policy is preventing the user from doing what she would otherwise like to. We attempt to preserve usability when the system is making the security decision for the user.

Besnard and Arief [3] look at both perspectives. They examine reasons why users might use systems in an insecure manner. They cover both why users might make poor security decisions and why users might intentionally go around security systems in order to get work done. This is relevant to our motivation because if the user obligation management system is too difficult to use, users will find ways to avoid using it, thus negating any security gains. However, they are focussed on understanding user behaviors and creating guidelines for effective programming rather than any particular system.

Stoller *et al.* [17] studied the mini-ARBAC policy through experimental analysis on the reachability property. The problem they deal with is also intractable like the action failure feedback problem. They identify different restrictions (only one positive pre-condition, no hierarchy among roles, no sessions) on the problem which make solving the problem practical. By contrast, we are analyzing the mini-ARBAC policy extended to support obligations.

6. CONCLUSION

Within the context of obligation systems in which obligations can require and affect authorizations, the action failure feedback plan problem attempts to provide suggestions to users how they could accomplish desired target actions that are currently denied. Our formulation of the problem is based on user obligation systems that use mini-RBAC and mini-ARBAC as their authorization system. We show that the action failure feedback problem is PSPACE-hard and propose an AI planner-based approach to solve this problem. We have constructed a tool based on this approach and evaluated it empirically on a diverse collection of problem instances. The results demonstrate that our approach could be useful for moderate size problem instances.

7. REFERENCES

[1] A. Adams and M. A. Sasse. Users are not the enemy. *Commun. ACM*, 42(12):40–46, 1999.

[2] L. Bauer, L. F. Cranor, R. W. Reeder, M. K. Reiter, and K. V. . Real life challenges in access-control management. In *CHI '09: Proceedings of the 27th international conference on Human factors in computing systems*, pages 899–908, New York, NY, USA, 2009. ACM.

[3] D. Besnard and B. Arief. Computer security impaired by legitimate users. *Computers & Security*, 23:253–264, 2004.

[4] A. Blum and M. Furst. Fast planning through planning graph analysis. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI 95)*, pages 1636–1642, 1995.

[5] L. F. Cranor. A framework for reasoning about the human in the loop. In *UPSEC'08: Proceedings of the 1st Conference on Usability, Psychology, and Security*, pages 1–15, Berkeley, CA, USA, 2008. USENIX Association.

[6] L. F. Cranor and S. Garfinkel, editors. *Security and Usability*. O'Reilly Media, 2005.

[7] L. F. Cranor, P. Guduru, and M. Arjula. User interfaces for privacy agents. *ACM Trans. Comput.-Hum. Interact.*, 13(2):135–178, 2006.

[8] Health Resources and Services Administration. Health insurance portability and accountability act, 1996. Public Law 104-191.

[9] K. Irwin, T. Yu, and W. H. Winsborough. On the modeling and analysis of obligations. In *CCS '06: Proceedings of the 13th ACM conference on Computer and communications security*, pages 134–143, New York, NY, USA, 2006. ACM.

[10] A. Kapadia, G. Sampemane, and R. Campbell. Know why your access was denied: Regulating feedback for usable security. In *In CCS 04: Proceedings of the 11th ACM conference on Computer and communications security*, pages 52–61. ACM Press, 2004.

[11] H. A. Kautz and B. Selman. Planning as satisfiability. In *Proceedings of the Tenth European Conference on Artificial Intelligence (ECAI'92)*, pages 359–363, 1992.

[12] X. Nguyen and S. Kambhampati. Reviving partial order planning. In *IJCAI*, pages 459–466, 2001.

[13] J. S. Penberthy. Ucpop: A sound, complete, partial order planner for adl. pages 103–114. Morgan Kaufmann, 1992.

[14] M. Pontual, O. Chowdhury, W. Winsborough, T. Yu, and K. Irwin. Toward practical authorization-dependent user obligation systems. In *Proceedings of the 5th International Symposium on ACM Symposium on Information, Computer and Communications Security (ASIACCS'2010)*, pages 180–191. ACM Press, 2010.

[15] R. S. Sandhu, V. Bhamidipati, and Q. Munawer. The ARBAC97 model for role-based aministration of roles. *ACM Transactions on Information and Systems Security*, 2(1):105–135, Feb. 1999.

[16] A. Sasturkar, A. Yang, S. D. Stoller, and

C. Ramakrishnan. Policy analysis for administrative role based access control. volume 0, pages 124–138, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[17] S. D. Stoller, P. Yang, C. R. Ramakrishnan, and M. I. Gofman. Efficient policy analysis for administrative role based access control. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, pages 445–455, New York, NY, USA, 2007. ACM.

[18] A. Whitten and J. Tygar. Why Johnny Can't Encrypt: A Usability Evaluation of PGP 5.0. In *Proceedings of the 8th USENIX Security Symposium*, Monterery, CA, Aug. 1999.

[19] M. Zurko, R. Simon, and T. Sanfilippo. A User-Centered, Modular Authorization Service Built on An RBAC Foundation. In *Proceedings of IEEE Symposium on Security and Privacy*, Oakland, CA, May 1999.