On the Management of User Obligations

[Technical Report: CS-TR-2011-001]

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

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

ABSTRACT

This paper is part of a project investigating authorization systems that assign obligations to users. We are particularly interested in obligations that require authorization to be performed and that, when performed, may modify the authorization state. In this context, a user may incur an obligation she is unauthorized to perform. Prior work has introduced a property of the authorization system state that ensures users will be authorized to fulfill their obligations. We call this property *accountability* because users that fail to perform authorized obligations are accountable for their non-performance. While a reference monitor can mitigate violations of accountability, it cannot prevent them entirely. This paper presents techniques to be used by obligation system managers to restore accountability. We introduce several notions of dependence among pending obligations that must be considered in this process. We also introduce a novel notion we call *obligation pool slicing*, owing to its similarity to program slicing. An obligation pool slice identifies a set of obligations that the administrator may need to consider when applying strategies proposed here for restoring accountability. The paper also presents the system architecture of an authorization system that incorporates obligations that can require and affect authorizations.

Categories and Subject Descriptors

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

General Terms

Security, Theory

Keywords

Obligations, RBAC, Policy, Authorization Systems, Accountability

1. INTRODUCTION

Obligations are actions that principals are required to perform, often within a pre-defined time interval. The majority of previous work focuses on obligations [7, 9] that are to be performed by computer systems instead of users. Relatively few researchers [11, 15, 18] have focused on obligations that are performed by users, where they can affect and depend on the system's authorization state. A correctly functioning system has predictable behavior, while a human does not. This raises complexity in designing and analyzing a user obligation system. In particular, it makes it essential to have deadlines. One such situation arises when a user Alice is assigned an obligation to perform an action in July of the next year, but she currently does not have the proper authorization to perform the action. Unless someone gives Alice the proper authorization beforehand, her request to perform the obligatory action will be denied. The inevitability of this outcome may not be discovered until she attempts that action.

Irwin *et al.* [11] were the first to study user obligations that affect and depend on authorizations. They introduced a property of the authorization state and the obligation pool called *strong accountability* that guarantees each obligatory action will be authorized during the entire time interval associated with the obligation. The property's name, accountability, reflects the fact that it is appropriate for an organization to hold accountable those that fail to perform authorized obligations. (A related notion called "weak accountability" was also introduced. However deciding weak accountability is intractable; when we refer to accountability, we mean strong accountability.) To the extent that it can be maintained, accountability is helpful for heightening the benefits derived by assigning user obligations. These benefits include facilitating effective planning, including obtaining early warning when plans are infeasible, as well as transparency and awareness of which users are failing to fulfill their duties and the impact of such failures.

Deciding whether a system state is accountable requires reasoning about future states of the authorization system.

It is reasonable to assign to a user an obligation that the user is not currently authorized to perform, provided that other obligations will grant that authorization prior to the time the first obligation must be performed.

As we have discussed elsewhere [11, 18], a reference monitor can help maintain accountability by preventing actions that would cause it to be violated. However, even with such a reference monitor in place, accountability is still violated when an obligation is not or will not be performed. For instance, if a user fails to fulfill an obligation, say, to grant Alice the rights she needs next July, the system will become unaccountable, and Alice will be unable to perform her own obligation. Thus, an obligation system manager needs strategies and support tools that she can use to restore accountability. Three of the four present contributions seek to address these needs.

The prior work that has studied accountability [11, 12, 18, 19] focused on several technical problems. However, the architecture of a user obligation management system has never been presented. This is the **first contribution** of this paper. We give a brief overview of each of the components of the framework and discuss their interaction. A central organizing principle of the architecture is that the system should be in an accountable state as much of the time as possible without interfering unnecessarily with usability. We also provide an example illustrating the utility of this goal.

As part of supporting an obligation system manager in restoring accountability, we present three forms of dependency that can exist among obligations within a system's obligation pool. While functional dependencies also exist, here we focus exclusively on dependencies that are based on authorization requirements. The three kinds of authorization dependencies we formalize are positive dependency, negative dependency, and antagonistic dependency. This is our **second contribution**.

Borrowing a term from programming languages, we introduce what we call a *slice* of an obligation pool. A slice of a program is a subset of the statements in a program that define a portion of the program's behavior [23]. In our context, a slice is a subset of the current pool of pending obligations. We introduce two kinds of slice. One is based on positive dependency among obligations. The other is based on all three forms of dependency mentioned above. An obligation system manager who is working on restoring accountability can use a slice of the current obligation pool to identify which obligations she needs to consider modifying. As we shall see, the choice of which kind of slice to use depends on the strategy being applied to the restoration of accountability. This is our **third contribution**.

Our **final contribution** consists of several strategies that an obligation system manager can use for accountability restoration. These strategies can be supported by AI planning techniques [19] and by tools that compute the kinds of obligation pool slice discussed above.

The remainder of this paper is organized as follows. Section 2 provides background necessary to understand our contributions. Section 3 presents our architecture for managing user obligations that depend and affect authorization. Section 4 introduces the notion of authorization dependency between obligations and also the slice properties. In section 5, we present approaches that can be used by an administrator for restoring accountability. Section 6 discusses related work. Section 7 discusses future work and concludes.

2. BACKGROUND

This section reviews the obligation model that we use for studying the interaction between authorizations and obligations that depend upon and affect them. It recalls simplified versions of the RBAC model and the administrative ARBAC model that fulfill our study's need for an authorization system that supports changes in authorization state. It summarizes the notion of accountability, which is a property of the obligation system state that ensures required authorizations are available to enable the fulfillment of obligations under the assumption that users are diligent in attempting to fulfill their obligations. Based on a software development-environment scenario, an example is presented that introduces some aspects of the problem of restoring accountability when it has been violated and preventing its violation ahead of time when doing so is possible. Building on that scenario, we finally show that even the simple approach of reassigning obligations, for instance when a user leaves the company, involves solving a PSPACE-hard problem. We show how some of our prior work will be used to address this aspect as part of a solution to the larger problem that we take it on in later sections.

2.1 An Obligation Model

This section summarizes the essential elements of the obligation system used in our study [11, 18]. At any given point in time, the set of users $U \subseteq \mathcal{U}$ in the system is finite, but unbounded (which requires that $\mathcal{U}$, the universe of users, be countably infinite). We denote users by u, possibly with subscripts. Objects follow the same pattern. The set of objects in the system is given by $O \subseteq \mathcal{O}$ and individuals are ranged over by o. To accommodate administrative operations, we require that $\mathcal{U} \subseteq \mathcal{O}$. The set of supported actions is given by $\mathcal{A}$. Each $a \in \mathcal{A}$ is parameterized by the user requesting the action and zero or more objects (denoted by $\vec{o}$) to which the action will be applied. The type of a will be given later. At any given point in time, the state of the system is given by $s = \langle U, O, t, \gamma, B \rangle$, in which $t \in \mathcal{T}$ is the current time, $\gamma \in \Gamma$ is an authorization state (in the current paper we use mini-RBAC as the authorization model, defined just below); B is a pool of *pending* obligations. An obligation has the form $b = \langle u, a, \vec{o}, \text{start}, \text{end} \rangle \in \mathcal{B}$ and $B \subseteq \mathcal{B}$, where $\mathcal{B} = \mathcal{U} \times \mathcal{A} \times \mathcal{O}^* \times \mathcal{T} \times \mathcal{T}$ is the universe of obligations[1]. Times start and end delimit the *interval* during which the obligation must be performed. We use record field-selection notations such as $b.u$ to select the user of b and require that $b.\text{start} < b.\text{end}$. Note that $b.\vec{o}$ is a tuple of objects to which the action $b.a$ must be applied by $b.u$.

At each point in time, an obligation b is in one the following states: pending, fulfilled or violated. We say b is *fulfilled* if the action identified in b has been executed during b's time interval. We say b is *violated* if b was not executed in the proper time and the current time is greater than b.end. Finally, we say b is *pending* if the current time is less than b.end and b has not yet been executed.

A pending obligation can be in one of the two states, available or unavailable. We say b is *available* if b is pending and all the resources, authorization and users required to fulfill b are available. On the other hand, b is *unavailable* if b is pending and one of the following is not available during b's

[1] We use $\mathcal{O}^*$ to represent the Cartesian product of zero or more copies of $\mathcal{O}$.

entire time interval: authorization, user, or resources. Of principal interest to us here is guaranteeing authorization availability. (Ensuring user availability and resource availability are matters for future work.)

2.2 mini-RBAC and mini-ARBAC

The widely studied ARBAC97 model [20] 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* [21]. The member of the family that we use, supports administrative actions that modify user-role assignments, but does not consider role hierarchies, sessions, mutual exclusion of roles, changes to permission-role assignments, or role administration operations. Because we are combining our meta-model with the mini-RBAC authorization state, some redundancy are shared between the models, more specifically the set of users U is the same for both models.

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). *A mini-ARBAC policy is a tuple $\psi = \langle CA, CR \rangle$ in which:*

- *$CA \subseteq R \times \mathcal{C} \times R$ is a set of can_assign rules, in which $\mathcal{C}$ is the set of preconditions. A precondition is a conjunction of positive and negative role memberships, denoted by c. 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 current role memberships of the target user u_t satisfy precondition c. In this case we write $u_t \models_\gamma c$. For instance $u_t \models_\gamma r_1 \wedge \neg r_2$ if $\langle u_t, r_1 \rangle \in \gamma.UA$ and $\langle u_t, r_2 \rangle \notin \gamma.UA$.*
- *$CR \subseteq R \times \mathcal{C} \times R$ is a set of can_revoke rules. Each $\langle r_a, c, r_t \rangle \in CR$ indicates that a user in role r_a can revoke the role r_t from any target user if her current role memberships satisfy precondition c.*

For examples of mini-RBAC model and mini-ARBAC policy consult appendix D and C, respectively.

2.3 State Transitions for Obligation Model

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 $B \subseteq \mathcal{B}$ of obligations incurred (by u or by others) when the action is performed under this rule. Each action a denotes a curried function of type $(\mathcal{U} \times \mathcal{O}^*) \rightarrow (\mathcal{FP}(\mathcal{U}) \times \mathcal{FP}(\mathcal{O}) \times \Gamma) \rightarrow (\mathcal{FP}(\mathcal{U}) \times \mathcal{FP}(\mathcal{O}) \times \Gamma)^2$. So, given a user u and a tuple of objects $\vec{o}$, $a(u, \vec{o})$ is a mapping that, when applied to the user set, object set, and authorization state of the current state, returns the values of these structures in the new system state. We require that $\forall a \in \mathcal{A} \cdot \exists p \in \mathcal{P} \cdot p.a = a$. Here, u and $\vec{o}$ are variables and the objects in $\vec{o}$ are the arguments given in the invocation of a. So, for instance, if read(bookA) is attempted, then a = read and $\vec{o} = \langle \text{bookA} \rangle$. We say that a is an administrative action (denoted by $a \in$ administrative) if it has the form $\langle u, \text{grant/revoke}, \vec{o} \rangle$, otherwise a is a non-administrative action (denoted by $a \notin$ administrative).

When one obligation leads to another being incurred, we call these *cascading* obligations. The obligation model discussed in this paper prevents cascading by partitioning $\mathcal{A}$ into those actions that can be discretionary and those that can be obligatory, and statically ensuring that policy rules for the latter do not add obligations. Preliminary investigations of allowing cascading suggest that the accountability problem is likely to be computationally expensive and that accountable system states are likely to be uncommon, making systems that maintain the property unusable. More specifically, three issues make supporting cascading difficult and unproductive with our current model. (i) Different policy rules can cause different (disjunctive) obligations to be incurred, making it computationally expensive to reason about the future state of the obligation pool and authorization system. (ii) Cycles can easily be formed that introduce the likelihood of infinite sequences of new obligations being incurred as the result of a single action. (iii) In the current model, the time intervals during which new obligations are to be performed depend on the time at which the action that causes them to be incurred is performed. As obligations are scheduled times further from the current time are considered, the time intervals in which obligatory actions could occur become longer. This makes it increasingly unlikely that these obligations will be authorized throughout the entire interval in which they must be to satisfy accountability.

Clearly, cascading obligations are important in many potential deployment environments. A design meeting this requirement is a matter of future work, in which we plan to alter and/or extend the current model.

Let us now consider the transition from state s to some state s' that occurs when an event e is handled according to policy rule p. The fact that this is the transition taken is denoted by $s \xrightarrow{\langle e,p \rangle} s'$. Letting $e = (u, a, \vec{o})$, we require that $u \in s.U$ and $\vec{o} \in s.O^*$. The action $a(u, \vec{o})$ determines the values $\langle s'.U, s'.O, s'.\gamma \rangle$ based on $(s.U, s.O, s.\gamma)$. Thus, actions can introduce and remove users and objects and change the authorization state. The condition, $cond(u, \vec{o}, a)$, in the policy rule p must be satisfied for p to be used in the transition; p determines any new obligations added in obtaining $s'.B$ from $s.B$. These points are formalized in Definition 4 below. There are three cases in which a user u is authorized to perform an action a on an object tuple $\vec{o}$. (i) When a is non-administrative, authorization depends on the permis-

[2] We use $\mathcal{FP}(\mathcal{X}) = \{X \subset \mathcal{X} | X \ \textit{is finite}\}$ to denote the set of finite subsets of the given set.

sions assigned to u's roles; (ii) when a grants a role, there must be a *can_assign* rule for one of u's roles such that the target user u_t satisfies the precondition; (iii) when a revokes a role, a similar requirement holds on the existence of a *can_revoke* rule, except there is no mutual exclusion issue.

Definition 3. *For all $u \in \mathcal{U}$ and $\vec{o} \in \mathcal{O}^*$, $\gamma \models cond(u, \vec{o}, a)$ if and only if the following holds.*

$$\begin{array}{ll} & (\exists r).(((u,r) \in \gamma.UA) \wedge \\ (i) & [a \notin \text{administrative} \rightarrow (\langle r, \langle a, \vec{o}\rangle\rangle \in \gamma.PA)] \wedge \\ (ii) & (\forall u_t, r_t).[a = grant \wedge \vec{o} = \langle u_t, r_t\rangle \rightarrow \\ & \qquad (\exists c).(((\langle r, c, r_t\rangle \in \psi.CA) \wedge (u_t \models_\gamma c))] \wedge \\ (iii) & (\forall u_t, r_t).[a = revoke \wedge \vec{o} = \langle u_t, r_t\rangle \rightarrow \\ & \qquad (\exists c).(((\langle r, c, r_t\rangle \in \psi.CR) \wedge (u_t \models_\gamma c))]) \end{array}$$

The transition relation defined below preserves the invariant over states $s = \langle U, O, t, \gamma, B\rangle$ given by $\forall b \in s.B \cdot (b.u \in s.U) \wedge (b.o^* \subseteq \mathcal{O}^*)$ and $s.U = s.\gamma.U$.

Definition 4 (Transition relation). *Given any sequence of event/policy-rule pairs, $\langle e, p\rangle_{0..k}$*[3]*, 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) $s_k \xrightarrow{\langle e,p\rangle_k} s_{k+1}$ holds if and only if, letting $p_k = a(u, \vec{o}) \leftarrow cond(u, \vec{o}, a) : F_{obl}(s, u, \vec{o})$, we have $s_k.\gamma \models cond(e_k.u, e_k.\vec{o}, e_k.a)$, and $s_{k+1} = \langle U'', O'', t'', \gamma'', B''\rangle$, in which $\langle U'', O'', \gamma''\rangle = a(u, \vec{o})(s_k.U, s_k.O, s_k.\gamma)$, $B'' = (s_k.B - \{e\})$ when $e_k \in \mathcal{B}$, and $B'' = s_k.B \cup F_{obl}(s_k, e_k.u, e_k.\vec{o})$ otherwise.

(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 $s_k \xrightarrow{\langle e,p\rangle_k} s_{k+1}$.

Note that a reference monitor in our obligation system is going to require more than simply that a transition is well defined to permit an action to be performed according to a given policy rule. It will further require that performing the action and adding any obligations required by the policy rule leaves the system in an accountable state. The next section specifies the property of accountability.

2.4 Strong Accountability Property

As discussed in the introduction, even though users are capable of failing to fulfill their obligations, it is very helpful to make use of a conditional notion of correctness that says, roughly, assuming the users fulfill their obligations diligently, all users will be authorized to perform their obligations. This section formalizes that notion. This notion is called (strong) accountability [11] [18].[4]

Accountability is defined in terms of hypothetical schedules according to which the given pool of obligations could be executed, starting in a given system state. Under the assumption that each prior obligation has been fulfilled during its specified time interval, accountability requires that each obligation be authorized throughout its entire time interval, no matter when during that interval 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 $b_{0..n}$ that enumerates B, for $n = |B| - 1$. A schedule of B is *valid* if for all i and j, if $0 \leq i < j \leq n$, 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 $b_{0..n}$ 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 5 (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 $s_{k+1} \xrightarrow{\langle b,p\rangle_{k+1}} s_{k+2}$.*

Please consult appendix A for an example of a strong accountable pool of obligations.

2.5 Illustrations and Utility of Accountability

This section presents three simple examples in a software-development environment that illustrate the use of accountability in detecting that obligatory actions are unauthorized when the obligation is introduced, rather than when the action is attempted. Eve is a project manager. She uses the action *assignProjObl* to assign obligations to team members. The action *assignProjObl* takes as input the values that are placed into the new obligation[5].

Suppose that Alice's only role is that of a developer, which enables her to develop software. In scenario 1, Eve creates an obligation that requires Alice to perform black-box testing. In doing so, Eve makes the state unaccountable, as Alice does not have the requisite roles to perform this action. Thus, a reference monitor that enforces accountability prevents Eve from performing this *assignProjObl*. Without this intervention, the inadequacy of Alice's roles would be discovered only when Alice attempts to perform the testing.

Now suppose Bob has the role blackBoxTester. Assume that the organization uses mandatory vacation to help prevent insiders committing fraud [3] and that Eve is responsible for ensuring her employees adhere to this policy. In scenario 2, Bob is required to take mandatory vacation in July. Eve adds obligations that Joan, in the role securityManager, remove Bob's roles while Bob is on vacation and restore them when he returns. Now, if Paul, another projectManager, tries to assign Bob an obligation to perform black-box testing in July, Paul will be prevented from doing so because Bob will not have the necessary roles that time.

[3] 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 $\langle e_0, p_0\rangle, \langle e_1, p_1\rangle, \ldots, \langle e_j, p_j\rangle$. We use "[" and "]" as an alternate form of parenthesis to aid the eye in recognizing the formula's syntactic structure

[4] A notion of *weak* accountability has also been proposed [11]. However, deciding weak accountability in general is co-NP-complete, so we do not discuss it here, as the current paper focuses on practical issues.

[5] Passing an action's parameter values into obligations that are assigned when the action is performed must be permitted only with care. Specifically, a parameter must not be permitted to define the action part of the new obligation. This is necessary for maintaining the separation of discretionary and obligatory actions and thereby preventing cascading. For example, *assignProjObl* cannot be made obligatory because it is a discretionary action. This means that *assignProjObl* must be authorized by a separate policy rule for each obligatory action that *assignProjObl* can be used to assign.

Paul discovers this when he tries to assign obligations to Bob rather than when Bob attempts to perform the task or fails to make the attempt due to being out of the office.

In scenario 3, suppose Bob already has an obligation to perform black-box testing of a software component. Should Joan attempt to revoke Bob's blackBoxTester role, she would be prevented because Bob will need it to fulfill his existing obligation. Under normal circumstances, this prevents Joan inadvertently removing a needed role. However, in some situations, such as when Bob leaves the company, Joan must be able to force the role revocation and in this case must remove or reassign Bob's obligations. Either of these courses of action could interact with other obligations already in the pool. For instance, there may be no other user to whom Bob's obligation can be assigned. Simply removing the obligation might be unacceptable, either because the action is required in its own right or because some later obligation depends on its having been performed. Sometimes there will be no satisfactory solution, as when a key employee leaves the company. However, to assist Joan in managing the situation when a solution exists, we provide the designs of tools that support finding such a solution. These designs will be presented below in section 5.

The Appendix provides tables that summarize the role assignments, *etc.*, that make up the instances of mini-RBAC (table 2) and mini-ARBAC (table 1) models used in the scenarios above. Appendix B contains the formal description of scenario 2.

2.6 Avoiding Violations of Accountability: Complexity and Planning

In the previous section, scenarios 1 and 2 led to an action being denied, while 3 led to a possible need to reassign obligations to users that may not currently be authorized to perform them. Suppose that Joan wants to give one of Bob's blackBoxTest obligations to Carl, but Carl does not currently have the roles he needs to perform it. Joan needs to find a sequence of administrative actions that can be performed to enable Carl to complete the test at the appointed time. This entails giving Carl a new role. In the simplest instance, doing so does not conflict with any other obligations already in the pool. So Joan just needs to find a sequence of administrative actions that can be performed by various administrators that results in Carl receiving the necessary role. In this case we assume that there are no administrative actions that will modify Carl's roles between the current time and the time at which the blackBoxTest obligation is to be performed.

The simple class of instances of our problem illustrated here defines a subproblem that is essentially identical to the *role reachability* problem [21]. In this problem, one seeks a sequence of grants and revokes that modifies the current role memberships of a given user to include a given role. Determining whether there is such a sequence that leads to the user having a given role is, in the unrestricted case, PSPACE-complete [21]. This is particularly daunting because RBAC systems used in practice often have hundreds of roles, thousands of users, and millions of objects [8].

The fact that the role reachability problem can be reduced trivially to this subproblem establishes the intractability of one of the techniques that we propose to use for accountability restoration, namely obligation reassignment. For dealing with a generalized variant of the role-reachability problem that arises in this context, we have developed an AI planning tool that is often able to find suitable action sequences (provided the sequences are not too long) [19]. There are several other aspects of our problem for which an administrator attempted to restore accountability will also require tool support. The design and use of such tools is taken up in section 5 below.

3. GENERAL ARCHITECTURE

This section presents our architecture for managing user obligations that can depend on and affect authorization (figure 1). We assume that obligations are triggered by two classes of events: *controllable events* and *uncontrollable events*. Controllable events are originated by actions taken by users of the local system. (*E.g.*, when Alice attends a conference, she may incur an obligation to later present a conference summary to members of her home organization. Presumably attending the conference is a controllable event.) Uncontrollable events are generated by the environment. (*E.g.*, a policy might require that when a court order arrives, a company lawyer must review and respond. Receiving a court order is an uncontrollable event.) In figure 1, arrows represent messages that can be exchanged among software components and users. The direction of the arrow indicates the direction of message flow. Let us consider the main components of the architecture.

Reference Monitor

The standard function of a reference monitor is to disallow actions that are not authorized. We augment this requirement so that the reference monitor also disallows controllable actions when the obligations that they cause to be incurred, or the action itself, would violate the accountability property of the system. The Reference Monitor is further divided into four main components.

(a) Authorization Checker. It checks standard authorization.

(b) Obligation Checker. It is responsible for denying actions that violate the accountability property of the system. The Obligation Checker can deny an action if one of the following cases occurs. (*i*) The action is administrative and can make an obligated user unable to perform her obligation. (*ii*) The action causes an obligation to be incurred that will not be authorized. (*iii*) The action introduces an administrative obligation that will make an obligated user unable to perform a subsequent obligation.

Denying an action that violates accountability is an attempt to maintain accountability incrementally as often as possible. By providing an algorithm for checking accountability and evaluating it empirically, it has been shown [18] that one can efficiently maintain accountability of a large scale system in practice.

An additional function of this component is to select or help a user to select among the policy rules that can be used to authorized the desired action (see section 2). When multiple rules preserve accountability, the appropriate rule to select may be application dependent. In some cases, it may even be appropriate to let the user requesting the action make the selection. When this is inappropriate, for example, due to performance issues, a range of policy-driven

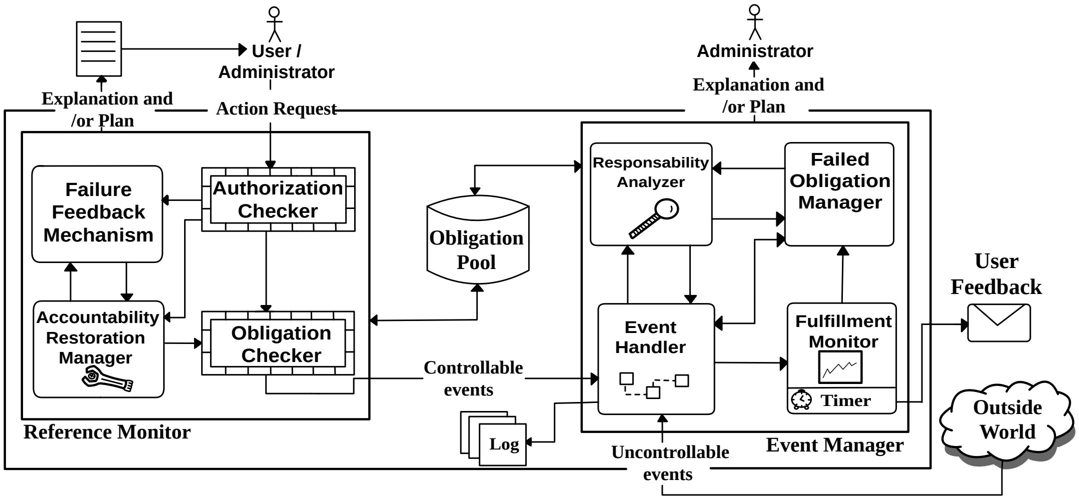

Figure 1: Obligation Architecture

alternatives are possible.

(c) Failure Feedback. When the Obligation Checker component denies an action because it would violate accountability, the failure feedback component attempts to present the user with an alternate plan of action that will enable her to accomplish her desired actions without violating accountability. The user has the option of accepting the plan, or not . If accepted, the actions in the plan become obligations. The plan can involve actions for the user herself and for others. If the plan contains actions for other users, they must also agree to the plan before the system will convert the plan into user obligations. If no plan is found, the action will be denied and the user will be notified. This problem is called the Action Failure Feedback Problem (*AFFP*). Pontual *et al.* [19] present a formal specification of the *AFFP*, complexity analysis, an AI-based approach that can encode instances of the AFFP problem as an input to a partial order AI planner, and empirical evaluation of the resulting tool.

(d) Accountability Restoration Manager. When an obligation is violated, unavailable, or some external uncontrollable event results in the violation of the system's accountability, this is detected by the failed obligation manager module (in the event manager) that will be discussed shortly in detail. When the administrator is notified of the violation, he uses the automatic tool support provided by the Accountability Restoration Manager to restore accountability. For that, we consider that the obligation pool is an object that an administrator can edit. Strategies for doing so are discussed in section 5.

Event Manager

The major responsibilities of the event manager are altering authorization state of the system according to administrative events, monitoring obligation status, and recognizing responsible users for obligation violation. The main components of it are presented next.

(a) Event Handler. It observes controllable and uncontrollable events, and is also responsible for performing administrative events, modifying the authorization state accordingly. It also adds new obligations to the obligation pool, per the policy rule requirements. All the events are also logged by the event handler.

(b) Fulfillment Monitor. It is responsible for checking whether an observed event constitutes the fulfillment of a pending obligation. It uses a timer to keep track of the obligations that are nearing their deadlines and notifies the appropriate users. Finally, it detects violated obligations and notifies the Failed Obligation Manager.

(c) Failed Obligation Manager. It is responsible for determining the violated and unavailable obligations. After that, it determines all the obligations that are affected by them. (See section 5.)

(d) Responsibility Analyzer. It is important for organizational managers to be aware of which employees are diligent and which are not. This component provides this information.

Assuming the system was initially in an accountable state, any single obligation violation is the responsibility of the user charged with fulfilling it. This also holds when multiple obligations are violated concurrently. When an administrative obligation is violated other obligations may inevitably also be violated due to lack of permissions.

This component may not be run immediately each time an obligation goes unfulfilled. Consequently, it is possible that this component is presented with multiple obligations that have been violated, some of which were supposed to be performed by users that had inadequate permissions. [6] In this case, the user that is ultimately responsible for a violation is the one that had the permissions required to

[6] The situation is actually slightly more complicated when the obligatory action is to grant a new role to a user. For instance, negative preconditions can prevent the user from being granted the new role. While this is an important case, we avoid discussing the intricacies of it here as it has been treated elsewhere [12]. Instead we focus simply on the permissions of the user responsible for initiating the action.

perform the administrative obligation to grant the missing permission. This module determines which users are responsible for causing each violation. Identifying users that are ultimately responsible for violations is known as the *blame assignment problem*, which is solved by this module.

Irwin *et al.* [12] provide a general approach for recognizing users responsible for accountability violation. However, they do not provide appropriate treatment to obligations incurred by unavailable users. We present techniques such as, reassignment of obligations, removal of obligations *etc.*, that an administrator can use to manage such obligations.

The architecture presented above is our vision for managing user obligations as part of the system's security policy. As mentioned at various points in the discussion above, several parts of this architecture (Obligation Checker, Failure Feedback, and Responsibility Analyzer) have been designed, implemented, and empirically evaluated elsewhere. The remaining components are subjects of on-going and future work. The current paper focuses on the Accountability Restoration Manager.

Restoring accountability is a complex problem. It requires consideration of characteristics of obligations such as their importance, purpose, and level of urgency. The capabilities of individual users that might be candidates for assuming obligations previously assigned to others must also be considered. For these reasons, a fully operational, deployed obligation system must include human actors to handle or help with accountability restoration. Thus the Accountability Restoration Manager includes a human, probably the same human as is generally responsible for obligation system management. It is on the techniques and tools used by this individual to restore accountability that the current paper focuses.

4. OBLIGATION DEPENDENCIES

Our system uses a reference monitor that attempts to maintain accountability by denying action requests that would violate it. Accountability can be violated nevertheless. For instance, suppose that obligations b_1 and b_2 are scheduled so that b_1 happens first and grants necessary permissions for performing b_2. If b_1 is violated, b_2 may no longer be authorized, so the obligation pool ceases to be accountable. A similar situation arises when a manager or administrator learns ahead of time that, if nothing is done to prevent it, b_1 will become violated owing to user or resource unavailability. As the violation is anticipated before it has occurred, in this latter case it may be easier to recover gracefully than in the former. In both cases, however, methods for restoring or preserving accountability are needed.

Ultimately, when accountability is violated, a human obligation system manager will generally have to participate in its restoration. To facilitate her task, we can provide information about dependencies among obligations that are relevant to the various approaches available to her for this purpose. For instance, in the previous example, b_2 cannot be fulfilled if b_1 is violated, as b_1 provides the necessary permissions for b_2. When an obligation is or will be affected by the (eventual) violation of another obligation, we say that the former has a dependency on the latter obligation. Changes in the obligation pool that affect accountability can have their impact on other obligations indirectly. The obligation system manager is best served by being provided an aggregation of dependencies that connect multiple obligations to the source of the disruption of accountability. We call the aggregate we define for this purpose an *obligation-pool slice*. In this section, after discussing each form of dependence, we return to the notion of a slice.

Various forms of dependence arise depending on the strategy one tries to use to restore accountability. In this section, we identify three different categories of dependencies on this basis. When one plans simply to let an obligation go unfulfilled ("removing" the obligation), then one is concerned only with the impact this will have on later obligations via the authorization state. We call this *positive dependence*. (Positive dependence can also arise when the violated obligation revokes a role if the second obligation is an administrative action that requires the target user not to have the role.) When one plans to reschedule an unfulfilled obligation, it may be necessary also to reschedule later obligations that have a positive dependence on it. When this is done, the second obligation is moved later in time, possible moving it past some other obligation that modifies the authorization state in a way that interferes with the execution of the second obligation. We call this interference *negative dependence*. Finally, when one obligation is rescheduled to occur after an obligation it once preceded, the authorization state in which a third, later obligation must be executed may be different as a result. For instance, if an obligation is moved past another that would reverse a change made by the first to the authorization state, the third obligation would no longer be exposed to an authorization state in which the reversal has been applied. We call this third form of dependence *antagonistic dependence*.

The notions of dependency among obligations presented here are based only on the authorization requirements of the obligations. There could be other notions of dependency, such as functional dependencies that expresses requirements on the temporal ordering of obligations. However, we do not consider this here.

EXAMPLE 6 (OBLIGATIONS). *We use the obligation pool pictured in figure 2 to illustrate the various notions of dependencies. Each obligation has a unique identifier, b_i. The roles r_j that the obligated user requires to perform his obligation are given on the left hand side of the colon, along with the obligated user u_k. In this example u_1 executes b_4, u_2 executes b_8, u_3 executes b_{12}, and u_0 is responsible for the rest. When multiple policy rules enable the obligation, the requirements for each rule are presented in separate annotations, as illustrated by obligation b_8. When one rule requires multiple roles, they are listed in the same annotation, as illustrated by obligation b_4. For economy of space, we have chosen an example that avoids the need to represent negative preconditions. When an obligation performs an administrative action (grant or revoke), the right hand side of the colon indicates the effect of this action. When an obligation grants a role r to a user u, this is indicated by $+r(u)$ on the right hand side of the colon; when it revokes r from user u, this is indicated by $-r(u)$. (E.g., obligation b_1 revokes the role r_1 from the user u_1). In the case of non-administrative obligations, this part of the annotation is empty. The current user role assignment we consider for the example is $\gamma.UA = \{\langle u_0, r_0\rangle \, \langle u_1, \emptyset\rangle \, \langle u_2, r_0\rangle \, \langle u_3, \emptyset\rangle\}$.*

DEFINITION 7 (POSITIVE DEPENDENCY). *Given a system state $s = \langle U, O, t, \gamma, B\rangle$, a set of policy rules $\mathcal{P}$, an obligation $b \in B$ and a set of pending obligations $\hat{B} \subseteq B$,*

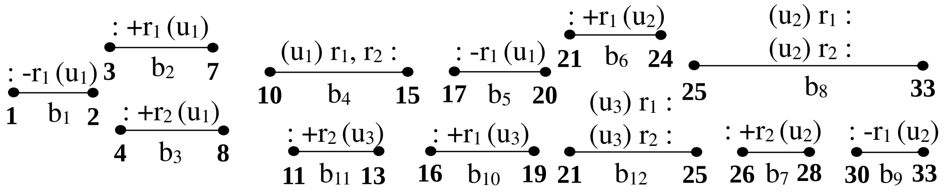

Figure 2: Dependencies among obligations

such that $(\forall \hat{b} \in \hat{B} \cdot \hat{b}.end < b.end)$. *We say* b *has a* positive dependence *on* $\hat{B}$, *denoted by* $\hat{B} \xrightarrow{+} b$, *if and only if removing* $\hat{B}$ *from* B *yields an obligation pool in which* b *is not guaranteed to be authorized during its entire time interval and* $\hat{B}$ *is a minimal set satisfying this property.*

The arrow direction signifies that $\hat{B}$ is establishing in the authorization state the necessary permissions required by b.

Example 8. *In figure 2 we have the following positive dependencies* $\{b_2\} \xrightarrow{+} b_4$, $\{b_3\} \xrightarrow{+} b_4$, $\{b_6\} \xrightarrow{+} b_8$, $\{b_7\} \xrightarrow{+} b_8$ *and* $\{b_{11}, b_{10}\} \xrightarrow{+} b_{12}$. *Here,* $\{b_2\} \xrightarrow{+} b_4$ *and* $\{b_3\} \xrightarrow{+} b_4$, *because without* b_2 *and* b_3, b_4 *cannot be performed. The same is true in the case of obligation* b_8. *Note that* b_{12} *will be authorized if one of* b_{10} *and* b_{11} *is absent, but when both of them are absent,* b_{12} *will be not be authorized. Thus,* b_{12} *does not have a positive dependence on* b_{10} *or* b_{11} *individually, but* does *have a positive dependence on the set* $\{b_{11}, b_{10}\}$.

Definition 9 (Negative Dependency). *Assume we are given a system state* $s = \langle U, O, t, \gamma, B \rangle$, *a set of policy rules* $\mathcal{P}$, *an obligation* $b \in B$ *and a set of pending obligations* $\hat{B} \subseteq B$. *We say that* b has a negative dependence on $\hat{B}$ *if* $\hat{B}$ *is a minimal set satisfying the following property. The start time of* b *is before that of each element of* $\hat{B}$ *(i.e.,* $\forall \hat{b} \in \hat{B} \cdot \hat{b}.start > b.start$*) and if* b *is rescheduled so that it starts after obligation in* $\hat{B}$, *then* b *is no longer guaranteed to be authorized throughout its entire time interval. In this case we write* $b \xleftarrow{-} \hat{B}$.

The direction of the arrow indicates that $\hat{B}$ yields an authorization state in which b may not be authorized during its entire time interval.

Example 10. *In figure 2, we have* $b_4 \xleftarrow{-} \{b_5\}$ *due to the fact that if we reschedule* b_4 *after* b_5, *it will not be authorized.*

Definition 11 (Antagonistic Dependency). *Given a system state* $s = \langle U, O, t, \gamma, B \rangle$, *a set of policy rules* $\mathcal{P}$, *and three obligations* $b_1, b_2, b_3 \in B$, *we say that* b_2 *has an* antagonistic dependence *on* b_1 *via* b_3, *denoted by* $b_2 \overset{b_3}{\text{—}} b_1$, *if inverting the order of* b_1 *and* b_2 *may result in there being a point during the interval of* b_3 *at which* b_3 *is not authorized.*

Example 12. *In figure 2, we have the following antagonistic dependencies* $(b_1 \overset{b_4}{\text{—}} b_2)$, $(b_6 \overset{b_8}{\text{—}} b_9)$ *and* $(b_7 \overset{b_8}{\text{—}} b_9)$. *Note that,* b_7 *and* b_9 *have an antagonistic dependency via* b_8 *although they consider different roles (viz.,* r_2 *and* r_1, *respectively).*

Having defined the dependence relations of interest, we now consider how to aggregate the dependencies at a higher level where they are more easily applied by the obligation system manager for the purpose to restoring accountability. The aggregation is a structure we call an *obligation-pool slice*, or simply *slice*. A slice is a subset $\hat{B}$ of a given obligation pool B. Intuitively, $\hat{B}$ consists of obligations that interact directly or indirectly with an input set of obligations $B_0 \subseteq B$ via various dependence relations relevant to the authorization requirements of obligations. The slice satisfies $B_0 \subseteq \hat{B} \subseteq B$ and is given by the closure of B_0 under some operation defined in terms of the dependence relation. The formal definition of each specific slice depends on the nature of the dependence relation used in its construction. We next provide these formal definitions, along with theorems that characterize them in terms of accountability, as needed for their use in accountability restoration.

Definition 13 (Positive Dependency Slice). *Assume we are given a system state* $s = \langle U, O, t, \gamma, B \rangle$, *a set of policy rules* $\mathcal{P}$, *and a set of pending obligations* $B_0 \subseteq B$. $PS_B(B_0)$ *is the* positive dependency slice of B with respect to B_0 *if it is given by* $PS_B(B_0) = B_p$ *in which* $B_p \subseteq B$ *is the smallest set that satisfies the following requirements:*

- $B_0 \subseteq B_p$
- $\forall b \in B \cdot (\exists \check{B} \cdot (\check{B} \subseteq B_p \wedge \check{B} \xrightarrow{+} b) \longrightarrow (b \in B_p))$

The relation $\xrightarrow{+}$ is of type $2^B \times B$ where B is the current pending pool of obligations. The intuition behind calculating the positive dependency slice is to start the slice to be the set of obligations B_v. Then, we consider all the subsets of the current slice and check whether there is any obligation that is not part of the current slice that has a positive dependency on one of those subset. If we can find such an obligation we add it to the current slice. We continue this process until the size of the slice does not grow anymore.

Please note that the above procedure is provided here solely to assist the reader's intuition regarding the slice definition. The design of an efficient algorithm remains an open problem. At minimum, it should compute the dependence relation in a lazy way. Moreover, minimality in the dependence relation will be less of an issue in a real algorithm, which will focus on constructing the slice, not on the dependence relation that defines it.

Note that this notion of slice would be useful when the administrator is considering removal of pending obligations as (part of) her strategy for restoring accountability. The following theorem shows the utility of this slice with respect the to administrators objective of leaving the obligation pool in an accountable state.

Theorem 14. *Given an accountable system state $s_0 = \langle U, O, t, \gamma, B\rangle$, a policy $\mathcal{P}$, a set of obligations $B_0 \subseteq B$, and $B_p = PS_B(B_0)$, the state given by $s = \langle U, O, t, \gamma, B \setminus B_p\rangle$ is accountable.*

Proof. Suppose for contradiction that s is not accountable. In this case, there must be an obligations $\tilde{b} \in (B \setminus B_p)$ that is not guaranteed by $B \setminus B_p$ and γ to be authorized during its entire time interval. By the assumption that s_0 is accountable, either (1) $\tilde{b}$ is authorized during its entire time interval by γ and no obligations in B modified that part of γ on which $\tilde{b}$'s authorization depends, or (2) some of the obligations in B modified the authorization state so as to make $\tilde{b}$ authorized throughout its time interval. In case (1), it is impossible to remove obligations from B with the result that $\tilde{b}$ becomes possibly unauthorized, contradicting the assumption that s is not accountable. In case (2), B_p has the property that removing it from B yields an obligation pool in which $\tilde{b}$ is not guaranteed to be authorized during its entire time interval. It follows that B_p has at least one minimal subset satisfying this property. Call it $\hat{B}$. By definition of positive dependency, $\tilde{b}$ has a positive dependence on $\hat{B}$. Now by definition of positive dependency slice, it follows that $\tilde{b} \in B_p$, giving us the desired contradiction with $\tilde{b} \in (B \setminus B_p)$. □

Definition 15 (Full Dependency Slice). *Assume we are given a system state $s = \langle U, O, t, \gamma, B\rangle$, a set of policy rules $\mathcal{P}$, and a set of obligations $B_0 \subseteq B$. The* full dependency slice *of B with respect to B_0, denoted by $FDS_B(B_0)$ is given by the smallest set B_f that satisfies the following properties:*

- $B_0 \subseteq B_f$
- $\forall b \in B \cdot (\exists \check{B} \cdot (\check{B} \subseteq B_f \wedge ((\check{B} \xrightarrow{+} b) \vee (b \xleftarrow{-} \check{B})))$ $\longrightarrow (b \in B_f))$
- $\forall b \in B \cdot (\exists b_1 \in B_f \cdot \exists b_2 \in B \cdot ((b \overset{b_2}{—} b_1) \vee b_1 \overset{b}{—} b_2)$ $\longrightarrow (b \in B_f))$

When the administrator is considering rescheduling the violated obligations and all its dependent obligations, this notion of slice would be used.

Theorem 16. *Given an accountable system state $s_0 = \langle U, O, t, \gamma, B\rangle$, a policy $\mathcal{P}$, a set of obligations $B_0 \subseteq B$, and $B_f = FDS_B(B_0)$, the state given by $s = \langle U, O, t, \gamma, B \setminus B_f\rangle$ is accountable.*

Proof. This theorem can be proved in a manner similar to that used in the proof of theorem 14. □

5. RESTORING ACCOUNTABILITY

In this section, we present several possible techniques by which an administrator can restore accountability. The selection among the techniques is application and system-requirement dependent. In practice, the administrator will use a combination of these techniques. Some obligations will, of course, be too important just to drop. Among these may be user-level obligations that do not change the authorization state. Achieving the intended changes to the authorization state might also influence the administrator's decision whether to drop obligations (including the violated ones), or instead to reschedule or reassign them.

It is important to bear in mind that restoring accountability while preserving all the desired obligations is not always possible. For instance, if an obligation with a hard deadline has been violated, this situation cannot be reversed. Furthermore, even when a solution exists, enabling us to reorganize existing obligations and add new obligations with the result that all desired obligations are fulfilled, it is not always going to be possible to find that solution in practice, as the problem is fundamentally intractable. Thus the support techniques and tools we discuss in section can at best increase the likelihood of finding a satisfactory solution. In the following we take B_0 to be the set of obligations that either have been violated or are unavailable.

Removal of Obligations.

When applying the *removal strategy*, the user removes the entire positive slice $B_p = PS_B(B_0)$ from the obligation pool B. The resulting obligation pool is accountable, as shown by theorem 14. Among the strategies for restoring accountability, this one modifies the fewest obligations, owing to the minimality of the sets in the forward dependency relation. Of course it may often be undesirable, depending on the importance of some of the obligations in the positive slice. However, sometimes there is really no alternative, since some deadlines are hard.

Example 17 (Removal of Obligations). *Using the example in figure 2, consider b_2 and b_{11} have been violated. If we use the removal strategy, we need to remove obligations b_2, b_{11}, and b_4. We have to remove b_4 as it has a positive dependency on b_2.*

Rescheduling of Obligations.

In this approach, we can take advantage of the fact that some pairs of obligations in $B_f = FDS_B(B_0)$ are independent. In particular, B_f can be partitioned into sets such that obligations from different sets are independent on one another. of obligations where obligations in different partitions are independent of each other. In this case, each partition can be rescheduled independently of one another. We denote each partition of B_f as $B_f^i \subseteq B_f$, $0 \leq i \leq |B_0|$.

Example 18 (Partitions of B_f). *Using the example in figure 2, let the current system time be 15, and that b_4 and b_{11} have been violated. Thus, $B_f = \{b_4, b_{11}, b_5\}$ creates two partitions $B_f^1 = \{b_4, b_5\}$ and $B_f^2 = \{b_{11}\}$.*

For each B_f^i, we find the set of obligations $B_0^i \subseteq B_f^i$ that have already been violated or are unavailable. We assume an obligation $\tilde{b}$ has already been violated if $\tilde{b}$.start $\leq t_c$ where t_c is the current system time. We use t_s and t_e to denote the earliest start time and the latest end time among all the obligations in B_0^i. Next, we find the obligation b_n with the earliest start time among all the obligations in $B_f^i \setminus B_0^i$. We then check whether it is possible to reschedule the obligations in B_0^i after t_c but before b_n.start ($(t_e - t_s) < (b_n.\text{start} - t_c)$). If so, we reschedule all the obligations in B_0^i after t_c keeping their original relative distance. If this is not the case, then we add b_n to the set B_0^i, and repeat the steps presented above (*i.e.*, compute t_e, and find a new b_n), until we find a time interval that is large enough to fit all the obligations in the current B_0^i. If no such intervals are found, we shift all the obligations in B_f^i so that the obligation with the earliest start time is scheduled at time $t_c + 1$ and the obligations maintain their original relative positioning. The intuition behind this approach is that all the obligations that can interact with each other will maintain their original relative positions and will be authorized.

EXAMPLE 19 (RESCHEDULING OF OBLIGATIONS). *the current time be 8 and b_2 and b_3 have been violated. If we reschedule b_2 and b_3, we need to reschedule the entire set $B_f = \{b_2, b_3, b_4, b_5\}$. The new time windows for the set could be $b_2 = [21, 25]$, $b_3 = [22, 26]$, $b_4 = [28, 33]$ and $b_5 = [35, 38]$.*

In some cases, it might not be possible to use the above approach. For instance, it may be essential that one of the obligations not be delayed. In such cases, the administrator must keep the time window of this obligation fixed, and attempt reschedule the other dependent obligations around it. However, if this is not possible, then the administrator may consider shrinking the width of some of the dependent obligations' time windows. Note that, every time the administrator tries to shrink the time window of an obligation, the reference monitor needs to check if the system is still accountable. It is up to the administrator to decide which obligations' time windows can be shrunk and by how much.

Reassignment of Obligations. In this strategy, the administrator reassigns new users to the obligations that are unavailable. When an obligation's window has already passed, this approach must be combined with rescheduling. This case is discussed below under "hybrid strategy." Along with the reassigning technique, the administrator may use an AI-planner [19] to check what other actions (*e.g.*, giving required permissions to the new users) are required in order to transfer these obligations to the new users.

EXAMPLE 20 (REASSIGNMENT OF OBLIGATIONS). *Returning again to the example in figure 2, suppose the current system time be 4 and that the user u_0 will be unavailable within the time window [10, 13]. The administrator can reassign obligations b_{10} to user u_2, since u_2 has the role r_0 required for performing b_{10}.*

Addition of Obligations. In this strategy the administrator adds new obligations in order to make the system accountable. (*E.g.*, let us consider obligation b_x needs a permission given by b_y which is scheduled before b_x. If b_y is violated, then the administrator can restore accountability by adding an obligation before b_x that grants the necessary permissions to it.) Again, the administrator can utilize the AI-planner presented in [19] to identify the new obligations she needs to add to the obligation pool to restore accountability when it has been violated.

Hybrid Strategy. Obligations can be deemed by the administrator to have different levels of importance. It may be reasonable to remove some obligations, while other must be performed according to their original schedule. Moreover, some obligations that have been violated may have had hard deadlines and cannot be rescheduled or reassigned. Thus, the administrator requires the flexibility to apply a mixture of strategies to restore accountability. We propose a *hybrid strategy* for this purpose. In it, the administrator takes an incremental approach to constructing a solution to the accountability violation. The techniques presented above are applied to different violated or unavailable obligations and different portions of their slices.

Suppose the administrator decides to divide B_0 into three subsets: obligations that must be removed (B_0^{Rem}); obligations that must be reassigned (B_0^{Rea}); and obligations that must be rescheduled (B_0^{Res}). Of course, these choices cannot be made independently of one another. For instance, it is not possible to reschedule an obligation that depends on an obligation that will be removed. On the other hand, some obligations can be both rescheduled and reassigned.

The administrator then computes the positive dependency slice of B_0^{Rem}, denoted by B_p^{Rem}, and the full dependency slice for B_0^{Res}, denoted by B_f^{Res}. As discussed in "removal of obligations", if the administrator needs to remove the obligations in B_0^{Rem}, she also has to remove the obligations in B_p^{Rem}. Moreover, for rescheduling obligations in B_0^{Res}, she also has to reschedule obligations in B_f^{Res}. If B_p^{Rem} and B_f^{Res} intersect then removing B_p^{Rem} could yield an authorization state where rescheduling obligations in $B_f^{\text{Res}} \setminus B_p^{\text{Rem}}$ would not yield an accountable system. When B_p^{Rem} either contains any non-administrative obligation that is important or contains any administrative obligation that yields an authorization state necessary for discretionary actions, then she can not also remove the set B_p^{Rem} to yield an accountable system. In such cases, she tries to find a maximal subset of B_0^{Rem}, denoted by $\hat{B}_0^{\text{Rem}}$, so that the positive slice of it has neither any intersection with the full dependency slice of $(B_0^{\text{Rem}} \setminus \hat{B}_0^{\text{Rem}}) \cup B_0^{\text{Res}}$ nor does it contain any obligations that the administrator is unwilling to remove.

If she can find such a maximal subset, she can remove the obligations in the positive slice of of $\hat{B}_0^{\text{Rem}}$. Then, she can reschedule the full dependency slice of $(B_0^{\text{Rem}} \setminus \hat{B}_0^{\text{Rem}}) \cup B_0^{\text{Res}}$. Finally, for reassigning the obligations in B_0^{Rea} she can use the approach discussed in "reassignment of obligations".

Rescheduling the full dependency slice of $(B_0^{\text{Rem}} \setminus \hat{B}_0^{\text{Rem}}) \cup B_0^{\text{Res}}$ can also introduce incompatibility. For instance, the administrator might not want to reschedule some obligations in the full dependency slice of $(B_0^{\text{Rem}} \setminus \hat{B}_0^{\text{Rem}}) \cup B_0^{\text{Res}}$ because of their urgency. We can address this problem in a manner similar to that discussed under "rescheduling of obligations".

Each time an administrator uses one of the techniques presented above to restore accountability, it is the system's responsibility to ensure that it is still in an accountable state. This can be checked using the algorithm in [18].

6. RELATED WORK

Many obligation models have been proposed, ranging from largely theoretical [7, 10, 14] to more practical [2, 5, 9, 13, 15, 22].

Minsky and Lockman [14] were the first to introduce obligations and their relationship with permissions. They proposed a very broad model that includes concepts of user obligations, refrainments, obligations triggered by events, deadline of obligations, compensatory actions for failed obligations, *etc.* The model is too abstract to be implemented in a real world system.

Nowadays, many security policy languages can specify the assignment of obligations as part of the policy. Some of these include XACML [24], EPAL [1] and Ponder [6]. XACML supports only system obligations that are triggered by events and have an immediate deadline. EPAL is somehow similar to XACML and assumes obligations are defined as actions that must be taken by the environment, and that they do not interfere with each other. Ponder, on the other hand, assumes user obligations that depend on authorization. However, obligations must be fulfilled immediately.

The model of Bettini *et al.* [4] formalizes authorization by using Datalog. Provisions and obligations are determined by the Datalog rules used in the proof of authorization. When multiple proofs of authorization exist, an algorithm selects which provisions and obligations are incurred.

Dougherty *et al.* [7] presented an abstract obligation model that relates a program execution path with obligations. The model is very expressive and can handle both positive and negative obligations. They also consider repeated obligations, penalties for violating an obligation and also stateful obligations. They perform static analysis by using Büchi automata to determine whether two obligations are contradictory or whether a run of the system fulfills a given obligation, *etc.* Thus, their analysis focuses on system obligations instead of user obligations.

Ni *et al.* [15] presented a concrete model for obligations based on PRBAC [16] that handles repeated obligations, pre and post obligations and also conditional obligations. They presented two algorithms to analyze the dominance and infinite obligation cascading properties. In their model, they do not consider the interaction of obligations with authorizations making it possible to incur obligations that are not authorized to be performed. The problems we deal in our work are inherently different than their work.

Casassa and Beato [5] provide a formal framework to enforce and specify privacy obligation policies. In their model, they allow user and system obligations. Obligations can be triggered by time or events, and they use a special kind of action called *on violation actions* for restoring the security state of the system when some user obligations are violated. Such actions allow the system to take some counter measures in order to fulfill the missed obligation. By contrast, we give the administrator some tool support in order to handle violated obligations (*e.g.*, finding the responsible user for the missed obligations, finding the obligations that can be affected by the missed obligations, and techniques to restore the accountability of the system). Differently of their work, we also check whether a user has the necessary permissions when he needs to execute an obligation.

Ali *et al.* [2] present an enforcement mechanism for obligations in service oriented architectures. Their model supports many different aspects of real obligation systems(*e.g.*, repeated obligations, conditional and pre-obligations), but do not support cascading of obligations. Although their model is more expressive than ours, they assume that obligations have all the necessary permissions.

Katt *et al.* [13] augmented UCON model [17] to support post-obligations. Their system considers two types of obligatory actions *non-trusted obligations* and *trusted obligations.* Trusted obligations are performed by the system, so they consider that they are never violated. Non-trusted obligations however are user obligations and can be violated. They proposed a mechanism that makes decisions based on the status of fulfilment of the non-trusted obligations (*e.g.*, if a client did not pay a bill, the system needs to send a email and a fine to the client). However, they do not consider interaction of authorization systems and obligations.

Hilty *et al.* [10] provide an obligation specification language (OSL) for distributed usage control. They also show how an OSL can be further translated into right expression languages that can be enforced by some existing DRM mechanisms. In contrast to our work, they consider obligations in a data containment mechanism, whereas we consider obligations and their interactions with authorization systems.

7. CONCLUSION

We have presented an architecture for managing user obligations, that affect and depend on authorization, as part of a system's security policy. We suggest to maintain accountability of a system by denying actions violating it. However, there are situations when the accountability property of a system can be violated. For this, we provide some restoration techniques that can be used by the system administrator to restore accountability. We also introduce different notions of authorization dependency among obligations. We also provide formal specification of two different notions of slice that calculates the set of obligations the administrator needs to consider when applying the different restoration techniques given a set of violated obligations.

We considered obligatory actions do not incur obligations and also assumed that the only dependencies obligations can have among them are based only on their authorization requirements. Relaxing these constraints is future work.

8. ACKNOWLEDGEMENT

Ting Yu is partially supported by NSF grant CNS-0716210. William H. Winsborough is partially supported by NSF grants CNS-0716750, CNS-0964710, and THECB ARP 010115-0037-2007. We would like to thank Dr. Lujo Bauer and the anonymous reviewers for their helpful suggestions.

9. REFERENCES

[1] Enterprise privacy authorization language (EPAL) version 1.2, Nov. 2003. http://www.zurich.ibm.com/pri/projects/epal.html.

[2] M. Ali, L. Bussard, and U. Pinsdorf. Obligation Language and Framework to Enable Privacy-Aware SOA. In *Data Privacy Management and Autonomous Spontaneous Security*, volume 5939 of *Lecture Notes in Computer Science*, pages 18–32. Springer Berlin, Heidelberg, 2010.

[3] R. J. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems.* Wiley, 2$^{\text{nd}}$ edition, 2008.

[4] C. Bettini, S. Jajodia, X. S. Wang, and D. Wijesekera. Provisions and obligations in policy rule management. *J. Netw. Syst. Manage.*, 11(3):351–372, 2003.

[5] M. Casassa and F. Beato. On Parametric Obligation Policies: Enabling Privacy-Aware Information Lifecycle Management in Enterprises. In *Policies for Distributed Systems and Networks.*, pages 51 –55, jun. 2007.

[6] D. Damianou, N. Dulay, E. Lupu, and M. Sloman. The Ponder Policy Specification Language. In *2nd International Workshop on Policies for Distributed Systems and Networks*, Bristol, UK, Jan. 2001. Springer-Verlag.

[7] D. J. Dougherty, K. Fisler, and S. Krishnamurthi. Obligations and their interaction with programs. In *Proceedings of the 12th European Symposium On*

Research In Computer Security, Dresden, Germany, September 24-26, Proceedings, pages 375–389, 2007.

[8] M. P. Gallaher, A. C. Oconnor, and B. Kropp. The Economic Impact of Role-Based Access Control, March 2002. Available at http://www.nist.gov/director/prog-ofc/report02-1.pdf.

[9] P. Gama and P. Ferreira. Obligation policies: An enforcement platform. In *6th IEEE International Workshop on Policies for Distributed Systems and Networks*, Stockholm, Sweden, June 2005. IEEE Computer Society.

[10] M. Hilty, A. Pretschner, D. Basin, C. Schaefer, and T. Walter. A policy language for distributed usage control. In J. Biskup and J. Lopez, editors, *Computer Security - ESORICS 2007*, volume 4734 of *Lecture Notes in Computer Science*, pages 531–546. Springer Berlin, Heidelberg, 2008.

[11] K. Irwin, T. Yu, and W. H. Winsborough. On the modeling and analysis of obligations. In *Proceedings of the 13th ACM conference on Computer and communications security*, pages 134–143, New York, NY, USA, 2006. ACM.

[12] K. Irwin, T. Yu, and W. H. Winsborough. Assigning responsibilities for failed obligations. In *IFIPTM Joined iTrust and PST Conference on Privacy, Trust Management and Security*, pages 327–342. Springer Boston, 2008.

[13] B. Katt, X. Zhang, R. Breu, M. Hafner, and J.-P. Seifert. A general obligation model and continuity: enhanced policy enforcement engine for usage control. In *Proceedings of the 13th ACM symposium on Access control models and technologies*, pages 123–132, New York, NY, USA, 2008. ACM.

[14] N. H. Minsky and A. D. Lockman. Ensuring integrity by adding obligations to privileges. In *Proceedings of the 8th international conference on Software engineering*, pages 92–102, Los Alamitos, CA, USA, 1985. IEEE Computer Society Press.

[15] Q. Ni, E. Bertino, and J. Lobo. An obligation model bridging access control policies and privacy policies. In *Proceedings of the 13th ACM symposium on Access control models and technologies*, pages 133–142, New York, NY, USA, 2008. ACM.

[16] Q. Ni, A. Trombetta, E. Bertino, and J. Lobo. Privacy-aware role based access control. In *Proceedings of the 12th ACM symposium on Access control models and technologies*, pages 41–50, New York, NY, USA, 2007. ACM.

[17] J. Park and R. Sandhu. The uconabc usage control model. *ACM Trans. Inf. Syst. Secur.*, 7(1):128–174, 2004.

[18] 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*, 2010.

[19] M. Pontual, K. Irwin, O. Chowdhury, W. H. Winsborough, and T. Yu. Failure feedback for user obligation systems. In *The Second IEEE International Conference on Information Privacy, Security, Risk and Trust*, pages 713 –720, 2010.

[20] 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.

[21] 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.

[22] V. Swarup, L. Seligman, and A. Rosenthal. A data sharing agreement framework. In *Information Systems Security, Second International Conference, Kolkata, India, December 19-21, Proceedings*, pages 22–36, 2006.

[23] M. Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[24] XACML TC. Oasis extensible access control markup language (xacml). *http://www.oasis-open.org/committees/xacml/*.

APPENDIX

A. ACCOUNTABILITY EXAMPLE

Example 21 (Accountability). *Let us assume that Joan has all the necessary permissions to grant developer role. On the other hand, Carl does not have yet 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.

B. OBLIGATION EXAMPLE

Example 22 (Obligation Example - Scenario 1). *In scenario 1 (Section 2.5), the project manager, Eve, performs a discretionary action that assigns an obligation to Alice. For this, she uses the action assignProjObl, which takes as arguments the elements of the new obligation, as specified by the following policy rule:*
$assignProjObl(admin, \langle oblAction, oblUser, \overrightarrow{oblObject}, oblStart, oblEnd\rangle) \leftarrow (admin \models_\gamma projectManager) : \{\langle oblUser, oblAction, \overrightarrow{oblObject}, [oblStart, oblEnd]\rangle\}$
So, when Eve creates an obligation that requires Alice to perform black-box testing, the following policy rule is created $assignProjObl(Eve, \langle test, Alice, \langle software\rangle, 01/01/2011, 02/01/2011\rangle$. *Eve satisfies* $Eve \models_\gamma projectManager$*, so the authorization system permits her to perform the action. However, the new obligation,* $\langle Alice, test, \langle software\rangle,$

[01/01/2011, 02/01/2011]⟩, *would make the system unaccountable, since Alice does not have the role of black-box tester. So Eve's attempt to add the obligation is prevented, rather than the inadequacy of Alice's roles being discovered only when Alice attempts to fulfill her obligation.*

C. MINI-ARBAC EXAMPLE

CA = {⟨securityManager, ¬ blackBoxTester, developer ⟩, ⟨securityManager, ¬ developer, blackBoxTester⟩}

CR = {⟨securityManager, True, blackBoxTester⟩}, {⟨securityManager, True, developer⟩}

Table 1: An example mini-ARBAC policy, ψ, for a software development life cycle.

D. MINI-RBAC EXAMPLE

U = {Joan, Carl, Alice, Bob, Eve}

R = {projectManager, developer, blackBoxTester, securityManager}

P = {⟨develop, sourceCode⟩, ⟨ test, software⟩, ⟨ assignProjObl, *⟩}

UA = {⟨Joan, securityManager⟩, ⟨Alice, developer⟩, ⟨Bob, blackBoxTester⟩, ⟨Eve, projectManager⟩}, ⟨Paul, projectManager⟩}

PA = {⟨developer, ⟨develop, sourceCode⟩⟩, ⟨projectManager, ⟨assignProjObl, *⟩⟩, ⟨blackBoxTester, ⟨test, software⟩⟩}

Table 2: An example mini-RBAC authorization model, γ, for a software development life cycle.