Ensuring Authorization Privileges for Cascading User Obligations

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

Murillo Pontual
The University of Texas at San Antonio
mpontual@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

Jianwei Niu
The University of Texas at San Antonio
niu@cs.utsa.edu

ABSTRACT

User obligations are actions that the human users are required to perform in some future time. These are common in many practical access control and privacy and can depend on and affect the authorization state. Consequently, a user can incur an obligation that she is not authorized to perform which may hamper the usability of a system. To mitigate this problem, previous work introduced a property of the authorization state, *accountability*, which requires that all the obligatory actions to be authorized when they are attempted. Although, existing work provides a specific and tractable decision procedure for a variation of the accountability property, it makes a simplified assumption that no *cascading obligations* may happen, *i.e.*, obligatory actions cannot further incur obligations. This is a strong assumption which reduces the expressive power of past models, and thus cannot support many obligation scenarios in practical security and privacy policies. In this work, we precisely specify the strong accountability property in the presence of cascading obligations and prove that deciding it is NP-hard. We provide for several special yet practical cases of cascading obligations (*i.e.*, repetitive, finite cascading, *etc.*) a tractable decision procedure for accountability. Our experimental results illustrate that supporting such special cases is feasible in practice.

Categories and Subject Descriptors

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

General Terms

Security, Theory

Keywords

Obligations, RBAC, Cascading Obligations, Authorization, Accountability

SACMAT'12, June 20–22, 2012, Newark, New Jersey, USA.

1. INTRODUCTION

Many access control and privacy policies contain some notion of actions that are *required* to be performed by a system or its users in some future time. Such required actions can be naturally modeled as obligations. Consider the following paraphrased regulation excerpt from section §164.524 of the Health Insurance Privacy and Accountability Act (HIPAA) [11]. A covered entity must respond to a request for access no later than 30 days after receipt of the request of the patient. As we can see from the regulation, the action of the covered entity is required when he receives a request from the patient. When we use obligations to capture this notion of required actions, we need a proper framework and mechanisms by which obligations can be managed efficiently.

The notion of obligations is not new. Several researchers [3, 4,6,13,15–17,19,25,26] have proposed frameworks for modeling and managing obligations. The majority of the existing work [3,4,6,15,19,25,26] focuses on policy specification languages for obligations rather than efficient management of obligations [3,7,10,12,14,18,21]. Even for works on the management of obligations, they mainly consider *system obligations*. Our goal is to address technical issues for efficient management of *user obligations*. A user (resp., system) obligation is an action that is to be carried out by a user (resp., the system) in some future time. Managing user obligations is challenging as system obligations can be assumed to be always fulfilled whereas this is often not the case for user obligations. More generally, we consider user obligations that can require authorization and can also alter the authorization state of the system. As a user obligation is an action, it is subjected to the authorization requirements imposed by the security policy of the system. We also consider that each of the user obligations has a time interval (*e.g.*, 30 days, *etc.*) which represents the alloted time window at which the obligation should be performed. Such intervals help detect obligation violation.

When managing user obligations that depend on and can affect authorization, we have to consider the case in which users can incur obligations that they are not authorized to perform. Otherwise, when an obligation goes unfulfilled, it is difficult to know if it is due to insufficient authorization or lack of diligence from the user. When it is ensured that all the obligatory actions are authorized, any obligation violation will only be caused due to user negligence. Irwin *et al.* [12] introduce a property of the authorization state and the current obligation pool, *accountability*, that tries to ensure that all the obligatory actions are authorized in some part of their stipulated time interval. They consider two variations of the accountability property (*i.e.*, *strong accountability* and *weak accountability*) based on when in the time intervals the obligatory actions should be authorized.

Irwin *et al.* [12] propose to maintain the accountability property as an invariant of the system. They propose to use the reference monitor of the system for maintaining accountability by denying actions that violate accountability. Extending the work of Irwin *et al.* [12], Pontual *et al.* [20] show that for an obligation system using mini-RBAC [23,24] and mini-ARBAC [23,24] as its authorization model, strong accountability can be decided in polynomial time whereas deciding weak accountability is co-NP complete. They also provide empirical evaluations for showing that a reference monitor can maintain the strong accountability property efficiently. They partition possible actions into two disjoint sets, *discretionary* and *obligatory* and only allow discretionary actions to incur further obligations. By doing this, they disallow *cascading obligations*.

The assumption of disallowing cascading obligations is restrictive. It significantly reduces the expressive power of the obligation model they use. For instance, consider the following scenario. When a sales assistant submits a purchase order, the clerk incurs an obligation to issue a check in the amount identified in the purchase order. As soon as the clerk issues the check, the manager incurs an obligation that requires him to check the consistency of the purchase order. If the purchase order is consistent and the manager approves it, then the accountant incurs another obligation to approve the check. Now, this situation can be easily modeled with cascading obligations, but it cannot be modeled by the obligation model of Pontual *et al.* [20]. Thus, one of the principal goals of this paper is to provide a concrete model in which the policy writers can specify cascading obligations easily. Furthermore, we also present a decision procedure which can be used to decide the strong accountability[1] property efficiently for special but practical cases of cascading obligations in the model.

The abstract obligation model that Irwin *et al.* [12] and Pontual *et al.* [20] use, allows specification of cascading obligations. However, their concrete model does not support the specification of cascading obligations. We adopt the concrete model of Pontual *et al.* [20] that uses mini-RBAC and mini-ARBAC as its authorization model and augment it in a way that cascading obligations can be specified. Furthermore, existing work [12, 20] does not discuss how to specify the user (obligatee) who incurs the new obligation when a user takes an action (obligatory or discretionary). We present several proposals for specifying the obligatee in a policy. The enhancement to the obligation model and proposals for obligatee selection comprise our ***first contribution***.

The specification for strong accountability presented by Pontual *et al.* [20] also takes advantage of the assumption that cascading obligations are not allowed. Our ***second contribution*** is to precisely specify the strong accountability property in presence of cascading obligations. There are two possible interpretations of strong accountability when considering cascading obligations. We define both interpretations, *existential* and *universal*, and give motivations for choosing the existential interpretation.

Our ***third contribution*** is to present a theorem which states that deciding accountability in presence of cascading obligations is in general NP-hard. We then consider several special cases which makes the problem tractable. We then provide a polynomial time algorithm (polynomial in the size of the policy, the size of the current obligation pool, and the new obligations to be considered) that can decide strong accountability for special cases of cascading obligations. This is our ***fourth contribution***.

We then present empirical evaluations of the accountability decision procedure allowing special cases of cascading obligations. Our empirical evaluations show that strong accountability can be efficiently decided for these special cases of cascading obligations. This is our ***final contribution***.

Section 2 reviews the background materials. Section 3 discusses the necessary enhancement of the obligation model to specify cascading obligations. Our main technical contribution is presented in section 4. It presents the refined definition of strong accountability, the complexity of deciding strong accountability, special cases that make deciding accountability feasible, and an algorithm for deciding strong accountability under these assumptions. Section 5 explains our input instance generation and presents empirical evaluation results. Related works are discussed in section 6. Section 7 discusses our future work and concludes.

2. BACKGROUND

In this section, we first summarize the restricted variation of the role-based access control (RBAC) and administrative role-based access control (ARBAC) model, mini-RBAC [23, 24] and mini-ARBAC [23, 24], respectively. We then discuss the obligation model presented by Pontual *et al.* [20] that uses mini-RBAC and mini-ARBAC as its authorization model.

2.1 mini-RBAC and mini-ARBAC

In the context of studying the role reachability problem, Sasturkar *et al.* [23] introduced mini-RBAC and mini-ARBAC which are simplified variations of the widely used RBAC [9] and ARBAC97 [22] model, respectively. The variation of mini-RBAC and mini-ARBAC model we use excludes sessions, role hierarchies, static mutual exclusion of roles, conditional revocation, changes to the permission-role assignment, and role administration operations. We use mini-RBAC and mini-ARBAC due its similarities with the widely popular RBAC and ARBAC model. We refer interested reader to [23, 24] for a more detailed presentation.

Definition 1 (mini-RBAC model). *A mini-RBAC model γ is a tuple $\langle U, R, P, UA, PA\rangle$ in which $U, R,$ and P represents the finite set of users, roles, and permissions, respectively. Each element of P is a pair $\langle a, o\rangle$ where a represents an action and o denotes an object. The formal type of a and o will be given later. $UA \subseteq U \times R$, denotes the set of user-role assignment and $PA \subseteq R \times P$, denotes the set of permission-role assignment.*

Definition 2 (mini-ARBAC policy). *A mini-ARBAC policy Φ is a pair $\langle CA, CR\rangle$ in which CA and CR denotes the set of can_assign and can_revoke rules, respectively. The following is the formal type of $CA \subseteq R \times C \times R$ in which*

[1]We only consider strong accountability in this work due to the complexity results of weak accountability.

C represents the set of all possible pre-conditions. Each can_assign rule $\langle r_a, c, r_t\rangle \in CA$ specifies that a user in role r_a is authorized to grant a target user the target role r_t provided that the target user satisfies the pre-condition c. A pre-condition c is a conjunction of positive and negative role memberships. The formal type of CR is $CR \subseteq R \times R$. Each $\langle r_a, r_t\rangle \in CR$ represents that a user in role r_a can revoke the r_t role from another user.

2.2 Obligation Model

We now summarize the obligation model proposed by Pontual *et al.* [20]. Note that, we augment this model for supporting cascading obligations in section 3. We use $U \subseteq \mathcal{U}$ to denote the finite set of users in the system at any given point of time. We use u possibly with subscripts to represents users. The finite set of objects in the system is denoted by $O \subseteq \mathcal{O}$. We use o with possibly subscripts to range over the elements of O. Note that, the universes $\mathcal{U}$ and $\mathcal{O}$ are countably infinite as we want to model systems of finite but unbounded sizes. For supporting administrative actions, we have $U \subseteq O$. The set of possible actions in the system is given by $\mathcal{A}$. The formal type of $\mathcal{A}$ is given below.

We denote a system state with $s = \langle U, O, t, \gamma, B\rangle$ where $t \in \mathcal{T}$ denotes the current system time, $\gamma \in \Gamma$[2] represents the mini-RBAC authorization state, and $B \subseteq \mathcal{B}$ represents the current pool of obligations. Obligations in the system has the form $b = \langle u, a, \vec{o}, t_s, t_e\rangle$, the universe of which, $\mathcal{B}$ has the formal type $\mathcal{U} \times \mathcal{A} \times \mathcal{O}^* \times \mathcal{T} \times \mathcal{T}$ [3]. For an obligation $b = \langle u, a, \vec{o}, t_s, t_e\rangle$, $[t_s, t_e]$ denotes the interval in which the obligation should be performed. Moreover, we require that $t_s < t_e$. We use $b.u$, $b.a$, and $b.o^*$, respectively, to denote the user, the actions, and the object(s) of the obligation b.

We consider two types of actions, namely, *discretionary* and *obligatory*. The system views them uniformly as events. The universe of discretionary action $\mathcal{D}$ has the formal type $\mathcal{U} \times \mathcal{A} \times \mathcal{O}^*$. Thus, the universe of all possible events is $\mathcal{E} = \mathcal{D} \cup \mathcal{B}$. Each action $a \in \mathcal{A}$ has the formal type $(\mathcal{U} \times \mathcal{O}^*) \to (\mathcal{FP}(\mathcal{U}) \times \mathcal{FP}(\mathcal{O}) \times \Gamma) \to (\mathcal{FP}(\mathcal{U}) \times \mathcal{FP}(\mathcal{O}) \times \Gamma)$[4]. Actions can add or remove users and objects and can also alter the authorization state. Thus, for a given a user u and object(s) $\vec{o}$, the action $a(u, \vec{o})$ is a mapping that that maps the current set of users, objects, and the current authorization state to a new set of users, objects, and authorization state.

Each action in our system is regulated by a fixed set of positive, policy rules $\mathcal{P}$. Each policy rule $p \in \mathcal{P}$ has the form $p = a(u, \vec{o}) \leftarrow cond(u, \vec{o}, a) : F_{obl}(s, u, \vec{o})$. This represents that a user u is authorized to perform an action a that is applied to object(s) $\vec{o}$, when the predicate $cond(u, \vec{o}, a)$ is satisfied in the current authorization state γ (denoted by $\gamma \vDash cond(u, \vec{o}, a)$) and this in turn incurs a set of obligations (possibly empty) for u or some other users. The predicate *cond* represents the authorization requirements imposed by the policy rule. $F_{obl}(s, u, \vec{o})$ is a function that takes as input s, u, and $\vec{o}$ and returns a set of obligations (possibly empty) when the policy rule p is used to authorize the action a. For a policy rule $p \in \mathcal{P}$ of form $p = a(u, \vec{o}) \leftarrow cond(u, \vec{o}, a) : F_{obl}(s, u, \vec{o})$, when $a \in \mathcal{B}$ and the $F_{obl}(s, u, \vec{o})$ is not empty, we call the obligatory action a, a *cascading obligation*. We also require that for each action, there is at least one policy rule that governs that action.

Recall that, we consider actions that can alter the authorization state of the system. Based on whether an action alters the authorization, we classify the actions in two possible categories, namely, *administrative* and *non-administrative*. An action a is called administrative (denoted by $a \in$ administrative) when it has the form $grant(u, \vec{o})$ or $revoke(u, \vec{o})$ and called non-administrative, otherwise.

Now, we turn our attention to how state transition occurs in the obligation system. We use $s \xrightarrow{\langle e,p\rangle} s'$ to denote the transition from state s to state s' when event e takes place and is authorized using the policy rule p. When e is of form $\langle u, a, \vec{o}\rangle$, we require that $u \in s.U$ and $\vec{o} \in s.O^*$. Furthermore, for each state s of the form $\langle U, O, t, \gamma, B\rangle$, the transition relation ensures that $\forall b \in s.B \cdot (b.u \in s.U) \wedge (b.\vec{o} \in s.O^*)$ and $s.U = s.\gamma.U$ holds. We first formalize what it means that the current authorization state γ satisfies the $cond(u, \vec{o}, a)$ predicate of a policy rule p (definition 3). We then formally specify the transition relation of the system.

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

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

Definition 4 (Transition relation). *Given any sequence of event/policy-rule pairs, $\langle e, p\rangle_{0..k}$[5], 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 $s_k.\gamma \vDash cond(e_k.u,\ e_k.\vec{o},\ e_k.a)$, and $s_{k+1} = \langle U'', O'', t'', \gamma'', B''\rangle$, in which $\langle U'', O'', \gamma''\rangle = a(u, \vec{o})(s_k.U, s_k.O, s_k.\gamma)$, $B'' = (s_k.B - \{e\}) \cup F_{obl}(s_k, e_k.u, e_k.\vec{o})$ when $e_k \in \mathcal{B}$, and $B'' = s_k.B \cup F_{obl}(s_k, e_k.u, e_k.\vec{o})$ otherwise. t'' denotes the system time when a is completed.

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

3. ENHANCEMENT OF THE MODEL

In this section, we extend the obligation model of Pontual *et al.* [20] to facilitate the specification and analysis of accountability in presence of cascading obligations.

3.1 Time Interval of the Incurred Obligation

In the previous obligation model [20], when a discretionary action a is taken at time t and it causes an obligation b to be incurred, the time interval of b depends on the time t. Thus, the time interval of b is calculated using a fixed offset from t and the interval size of b. Let us assume the fixed offset is δ

[2] Γ here denotes the set of abstract authorization states.

[3] $\mathcal{O}^*$ is the Cartesian product of zero or more copies of $\mathcal{O}$.

[4] $\mathcal{FP}(\mathcal{X}) = \{X \subset \mathcal{X} | X \textit{ is finite}\}$ denotes the set of finite subsets of the given set $\mathcal{X}$.

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

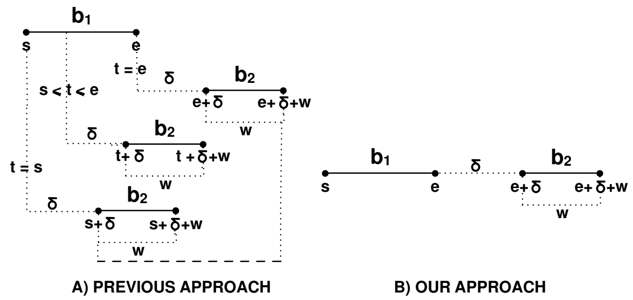

Figure 1: Time Interval of the Incurred Obligation.

and the interval size of b is w. So, the time interval of b will be $[t+\delta, t+\delta+w]$. Now, consider the case where an obligation b_1 with time interval $[s, e]$ incurs another obligation b_2. The time t at which b_1 can possibly be performed can be any value between s and e, inclusive. Thus, we have several possible intervals for b_2 considering each possible values of t (see figure 1(a)). For deciding strong accountability, we have to check whether b_2 is authorized in each of the possible time intervals. One possibility is to consider the interval $[s+\delta, e+\delta+w]$ to be the time interval of b_2, as all the possible time intervals are inside this interval. However, when b_1 is performed (we know t) we get b_2's original time interval and have to shrink the large time interval $[s+\delta, e+\delta+w]$ appropriately with respect to t. When we use this approach, it will yield runtime overhead for managing obligations and accountability will be less likely to hold due to increasingly large obligation time intervals.

To mitigate this problem, we assume that b_2's time interval will be at a fixed distance $\delta \in \mathbb{N}$ from the time interval of b_1 (see figure 1(b)). We assume δ is measured from the end time of b_1's interval. Thus, b_2's stipulated time interval in our approach will be $[e+\delta, e+\delta+w]$. This approach will ensure that the cascading obligation's time interval is fixed. For a discretionary action incurring an obligation, we replace e with t, the time at which the action is performed. Thus, in our model obligations have the form $b = \langle u, a, \vec{o}, t_s, t_e, \delta, w \rangle$. In section 4.4.2, we further augment our obligations to contain one additional field (repetition). We also extend the notion of the transition relation to allow cascading obligations.

3.2 Selection of Obligatee

We now present some strategies by which a user who incurs obligations, can be specified in the policy. When a user executes an action, this can generate other obligations to the user who initiated the action, or for other users. The user who incurs an obligation is called an *obligatee*. Existing work [12, 20] does not discuss how obligatees are specified in the policy. To allow the specification of obligatees, we extend the policy rules to include an extra field called "obligatee". Thus, policies now have the following form: $p = a(u, \vec{o}) \leftarrow cond(u, \vec{o}, a) : F_{obl}(obligatee, s, u, \vec{o}, \delta, w)$. Note that, F_{obl} function returns a set of obligations and is guaranteed to terminate in a constant time.

Explicit User: In this strategy, the obligatee is hard-coded in the policy rule.

EXAMPLE 5 (EXPLICIT USER). *Let us consider the following policy rule,* $p_0 : check(u, log) \leftarrow (u \in manager) : F_{obl}(Bob, s, u, log, \delta = 10, w = 5)$. *This rule authorizes a user in the role of manager to check the log and it will incur an obligation*[6] *for Bob.*

Self, Target, and Explicit User: In this strategy, the obligatee field can contain "Self", "Target", or an explicit user. When a policy rule's obligatee field contains "Self", it represents that the user who initiates the action, authorized by the current policy, will incur the associated obligations.

EXAMPLE 6 (SELF). *Let us consider a policy rule,* $p_1 : grant(u, \langle u_t, programmer\rangle) \leftarrow ((u \in manager) \wedge (u_t \in employee)) : F_{obl}(Self, s, u, \langle u_t, programmer\rangle, \delta = 10, w = 5)$. *This rule authorizes a user u in the role of manager to grant a new role programmer to a target user u_t in the role of employee and this will incur an obligation for u. Let us consider that manager Bob grants the employee Alice the role programmer. This will generate a new obligation for Bob.*

On the other hand, whenever the policy rule is authorizing an administrative action and the obligatee field of that policy rule contains "Target", it signifies that the target of the original administrative action authorized by this policy would incur the obligations specified by it.

EXAMPLE 7 (TARGET). *Let us consider a policy rule,* $p_2 : grant(u, \langle u_t, programmer\rangle) \leftarrow ((u \in manager) \wedge (u_t \in employee)) : F_{obl}(Target, s, u, \langle u_t, programmer\rangle, \delta = 10, w = 5)$. *The policy rule p_2 is similar to p_1 except it incurs an obligation for the target. As in the previous example, when Bob grants the role programmer to Alice, Alice will incur an obligation as she is the target of the action.*

Role Expression: In this approach, the obligatee field can contain a boolean role expression. Each literal in the boolean expression is either a positive or a negative role membership test. The system can select a user to be the obligatee provided that the user satisfies the role expression when the original action is performed. A comprehensive example of this strategy is presented in appendix A.

In the current work, we use the "Self, Target, and Explicit User" scheme to specify the obligatee. Although this approach is not the most general strategy to specify an obligatee, our accountability decision procedure requires every obligation to have an individual user statically associated with it. However, in the "Role expression" scheme multiple users can satisfy the role expression specified in the obligation policy rule. Thus, we have two possible interpretations of strong accountability. One of which says that the newly incurred obligation will maintain accountability if at least one of the users satisfying the role expression is authorized to perform the obligation during its whole time interval. The other interpretation requires that every user who satisfies the role expression must be authorized to perform the obligation during its whole time interval. Although both of the interpretations have practical utility, the choice of interpretation will influence the time complexity of the accountability decision procedure. We leave the adoption of the role expression scheme for specifying the obligatee as a future work.

4. STRONG ACCOUNTABILITY

When considering user obligations that depend on and affect authorization, we can have a situation where a user can

[6]The action associated with the obligation will be specified in the body of the F_{obl} function. For clarity, we do not show the body of the F_{obl} function.

incur obligations which she is not authorized to fulfill. However, without any preemptive approach, the obligatee will realize the absence of proper authorization in the time she attempts the obligation. This can hinder the proper functioning of the system. To mitigate this, Irwin *et al.* [12] introduced a property of the authorization state and the current obligation pool, accountability, that ensures that all the obligatory actions are authorized in some part of their time interval. Based on when they are supposed to be authorized in their time intervals, they introduced two variations of the accountability property, weak and strong. Pontual *et al.* [20] have shown that deciding weak accountability is co-NP complete for a model using mini-RBAC and mini-ARBAC, whereas deciding strong accountability is polynomial. Due to its high complexity, we do not consider weak accountability. Roughly, strong accountability requires that as long as prior obligations have been performed in their stipulated time interval, each obligatory action must be authorized no matter what policy rules are used to authorize the other obligations and no matter when they are performed in their time interval.

In this section, we first present the definition of strong accountability presented by Pontual *et al.* [20]. As mentioned before, their definition of strong accountability does not take into account cascading obligations. We call their notion of the property *restricted strong accountability.* We then refine their notion of the property and give a recursive definition of it considering the presence of cascading obligations. We go on to show that deciding strong accountability in presence of cascading obligations in general is NP-hard. We then consider some special cases of cascading obligations and give a tractable decision procedure for deciding strong accountability in their presence.

4.1 Restricted Strong Accountability

Roughly stated, under the assumption that all previous obligations have been fulfilled in their time interval, strong accountability property 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 pool of obligations B, a *schedule* of B is a sequence $b_{0..n}$ that enumerates B, for $n = |B| - 1$ (including the possibility that B may be countably infinite). 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 8 (Restricted Strong accountability). *Given a state $s_0 \in \mathcal{S}$ and a policy $\mathcal{P}$, we say that s_0 is* strongly accountable *(denoted by $RStrongAccountable(s_0, \mathcal{P})$) 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 $s_{k+1} \xrightarrow{\langle b,p\rangle_{k+1}} s_{k+2}$.*

4.2 Unrestricted Strong Accountability

In this section, we provide a formal specification of the strongly accountability property with cascading obligations. The strong accountability definition presented by Pontual *et al.* [20] disallowed cascading obligation. We extend their model to allow them. We define three auxiliary functions that will be used in the definition of strong accountability.

Definition 9 (Ψ function). *Ψ is a function that takes as input an obligation $\hat{b}$ and a fixed set of policy rules $\mathcal{P}$ and returns a set of sets of obligations $\hat{B}$ in which each element represents a set of obligations that $\hat{b}$ can incur according to the F_{obl} function of a policy rule authorizing it. The formal specification and the type of Ψ are precisely shown below.*

$$\Psi : B \times \mathcal{FP}(\mathcal{P}) \to \mathcal{FP}(\mathcal{FP}(B))$$
$$\Psi(b = (u, a, \vec{o}, t_s, t_e, \delta, w), \mathcal{P}) =$$
$$\Big\{ F_{obl}(\mathit{obligatee}, s, u, \vec{o}, \delta, w) \;\Big|\; p = (a(u, \vec{o}) \leftarrow \mathit{cond}(u, \vec{o}, a) :$$
$$F_{obl}(\mathit{obligatee}, s, u, \vec{o}, \delta, w)) \wedge (p \in \mathcal{P}) \Big\}$$

Definition 10 (Π function). *Π is a function that takes as input a set of obligations $\bar{B}$ and a fixed set of policy rules $\mathcal{P}$ and returns a set of sets of obligations $\tilde{B}$ in which each element is a possible set of obligations that all the obligations of $\bar{B}$ can incur. In short, $\tilde{B}$ is the set containing all possible combination of obligations that $\bar{B}$ can incur. The formal specification of Π and its type are shown below.*

$$\Pi : \mathcal{FP}(B) \times \mathcal{FP}(\mathcal{P}) \to \mathcal{FP}(\mathcal{FP}(B))$$
$$\Pi(b_{1\ldots n} = (u_{1\ldots n}, a_{1\ldots n}, \vec{o}_{1\ldots n}, t_{s_{1\ldots n}}, t_{e_{1\ldots n}}, \delta_{1\ldots n}, w_{1\ldots n}), \mathcal{P})$$
$$= \{\mathcal{B} \subseteq \mathcal{B} | \forall i \in 1 \ldots n. \Psi(b_i, \mathcal{P}) \neq \varnothing \to \exists f \in \Psi(b_i, \mathcal{P}). f \subseteq BB\}$$

Definition 11 (Ξ function). *Ξ is a function that takes as input a set of sets of obligations and a set of policy rules and applies Π to each of set of obligations and then combines the results. This allows us to find the set of all possible sets of obligations generated by a given set of possible obligations. For simplicity in later definitions, we also include in the output sets, the original sets which generated those obligations.*

$$\Xi : \mathcal{FP}(\mathcal{FP}(B)) \times \mathcal{FP}(\mathcal{P}) \to \mathcal{FP}(\mathcal{FP}(B))$$
$$\Xi(\tilde{B}, \mathcal{P}) = \{\forall \bar{B} \in \tilde{B}, \bigcup_{B \in \Pi(\bar{B})} B \cup \bar{B}\}$$

Note that, each action a in our system can be authorized by multiple policy rules. Each of the policy rules authorizing a can incur different obligations. Furthermore, it can be the case that among different possible obligations incurred due to a, some of them maintain accountability and some of them do not. Provided that the policy allows infinite cascading obligations and a is authorized by multiple policy rules, each of which incurs different obligations, then all possible obligations incurred due to a can be modeled as a tree (possibly infinite). Based on this, we can have two interpretations of strong accountability, *existential* and *universal.* The existential interpretation requires that there exists a single path in the tree in which all the obligatory actions maintain accountability when added to the current pool of obligations. The universal interpretation is the dual and requires that all the paths in the tree maintain strong accountability. We think the universal interpretation is too strong. As a result of which, we use the existential interpretation of the strong accountability property and define it just below. However, the following definition can be extended to express the universal interpretation of strong accountability.

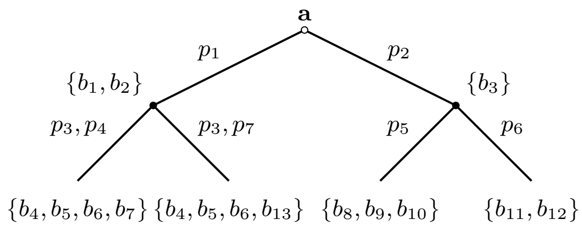

Figure 2: Possible obligations incurred by action a

In the example (in figure 2), let us consider the current accountable pool of obligations is B. We want to know whether performing a would maintain accountability. Let us consider that a can be authorized by policy rule p_1 or p_2. When a is authorized using p_1, it incurs obligations b_1 and b_2. However, when p_2 is used to authorize a, it incurs obligation b_3. Then, b_1 can be authorized by p_3 and b_2 can be authorized by either p_4 or p_7 and so on. In the existential interpretation, if one of the following sets is accountable then adding a would maintain accountability: $B \cup \{b_1, b_2, b_4, b_5, b_6, b_7\}$, $B \cup \{b_1, b_2, b_4, b_5, b_6, b_{13}\}$, $B \cup \{b_3, b_8, b_9, b_{10}\}$, and $B \cup \{b_3, b_{11}, b_{12}\}$. The universal interpretation requires all the above sets to be accountable.

In order to formalize this, we need to first define the set of possible future sets of obligations using the Ξ function. In particular, we wish to define a series of sets of sets of obligations. We will define $\Xi^0(\tilde{B}) = \tilde{B}$ and for all integers $i > 0$, we define $\Xi^i(\tilde{B}) = \Xi(\Xi^{i-1}(\tilde{B}))$. Further, we define $\Xi^\infty = \lim_{i\to\infty} \Xi^i$. Thus, given a starting set of obligations B, we can define the set of all sets of possible obligations which can arise from B as $\Xi^\infty(\{B\})$. Note that this is a countable, but possibly infinite, set of countable, but possibly infinite, sets of obligations. Because we are allowing for potentially infinitely cascading obligations, this is necessary.

Definition 12 (Strong accountability). *Given a state $s_1 \in \mathcal{S}$, in which $s_1.B$ is a strongly accountable pool of obligations, a policy $\mathcal{P}$, a set of new obligations B_c that can generate cascading obligations, we say that the state s (where $s.B := s_1.B \cup B_c$) is existentially strongly accountable (denoted by StrongAccountable$(s, \mathcal{P})$) if and only if*

$$\exists B_c' \in \Xi^\infty(s.B, \mathcal{P}).RStrongAccountable(s[B := B_c' \cup s.B], \mathcal{P})$$

4.3 Computational Complexity

This section discusses the computational complexity of deciding strong accountability in presence of cascading obligations. Prior work [12,20] disallowed cascading obligations while deciding strong accountability. They give intuitive discussions about why deciding strong accountability in presence of cascading obligation is difficult. However, they did not present any theoretical results regarding this.

We have the following theorem which states that the strong accountability decision problem is NP-hard. We reduced the Hamiltonian path problem for graphs to the decision of unrestricted strong accountability property. We present a detailed proof of the theorem in the technical report [5].

Theorem 13. *Given a strongly accountable pool of obligations B, a new obligation b, an initial authorization state γ, and a mini-ARBAC policy Φ that allows cascading obligations and also allows each action to be authorized by multiple policy rules, deciding whether $B \cup B_c \cup \{b\}$ is strongly accountable (either in existential or universal interpretation) is NP-hard in the size of B, γ, and Φ, where B_c is the set of cascading obligations incurred by b.*

4.4 Special Cases of Cascading Obligation

As in section 4.3, deciding accountability in presence of cascading obligations is NP-hard. Our goal is to find certain special cases of cascading obligations for which the accountability decision is tractable. This section introduces two such special cases.

4.4.1 Finite Cascading Obligation

In this special case of cascading obligation, we consider that the policy is written in a way that the maximum number of new obligations incurred by a single obligation is bounded by a constant (see appendix A). Furthermore, we also consider that each action, object pair is authorized by only one policy rule. We also assume that the policy rules are free of cycles prohibiting infinite cascading. This can be achieved by a static checking for cycles in the policy.

4.4.2 Repetitive Obligation

Repetitive obligations occur recurrently after a fixed amount of time. A real life example of repetitive obligation can be found in the chapter 6803(a) of Gramm-Leach-Bliley Act (GLBA) [1]. According to the regulation, a financial institution must send a customer an annual privacy notice as long as the individual is a customer. Note that, we cannot specify repetitive obligations in our model directly. For this, we follow Ni *et al.* [18] to augment our obligations with an extra field that specifies the number of repetition (denoted by ρ). We allow both finite and infinite repetitive obligation. Now, let us consider an obligation $b = \{u, a, \vec{o}, t_s, t_e, \delta, \rho, w\rangle\}$. This obligation is considered to be infinite repetitive when $\rho = I$ or finite repetitive when $\rho \in \mathbb{N}$ and $\rho > 1$.

Finite Repetitive Obligations. This kind of obligation recurs finitely after a fixed amount of time. For instance, $b = \{Bob, check, log, t_s = 5, t_e = 8, \delta = 2, \rho = 3, w = 3\}$ will generate 3 obligations $\{Bob, check, log, 5, 8\}$, $\{Bob, check, log, 10, 13\}$, and $\{Bob, check, log, 15, 18\}$.

Infinite Repetitive Obligations. This kind of obligations on the other hand recurs indefinitely. For example, $b = \{Bob, check, log, t_s = 5, t_e = 8, \delta = 2, \rho = I, w = 3\}$ will generate the following infinite number of obligations: $\{Bob, check, log, 5, 8\}$, $\{Bob, check, log, 10, 13\}$, $\cdots$.

4.5 Algorithm

As deciding accountability in presence of cascading obligations is NP-hard, we simplify our accountability decision problem by imposing several restrictions on the problem. The restrictions are: (1) We consider each action, object pair is authorized by one policy rule, prohibiting disjunctive choices. (2) We require that the policy is free of cycles which prohibits obligations which incur an infinite number of new obligations. (3) We disallow role expressions to specify the obligatee of the new obligation. (4) We also disallow finite cascading obligations which incur repetitive obligations. (5) We also disallow repetitive obligations which incur nonrepetitive cascading obligations.

Under restrictions, strong accountability can be decided in polynomial time of the policy size, number of obligations, and the number of new obligations that need to be considered. The algorithm (algorithm 1) decides whether adding an obligation to an accountable pool of obligations maintains accountability. The algorithm takes as input the accountable pool of obligations B (containing the finite cascading, finite repetitive, and infinite repetitive obligations), the current authorization state γ of the system, a mini-ARBAC policy Φ, and the new obligation b. It returns true when adding $B \cup \{b\} \cup B_c$ is strongly accountable where B_c is the new set of obligations incurred by b. Note that, the time complexity of the algorithm additionally depends on the type of the obligation to be added and the number of infinite repetitive obligations that need to be unrolled. The complexity of the algorithm is precisely described in appendix C.

In the algorithm 1, the new obligation b can either incur no new obligations, finite cascading obligations, finite repetitive obligations, or infinite repetitive obligations. Based on what kind of new obligation(s) b incurs, we have to take different course of actions. The main idea behind the algorithm is to unroll a finite amount of new obligations and use the non-incremental algorithm presented by Pontual *et al.* [20] to decide whether the original pool of obligation in addition with the new obligation and finitely unrolled obligation is strongly accountable. The way in which each type of obligation is unrolled is presented in the following discussion.

Algorithm 1 *StrongAccountableCascading* (γ, Φ, B, b)

Input: A policy $\langle\gamma, \Phi\rangle$, a strongly accountable obligation set B, and a new obligation b that generates cascading obligations.
Output: returns **true** if addition of b to the system preserves strong accountability.

if $b.\rho = 1$ **then**
  $B_{final} \coloneqq B \cup$ *UnrollCascading*(γ, Φ, b);
else if $b.\rho = I$ **then**
  $B_{final} \coloneqq B \cup \{b\}$;
else
  $B_{final} \coloneqq B \cup$ *UnrollFiniteRepetitive*(γ, Φ, b);
$m \coloneqq$ *MaxEndTime*(B_{final});
$B_{final} \coloneqq B_{final} \cup$ *UnrollInfiniteRepetitive*$(\gamma, \Phi, B_{final}, m)$;
for each obligation $b^* \in B_{final}$ **do**
  if $b^*.a$ = grant or revoke **then**
    InsertIntoDataStructure(b^*);
for each obligation $b^* \in B_{final}$ **do**
  if *Authorized* $(\gamma, \Phi, B_{final}, b^*)$= **false then**
    return false
return true

Unrolling Finite Cascading Obligations. To unroll the chain of cascading obligations incurred by b, Algorithm 1 uses procedure *UnrollCascading* described in Algorithm 2. This procedure is an adapted breadth-first search algorithm. Recall that we disallow infinite cascading obligations which guarantees that the procedure *UnrollCascading* will terminate. Furthermore, we also impose the restriction that each action, object pair can be authorized by only one policy rule. Thus, the new obligations incurred by a fixed obligation will be finite and fixed. For this, we use the function Ψ (discussed in section 4.2) that takes an obligation b and set of policy rules and returns a set of set of obligations which can be possibly incurred by b. Due to the restriction above, the result of Ψ will be a single set of obligations B_f that can be incurred by b. The different fields of each obligation $\hat{b} \in B_f$ will depend of the fields of b and the policy rule that authorizes b.

Unrolling Finite Repetitive Obligations. When the new obligation we want to add (b) is a finite repetitive obligation ($b.\rho \in \mathbb{N}$ and $b.\rho > 1$), we use the procedure *UnrollFiniteRepetitive* described in Algorithm 3 to unroll it appropriately. We follow the procedure presented in appendix B to unroll finite repetitive obligations. Thus, for the obligation b, the procedure *UnrollFiniteRepetitive* clones b, varying only the time intervals of the new obligations based on $b.\delta$. The exact number of copies of b that are unrolled will depend on $b.\rho$.

Algorithm 2 *UnrollCascading* (γ, Φ, b)

Input: A policy $\langle\gamma, \Phi\rangle$ and a new obligation b.
Output: returns a set of cascading obligations B that is generated by b.

$B = \varnothing$;
queue < *obligation* > q;
q.push(b);
while $!q.empty()$ **do**
  $b = q.front()$; $B \coloneqq B \cup \{b\}$;
  $q.pop()$; $B' \coloneqq \Psi(b, \Phi)$;
  for each obligation $b^* \in B'$ **do**
    $q.push(b^*)$;
return B

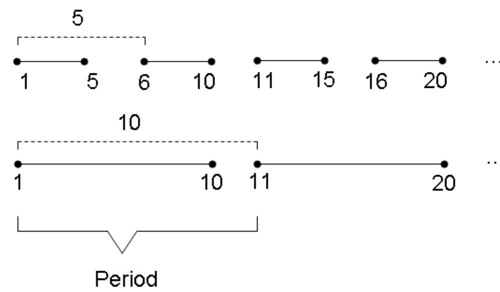

Figure 3: Computing Period of Infinite Repetitive

Algorithm 3 *UnrollFiniteRepetitive* (γ, Φ, b)

Input: A policy $\langle\gamma, \Phi\rangle$ and a finite repetitive obligation b.
Output: returns a set of unrolled obligations B that is generated by b.

$B = \varnothing$; $i \coloneqq 1$;
while $i \leq b.\rho$ **do**
  $b_i \coloneqq b$; $b_i.t_e \coloneqq (b.w - b.\delta) \times i + b.t_s - b.\delta$;
  $b_i.t_s \coloneqq b_i.t_e - w$; $B \coloneqq B \cup \{b_i\}$; $i \coloneqq i + 1$;
return B

Unrolling Infinite Repetitive Obligations. When the obligation we want to add (b) is an infinite repetitive obligation, Algorithm 1 uses procedure *UnrollInfiniteRepetitive*, described in algorithm 4, to unroll a finite amount of it. Let us consider $B_i \subseteq B$ is the set of infinite repetitive obligations. Note that, $b \in B_i$. First, we find the *overall period* of all the obligations in B_i at which the infinite repetitive obligations repeat themselves. In figure 3 we have two infinite repetitive obligations, $b_1 = \{u_1, a, o, t_s = 1, t_e = 5, \delta = 1, \rho = I, w = 4\}$ and $b_2 = \{u_1, a, o, t_s = 1, t_e = 10, \delta = 1, \rho = I, w = 9\}$. It is clear that after time 11, we see a pattern formed by the obligations, this is the overall period. The overall period is the least common multiple (LCM) of the periods of each $b_i \in B_i$. For each infinite repetitive obligation b_i, the period of b_i is given by $b_i.\delta + b_i.w$. Once the period is computed, we check to see whether the overall period is greater than the maximum end time of the finite obligations (repetitive or non-repetitive). If this is the case, we just need to unroll the infinite repetitive obligations three periods (to be safe). Otherwise, we unroll the infinite obligations until the maximum

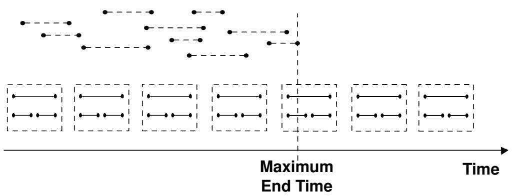

Figure 4: Unrolling Infinite Repetitive Obligations

time, and then we unroll two additional periods (figure 4). In the current pool of obligations let us assume that the only type of obligations present are the infinite repetitive obligations. When we have calculated the overall period of these infinite repetitive obligations, part of the authorization state influencing the permissibility of the infinite repetitive obligations, after each of these period should be equivalent to the authorization state before, if the system is accountable. If the authorization state is not necessarily equivalent, this will be revealed when the second repetition is analyzed. Thus, we do not need to analyze the infinite repetitive obligations beyond two repetitions. Similarly, when we have other types of obligations residing in the current pool of obligations, we can safely unroll the infinite repetitive obligations for two additional period after the maximum end time of the finite obligations and soundly decide accountability.

Algorithm 4 *UnrollInfiniteRepetitive* (γ, Φ, B, m)

Input: A policy $\langle\gamma, \Phi\rangle$, a set of obligations, where $B_i \subseteq B$ is a set of infinite repetitive obligations, and m representing the last time point where a non-infinite obligation happens.
Output: returns a set of unrolled obligations B' that is generated by B.

$B' = \varnothing$; $period = LCM(B)$
if $period > m$ **then**
  $finalTime := period * 3$;
else
  $finalTime := (\lceil (m/period) \rceil + 2) \times period$;
for each obligation $b' \in B_i$ **do**
  $end := b'.t_e$;
  while $end <= finalTime$ **do**
    $b_i := b'$; $b_i.t_e := (b'.w - b'.\delta) \times i + b'.t_s - b'.\delta$;
    $b_i.t_s := b_i.t_e - w$; $B' := B' \cup \{b_i\}$; $end := b_i.t_e$;
return B'

We now briefly summarize the non-incremental algorithm for deciding strong accountability due to Pontual *et al.* [20] which we use as a procedure for deciding accountability in presence of special cases of cascading obligations. We refer readers to Pontual *et al.* [20] for a detailed presentation.

The non-incremental algorithm takes as input a set of obligations, an authorization state, and a mini-ARBAC policy and returns true when the set of obligations is strong accountable. For this, the algorithm inserts all the administrative obligations in the set to a modified interval search tree. Then it checks whether each of the obligation is authorized in its whole time interval. To do this, the algorithm inspects whether the user performing the obligation has the necessary roles in the whole time interval. For simplicity, let us consider the user u needs the role r to perform the obligation. Then, the algorithm checks whether u has role r in the current authorization state. If so, then it checks whether there is an obligation overlapping with the current obligation that revokes r. If not, then u is guaranteed to have role r in the whole time interval. In case, u currently does not have role r, then the algorithm checks whether there is a grant of the role r to u and no one is revoking it. If that is the case, then u is guaranteed to have role r in the whole time interval.

5. EMPIRICAL EVALUATION

The goal of the empirical evaluations is to determine whether strong accountability can be decided efficiently for some special cases of cascading obligations. For those cases, our empirical evaluations illustrate that it is actually feasible to decide the strong accountability property.

The algorithm for deciding strong accountability for special cases of cascading obligations is implemented using C++ and compiled with g++ version 4.4.3. All experiments are performed using an Intel i7 2.0GHz computer with 6GB of memory running Ubuntu 11.10.

5.1 Input Instance Generation

As in the case for many security researchers, we do not have access to real life access control policies that contain obligations. Thus, we synthetically generate problem instances for our empirical evaluations. We believe the values of the different parameters we assume are appropriate for a medium sized organization.

In our experiments, we consider 1007 users, 1051 objects, and 551 roles. We also consider 53 types of actions, 2 of which are administrative (grant and revoke). We handcrafted a mini-RBAC and mini-ARBAC policy with 1251 permission assignment rules, 560 role assignment rules (maximum 5 pre-conditions in each), and 560 role revocation rules. Among the policy rules, 100 of them can incur new obligations. Each of which can incur a maximum of 10 new obligations totaling 1000 new cascading obligations.

To generate the obligations, we handcrafted 6 strongly accountable sets of obligations in which each set has 50 obligations. Each set has a different ratio of administrative to non-administrative obligations (rat). We then replicated each set of obligations for different users to obtain the desired number of obligations. Similarly, we generate the infinite and finite repetitive obligations, we use 6 sets of repetitive obligations that are strongly accountable. The execution times shown are the average of 100 runs of each experiment.

5.2 Empirical Results

Our accountability decision procedure takes as input an accountable pool of obligations B, the current authorization state γ, a mini-ARBAC policy Φ, and a new obligation b. It returns true when adding b and its associated new obligations maintain accountability. In these empirical evaluations, we consider cases where b can incur a finite amount of new obligations and can be finitely (infinitely) repetitive.

Finite Cascading Obligations.
In this case, we add an obligation to a strongly accountable set of obligations. This obligation in turn incurs 1000 new obligations. Then, the algorithm needs to decide whether these 1000 obligations along with the original strongly accountable obligation set is still strongly accountable. Figure 5 presents the results for the strong accountability algorithm for this case. Although the number of cascading obligations is fixed (1000) throughout this experiment, we vary the number of obligations by changing the number of pending obligations in the pool from 0 to 99000. We follow the same strategy for all the other cases.

The time required by the strong accountability algorithm grows roughly linearly in the number of obligations. In the

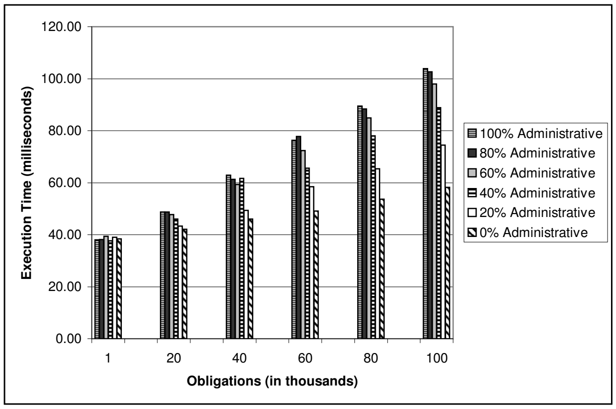

Figure 5: Finite Cascading Obligations.

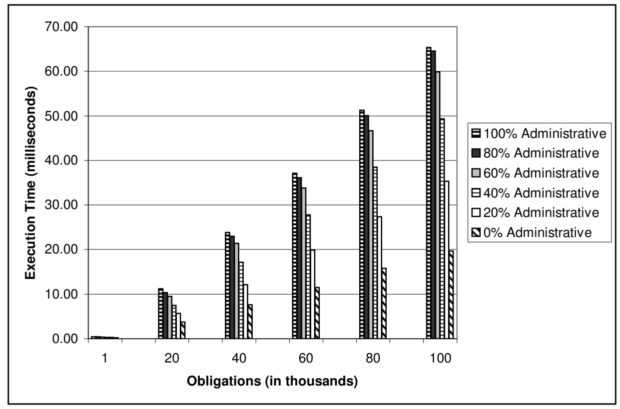

Figure 6: Infinite Repetitive Obligations.

worst case (99,000 administrative obligations plus 1000 finite cascading obligations), the algorithm runs in 103 milliseconds to determine that the set is strongly accountable. This is roughly two times slower than the non-incremental strong accountability algorithm presented by Pontual *et al.* [20] without cascading of obligations. This is due to the overhead of unfolding the cascading obligations (algorithm 3). As the algorithm must inspect every obligation following each administrative obligation, rat influences the execution time of the algorithm. In addition, we have also simulated (not shown) the same case when the original set of obligations have infinite repetitive obligations, in this case the worst execution time is still 103 milliseconds. This is due to the fact that the time of procedure *UnrollCascading* dominates the time of procedure *UnrollInfiniteRepetitive*.

Finite Repetitive Obligations.
In this experiment, we add a finite repetitive obligation to a strongly accountable obligation set. This new obligation repeats 1000 times ($\rho = 1000$). The algorithm decides whether the old set of obligations plus the 1000 copies of the repetitive obligation is strongly accountable. The execution time of the strong accountability algorithm grows roughly linearly in the number of obligations. In the worst case, the algorithm runs in 66 milliseconds to decide whether the set is strongly accountable. In general, if the number of obligations generated by the finite repetitive obligations is not too large (when compared with the original set), the time necessary to decide accountability is not affected by the addition of finite repetitive obligations. As algorithm 3 can unroll the repetitive obligations in a trivial way, the overhead of this procedure will be small provided that the number of repetition is small. In addition, we have also simulated (not shown) the same case when the original set of obligations have infinite repetitive obligations. The worst case execution time, for this case, is 66 milliseconds.

Infinite Repetitive Obligations.
In these experiments, we add an infinite repetitive obligation to a strongly accountable obligation set that already contains some infinite repetitive obligations. These infinite repetitive obligations together with b is cloned for a total of 519 times. Figure 6 shows the results for the strong accountability algorithm for this case. The execution time of the strong accountability algorithm grows roughly linearly in the number of obligations. In the worst case, the algorithm runs in 66 milliseconds.

6. RELATED WORK

Obligations have received a lot of attention from different researchers [3,4,6,7,10,12–19,21,25,26]. Some of them are interested in efficiently specifying obligatory requirements [3, 4, 6, 15, 19, 25, 26] and others are interested in the management of obligations [3, 7, 10, 12, 14, 18, 21].

Ni *et al.* [18] presented a user obligation model based on an extended role based access control for privacy preserving data mining (PRBAC) [19]. Their model supports repeated obligations, cascading obligations, pre and post-obligations and also conditional obligations. In addition, they also present how to detect infinite obligation cascading in a policy. Their work is complimentary to ours, since we study the impact of different types of cascading obligations when deciding accountability.

Ali *et al.* [2] presented an enforcement mechanism for obligations in service oriented architectures. Their model supports repetitive obligations, conditional and pre-obligations, but do not support finite cascading obligations. Although their model is more expressive than ours, they assume that obligations have all the necessary permissions.

Elrakaiby *et al.* [8] borrow the concepts of Event Condition Action from the area of database to present an obligation model. It supports pre and post-obligations, on-going, and continuous obligations. Obligations can have relative or absolute deadlines. To cope with violations, conflicts, and lack of permissions, they adopt a set of strategies such as sanctions for users that violate obligations, cancellation of obligations, delay of obligations, and re-compensation for users that fulfill their obligations. In contrast, we use accountability to detect violations before they occur.

Li *et al.* [14] extended XACML [26] to support a richer notion of obligations. They view obligations as state machines and can express pre-obligations, post-obligations, stateful-obligations, *etc.* However, they do not consider obligations requiring authorizations and in turn do not concentrate on deciding accountability. In this sense, our view of managing obligations is different than theirs.

7. CONCLUSION AND FUTURE WORK

In current work, we have refined the notion of strong accountability due to Irwin *et al.* [12] to allow cascading obligations. We also enhance the obligation model used by Pontual *et al.* [20] to support the specification of cascading obligations. We present several proposals to specify the obligatee in the policy. We then show that deciding accountability in general is NP-hard. Thus, we consider several simplifications for which the strong accountability decision becomes tractable. We provide an algorithm, its complexity, and also present empirical evaluations of the algorithm. Our experiments show that accountability can be efficiently decided for special cases of cascading obligations.

We want to explore other approaches for obligatee specification and understand their impact on accountability decision. Furthermore, we want to explore how to specify different kinds of obligations, namely, negative obligations, stateful obligations, group obligations, *etc.*, in our model and also study their impact on accountability decision.

8. ACKNOWLEDGEMENT

Ting Yu is partially supported by NSF grant CNS-0716210. Jianwei Niu is partially supported by NSF grant CNS-0964710. We would like to thank the anonymous reviewers and Andreas Gampe for their helpful suggestions.

9. REFERENCES

[1] Senate banking committee, Gramm-Leach-Bliley Act, 1999. Public Law 106-102.

[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] A. Barth, A. Datta, J. C. Mitchell, and H. Nissenbaum. Privacy and contextual integrity: Framework and applications. *Security and Privacy, IEEE Symposium on*, 0:184–198, 2006.

[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] O. Chowdhury, M. Pontual, W. H. Winsborough, T. Yu, K. Irwin, and J. Niu. Ensuring authorization privileges for cascading user obligations. Technical Report CS-TR-2012-005, UT San Antonio, 2012.

[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] Y. Elrakaiby, F. Cuppens, and N. Cuppens-Boulahia. Formal enforcement and management of obligation policies. *Data Knowl. Eng.*, 71:127–147, Jan. 2012.

[9] D. F. Ferraiolo, R. S. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli. Proposed NIST standard for role-based access control. *ACM Transactions on Information and Systems Security*, pages 224–274, Aug. 2001.

[10] 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.

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

[12] 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.

[13] A. J. I. Jones. On the relationship between permission and obligation. In *ICAIL '87*, New York, NY, USA. ACM.

[14] N. Li, H. Chen, and E. Bertino. On practical specification and enforcement of obligations. In *Proceedings of the second ACM conference on Data and application security and privacy*, 2012.

[15] M. J. May, C. A. Gunter, and I. Lee. Privacy APIs: Access control techniques to analyze and verify legal privacy policies. In *CSFW '06*, Washington, DC, USA, 2006. IEEE Computer Society.

[16] L. McCarty. Pemissions and obligations. In *Proceedings IJCAI-83*, 1983.

[17] 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.

[18] Q. Ni, E. Bertino, and J. Lobo. An obligation model bridging access control policies and privacy policies. In *SACMAT' 08*, New York, NY, USA. ACM.

[19] Q. Ni, A. Trombetta, E. Bertino, and J. Lobo. Privacy-aware role based access control. In *Proceedings of the SACMAT'07*, New York, NY, USA. ACM.

[20] M. Pontual, O. Chowdhury, W. Winsborough, T. Yu, and K. Irwin. Toward Practical Authorization Dependent User Obligation Systems. In *ASIACCS' 10*, pages 180–191. ACM Press, 2010.

[21] M. Pontual, O. Chowdhury, W. H. Winsborough, T. Yu, and K. Irwin. On the management of user obligations. SACMAT '11, New York, NY, USA. ACM.

[22] 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.

[23] A. Sasturkar, P. Yang, S. Stoller, and C. Ramakrishnan. Policy analysis for administrative role based access control. In *Computer Security Foundations Workshop, 2006. 19th IEEE*, 2006.

[24] S. D. Stoller, P. Yang, C. R. Ramakrishnan, and M. I. Gofman. Efficient policy analysis for administrative role based access control. In *CCS '07*, New York, NY, USA, 2007. ACM.

[25] A. Uszok, J. Bradshaw, R. Jeffers, N. Suri, P. Hayes, M. Breedy, L. Bunch, M. Johnson, S. Kulkarni, and J. Lott. Kaos policy and domain services: Toward a description-logic approach to policy representation, deconfliction, and enforcement. In *POLICY '03*, Washington, DC, USA, 2003. IEEE Computer Society.

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

APPENDIX

A. CASCADING OBLIGATIONS EXAMPLE

Let us assume the following policy rules. Policy rule p_1 allows a registered user to submit a paper and this in turn creates an obligation for a user (obligatee) in role reviewer to submit the review of the paper. Rule p_2 authorizes a user in role reviewer to submit a review of a paper and it incurs an obligation for a user in the role PC_chair that requires him to make a decision on the paper. Rule p_3 authorizes a user in role PC_chair to submit a decision for a paper and it incurs an obligation for the same user submitting the decision to notify the corresponding author. Rule p_4 authorizes a user in the role PC_chair to notify the author of a paper. Now,

$$
\begin{aligned}
&p_1 : submit(u, paper) \leftarrow (u \in registeredUser) :\\
&F_{obl}(reviewer, s, u, paper, 2\ days, 1\ week)\\
&\{\\
&\quad (Choose\, u_1\, such\, that\, u_1 \in reviewer)\\
&\qquad submitReview(u_1, \langle u, paper\rangle)\\
&\}\\
&p_2 : submitReview(u, \langle author, paper\rangle) \leftarrow (u \in reviewer) :\\
&F_{obl}(PC\_Chair, s, u, \langle author, paper\rangle, 1\ day, 1\ day)\\
&\{\\
&\quad (Choose\, u_1\, such\, that\, u_1 \in PC\_Chair)\{\\
&\qquad submitDecision(u_1, \langle author, paper\rangle);\\
&\qquad \}\\
&\}\\
&p_3 : submitDecision(u, \langle author, paper\rangle) \leftarrow (u \in PC\_Chair) :\\
&F_{obl}(Self, s, u, \langle author, paper\rangle, 1\ day, 1\ day)\\
&\{\\
&\qquad notify(u, \langle author, paper\rangle)\\
&\}\\
&p_4 : notify(u, \langle author, paper\rangle) \leftarrow (u \in PC\_Chair) : \varnothing
\end{aligned}
$$

consider the following situation. The set of current users of the system is $\gamma.U = \{Alice, Bob, Carol\}$ and their current role assignments are $\gamma.UA = \{\langle Alice, registeredUser\rangle, \langle Bob, reviewer\rangle, \langle Carol, PC\_Chair\rangle\}$. Let us assume Alice submits a paper on 07/01/2012 and according to p_1 Bob (in role reviewer) will get the following obligation $\langle Bob, submit\text{–}Review, \langle Alice, paper\rangle, 07/03/2012, 07/10/2012\rangle$. According to p_2, this obligation in turn will incur the obligation $\langle Carol, submitDecision, \langle Alice, paper\rangle, 07/11/2012, 07/12/2012\rangle$ for Carol (in role PC_chair). According to p_3 when Carol submits the decision, she incurs the obligation $\langle Carol, notify, \langle Alice, paper\rangle, 07/13/2012, 07/14/2012\rangle$.

B. REPETITIVE OBLIGATIONS

Let us consider an obligation $b = \{u, a, \vec{o}, t_s, t_e, \delta, \rho, w\rangle\}$. This obligation is considered to be infinite repetitive when $\rho = I$ or finite repetitive when $\rho \in \mathbb{N}$ and $\rho > 1$. For finite and infinite repetition of the obligation the possible time intervals of the recurring obligation are the following.

- **Finite Repetitive:** $[t_s, t_e], [t_e + \delta, t_e + \delta + w], \cdots [t_s + (\rho - 1)(w + \delta), t_e + (\rho - 1)(w + \delta)]$.
- **Infinite Repetitive:** $[t_s, t_e], [t_e + \delta, t_e + \delta + w], \cdots [t_s + (n - 1)(w + \delta), t_e + (n - 1)(w + \delta)], \cdots$ where $n \in \mathbb{N}$.

C. COMPLEXITY ANALYSIS OF THE ALGORITHM

Let us consider the current pending pool of obligations is B where $|B| = n$. Moreover, let us consider $B_i \subseteq B$ denotes the set of infinite repetitive obligations in the current pending pool of obligations where $|B_i| = d$. Let us consider the number of policy rules Φ is k. (1) When the new obligation b we want to add incurs a finite number of cascading obligations, the number of finite cascading obligations due to b can be approximated by k. This is due to our restriction that our policies are free of cycles. Furthermore, let us consider that the number of times the infinite repetitive obligations are unrolled is α. Thus, the total number of obligations for which we need to check accountability in this case is $\eta_c = \alpha \times d + k + (n - d)$. Then, we check each of the η obligations are all authorized, which can be done using the non-incremental algorithm presented by Pontual *et al.* [20] in $\mathcal{O}(k\eta_c^2 \times \log(\eta_c))$. (2) In the case b being a finite repetitive obligation, the number of times b needs to be unrolled is $b.\rho$. Let us denote it by m. Thus, the total number of obligations for which we need to check accountability in this case is $\eta_r = \alpha \times d + m + (n - d)$. The resulting complexity of the algorithm in this case will be $\mathcal{O}(k\eta_r^2 \times \log(\eta_r))$ (3) For the case, b is an infinite repetitive obligation, we have to compute the overall period of the obligations in B_i and b. Let, β denote the number of times the obligations in B_i and b needs to be unrolled. Thus, the total number of obligations for which we need to check accountability in this case is $\eta_i = \beta \times (d + 1) + (n - d)$. This results in a time complexity of $\mathcal{O}(k\eta_i^2 \times \log(\eta_i))$.