Toward Practical Authorization-dependent User Obligation Systems

[Technical Report : CS-TR-2009-011]

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

Many authorization system models include some notion of obligation. Little attention has been given to user obligations that depend on and affect authorizations. However, to be usable, the system must ensure users have the authorizations they need when their obligations must be performed. Prior work in this area introduced *accountability* properties that ensure failure to fulfill obligations is not due to lack of required authorizations. That work presented inconclusive and purely theoretical results concerning the feasibility of maintaining accountability in practice. The results of the current paper include algorithms and performance analysis that support the thesis that maintaining accountability in a reference monitor is reasonable in many applications.

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

Maintaining security in modern organizations depends on security procedures being faithfully carried out, both by computer systems and by humans. Security of computer systems relies on authorization systems to prevent malicious or accidental violation of confidentiality and integrity requirements. However, they also rely on human users and administrators performing actions that range broadly, including tasks such as business functions and administrative operations, and are an obligatory part of the humans' job descriptions. We call these actions *user obligations.* Most user obligations require corresponding system authorizations. Many also affect authorizations, as when an employee is reassigned to a different division, or a new project or business function is added, and an administrator must adjust authorizations accordingly. As automated tools seek to provide increasing support for managing personnel and projects, there is an increasing need for individual tasks to be assigned and coordinated with authorizations, and for supporting automated techniques. Thus, the management of user obligations that depend on and affect authorizations is a significant issue for the field of computer security. Nowadays, many governmental privacy regulations have some notion of user obligations which depends on authorization. For instance, the 45 CFR part 164 HIPAA [11] states that an individual has the right to request a Covered Entity (CE) (e.g., a hospital) to amend his private health information (PHI). After this request, the CE is obliged no later than 60 days to correct the individual's PHI (for doing this, some CE's employee must have the appropriate permissions) or to provide the individual the reasons for not amending his PHI.

Work on computer managed obligations goes back several decades [14, 16, 17]. Many works focus on policy determination of obligations [1, 5, 6, 8, 15, 20, 25, 26]. Relatively little work has focused on specification of the proper discharge of obligations [5, 9, 10, 12, 18]. Even less has focused on user obligations.

Working in the context of user obligations, Irwin *et al.* [12, 13] were the first to study obligations that depend on and affect authorizations. They observe that deadlines are needed for user obligations in order to be able to capture the notion of violation of obligations[1]. Associating a start time as well

[1] Users are unlike machines in that their future behavior cannot be determined, so only safety properties can be enforced, not liveness properties.

as an end time with each obligation, Irwin *et al.* introduce properties called strong and weak accountability [12] that ensure that each obligatory action will be authorized during its appointed time interval. For a given authorization policy, accountability is a property of the authorization state and the obligation pool. Roughly stated, accountability holds if each obligatory action will be authorized, no matter when in their associated time intervals all the other obligations are fulfilled. The strong version of accountability requires that each obligation be authorized throughout its entire time interval. Weak accountability allows an obligation to be unauthorized during part of its time interval, provided that if the obligated user waits for other obligations to be fulfilled, it is guaranteed that the action will become authorized before its deadline.

Thus, accountability properties can be viewed as invariants that the system attempts to maintain. To this end, it may be necessary to prevent discretionary (non-obligatory) actions being performed if they would disturb accountability[2]. This can happen in three ways. (1) An administrative action can change the current authorization state in a way that makes an obligated user unable to perform an obligatory action. (2) A discretionary action can cause an obligation to be incurred to perform an action that will not be authorized. (3) A discretionary action can cause an obligation to be incurred to perform an administrative action that will change the authorization state in a way that makes an obligated user unable to perform a subsequent obligation.

The results presented by Irwin *et al.* were inconclusive from the standpoint of practicality. They show that when the authorization system used in the obligation model is fully abstract, accountability is undecidable. When it is instantiated with an access control (AC) matrix model, the problem of determining strong accountability becomes polynomial. However for the algorithm they present, this polynomial has degree 4 in the number of obligations and degree 2 in the policy size. No performance evaluation is provided to determine the limits of problem size that can be handled. Moreover, determining weak accountability in the instantiated case remains intractable (co-NP hard) and no decision procedures are presented.

The results of the current paper support the thesis that maintaining strong accountability in the reference monitor is reasonable in most applications, and that in many, even weak accountability can be supported adequately.

In this paper we instantiate the authorization portion of the obligation model of Irwin with a previously studied administrative role-based access control model [21, 22] called mini-ARBAC [23]. Using mini-ARBAC instead of the AC matrix simplifies the problem of determining accountability, largely because obligatory actions are limited to making at most one change to the authorization state per action. The **first contribution** is an algorithm that determines strong accountability of a set of pending obligations under this instantiated model. This algorithm has complexity that is $n^2 \log n$ times the policy size, in which n is the number of pending obligations. The empirical evaluation that we have conducted indicates that the algorithm runs in less than $1/10^{th}$ seconds, even for very large policies and obligation sets, when the algorithm is used in an incremental fashion to determine whether a single obligation can be added to an accountable obligation pool. This result supports the thesis stated above.

Interestingly, weak accountability remains theoretically intractable, even with the simplification of administrative actions that can be obligatory. Our **second contribution** is the result that the problem is co-NP complete in the simplified model, and remains so when only one policy rule enables each action and the condition expressed by that rule is purely conjunctive. (Both positive and negative role memberships can be tested, however.)

We also study the question whether in practice weak accountability can be decided despite the problem's theoretical intractability. We explore two approaches, one that designs an algorithm specifically to solve this problem, and one that uses model checking. Design, specification, optimization, prototypes, and empirical evaluation of these techniques form the **third contribution** of this paper. Our empirical evaluation of these techniques indicate that they are effective in many cases for obligation sets and policies of moderate size, and that there are cases in which each one outperforms the other. In short, the special-purpose algorithm can handle larger problem instances (more obligations, users, roles, and policy rules), provided interdependent obligations do not overlap too much, while the model checker is much better able to deal with the many possible interleavings of overlapping obligations.

Our recapitulation of Irwin's obligation system and accountability definitions makes some improvements on the original. The most significant of these is that we formalize the scheduling of obligations in terms of traces (sequences of states and actions), rather than by assigning times at which actions are performed. Assigning times raises the issue of two actions being scheduled at the same time. Since actions must be atomic, Irwin *et al.* introduced a fixed, arbitrary order on actions that defines the order in which actions scheduled for the same time are performed. Not only does this make the presentation cumbersome. This order can actually affect whether a given obligation pool is accountable. In the current formulation, time is used only to define the relative order in which obligations may be carried out. This is our **fourth contribution**.

Section 2 provides background necessary to understand our contributions. Section 3 presents our algorithm for determining strong accountability. Section 4 presents our complexity result for weak accountability with the mini-ARBAC authorization system, as well as our techniques for determining weak accountability: the special-purpose algorithm, including a very powerful optimization, and the model checking approach. Section 5 presents an empirical evaluation of all these techniques. Section 6 discusses related work. Section 7 discusses future work and concludes.

2. BACKGROUND

This section reviews mini-ARBAC. It then defines the obligation system model and accountability properties, both of which are abstract with respect to the authorization system. It puts our contributions in context with respect to de-

[2] In many cases it will be necessary to enable users (especially administrators) to force discretionary actions to be performed, even when doing so violates accountability. Also, since it is impossible to ensure that users will fulfill their obligations, accountability guarantees obligatory actions will be authorized only under the assumption that other obligations on which they depend are faithfully discharged. Issues such as how to restore accountability when it becomes violated remain open. Section 2.4 discusses these and other topics that place the current contribution in context.

velopment of obligation systems in which obligations depend on and affect authorizations. It concludes by showing how to instantiate the obligation system with the mini-ARBAC authorization system, and by giving an example obligation system that uses a mini-ARBAC authorization policy.

2.1 mini-ARBAC

The widely studied ARBAC97 model [22] has been simplified somewhat by Sasturkar *et al.* for the purpose of studying policy analysis, forming a family of languages called *mini-ARBAC* [23]. The member of the family that we use supports administrative actions that modify user-role assignments, but does not consider role hierarchies, sessions, changes to permission-role assignments, or role administration operations, such as creation or deletion of roles. Constraints can be placed on the current role memberships of users that are candidates for assignment to new roles; other constraints are not supported. The model does not distinguish general and administrative roles. The presentation of mini-ARBAC here is based on the one given by Sasturkar *et al.* [23]. An example mini-ARBAC policy is presented in table 1.

We use mini-ARBAC in this study in part because of its relationship with RBAC, an access control model that has gained wide acceptance in many sectors. Previous work [12] in accountability analyzed obligations in the context of the AC matrix model. It is interesting and useful investigate issues that arise in integrating the user obligation systems with another popular authorization system. It turns out that mini-ARBAC is simpler than the AC matrix model, because administrative operations are more restricted. In the AC matrix model, individual administrative actions can make multiple changes to the access control state, whereas in mini-ARBAC, actions can change only one permission or role assignment at a time. Moreover, the impact of those changes are simpler, as authorization depends on individual role memberships, rather than on a combination of matrix entries. As we shall see, this simplification has a beneficial impact on the complexity of deciding strong accountability. The algorithms we present in sections 3 and 4 support obligations that make administrative changes to the user-role assignment. In section 3.3, we discuss the applicability of our strong accountability algorithm to other administrative changes in mini-ARBAC, ARBAC with role hierarchies, and to the AC matrix model.

Definition 1 (mini-ARBAC model:). *A mini-ARBAC model γ is a tuple $\langle U, R, P, UA, PA, CA, CR\rangle$ where:*

- U, R and P are the finite sets of users, roles and permissions respectively, where a permission $\rho \in P$ is a pair $\langle action, object\rangle$. Note that we allow the use of wildcards(*) in the expression of permissions to denote collections of permission pairs.
- $UA \subseteq U \times R$ is a set of user-role pairs. Each $\langle u, r\rangle \in UA$ indicates that user u is a member of role r.
- $PA \subseteq R \times P$ is a set of permission-role pairs. If $\langle r, \rho\rangle \in PA$, users in role r are granted the permission ρ.
- $CA \subseteq R \times \mathcal{C} \times R$ is a set of the *can_assign* rules, in which $\mathcal{C}$ is the set of preconditions, the structure of which is discussed presently. Each $\langle r_a, c, r_t\rangle \in CA_\gamma$ indicates that users in role r_a are authorized to assign a user to the *target role* r_t, provided the target user's current role memberships satisfy precondition c. A precondition is a conjunction of positive and negative roles. Target user u_t satisfies c in γ (written $u_t \vDash_\gamma c$) if for each literal l in c, $u_t \vDash_\gamma l$, which is defined by $u \vDash_\gamma r \equiv \langle u, r\rangle \in \gamma.UA$ and $u \vDash_\gamma \neg r \equiv \langle u, r\rangle \notin \gamma.UA$. So the action $grant(u_a, r_t, u_t)$ is authorized if there exists $\langle r_a, c, r_t\rangle \in CA$ such that $u_a \vDash_\gamma r_a$ and $u_t \vDash_\gamma c$.
- $CR \subseteq R \times R$ is a set of *can_revoke* rules. Each $\langle r_a, r_t\rangle \in CR_\gamma$ indicates that a user belonging to the (administrative) role r_a has the capability to revoke the role r_t from any target user (*i.e.*, there exists $\langle r_a, r_t\rangle \in CR$ such that $u_a \vDash_\gamma r_a$). There are no constraints on revocation based on the other roles held by the target user. Our treatments in later sections can be easily extended to accommodate them, however.

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⟩}
PA	=	{⟨developer, ⟨develop, sourceCode⟩⟩, ⟨projectManager, ⟨assignProjObl, *⟩⟩, ⟨blackBoxTester, ⟨test, software⟩⟩}
CA	=	{⟨ securityManager, ¬ blackBoxTester, Developer⟩, ⟨ securityManager, ¬ developer, blackBoxTester⟩}
CR	=	{⟨ securityManager, blackBoxTester⟩}

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

2.2 Obligations

This section presents an abstract meta-model that encompasses the basic constructs of an authorization system that supports obligations. Our presentation in this section is derived from that of Irwin *et al.* [12]. An obligation system consists of the following components:

- $\mathcal{U}$: a universe of users.
- $\mathcal{O}$: a universe of objects with $\mathcal{U} \subseteq \mathcal{O}$.
- $\mathcal{A}$: a finite set of actions that can be initiated by users. The structure of actions is given just below.
- $\mathcal{T}$: a countable set of time values.
- $\mathcal{B} = \mathcal{U} \times \mathcal{A} \times \mathcal{O}^* \times \mathcal{T} \times \mathcal{T}$: the universe of obligations users can incur. Given $b = \langle u, a, \vec{o}, \text{start}, \text{end}\rangle \in \mathcal{B}$, $b.u$ denotes the obligated user, $b.a$ the action the user must perform, $b.\vec{o}$ the finite sequence of zero or more objects that are parameters to the action, and $b.$start and $b.$end the start and end times of the interval during which the action must be performed[3]. A *well formed* obligation b satisfies $b.\text{start} < b.\text{end}$.

  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 $\mathcal{E} = \mathcal{D} \cup \mathcal{B}$.

[3] Throughout, we refer to components of structured objects such as b with notation such as $b.\vec{o}$.

- Γ : fully abstract representation of authorization state (*e.g.*, AC matrix, UA).
- $\mathcal{S} = \mathcal{FP}(\mathcal{U}) \times \mathcal{FP}(\mathcal{O}) \times \mathcal{T} \times \Gamma \times \mathcal{FP}(\mathcal{B})$: the set of system states[4]. We use $s = \langle U, O, t, \gamma, B\rangle$ to denote system states. U is the finite set of users currently in the system, O, the finite set of objects, t, the current time, B, the set of pending obligations, and $\gamma \in \Gamma$.
- $\mathcal{P}$: a fixed set of policy rules. 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 $\gamma \vDash cond(u, \vec{o}, a)$) in the current authorization state γ when the rule is used to authorize the action. F_{obl} is an *obligation function*, which returns a finite set $B \subset \mathcal{B}$ of obligations incurred (by u or by others) when the action is performed under this rule.

Each action a denotes a higher order function of type $(\mathcal{U} \times \mathcal{O}^*) \rightarrow (\mathcal{FP}(\mathcal{U}) \times \mathcal{FP}(\mathcal{O}) \times \Gamma) \rightarrow (\mathcal{FP}(\mathcal{U}) \times \mathcal{FP}(\mathcal{O}) \times \Gamma)$. When, in state s, user $u \in s.U$ performs action a on the objects $\vec{o} \in s.O^*$, $a(u, \vec{o})(s.U, s.O, s.\gamma)$ returns $\langle s'.U, s'.O, s'.\gamma\rangle$ for the new state s'. Thus, actions can introduce new users and objects, have side effects, and change the authorization state. Note that in general performing an action also introduces new obligations; these depend on the policy rule used, as well as on the action, and are handled in Definition 2 below.

A trace $\sigma \in \Sigma^\omega$ is an infinite[5] sequence of elements of $\Sigma = \mathcal{S} \times \mathcal{E} \times \mathcal{P}$. We use $e \in \mathcal{E}$ to range over both obligatory and discretionary events. For $i \geq 0$, each element σ_i of a trace σ has the form $\sigma_i = \langle s_i, e_i, p_i\rangle$, where p_i is the policy rule according to which the action e_i is performed to arrive at s_{i+1}.

Each trace element must follow from its predecessor according to the effect of the corresponding action, which is denoted by $s_i \xrightarrow{e_i, p_i} s_{i+1}$ for $i \geq 0$, and is defined as follows.

Definition 2 (Transition relation). *The relation* $\longrightarrow \subseteq \mathcal{S} \times \mathcal{E} \times \mathcal{S} \times \mathcal{P}$ *is defined by* $s' \xrightarrow{e,p} s''$ *if and only if the policy rule* $p = a(u, \vec{o}) \leftarrow cond(u, \vec{o}, a) : F_{obl}(s, u, \vec{o}) \in \mathcal{P}$ *satisfies* $a = e.a$, *and* $s'' = \langle U'', O'', t'', \gamma'', B''\rangle$ *satisfies*

$$s.\gamma \vDash cond(u, \vec{o}, a)$$

$$e.u \in s'.U \text{ and } e.\vec{o} \in s'.O^*$$

$$\langle U'', O'', \gamma''\rangle = a(u, O)(s'.U, s'.O, s'.\gamma)$$

$$B'' = \begin{cases} (s'.B - \{e\}) \cup F_{obl}(s', e.u, e.O), \ \textit{if } e \in \mathcal{B} \\ s'.B \cup F_{obl}(s', e.u, e.O), \ \textit{otherwise} \end{cases}$$

2.3 Accountability Properties

In this section, we present a reformulation of the definitions of strong and weak accountability [12]. Accountability properties are defined in terms of hypothetical schedules according to which the given pool of obligations could be executed, starting in the given state. Strong accountability requires that each obligation be authorized throughout its entire time interval, no matter when during their intervals the other obligations are scheduled, and no matter which policy rules are used to authorize them.

On the other hand, weak accountability allows an obligation to be unauthorized during part of its time interval, provided that if the obligated user waits for other obligations to be fulfilled, it is guaranteed that the action will become authorized before its deadline. More formally, as we consider schedules according to which users may attempt their obligations, weak accountability allows us to ignore those schedules in which an obligatory action is unauthorized if other obligations scheduled to come after it have earlier end times. Weak accountability is violated if some schedule attempts an unauthorized, obligatory action after all obligations with earlier deadlines have been performed.

Example 3 (Accountability). *Using the mini-ARBAC policy presented in table 1, let us assume 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* $[5, 20]$

The example obligation system is weakly accountable, but not strongly accountable. It is not strongly accountable as obligation b_2 is not authorized before time 7 and is not guaranteed to be authorized until time 9: initially Carl does not have the role developer (table 1) and Joan may not grant him the role until time 9. On the other hand, the system is weakly accountable because obligation b_1 can be fulfilled any time in its time interval, since Joan already has the role of security manager and Carl satisfies the CA rule constraints, and b_2 can be fulfilled anytime after time 9.

Let us now formalize the accountability properties. As we discuss further in the next section, our treatment here ignores the possibility of "cascading" obligations. Specifically, we assume that actions that can cause new obligations to be incurred are disjoint from actions that can be obligatory. Space considerations prevent a more general treatment here, though such a treatment is planned to be presented in the very near future, along with techniques for handling such systems.

Given a set of obligations B, a *schedule* of B is a sequence $\{b_i\}_{0 \leq i \leq n}$ that enumerates B, for $n = |B| - 1$. Given a schedule $\{b_i\}_{0 \leq i \leq n}$, for any $k \in \{0..n\}$, we write $b_{0..k}$ for $\{b_i\}_{0 \leq i \leq k}$. In this case, we call $b_{0..k}$ a *prefix* of $b_{0..n}$; when $k < n$, we say the prefix is *proper*. A schedule of B is *valid* if $(\forall i, j).(0 \leq i < j \leq n \rightarrow b_i.\text{start} \leq b_j.\text{end})$. This prevents scheduling b_i before b_j if $b_j.\text{end} < b_i.\text{start}$. Given an authorization state s_0, and a policy $\mathcal{P}$, a schedule $b_{0..n}$ for $s.B$ is *authorized by* policy-rule sequence $p_{0..n} \subseteq \mathcal{P}$ and *yields* s_{n+1} (denoted $s_0 \xrightarrow{\langle b,p\rangle_{0..n}} s_{n+1}$) if

each obligation in it is authorized by the corresponding policy rule when it occurs. Assume we are given $b_{1..k}$, an initial state s_0 and $k \in \{0..n\}$ (*Note: $b_{1...0}$ denotes the empty sequence*). We denote by $s_0 \xrightarrow{\langle b,p\rangle_{1..k}} s_k$ the assertion

$(\exists s_1, s_2, \ \ldots \ , s_n).(s_0 \xrightarrow{b_0, p_0} s_1 \xrightarrow{b_1, p_1} \ \ldots \ \xrightarrow{b_{n-1}, p_{n-1}} s_n \xrightarrow{b_n, p_n} s_{n+1})$.

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

[4] 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.

[5] We use the standard convention of denoting by ω the least infinite ordinal number.

Given a schedule $b_{0..n}$, a proper prefix $b_{0..k}$ is a *critical prefix* if for all j such that $k+1 < j \leq n$, b_{k+1}.end $\leq b_j$.end. Weak accountability requires much the same thing as strong accountability, but only for critical prefixes.

Definition 5 (Weak accountability). *Given a state $s_0 \in \mathcal{S}$ and a policy $\mathcal{P}$, we say that s_0 is* weakly accountable *if for every valid schedule, $b_{0..n}$, and every critical prefix of it, $b_{0..k}$, for every policy-rule sequence $p_{0..k} \subseteq \mathcal{P}$ and 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 $s_{k+1} \xrightarrow{b_{k+1},p_{k+1}} s_{k+2}$.*

2.4 Context of Current Contribution

There are many research issues that must be addressed by any system that manages obligations that interact with authorization. This section summarizes many of them in an effort to place the issues addressed here in context.

As mentioned in the introduction, the ideal of preserving accountability as a system invariant cannot be achieved, as users may fail to fulfill their obligations. Furthermore, it may be necessary to revoke a user's authorization to fulfill their obligations, as when a user leaves an organization or is transfered to a different position. Tools are needed that enable administrators and other authorities to remove or reassign obligations, or to add new obligations that replace old ones in a manner that authorizes fulfillment of new or existing dependent obligations. These may include tools that analyze dependencies among existing obligations or that propose plans whereby accountability could be maintained or restored. While important research issue, the design of such tools are not within the current scope, and will be addressed separately.

Irwin *et al.* showed that when the obligation model studied here is instantiated with the access control matrix model of authorization, it is possible to develop a polynomial time algorithm for determining strong accountability. To accomplish this, they made an additional assumption, which we also make. Specifically, it is helpful to prevent *cascading* obligations, by which we mean obligatory actions that cause additional obligations to be incurred. Several issues complicate handling such situations. For example, different policy rules can cause different obligations to be incurred, making it difficult to reason about the future state of the obligation pool. Also, the time intervals of the new obligations depend on the time at which the action is performed that causes them to be incurred, making it difficult to reason about when the new obligations must be authorized. To simplify matters, we follow Irwin *et al.* in partitioning actions into those that can be *obligatory*, which cannot themselves cause new obligations to be incurred, and those actions that are purely *discretionary*, which can.

Admittedly, this is a strong assumption, and one that we intend to investigate relaxing in the future. Doing so is a rich research problem unto itself. It seems highly unlikely that permitting arbitrary cascading of obligations would lead to a tractable accountability decision problem. Several models might be explored to permit a more restricted notion, however. For instance, the length of cascading chains might be bounded, say, by establishing a partial order on actions, and requiring that obligatory actions incurred must be strictly dominated by the actions that cause them to be incurred. Recurring obligations [18] also provide a form of expressive power that is related to a restricted form of cascading.

There are other issues as well. For instance, since our vision is to incorporate accountability checks into a reference monitor, what happens when different policy rules for the same action cause different obligations to be incurred. Certainly, if one rule leads to an accountable obligation pool, and the other does not, one would expect the former rule to be used. When both rules lead to accountability, the appropriate course of action seems to be application dependent. In some cases, it may be appropriate to let the user requesting the action make the decision. When this is inappropriate, for example, due to performance issues, a range of policy-driven alternatives present themselves.

We are interested in exploring these and other problems. However, the most urgent research question at this juncture is whether either strong or weak accountability can be decided fast enough to make such systems potentially viable.

In this paper, we show that when the authorization system is mini-ARBAC, strong accountability can be decided very quickly even for large problem instances. Perhaps more surprisingly, we show that for a very large collection of medium-size problem instances, weak accountability can be decided with a speed that would be adequate for many applications.

2.5 Concrete Model Using mini-ARBAC

In this section, we instantiate the obligation system of Irwin *et al.*, summarized above in section 2.2, to use mini-ARBAC as the authorization model. We also present an example of an obligation-system policy that uses a mini-ARBAC policy.

In the instantiated, concrete model we introduce here, $\mathcal{O}$ is a set of objects with $\mathcal{U} \cup R \subseteq \mathcal{O}$, in which R is the set of roles and $\mathcal{U}$ is the set of users of the system. The finite set of actions $\mathcal{A}$ comprises two different types of actions, administrative actions (*e.g.*, *grant* and *revoke*) and non-administrative actions (*e.g.*, *read*, *write*, *etc.*). An event e is a tuple $\langle u, a, \vec{o}\rangle$ in which u is the user performing the action, a is the action (*i.e.*, *grant*, *revoke*, *read*, *etc.*), $\vec{o}$ is a tuple of objects, the subtype of which depends on the action. For instance, $\vec{o} = \langle r_t, u_t\rangle$ when a is an administrative action, and $\vec{o}$ may be $\langle \mathit{book}\rangle$ when a is the non-administrative action (*e.g.*, *read*).

In the concrete model, we use a mini-ARBAC policy $\gamma = \langle P, R, \mathit{UA}, \mathit{PA}, \mathit{CA}, \mathit{CR}\rangle$. We instantiate the set of permissions P by assuming $P \subseteq \mathcal{A} \times \mathcal{O}^*$, the set of action, object-tuple pairs. We omit $\mathcal{U}$ because it occurs elsewhere in the obligation-system state s. In our context, the policy γ is modified dynamically: the techniques we introduce in later sections for determining accountability support administrative changes to UA. The set of obligation-system policy rules $\mathcal{P}$ consists of policy rules of the form $p = a(u, \vec{o}) \leftarrow \mathit{cond}(u, \vec{o}, a) : F_{obl}(s, u, \vec{o})$. The way in which it is determined whether the condition $\mathit{cond}(u, \vec{o}, a)$ is satisfied in a given policy state γ depends on whether a is an administrative action. When a is grant (respectively, revoke), $\vec{o}$ is a pair $\langle u_t, r_t\rangle$ and u is attempting to grant role r_t to (respectively, revoke r_t from) user u_t. In this case u's authorization to do so is determined by $\gamma.\mathit{UA}$ and $\gamma.\mathit{CA}$ (respectively, $\gamma.\mathit{CR}$). When action a is not administrative, u's authorization is determined by $\gamma.\mathit{UA}$ and $\gamma.\mathit{PA}$. Formally, $\gamma \vDash \mathit{cond}(u, \vec{o}, a)$

is defined as follows:

$$\begin{aligned}
&\gamma \vDash \mathit{cond}(u,\vec{o},a) \equiv (\exists r).(((u,r) \in \gamma.UA)\wedge\\
&\quad (a \notin \text{administrative} \rightarrow (\langle r, \langle a,\vec{o}\rangle\rangle \in \gamma.PA))\wedge\\
&\quad (\forall u_t, r_t).((a \in \text{administrative} \wedge \vec{o} = \langle u_t, r_t\rangle) \rightarrow (\\
&\qquad (a = \mathit{grant} \rightarrow ((\exists c).(((\langle r, c, r_t\rangle \in \gamma.CA) \wedge (u_t \vDash_\gamma c))))\wedge\\
&\qquad (a = \mathit{revoke} \rightarrow (((\langle r, r_t\rangle \in \gamma.CR))))))
\end{aligned}$$

Note that u_t and r_t are guaranteed to exist and to be unique.

Example 6 (Obligation System). *We use the mini-ARBAC policy presented in table 1 to illustrate three scenarios that demonstrate how an obligation system can be used to manage a software development cycle. For simplicity, we assume all the roles and permissions in this example are associated with a specific software development project.*

In scenario 1, Bob has an obligation to perform black-box testing of some software. Should the security manager, Joan, attempt to revoke Bob's black-box tester role, she would be prevented from doing so. This is because Bob needs the role to fulfill his obligation, so revoking it would make the system unaccountable. (Of course, in some situations, such as Bob leaving the company, Joan would have to be able to force revocation. This requires handling the violation of accountability, and is beyond the scope of the current contribution.)

In scenarios 2 and 3, Eve, the project manager performs discretionary actions that assign obligations to team members. For this, she uses the action assignProjObl, which is governed by the following policy rule in our framework:

- $\mathit{assignProjObl}(pm,\ \langle \mathit{oblAction}, \mathit{oblUser}, \overrightarrow{\mathit{oblObject}}, \mathit{oblStart}, \mathit{oblEnd}\rangle) \leftarrow (pm \vDash_\gamma \mathit{projectManager}) : \{\langle \mathit{oblUser}, \mathit{oblAction}, \overrightarrow{\mathit{oblObject}}, [\mathit{oblStart}, \mathit{oblEnd}]\rangle\}$

In scenario 2, Eve creates a new obligation that requires Alice to perform black-box testing within 1 month. The action Eve performs is given by assignProjObl(Eve, ⟨test, Alice, ⟨software⟩, 01/01/2010, 02/01/2010*⟩. Eve satisfies Eve* $\vDash_\gamma$ *projectManager, so the authorization system permits her to perform the action. However, the new obligation, ⟨Alice, test, ⟨software⟩,* [01/01/2010, 02/01/2010]*⟩, would make the system unaccountable, since Alice does not have the role of black-box tester. So Eve's action is prevented.*

In scenario 3, after discovering that Alice does not have the black-box tester role, Eve attempts to create a new obligation that obligates Joan to grant Alice the role of black-box tester. For this, Eve attempts the action, assignProjObl(Eve, ⟨Grant, Joan, ⟨blackBoxTester, Alice⟩, 01/01/2010, 02/01/2010*⟩). This would generate the new obligation ⟨Joan, Grant, ⟨blackBoxTester, Alice⟩,* [01/01/2010, 02/01/2010]*⟩. However, as Alice does not satisfy the conditions required for assignment to this role, Joan would be unable to fulfill this obligation.*

Recall from section 1 that there are three reasons that a discretionary action may be prevented so as to preserve accountability. The scenarios above illustrate each of these, in order. Note that in scenarios 2 and 3, our system alerts Eve immediately that the tasks she is attempting to assign cannot be completed, and enables her to revise her plans accordingly. Without the accountability checks, Eve would not be alerted to the fact that the obligations cannot be fulfilled until the obligated users attempted the obligatory actions and were prevented from performing them by the authorization system.

Algorithm 1 *StrongAccountable* (γ, B, b)

```
Input: A policy $\gamma$, a strongly accountable obligation set $B$, and
    a new obligation $b$.
Output: returns true if addition of $b$ to the system preserves
    strong accountability.
if Authorized ($\gamma$, $B$, $b$)= false then
    return false
if $b.a \neq$ grant or revoke then
    return true
$After = \{b' | b' \in B \wedge b'.\text{end} > b.\text{start}\}$
$B = B \cup b$
for each obligation $b^* \in After$ do
    if Authorized ($\gamma$, $B$ , $b^*$)=false then
        $B = B \setminus b$ /* Restore representation of $B$ */
        return false
return true
```

3. STRONG ACCOUNTABILITY

This section begins by presenting our incremental algorithm for determining whether adding a new obligation to a strongly accountable obligation pool preserves the property. It then discusses the complexity of the algorithm, comparing it with that of the algorithm presented by Irwin *et al.* A non-incremental version of the algorithm is then discussed. The algorithms presented here and in the next section are specialized to mini-ARBAC, but can be generalized to support other models in which user rights are modified by administrative actions. As discussed in section 2.4, all our techniques make the assumption that obligatory actions cannot cause new obligations to be incurred.

3.1 The Algorithm

Algorithm 1 is designed to be used incrementally for maintaining strong accountability. Below, we discuss using it to determine whether a given set of obligations is strongly accountable. Algorithm 1 takes as input a strongly accountable set of obligations B, a mini-ARBAC policy γ, and a new obligation b. It returns *true* if adding b to B preserves strong accountability, and *false* otherwise.

The intuition behind Algorithm 1 is as follows. To determine whether adding b to B preserves strong accountability, the algorithm inspects the current policy γ and each obligation $b' \in B$ that could be performed prior to b ($b' \in B \wedge b'.\text{end} \leq b.\text{start}$) to determine whether b will be authorized during its entire time interval, $[b.\text{start}, b.\text{end}]$. The algorithm uses procedure *Authorized* (Algorithm 2) for this purpose. If $b.a$ is an administrative action, the algorithm also determines whether b interferes with authorizations required by later obligations b^* ($b^* \in B \wedge b^*.\text{end} > b.\text{start}$). When $b.a$ is not administrative, this is not necessary, as it does not affect authorizations.

Because $b.u$'s roles can change during the interval $[b.\text{start}, b.\text{end}]$, Algorithm 2 must check each subinterval defined by start and end points of obligations in B to check whether the obligatory action is authorized during that period. For this it uses the set $\mathit{subint}(B)$, which we construct as follows. Let $\{tp_i\}_{1 \leq i \leq w}$ enumerate $\mathit{timepoints}(B)$ in sorted order, in which $\mathit{timepoints}(B)$ is defined by $\mathit{timepoints}(\varnothing) = \varnothing$ and $\mathit{timepoints}(B \cup \{b\}) = \mathit{timepoints}(B) \cup \{b.\text{start}, b.\text{end}\}$. Here, w denotes the size of the set $\mathit{timepoints}(B)$. The set $\mathit{subint}(B)$ is now defined by $\mathit{subint}(B) = \{[tp_i, tp_{i+1}] \mid 1 \leq i < w\}$.

Algorithm 2 uses procedure *hasRole* (Algorithm 3) to determine whether a user's role memberships conform to re-

Algorithm 2 *Authorized* (γ, B, b)

```
Input: A policy $\gamma$, an obligation set $B$, and an obligation $b$.
Output: returns true if $b$ is authorized with respect to $\gamma.UA$ and $B$
if $b = \langle u, grant, \langle r_t, u_t \rangle, [start, end] \rangle$ then
    return $(\forall [s,e] \in subint(B \cup \{\text{start}, \text{end}\})).$
    $(overlaps([s,e], [\text{start}, \text{end}]) \rightarrow$
      $(\exists \langle r_a, c, r_t \rangle \in \gamma.CA).(hasRole(\gamma, B, u \vDash r_a, [s,e])$
      $\wedge (\forall l \in c).(hasRole(\gamma, B, u_t \vDash l, [s,e]))))$
else if $b = \langle u, revoke, \langle r_t, u_t \rangle, [start, end] \rangle$ then
    return $(\forall [s,e] \in subint(B \cup \{\text{start}, \text{end}\})).$
    $(overlaps([s,e], [\text{start}, \text{end}]) \rightarrow$
      $(\exists \langle r_a, r_t \rangle \in \gamma.CR).(hasRole(\gamma, B, u \vDash r_a, [s,e]))$
else /* $b = \langle u, a, \langle \vec{o} \rangle, [start, end] \rangle$ */
    return $(\forall [s,e] \in subint(B \cup \{\text{start}, \text{end}\})).$
    $(overlaps([s,e], [\text{start}, \text{end}]) \rightarrow$
      $(\exists \langle r_a, \langle a, \vec{o} \rangle \rangle \in \gamma.PA).$
      $(hasRole(\gamma, B, u \vDash r_a, [s,e]))$
```

quirements for $b.a$ to be authorized. Recall that to be authorized, a grant action $grant(u, r_t, u_t)$ requires that $u_t \vDash_\gamma c$, in which c is given by a *can_assign* rule, $\langle r_a, c, r_t \rangle \in \gamma.CA$. This requires the ability to test that u_t is *not* in certain roles, as required to satisfy a query such as $u \vDash_\gamma \neg r$. Thus *hasRole* takes a query and the time interval of b, during which the query should be satisfied. When a pending obligation $b' \in B$ can change whether the query is satisfied during the time interval of b, $b.a$ is not guaranteed to be authorized during its full time interval, indicating that strong accountability is not satisfied. Otherwise, the current policy $(\gamma.UA)$ is investigated to determine whether the query is satisfied at present. Then the last pending obligation scheduled to be performed before b that affects the role membership in question is found and inspected. The result of *hasRole* is then determined on this basis.

The procedure, *hasRole*, must find the last grant or revoke of the given role to the given user. To support this, we use a modified interval search tree, which performs such lookups in time $\mathcal{O}(\log n)$, in which $n = |B|$. Using it, the time complexity of *hasRole* is also $\mathcal{O}(\log n)$. For each time interval in $subint(B \cup \{b.\text{start}, b.\text{end}\})$ that overlaps $[b.\text{start}, b.\text{end}]$, *Authorized* calls *hasRole* once for each policy rule and once for each literal in the associated constraint c, making the time complexity of *Authorized* $\mathcal{O}(qmn \log n)$, in which q is the number of policy rules ($q = Max\{|CA|, |CR|, |PA|\}$) and m is the maximum size of the role constraints in the *can_assign* rules in *CA*. In the worst case, in which the obligatory action is administrative (*grant* or *revoke*), the main procedure *StrongAccountable* calls *Authorized* n times. This results in an overall worst-case time complexity of $\mathcal{O}(qmn^2 \log n)$. When the obligation to be added, b, is non-administrative, the time complexity of *StrongAccountable* is reduced to $\mathcal{O}(qmn \log n)$. The memory complexity, when implemented with a modified interval search tree is $\mathcal{O}(|\mathcal{R}| \cdot |\mathcal{U}| + n)$.

3.2 Non-incremental Version

Algorithm 2 can be used in a non-incremental fashion to determine whether a given obligation set B is strongly accountable. This is achieved by adding each administrative obligation to an empty modified interval search tree and then calling *Authorized* for each obligation $b \in B$ to see whether it is authorized in the context of γ and B. This can be done in $\mathcal{O}(qmn^2 \log n + a \log n) = \mathcal{O}(qmn^2 \log n)$, in which a is the number of obligations to perform administrative actions. (The term $a \log n$ is the cost of constructing the search tree.)

Algorithm 3 *hasRole* $(\gamma, B, u \vDash_\gamma l, [s,e])$

```
Input: A policy $\gamma$, an obligation set $B$, a query $u \vDash_\gamma l$ in which
    $l$ has either the form $r$ or $\neg r$, and a time interval $[s,e]$.
Output: Returns true if $u \vDash_\gamma l$ is guaranteed to hold throughout
    the interval $[s,e]$.
if $l = r$ then /* positive role constraint */
    if $(\exists \langle u', revoke, \langle r, u \rangle, [start, end] \rangle \in B).($
    $overlap([s,e], [start, end]))$ then
        return false
    if $\langle u, r \rangle \in \gamma.UA$ then
        if $(\exists \langle u', revoke, \langle r, u \rangle, [start, end] \rangle \in B).(end < s)$
        then
            Select such a tuple so that end is maximized
            if $(\exists \langle u'', grant, \langle r, u \rangle, [start', end'] \rangle \in B).$
            $(start' > end \wedge end' < s)$ then
                return true
            else
                return false
        else
            return true
    else /* $\langle u, r \rangle \notin \gamma.UA$ */
        if $(\exists \langle u', grant, \langle r, u \rangle, [start, end] \rangle \in B).$
        $(end < s)$ then
            Select such a tuple so that start is maximized
            if $(\exists \langle u'', revoke, \langle r, u \rangle, [start', end'] \rangle \in B).$
            $(overlap([start, e], [start', end']))$ then
                return false
            else
                return true
        else
            return false
else /* $l = \neg r$ negative role constraint */
    In case of negative role checking ($u \vDash_\gamma l$ where $l = \neg r$),
    the algorithm follows similar steps, reversing the roles of
    "GRANT" and "REVOKE" and reversing the negative
    and positive role tests.
```

3.3 Generalizing the Authorization Model

Algorithm 1 can be extended to support an authorization model based on an access control matrix (ACM) by replacing the notion of roles with the notion of permissions and changing the structure of the authorization state accordingly. Suppose that we assume that policy conditions remain purely conjunctive and that each obligatory action adds or removes at most one right from one cell. These correspond to assumptions we make in the algorithm presented above for mini-ARBAC. The resulting algorithm for the ACM model determines whether adding an obligation to perform an administrative action preserves strong accountability in time $\mathcal{O}(kn^2 \log n)$, in which k is the number of the policy rules. (Note: $k \approx qm$.) For adding an obligation to perform a non-administrative action, it makes the determination in time $\mathcal{O}(kn \log n)$.

These assumptions make the supported form of obligations less expressive than those supported by the algorithm presented by Irwin *et al.* That prior algorithm differs from this ACM variant of our algorithm by allowing disjunctions as well as conjunctions in policy-rule conditions and by allowing administrative actions to modify multiple rights in multiple cells. So the Irwin algorithm supports an obligation model that is strictly more expressive than that supported by ours. The Irwin algorithm runs in time $\mathcal{O}(k^2 n^4)$. The additional expressivity of obligations supported by the Irwin algorithm explains a factor of $\mathcal{O}(kn)$ in the difference between the complexities of these algorithms. The remaining factor of $\mathcal{O}(n/\log n)$ arises because we obtain a performance improvement by using a data structures based on binary interval search tree. Were a similar data structure used in the Irwin algorithm, this difference would disappear.

Algorithm 1 and the special-purpose weak accountability algorithm presented below in section 4.1 can, without modifying the time complexity, be extended to support administrative actions that modify the *PA*, *CA* and *CR* components of the authorization state.

We believe greater generalization is also possible, for instance, to support ARBAC models that support role hierarchies and in which the role hierarchy can be modified. However, doing so seems likely to increase the algorithm's complexity by a factor of the number of roles in the system.

4. WEAK ACCOUNTABILITY

In the previous section, we presented a polynomial time algorithm for the strong accountability problem. However, for the weak accountability problem instantiated with mini-ARBAC, we present the following theorem.

THEOREM 7. *Given a set of obligations B, a mini-ARBAC policy γ, including an initial authorization state $\gamma.UA$, deciding whether the given system is weakly accountable is co-NP complete.*

We have proved this theorem by reducing 3-SAT to the problem of determining that a given problem instance is not weakly accountable. (See appendix B for the proof details.) This theorem casts doubt on the feasibility of supporting weak accountability in practice. To gain insight into the limits of this feasibility, we evaluate two approaches to determining weak accountability, and find that they efficiently solve many instances.

The approaches we use are, respectively, an algorithm designed specifically for this purpose and the general-purpose technique of model checking. Model checking is a formal verification method for determining whether a FSM model satisfies a temporal logic property.

The special-purpose algorithm explicitly considers all authorized[6] critical prefixes of valid schedules and checks whether the next obligation in the schedule is authorized. On the other hand, the model checking approach models the execution of a set of obligations as a finite state machine (FSM) and checks the accountability property, as specified by a temporal logic formula. We use a symbolic [7] model checker (*viz.*, Cadence SMV [2]). This approach has the advantage that, without constructing actual traces, it computes a characterization of states starting from which a trace can reach a state that violates accountability. An efficient dynamic programming algorithm avoids considering multiple interleavings of actions when order makes no difference to the result.

When the set is not weakly accountable, both methods generate a counter example. If a given state s_0 is not weakly accountable, then a *counter example* is an authorized critical prefix $b_{0..k}$ of a valid schedule such that the next obligation in the schedule is not authorized.

4.1 Special-Purpose Algorithm for Determining Weak Accountability

In this section, we present an algorithm (see Algorithm 4) designed specifically to solve the weak accountability problem. The algorithm takes as input a list of pending obligations $B = b_1, \ldots, b_n$, $n = |B|$, and a mini-ARBAC policy γ. It returns *true* if the list of pending obligations B is weakly accountable and *false* otherwise.

As mentioned above, the algorithm investigates all authorized critical prefixes of valid schedules of B and checks whether the next obligation in the schedule is authorized. If not, a counter example has been found and is returned. Otherwise, it returns *true*, indicating that B is weakly accountable.

The algorithm uses a recursive procedure, *solve*, to incrementally explore valid schedules in a depth-first manner. Each invocation of *solve* extends a prefix that is known to be authorized and extensible to at least one valid schedule. For each obligation that, under the validity constraint, is a candidate to extend the prefix, the algorithm determines whether that obligation is authorized in the current authorization state. (The current authorization state is maintained so as to reflect the initial authorization state provided by γ and the effect of obligations already in the prefix.) If the obligation is authorized, it is appended to the prefix, the authorization state is updated, and the procedure is invoked recursively to explore further extensions of the prefix. If it is not authorized, the algorithm determines whether the obligation's end time is later than that of some other obligation that is not in the current prefix. If so, the obligation is skipped, as the current prefix is not a critical prefix of schedules in which this obligation comes next. If not, a counter example has been found; it is reported, and the algorithm terminates. Unless a counter example has been found, the algorithm proceeds by examining any remaining candidates to be added to the current schedule.

The algorithm uses a boolean array *executed*$[1..n]$ to represent the set of obligations that have been successfully incorporated into the current partial schedule.

Recall that a valid schedule must execute b_1 before b_2 if b_1.end $\leq$ b_2.start. As *solve* incrementally constructs a possible schedule, an obligation is not yet ready to be incorporated if some other unscheduled obligation has to go before it. On the other hand, an obligation b is ready to be scheduled if b.start $\leq$ *minTime* in which *minTime*, the least end time of all unincorporated obligations, is given by $minTime = min\{b_i.\text{end} \mid executed[i] = false\}$.

The algorithm assumes that the obligation set is represented by an array B and begins by sorting B by time interval start time. This supports identifying obligations that are ready to be scheduled very efficiently by a single index, *ready*, which is maintained so as to preserve the invariant that *ready* is the greatest i such that $executed[i] = false \wedge b_i.\text{start} \leq minTime$.

The multi-set of end time points of as yet unscheduled obligations is implemented by using a balanced binary search tree T, which supports finding the minimum value, inserting a new value, and deleting a value, each in $\mathcal{O}(\log n)$. We represent $\gamma.UA$ by a 2-dimensional array representing its characteristic function, which enables us to perform lookup, update and restore operations in constant time.

[6] Given a state s_0 and a valid schedule of $s_0.B$, a critical prefix $b_{0..k}$ of that schedule is *authorized* if there exists a policy-rule sequence $p_{0..k} \subseteq \mathcal{P}$ and state s_{k+1} satisfying $s_0 \xrightarrow{\langle b,p\rangle_{0..k}} s_{k+1}$.

Algorithm 4 *WeaklyAccountable*(γ, B)

```
Input: A policy $\gamma$, current pending obligations $B$
Output: return true $B$ is weakly accountable.
sort the obligation list $B$ according to the non-decreasing
    order of end time of the obligations.
$numEx \Leftarrow 0$
$executed\ [1...n] \Leftarrow$ false
$T \Leftarrow null$
return $solve(0, \gamma, B, T, numEx, executed)$
```

Algorithm 5 $solve(ready, \gamma, B, T, numEx, executed)$

```
Input: The index $ready$, represents the index of the last obligation that is ready, a policy $\gamma$, obligation array $B$, multi-set of end times $T$, total number of obligation executed so far $numEx$, status of obligations $executed$.
Output: returns false if there is a counter example which ensures that $B$ is not weakly accountable.
if $numEx \geq |B|$ then
    return true
$i \Leftarrow ready + 1$
if $T = null$ then
    $T.insert(B[i].\text{end})$
    $ready \Leftarrow ready + 1$; $i \Leftarrow i + 1$
while $i \leq |B|$ and $T.minValue() \geq B[i].\text{start}$ do
    $T.insert(B[i].\text{end})$
    $ready \Leftarrow ready + 1$; $i \Leftarrow i + 1$
for all $j \in [1, ready]$ do
    /* For $B[j] = \langle u, a, \vec{o}, \langle s, e \rangle \rangle$, $a(u, \vec{o})(\gamma)$ is the policy
    state obtained after the action */
    if $executed[j] = $ false then
        if $(\exists p \in \mathcal{P}).(\gamma \vDash p.cond(u, \vec{o}, a))$ then
            /* obligation $B[j]$ is currently authorized */
            $executed[j] \Leftarrow$ true
            $T.delete(B[j].\text{end})$
            if $\neg solve(ready, a(u, \vec{o})(\gamma),$
            $B, T, numEx + 1, executed)$ then
                return false
            $executed[j] \Leftarrow$ false
            $T.insert(B[j].\text{end})$
        else
            if $T.minValue() \geq B[j].\text{end}$ then
                print counter example
                return false
return true
```

Optimizing the Algorithm

The special-purpose weak accountability algorithm becomes impractical when the number of overlapping obligations is greater than about 10. However, in practice, usually the overlapping obligations are not related directly or indirectly. Using that intuition, we can optimize the above algorithm by using a notion called *dependence of obligations*. The idea behind the optimization is to partition the set of pending obligations so that no two sets in the partition contain obligations that are mutually dependent. We then execute Algorithm 4 separately on each set in the partition.

We say two obligations are *dependent* on each other if one of the obligations grants or revokes a role that the other obligation uses (or might use according to the policy) (*i.e.*, as a pre-condition or as part of a permission).

According to the definition of dependence, we write $b_1 \sim b_2$ if there is a dependency between obligation b_1 and obligation b_2. We write $\approx$ for the reflexive transitive closure. As $\sim$ is also symmetric, this is an equivalence relation. We say obligation set B_1 and B_2 are independent ($B_1 \not\approx B_2$) if $\neg(\exists b_1 \in B_1).(\exists b_2 \in B_2).(b_1 \approx b_2)$. The proof of the following theorem is given is appendix C.

THEOREM 8. *Let $\{B_1, B_2\}$ partition a pending set of obligations, B, such that $B_1 \not\approx B_2$. There is a counter example of weak accountability for the obligation set B_1 or B_2 or both if and only if there is a counter example for B.*

Even with this optimization, the special-purpose algorithm becomes impractical when it is applied to a set of obligations that have a high degree of overlapping among obligations that are also dependent on one another. In that case, the optimization gains little advantage. Fortunately, this circumstance seems to be quite rare in practice.

4.2 Model Checking Approach for Determining Weak Accountability

In this section, we describe an approach to determining weak accountability by using model checking techniques. Model checking is a formal verification method for determining whether a FSM model satisfies a temporal logic property. In our study, we use the symbolic [7] model checker, Cadence SMV [2]. For each input problem instance, we construct a FSM that encodes the obligations and explores all possible valid, weakly authorized schedules. In effect, it checks each one to determine whether they contain any unauthorized obligations. However, symbolic model checkers, such as the one we use, construct an internal representation, based on binary decision diagrams, that characterizes FSM traces without explicitly constructing them. As we report below in section 5, this enables the technique to solve many problem instances that involve too many schedules for our special-purpose algorithm to handle.

4.2.1 FSM State Variables

The FSM we construct uses the following 5 state variables.

- t (Integer) — t represents the current system time. Its value ascends through the start and end points of input-obligation time intervals. It is used to identify obligations that are ready to execute (their time interval has been entered) and those that must be executed before time advances (their time interval is about to end).
- n (Integer) — This represents the index of the obligation that is selected by the scheduler to be executed.
- *obl* (Array of boolean) — Indicates for each obligation i whether it has been performed (0 for no, 1 for yes).
- *UA* (Two dimensional array of boolean) — *UA* represents the current authorization state. $UA[u][r]$ takes the value 1 when user u is in role r, and 0 otherwise.
- *aco* (Boolean) — Records whether a counter example has been found.

4.2.2 The FSM for Checking Weak Accountability

The FSM is divided in two main modules as pictured in Figure 1:

The scheduler comprises two parts. *The timer* increments the system time when the end times of the remaining obligations all exceed the current system time. *The obligation selector* nondeterministically selects one of the pending obligations that is currently ready to be executed. An obligation b is ready at time t if it satisfies $t \geq b.\text{start} \wedge t \leq b.\text{end}$.

The monitor receives an obligation b from the scheduler, and checks whether it is authorized according to *UA*. If so, it records that this obligation has been fulfilled by setting $obl[b] = 1$. If b is an administrative action, the monitor

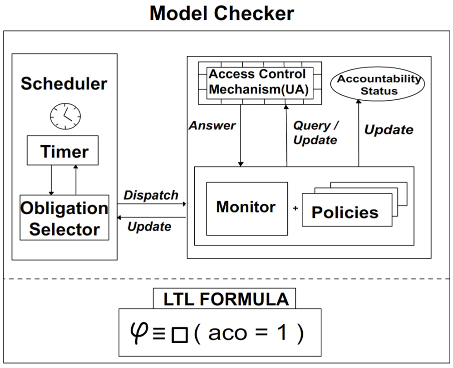

Figure 1: Organization of the FSM

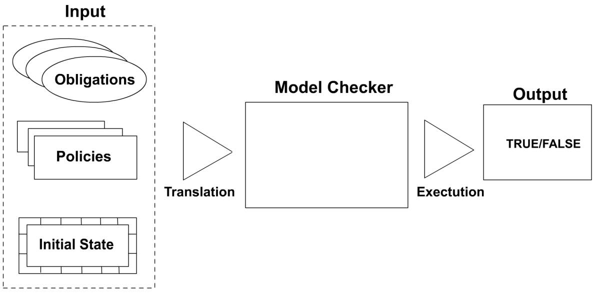

Figure 2: Translation Model

changes UA accordingly. On the other hand, if b is not authorized, this may or may not represent a counterexample to weak accountability. If $t < b$.end, weak accountability can still be satisfied if b becomes authorized later, before the end of its time interval. In this case, the monitor returns control to the scheduler to select another obligation. However, if $t = b$.end, the monitor sets $aco = 0$, signaling that the system is not weak accountable that is captured by checking whether it is always the case that the variable aco is true.

4.2.3 *Translation and Optimizations*

We use a C++ program to translate a mini-ARBAC security policy γ, the initial state of a given system $\gamma.UA$, and a set of obligations B into a Cadence SMV finite-state-machine (FSM). The translator hard-codes the security policy γ and the obligation set into a FSM. Then, we execute the SMV code to determine whether the obligation set is weak accountable(Figure 2).

To reduce the number of states generated, the translator scans the obligations and represents as a constant each entry of the authorization state that is not modified during execution. For optimization purposes, instead of the system time assuming arbitrary values; we normalize the end point of each obligation to differ by one. To decrease the number of the states generated, we also make sure that the nonadministrative obligations are scheduled just in their end times as they do not alter the authorization state of the system.

5. EVALUATION RESULTS

A central goal of our empirical evaluation is to determine how practical it is to use the strong accountability algorithm as a part of a reference monitor. This section presents results of experiments designed to assess the adequacy of our algorithms and techniques with respect to performance. When a discretionary action is attempted, the problem that a reference monitor must solve is to determine whether the action, if permitted, would cause accountability to be violated. The discretionary action could do this by changing the current authorization state, causing new obligations to be incurred, or both. It is the performance of using our techniques to determine whether the discretionary action would lead to a violation of accountability that we wish to evaluate. When the determination must be made, there is an existing obligation pool, which in general includes new obligations that would be incurred if the discretionary action were permitted, and a current authorization state, which again reflects the state that would result if the discretionary action were carried out.

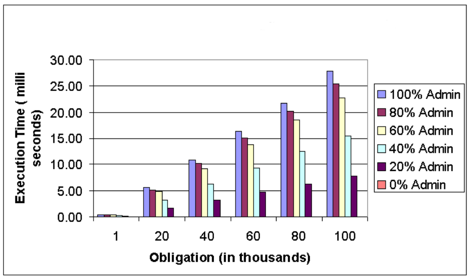

Figure 3: Performance of incremental strong accountability algorithm

In our evaluation, we use a system with a moderate-size policy and 1000 users. For this, we perform two sets of experiments. The first evaluates the efficiency of the incremental algorithm. For an obligation set of size 100,000, the algorithm runs in 25-30 milliseconds. The second experiment is done on the non-incremental algorithm. On the same size input, it requires about 450 milliseconds. The number of users and roles have little impact on the algorithms' execution times, and the effect of the number of policy rules is roughly linear. Thus we conclude that when the mini-ARBAC authorization system is used, our algorithm for strong accountability provides adequate performance to be incorporated into reference monitors for most applications.

Another goal is to determine the effectiveness of all our approaches to weak accountability. We find that if the obligations overlap little and have a low degree of mutual dependence, the optimized special-purpose algorithm outperforms the model-checking approach; however, when obligations are clustered and have a high degree of mutual dependence, the model-checking approach tends to perform better. We introduce a metric called the Degree of OVErlapping (DOVE) that is the number of pairs of overlapping obligations, normalized with respect to the number of possible overlaps. Imagine a graph with nodes given by obligations and an edge connecting each pair of obligations that overlap in time. The DOVE is the size of the edge set divided by the number of edges the graph would have if it were complete.

All the experiments are performed using an Intel Core 2 Duo 2.0GHz computer with 2GB of memory running Ubuntu 8.10. The algorithms for strong and weak accountability are implemented in C++ and built with gcc 4.2.4.

policy	R	O	A	CA	CR	PA	C	Generated
γ_0	50	50	50	60	60	250	10	By Hand
γ_1	50	50	50	2500	2500	120,000	10	Random
γ_2	27	12	37	37	38	1,200	4	By Hand

Table 2: Policies used in experiments. C represents the number of roles in the pre-conditions of CA rules

5.1 Evaluation of the Strong Accountability Algorithm

To evaluate the strong accountability algorithms, we assumed 1000 users and used a handcrafted mini-ARBAC policy γ_0 summarized in table 2. 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. The execution times shown are the average of 100 runs of each experiment.

Figure 3 presents results for the incremental algorithm. As we can see, the time required by the incremental algorithm grows roughly linearly in the number of obligations. The impact of *rat* on the execution time of algorithm 1 arises largely because the algorithm must inspect every obligation following each administrative obligation.

In the experiments for the non-incremental algorithm (SA), we found that the execution time (not shown) grows roughly linearly with the size of the obligation set. As with the incremental algorithm, a higher *rat* value leads to a higher execution time.

5.2 Evaluation of the Weak Accountability Approaches

In this section, we compare the performance of the special-purpose algorithm (algorithm 4) (SP), the optimized special-purpose (OSP) algorithm, and the model checking approach (MC). We used two policies, γ_1 and γ_2, summarized in table 2. (See [3] for the detailed policies.) To generate obligations, we used two algorithms, A_1 and A_2. A_1 uses policy γ_1 to generate a random set of obligations for given values of DOVE, number of obligations (n), and *rat*. A_2 uses policy γ_2 and the parameters presented above, and generates obligations like in section 5.1. In the experimental results presented in table 3 and 4, the entry x means the approach was unable to terminate within a reasonable time (60 minutes) for the input set.

Table 3 (a) shows the execution time vs. number of obligation(n) for MC, SP and OSP. Obligations were generated by A_1 using DOVE =0.002 and rat = 5%. The execution time for MC grows rapidly with n, as this and the number of users increases the size of the state space. On the other hand, execution time of SP and OSP grows slowly with n and OSP has a much better performance than SP.

Table 3 (b) shows execution time vs. DOVE for MC, SP and OSP. We used 100 obligations generated by A_2 with a *rat* value of 20%. The execution time of MC and OSP are not affected by DOVE, whereas that of SP grows exponentially. We empirically determined that the value of DOVE 0.158 forms a threshold for SP beyond which it does not perform adequately. Again, the execution time for the OSP approach is much faster than the MC approach.

n	100	300	700
MC	1.0	16.1	324.8
SP	0.006	0.025	1.42
OSP	0.002	0.003	0.016

(a) Time in seconds

DOVE	0.02	0.06	0.19
MC	0.80	0.85	0.83
SP	0.001	179.0	x
OSP	0.0006	0.0006	0.0006

(b) Time in seconds

Table 3: (a) Execution time of the MC, SP and OSP approach vs. the number of obligations (n). (b) Execution time of the MC, SP and OSP approach vs. DOVE.

rat	0%	10%	20%
MC	97.9	99.3	104.1
SP	91.1	92.8	96.8
OSP	0.004	0.004	0.005

(a) Time in seconds

n	10^3	10^4	3×10^4
MC	1831.5	x	x
SP	0.003	0.055	1.859
OSP	0.008	0.618	5.27
SA	0.001	0.006	0.052

(b) Time in seconds

Table 4: (a) Execution time of the MC, SP, OSP approach vs. different *rat* value. (b) Execution time of the MC, SP, SA and OSP vs. n.

Table 4 (a) presents execution time vs. *rat* for all the techniques using A_2 where DOVE = 0.01 and n = 500. The *rat* affects MC and SP techniques' execution time, whereas the OSP does not seem to be affected and has a speedup of magnitude 25,000 compared to both MC and SP. In our experiments, we used A_2 to generate obligations for most of our inputs, as it is difficult in practise to randomly generate large sets of obligations that are weakly accountable.

In our experiments, we found problem instances where the MC approach outperforms the OSP approach (*viz.*, when the obligations are clustered together and are dependent on each other) and vice versa. We use this fact and introduce a hybrid approach (see Figure 4). In this approach, we use the fact that strong accountability implies weak accountability so we check if the obligation set is strongly accountable. If not, then we perform some quick tests (presented in the appendix A) which can identify some of the problem instances that are definitely not weakly accountable. If the test is unable to decide the accountability query, then we compute two metrics: DOVE and dependency; based on these metrics we choose whether we use the MC, SP or OSP approach. In case that both metrics are unsatisfied, we execute both approaches. If neither of the approaches terminates within a specified amount of time, we say that the problem instance can not be solved.

6. RELATED WORK

Many obligation models have been proposed, ranging from largely theoretical [9,12,17] to more practical [10,13,15,18, 24].

Minsky and Lockman [17] were the first to suggest incorporating obligations into authorization models. They proposed a very broad model that includes concepts of positive obligations, negative obligations, deadline of obligations, obligations triggered by events, compensatory actions for failed obligations, *etc.* The model is probably too abstract to be implemented in a real world system.

Nowadays, many policy languages can specify the assignment of obligations as part of the policy. Some of these include XACML [26], EPAL [1], KAoS [25] and Ponder [8].

Bandara *et al.* [4] extended the model proposed by Irwin *et al.* [12] for obligations in order to create more complex policy

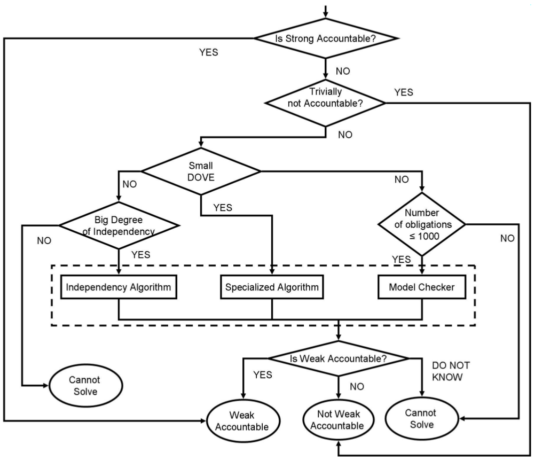

Figure 4: The Hybrid Approach for solving Weak accountability

rules. The language proposed is more expressive, yet the strong accountability problem remains tractable.

Gama and Ferreira [10] presented an implementation of an obligation model called Heimdall, which is based on policies written in xSPL. Obligations are triggered by specific events.

The model of Bettini *et al.* [6] associates obligations assigned with logic-programming style policy rules used to prove each given authorization. Dougherty *et al.* [9] 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 states for obligations. They do static analysis by using Büchi automata to answer questions such as determining whether two obligations are contradictory or whether a run of the system fulfills a given obligation, *etc.* Thus, their analysis work focuses on system obligations, rather than on user obligations.

Ni *et al.* [18, 19] have created a concrete model for obligations based on PRBAC [20] that handles repeated obligations, pre and post obligations and conditional obligations. They also presented two algorithms to analyze the dominance and infinite obligation cascading property. The problems we deal in our work are inherently different from those they consider in their work.

May *et al.* [15] presented a formal model for legal privacy policies of HIPAA. They consider a basic model of obligations without considering many theoretical issues. They use model checking to analyze properties with respect to some fixed policy rules. They do not give any experimental results of the efficiency of their approach to the problem.

Swarup *et al.* [24] presented a model for data sharing agreements in which a set of obligation is considered as constraints. They also suggested using model checking to verify some properties in their model, though they did not mention any practical implementation.

7. CONCLUSION

In this paper, we have presented an algorithm to decide strong accountability efficiently and two methods to decide weak accountability. These are based on an obligation system that uses mini-ARBAC as its authorization system. We have given experimental results that demonstrate that the performance of the algorithm for strong accountability is excellent and that show the methods for weak accountability are adequate for most medium-size problem instances. This is despite our result that, even using the simple authorization model, mini-ARBAC, the weak accountability problem is co-NP complete. Algorithm 1 and the special-purpose weak accountability algorithm presented in section 4.1 can be extended to support administrative actions that can modify *PA*, *CA* or *CR* and it does not effect the time complexity.

In section 2.4, we have discussed several open problems that need to be addressed to support user obligations that depend on and affect authorizations. These problems are the subject of future work.

Acknowledgment

This research was supported by the NSF under awards CNS-0716210 and CNS-0716750. We would like to take the opportunity to thank Dr. Alexander Pretschner for giving the idea of using model checking for the weak accountability problem. We also thank Andreas Gampe and the anonymous reviewers for their helpful suggestions.

8. REFERENCES

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

[2] Cadence SMV, 2009. http://www.kenmcmil.com/.

[3] Description of the policies used in the experiments, 2009. https://galadriel.cs.utsa.edu/policies/.

[4] A. Bandara, J. Lobo, S. Calo, E. Lupu, A. Russo, and M. Sloman. Toward a Formal Characterization of Policy Specification Analysis. In *Annual Conference of ITA (ACITA), University of Maryland, USA*, September 2007.

[5] 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.

[6] 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.

[7] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Inf. Comput.*, 98(2):142–170, 1992.

[8] 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.

[9] D. J. Dougherty, K. Fisler, and S. Krishnamurthi. Obligations and their interaction with programs. In *CESORICS '07: Proceedings of the 12th European Symposium On Research In Computer Security, Dresden, Germany, September 24-26, 2007, Proceedings*, pages 375–389, 2007.

[10] P. Gama and P. Ferreira. Obligation policies: An enforcement platform. In *6th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2005)*, 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 *CCS '06: Proceedings of the 13th ACM conference on Computer and communications security*, pages 134–143, New York, NY, USA, 2006. ACM.

[13] K. Irwin, T. Yu, and W. H. Winsborough. Assigning responsibilities for failed obligations. In *iTrust '08: IFIPTM Joined iTrust and PST Conference on Privacy, Trust Management and Security*, pages 327–342. Springer Boston, 2008.

[14] A. J. I. Jones. On the relationship between permission and obligation. In *ICAIL '87: Proceedings of the 1st international conference on Artificial intelligence and law*, pages 164–169, New York, NY, USA, 1987. ACM.

[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: Proceedings of the 19th IEEE workshop on Computer Security Foundations*, pages 85–97, 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 *ICSE '85: 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 2008: Proceedings of the 13th ACM symposium on Access control models and technologies*, pages 133–142, New York, NY, USA, 2008. ACM.

[19] Q. Ni, D. Lin, E. Bertino, and J. Lobo. *Conditional Privacy-Aware Role Based Access Control.* Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2007.

[20] Q. Ni, A. Trombetta, E. Bertino, and J. Lobo. Privacy-aware role based access control. In *SACMAT '07: Proceedings of the 12th ACM symposium on Access control models and technologies*, pages 41–50, New York, NY, USA, 2007. ACM.

[21] R. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.

[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, 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.

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

[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: Proceedings of the 4th IEEE International Workshop on Policies for Distributed Systems and Networks*, page 93, 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. PROPERTIES FOR HYBRID APPROACH

1. We simulated the schedule in which each obligation is executed at its end time. If under this schedule any obligation is not authorized, the obligation set is not weakly accountable. (The converse is not true because other valid schedules may also provide counter examples.)
2. If some obligation b requires a user to be in a role r and another obligation b' that overlaps with b with b.end $\le b'$.end and that revokes u the role, then the set of obligations is not weakly accountable. (No third obligation could restore the role, should b' be executed before b.) Similarly, if b requires the user not to occupy the role, and b' grants it, the set is not weakly accountable.

B. PROOF OF CO-NP COMPLETENESS OF WEAK ACCOUNTABILITY PROBLEM

THEOREM 7. *Given a set of obligations B, a mini-ARBAC policy $\mathcal{P}$ and an initial authorization state γ, deciding whether the given system is weakly accountable is co-NP complete.*

PROOF. We are given a 3-SAT expression $S = d_1 \wedge d_2 \wedge \cdots \wedge d_n$ where each $d_i = l_1^i \vee l_2^i \vee l_3^i$ for $i \in [1,\ n]$. Let the set of variables involved in this expression be given by $X = \{x_1, x_2, \ldots, x_m\}$. Thus each literal $l_j^i = x_k$ or $\neg x_k$ where $i \in [1,\ n]$, $j \in [1,\ 3]$ and $k \in [1,\ m]$.

We construct a system of obligations within the context of mini-ARBAC. This system will be weakly accountable if and only if S is not satisfiable. The system has two administrative roles, ar_1 and ar_2. For every disjunct d_i of S we have a corresponding role r_{d_i} and for every variable x_k of S we have an associated role r_{x_k} in the system. Along with these roles, the system has two more roles which are important to us r' and r_{goal}.

We assume that the system has three users, u_0, u_1 and u_2. The initial user-role assignment in the system is as follows:

$$\begin{gathered} \mathit{user\_assignment}(u_0,\ \emptyset) \\ \mathit{user\_assignment}(u_1,\ \{ar_1\}) \\ \mathit{user\_assignment}(u_2,\ \{ar_2\}) \end{gathered}$$

We ignore the role-permission assignment rules, as it has no impact on the argument. Thus the policy rules are as follows:

$$\begin{gathered} \mathit{can\_assign}(ar_1,\ true,\ r_{x_1}) \\ \vdots \\ \mathit{can\_assign}(ar_1,\ true,\ r_{x_m}) \\ \mathit{can\_revoke}(ar_1,\ r_{x_1}) \\ \vdots \\ \mathit{can\_revoke}(ar_1,\ r_{x_m}) \end{gathered}$$

We also add one *can_assign* rule for each l_j^i, the form of which depends on the form of l_j^i as follows:

$$\mathit{can\_assign}(ar_1,\ r_{x_k},\ r_{d_i}) \qquad \mathit{when}\ l_j^i\ =\ x_k \tag{1}$$
$$\mathit{for\ some}\ k \in [1,\ m],\ j \in [1,\ 3]\ \mathit{and}\ i \in [1,\ n].$$

$$\mathit{can\_assign}(ar_1,\ \neg r_{x_k},\ r_{d_i}) \qquad \mathit{when}\ l_j^i\ = \neg x_k \tag{2}$$
$$\mathit{for\ some}\ k \in [1,\ m],\ j \in [1,\ 3]\ \mathit{and}\ i \in [1,\ n].$$

$$\mathit{can\_assign}(ar_1, \bigwedge_{i=1}^{n} r_{d_i}, r') \tag{3}$$

$$\mathit{can\_assign}(ar_1, \neg r', r_{goal}) \tag{4}$$

$$\begin{gathered} \mathit{can\_assign}(ar_2,\ true,\ r_{d_1}) \\ \vdots \\ \mathit{can\_assign}(ar_2,\ true,\ r_{d_n}) \\ \mathit{can\_assign}(ar_2,\ true,\ r_{x_1}) \\ \vdots \\ \mathit{can\_assign}(ar_2,\ true,\ r_{x_m}) \\ \mathit{can\_revoke}(ar_2,\ r_{x_1}) \\ \vdots \\ \mathit{can\_revoke}(ar_2,\ r_{x_m}) \end{gathered}$$

The system currently has some pending obligations where each obligation is of form $(action,\ [t_s,\ t_e])$. Intuitively, the first ones nondeterministically select a truth assignment.

$$\begin{aligned} b_{1,\ t} &= (u_1\ \mathit{Grants}\ r_{x_1}\ \mathit{to}\ u_0,\ [1,\ 2]) \\ b_{1,\ f} &= (u_1\ \mathit{Revokes}\ r_{x_1}\ \mathit{from}\ u_0,\ [1,\ 2]) \\ &\vdots \\ b_{m,\ t} &= (u_1\ \mathit{Grants}\ r_{x_m}\ \mathit{to}\ u_0,\ [1,\ 2]) \\ b_{m,\ f} &= (u_1\ \mathit{Revokes}\ r_{x_m}\ \mathit{from}\ u_0,\ [1,\ 2]) \end{aligned}$$

For each literal l_j^i we add one obligation. The time interval of this obligation depends on whether the literal is positive or negative, which assists in ensuring the obligations can be fulfilled during the cleanup phase.

$$b_{l_j^i}\ =\ \begin{cases} (u_1\ \mathit{Grants}\ r_{d_i}\ \mathit{to}\ u_0,\ [3,\ 21]) & \text{if } l_j^i = x_k \\ (u_1\ \mathit{Grants}\ r_{d_i}\ \mathit{to}\ u_0,\ [3,\ 31]) & \text{if } l_j^i = \neg x_k \end{cases}$$

The time line during which all the obligations in the system should be fulfilled can be divided into 6 time periods as shown in Table 5.

$$\begin{aligned} b^* &= (u_1\ \mathit{Grants}\ r'\ \mathit{to}\ u_0,\ [5,\ 31]) \\ b &= (u_1\ \mathit{Grants}\ r_{goal}\ \mathit{to}\ u_0,\ [10,\ 15]) \\ b_{d_1}^{cleanup} &= (u_2\ \mathit{Grants}\ r_{d_1}\ \mathit{to}\ u_0,\ [16,\ 20]) \\ &\vdots \\ b_{d_n}^{cleanup} &= (u_2\ \mathit{Grants}\ r_{d_n}\ \mathit{to}\ u_0,\ [16,\ 20]) \\ b_{x_1}^{cleanup} &= (u_2\ \mathit{Grants}\ r_{x_1}\ \mathit{to}\ u_0,\ [16,\ 20]) \\ &\vdots \\ b_{x_m}^{cleanup} &= (u_2\ \mathit{Grants}\ r_{x_m}\ \mathit{to}\ u_0,\ [16,\ 20]) \\ b_{\neg x_1}^{cleanup} &= (u_2\ \mathit{Revokes}\ r_{x_1}\ \mathit{from}\ u_0,\ [22,\ 30]) \\ &\vdots \\ b_{\neg x_m}^{cleanup} &= (u_2\ \mathit{Revokes}\ r_{x_m}\ \mathit{from}\ u_0,\ [22,\ 30]) \end{aligned}$$

The logic behind the reduction is as follows. The obligations $b_{k,\ t}$ and $b_{k,\ f}$ for $k \in [1,\ m]$ can always be executed before their deadline. Obligations $b_{d_i}^{cleanup}$, $b_{x_k}^{cleanup}$ and $b_{\neg x_k}^{cleanup}$ for $i \in [1,\ n]$, $j \in [1,\ 3]$ and $k \in [1,\ m]$ can also be executed before their stipulated deadline enabling b^* and each $b_{l_j^i}$ to be carried out eventually. Now whether b can be carried out depends on how it is scheduled with respect to obligation b^*. If b^* is performed before b then b can not be executed because b^* adds u_0 to a role that prevents u_0 to receive the role granted in b. We show that S is satisfiable if and only if there exist a schedule in which b^* is performed before b; resulting the system being in weakly unaccountable state.

For the *only if* part, suppose the 3-SAT problem is satisfiable. We construct a schedule that demonstrates the system is not weakly accountable. Such a schedule begins in period 1 by selecting an assignment that satisfies the 3-SAT problem, representing this in the assignment of each r_{x_k} to u_0 just in case x_k is true. Then, in period 2, u_1 grants each r_{d_i} to u_0, as permitted by statements (1) and (2). In period 3, u_1 grants r' to u_0, as permitted by statement (3). Finally, in period 4, u_1 is unable to fulfill obligation b due to statement (4)

For the *if* part of our reduction, we show the converse. Suppose there is no satisfying assignment for S. In this case, there is no scheduling of obligations $b_{1,t} \cdots b_{m,t}$ and $b_{1,f} \cdots b_{m,f}$ such that every $b_{l_j^i}$ can be performed before time 14. Thus, b^* can not be performed before time 15. This ensures b can execute in every schedule. The pending obligations at time 15 are then each $b_{l_j^i}$ and b^*. By time 20, u_2 must grant each r_{d_i} to u_0, enabling u_1 to fulfill b^*. Also by time 20, u_2 must grant each r_{x_k} to u_0, enabling u_1 to fulfill each obligation of form $(u_1\ \mathit{Grants}\ r_{d_i}\ \mathit{to}\ u_0,\ [3,\ 21])$. Then, by time 30, u_2 must revoke each r_{x_k} from u_0 enabling u_1 to fulfill each obligation of form $(u_1\ \mathit{Grants}\ r_{d_i}\ \mathit{to}\ u_0,\ [3,\ 31])$. Thus all the obligations can be carried out, resulting the system being in a weakly accountable state. We can say that, when there is no satisfying assignment for S, the system is in weakly accountable state.

The above reduction shows that checking weak accountability in a system of obligations governed by mini-ARBAC policy is *co-NP Hard*. Now we want to show that checking weak accountability in such a system is in *co-NP*. A nondeterministic algorithm to identify systems that are not weakly accountable begins by guessing a schedule of the obligations indicating for each obligation when it is to be carried out, then we can check if all the obligations of the system are present in the schedule. Then for each obligation, we check if it authorized. If it is authorized, we simulate the behavior of the obligation and update the authorization state of the system. Otherwise we check if the obligation is in its last time. If it is, we stop and declare the system is not weakly accountable. So given a schedule of the obligations, we can

Period	Interval	Activity
1	1...2	Nondeterministically select a truth assignment. T he represented assignment makes x_k true just in case u_0 has role r_{x_k}.
2	3...4	It is possible to grant r_{d_i} to u_0 during this period just in case one of the literals in d_i is true under the selected assignment.
3	5...9	It is possible that u_0 gets r' during this period (via obligation b^*) just in case each disjunction in the 3-SAT problem is satisfied.
4	10...15	u_1 is unable to fulfill obligation b just in case u has been assigned r' in period 3 (via obligation b^*).
5	16...21	In this period, u_2 grants each of the r_{d_i}'s to u_0 (via obligation $b^{cleanup}_{d_i}$) enabling obligation b^* to be fulfilled at its last time period if b^* has not been carried out before. Moreover u_2 also grants each r_{x_k} to u_0 (via obligation $b^{cleanup}_{x_k}$) enabling the obligation $b_{l^i_j}$ where $l^i_j = x_k$ to be fulfilled in time 21 if it has not been fulfilled before.
6	22...31	Here u_2 revokes each r_{x_k} from u_0 (via obligation $b^{cleanup}_{\neg x_k}$) enabling the obligation $b_{l^i_j}$ where $l^i_j = \neg x_k$ to be fulfilled in time 31 if it has not been fulfilled before.

Table 5: Time periods in the proof of Theorem 7.

check if the schedule is a valid one in $\mathcal{O}(n)$. Thus checking weak accountability in a system of obligations governed by mini-ARBAC is *co-NP Complete.* □

C. PROOF OF CORRECTNESS OF SPECIAL-PURPOSE WEAK ACCOUNTABILITY ALGORITHM

We write $b_1 \sim b_2$ if there is a dependency between obligation b_1 and obligation b_2. We write $\approx$ for the reflexive transitive closure. As $\sim$ is also symmetric, this is an equivalence relation. We say obligation set B_1 and B_2 are independent ($B_1 \not\approx B_2$) if $\neg(\exists b_1 \in B_1).(\exists b_2 \in B_2).(b_1 \approx b_2)$. We refer to counter examples of weak accountability as those weakly authorized schedules that are not authorized.

THEOREM 8. *Let $\{B_1, B_2\}$ partition a pending set of obligations, B, such that $B_1 \not\approx B_2$. There is a counter example of weak accountability for the obligation set B_1 (or B_2 or both) if and only if there is a counter example for B.*

PROOF. $\Rightarrow$ direction :

Without loss of generality, we will show that if there is a counter example of weak accountability within obligations of set B_1 then we can find a counter example of weak accountability in obligation set B.

Thus we can fix, σ^{B_1} to be a counter example of weak accountability for obligation set B_1 and let b^* refer to the obligation in the schedule σ^{B_1} that is not authorized. On the other hand, for obligation set B_2 we chose any weakly authorized schedule σ^{B_2}. We next construct an interleaving σ^B of σ^{B_1} and σ^{B_2}, and show that it is valid, weakly authorized, and b^* is not authorized in σ^B.

We start with σ^{B_2} and, starting with $\sigma^{B_1}_1$, add each element $\sigma^{B_1}_k$ to the schedule being constructed in the least position that satisfies the following properties :

1. If $k > 1$, it is after $\sigma^{B_1}_{k-1}$
2. It is placed after each $\sigma^{B_2}_o$, if any such o exists, such that $\sigma^{B_2}_o$.end $\leq \sigma^{B_1}_k$.start
3. If $\sigma^{B_1}_k$ is not authorized in σ^{B_1}, it is placed after each $\sigma^{B_2}_m$ if any such m exists, such that $\sigma^{B_2}_m$.end $< \sigma^{B_1}_k$.end

We now use induction on k to show that the schedule of $B_2 \cup \{\sigma^{B_1}_i \mid 1 \leq i \leq k\}$ constructed at stage k is valid and weakly authorized. The base case follows immediately from assumptions on σ^{B_2}. For the step, recall that the requirement of validity of a schedule ensures that each unordered pair of obligations is ordered in a manner that respects their time intervals. The induction assumption that the schedule of $(B_2 \cup \{\sigma^{B_1}_i \mid 1 \leq \ i \leq k-1\})$ that is constructed at stage $k-1$ is valid. Thus, when $\sigma^{B_1}_k$ does not occur in the pair the validity constraint is satisfied.

Let us consider the pairs that include $\sigma^{B_1}_k$. For validity, we must show that each obligation b' scheduled after $\sigma^{B_1}_k$ satisfies $\sigma^{B_1}_k$.start $< b'$.end and each obligation, b, scheduled before $\sigma^{B_1}_k$ satisfies b.start $< \sigma^{B_1}_k$.end. For the former, requirement (2) ensures validity for all such pairs of obligations b' and $\sigma^{B_1}_k$. For the latter, when $b \in B_1$, this follows from requirement (1) and the assumption that σ^{B_1} is valid. For the case in which $b \in B_2$, consider the last $b' \in B_2$ in the constructed schedule that precedes $\sigma^{B_1}_k$. The $b'' \in B_1$ that immediately follows b' must satisfy one of two cases : either (a) b'.end $\leq b''$.start or (b) b'.end $< b''$.end and b'' is not authorized in σ^{B_1}. From validity of σ^{B_2} (or from well-formednesss, if $b = b'$), we have b.start $< b'$.end. In case (a), we use the fact that b''.start $< \sigma^{B_1}_k$.end follows from the validity of σ^{B_1}, or the well-formedness when $b'' = \sigma^{B_1}_k$. In case (b), we use the fact that σ^{B_1} is weakly authorized and b'' is unauthorized in σ^{B_1} to show that b''.end $< \sigma^{B_1}_k$.end.

To show the schedule is weakly authorized, we must show that

1. If $\sigma^{B_1}_k$ is not authorized in σ^{B_1} then it comes after any of $\sigma^{B_2}_m$ such that $\sigma^{B_2}_m$.end $< \sigma^{B_1}_k$.end
2. If some $b \in B_2$ is not authorized in σ^{B_2} and b comes

before $\sigma_k^{B_1}$, then b.end $\leq \sigma_k^{B_1}$.end

For (1), we can see that requirement (3) achieves exactly what is required by it.

For (2), we again consider b' and b'' as in the previous part of the proof and the two cases, (a) and (b). In both cases, because b is unauthorized in σ^{B_2}, yet σ^{B_2} is weakly authorized, we have

$$b.\text{end} \leq b'.\text{end} \tag{5}$$

case a: b'.end $\leq b''$.start. Here we use the fact that σ^{B_1} is valid to obtain b''.start $< \sigma_k^{B_1}$.end, which yields the desired result, b.end $\leq \sigma_k^{B_1}$.end. case b: b'.end $< b''$.end and b'' is not authorized in σ^{B_1}. Now because σ^{B_1} is weakly authorized, we obtain b''.end $\leq \sigma_k^{B_1}$.end completing this case as well.

We now show that b^* is not authorized in σ^B. Suppose for contradiction that it were authorized. In this case, we show that ther must be some obligation $b^\diamond \in \sigma^{B_2}$ that influenced the authorization state of obligation b^* to make it become authorized during the execution of obligation set B. If $b^\diamond \in \sigma^{B_1}$ then during the execution of σ^B, $b^\diamond$ will not be authorized. As $b^\diamond \in \sigma^{B_2}$, $b^\diamond$ is not authorized during the execution of σ^{B_1} whereas it is authorized during execution of σ^B.

There are several scenarios which can arise when $b^\diamond$ can influence the authorization state of b^*. Let us investigate one of the possible scenarios, when b^* is not authorized during the execution of obligation set B_1 according to the initial authorization state. There is no previous obligation in B_1 which grants the necessary permissions to obligation b^* resulting it to be not authorized. During the execution of obligation set B, $b^\diamond \in B_2$ can grant $b^* \in B_1$ the necessary permissions resulting b^* to become authorized during the execution of B.

There is another possible scenario that arise during execution of obligation B by which $b^\diamond$ can influence the authorization state of b^*. In the scenario, b^* is authorized according to the initial authorization state during the execution of B_1 but there is a previous obligation $b_r \in B_1$ which revoked the necessary permissions from b^* resulting it to be unauthorized. During the execution of B, $b^\diamond$ can grant the permission revoked by b_r after b_r resulting b^* to be authorized or $b^\diamond$ can grant some other permission which can enable b^* to be authorized using a different policy rule.

According to our definition of dependence, in all the scenarios described above the obligations $b^\diamond$ and b^* are dependent ($b^\diamond \sim b^*$) so $B_1 \approx B_2$ which is a contradiction to our assumption of $B_1 \not\approx B_2$.

Thus, when there is a counter example of weak accountability within obligations of set B_1 then there is a counter example of weak accountability present in B.

$\Leftarrow$ direction:

If there is a counter example of weak accountability within obligations of set B then there is a counter example present in either set B_1 or B_2 or both.

We assume that obligation set B is not weakly accountable and σ^B be a counter example for B. Let b^* be the obligation which is not authorized according to σ^B. For $i \in \{1, 2\}$, let σ^{B_i} enumerate B_i in the same order as it is enumerated by σ^B. Without loss of generality, we also assume $b^* \in B_1$. We show that σ^{B_1} is a counter example. If it is not then there exists an obligation $b^\diamond \in B_2$ which influenced the authorization state of obligation b^* during execution in obligation set B. Several scenarios can arise where b^*'s authorization state can be influenced by $b^\diamond$. One possible scenario is when b^* is authorized according to the initial authorization state but $b^\diamond$ scheduled before b^* revokes the necessary permissions from b^* resulting it to be unauthorized during execution of B but as $b^\diamond \notin B_1$, b^* is authorized during execution of B_1.

Another possible scenario arises when b^* is authorized according to the initial authorization state. But there is another obligation $b^\square \in B_1$ scheduled before b^* and $b^\diamond \in B_2$, that grants the necessary permissions making b^* to be authorized. But $b^\diamond$ scheduled before b^* and after $b^\square$, revokes the necessary permission from b^* resulting it to be unauthorized eventually during execution of B. As $b^\diamond \notin B_1$, b^* becomes authorized during execution of B_1.

Thus according to our definition of dependence between obligations, in all the scenarios mentioned here $b^\diamond$ and b^* can not be independent ($b^\diamond \sim b^*$) resulting obligation B_1 and B_2 to be dependent ($B_1 \approx B_2$), which is a contradiction to our assumption of $B_1 \not\approx B_2$.

Thus when there is a counter example of weak accountability within obligations of set B then there is a counter example present in either set B_1 or B_2 or both.

□