On the Modeling and Analysis of Obligations

Keith Irwin
North Carolina State University
kirwin@ncsu.edu

Ting Yu
North Carolina State University
yu@csc.ncsu.edu

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

Abstract

Traditional security policies largely focus on access control requirements, which specify who can access what under what circumstances. Besides access control requirements, the availability of services in many applications often further imposes obligation requirements, which specify what actions have to be taken by a subject in the future as a condition of getting certain privileges at present. However, it is not clear yet what the implications of obligation policies are concerning the security goals of a system.

In this paper, we propose a formal metamodel that captures the key aspects of a system that are relevant to obligation management. We formally investigate the interpretation of security policies from the perspective of obligations, and define secure system states based on the concept of *accountability*. We also study the complexity of checking a state's accountability under different assumptions about a system.

Categories and Subject Descriptors: K.6.5 [Management of Computing and Information Systems]: Security and Protection

General Terms: Security, Theory

Keywords: Policy, Obligations

1 Introduction

Security policies are widely used in the management of sensitive information and valuable resources in a variety of applications and systems. Traditional security policies largely focus on the specification and management of access control requirements, i.e., what principals are allowed to access what objects and when. Besides controlling principals' privileges, the availability of services as well as the correct operation of a system often imposes obligation requirements which specify what actions a subject is *obliged* to perform in the future in order to allow certain actions to be taken at present. For example, if a customer is allowed to subscribe to a certain service, then she is obliged to pay the subscription fee by the end of each month. In some situations, one's obligations may also result from the action performed by others instead of by oneself. For example, in a conference reviewing system, once a paper is assigned by the program chair to a reviewer, the reviewer is obliged to submit her review before a certain deadline.

Obligation requirements have traditionally been hard-coded into applications. But recently, obligations are increasingly being expressed explicitly as part of security policies. As is the case for many policy-based systems, this approach offers advantages in terms of flexible management and easy maintenance of obligations. It allows a system to change obligation requirements quickly when flaws in existing policies are found, to react to new circumstances promptly, and to accurately enforce complex obligation requirements. In particular, supporting obligations in policies can be an important part of translating high level security goals into low level policies.

As traditional access control policies are only concerned with permitting or denying subjects the ability to take certain actions, they cannot be used directly to express obligation requirements. Recently, several security policy languages have been proposed to support the specification of obligation requirements [9, 12, 15, 18]. Works have also been done on monitoring the fulfillment of obligations [4, 5].

The introduction of obligations inevitably complicates the management of security policies. Tradition-

ally, a reference monitor of a system only needs to determine whether an access request should be allowed, purely from an access control perspective. When obligations are added, it is not immediately clear how they should affect the reference monitor's decision. In other words, the relationship between access control and obligations is not yet well studied. Further, existing work on obligations focuses on the responsibility of the subjects who receive obligations. However, to ensure the correct operation of a system, a reference monitor should not blindly assign obligations to subjects and eventually check whether they have been fulfilled. Instead, a system should only allow obligations to be assigned when the receiving subject will have sufficient privileges as well as other resources in the system to successfully fulfill the obligation. That is, a diligent user should always be able to fulfill her obligations. Otherwise, the obligation should be not assigned in the first place.

Conceptually, a security policy defines what the secure states are for a system, and the job of a reference monitor is to ensure the system stays in a secure state and prevent it from transitioning into insecure states. With obligations introduced in security policies, the questions of how to define secure states and how to ensure the security of a system, to the best of our knowledge, have not yet been adequately investigated. These are the focus of this paper. More specifically, the contributions of this paper are as follows.

1. We abstract common system components that are relevant to the management of obligations, and propose a formal metamodel to capture a system and its possible states from the perspective of obligations.
2. Built on the above metamodel, we propose a formal definition of secure states for obligation management. Our definition is based on the concept of "accountability". Intuitively, considering subjects as autonomous entities, a system cannot prevent a subject from failing to fulfill its obligations. Violation of obligations may not be avoidable. However, once an obligation is violated, the system should be able to clearly identify who is responsible. In this paper, we view an obligation as a contract between a system and a subject. A system is accountable if and only if all the obligations will be fulfilled supposing all the subjects are diligent. In other words, a secure state implies that any obligation violation can only be due to the lack of diligence of subjects.
3. We further study the problem of checking whether a state is accountable. We identify a set of conditions, which, when satisfied, allow the efficient checking of a state's accountability. We also show that when some of the conditions do not hold, the problem becomes intractable in the worst case, when only considering the abstract construct of the metamodel.
4. We study the accountability problem in the context of a simple yet expressive authorization system with obligations, and show the tractability of the accountability checking problem even though the system does not satisfy the above identified conditions.

The rest of the paper is organized as follows. In section 2, we propose a set of criteria for the specification and management of obligations, and further elaborate the problem addressed in the paper. Section 3 presents a formal metamodel for systems with obligations as well as obligation policies. In section 4, we formally define the concept of accountability in terms of state transitions. In section 5, we study the problem of determining a state's accountability, based on the proposed model. In section 5, we show that the accountability problem can be efficiently solved in an authorization system with obligations. We report closely related work in section 6, and conclude this paper in section 7.

2 Properties of Obligations

An obligation is a requirement for a subject to take some action at some time in the future. Such obligations are sometimes also referred to as positive obligations, in contrast to "negative obligations" (or *refrainments*), where a subject is required not to take some action at some time in the future. For example, in privacy-enhanced systems, one common negative obligation is that a company should not share users' private information without their consent. In this paper, we focus on positive obligations. Although we believe that our model could handle negative obligations, we do not include them because in centralized systems negative obligations can be easily transformed into access control requirements, and thus can be enforced directly by the reference monitor. Positive obligations, on the other hand, cannot be enforced by a system in a direct way. After all, if a system could cause an action to happen, then it would no longer be a user action, but instead a sys-

tem action. Negative obligations are more necessary in distributed systems, a topic we will explore in later papers.

Unenforcable. As mentioned above, an action required by an obligation cannot be forced to happen by a system. For example, a system cannot force a subscriber to pay his fee on time. Similarly, a program chair cannot ensure that every reviewer will submit their reviews before the deadline. This is an essential difference from access control requirements, which must be enforcable by the systems.

Monitorable. Though a system cannot force an obligation to be fulfilled, it should be able to monitor the status of an obligation, i.e., whether it has been fulfilled. Although it is conceivable that a system could give a user an obligation which it cannot monitor, such obligations are clearly not relevant to any decisions or analysis which the system undertakes.

If an obligation is to be feasibly monitored, there are two conditions which must be met. The first is that it must be clear when an obligation has been fulfilled. And equally importantly, it must be clear when a user has failed to fulfill an obligation with which he has been charged. One implication of this requirement is that an obligation must have some sort of deadline before which it must be accomplished. Otherwise there is no point at which it can be said that a user has violated his obligation. As such, there is no real motivation for users to fulfill their obligations. Therefore, a time window should be an intrinsic property of an obligation.

Obligations arise in a system when a user is allowed to take an action with the condition that they also accept an obligation or when a user's action causes someone else to receive an obligation. For instance, a policy may specify that users are allowed to run certain tests, but only if they agree to submit a report about the results of those tests within a week afterwards. Or a policy might say that employees are allowed to submit vacation requests, but when they do, this gives the human resources department an obligation to review those requests and take action on them within a week.

3 A Metamodel for Systems with Obligations

In this section, we present a formal metamodel that encompasses the basic constructs of a system from the perspective of obligations. The model serves as the foundation for our later discussion on the accountability of states and checking of accountability.

First let us discuss how we model obligations themselves. An obligation is an action which some subject must carry out during some timeframe. Thus, we model an obligation as a tuple $obl(s, a, O, [t_s, t_e])$, where s is a subject, a is an action, $[t_s, t_e]$ is a time window during which s is obliged to take action a, and O is a finite sequence of zero or more objects on which the action must be performed.

An obligation system consists of the following components:

- $\mathcal{T}$: a countable set of time values. For simplicity we take $\mathcal{T}$ to be the non-negative integers, with 0 indicating the system start time and each value indicating a point in time after that by a multiple of some appropriate, unspecified time interval.
- $\mathcal{S}$: a set of subjects that could be added to the state of the system.
- $\mathcal{O}$: a set of objects with $\mathcal{S} \subseteq \mathcal{O}$.
- $\mathcal{A}$: a finite set of actions that can be initiated by subjects. Each action's behavior is given by a function that takes as input the current system state (defined just below), the subject performing the action, and a finite sequence of zero or more objects. It outputs a new system state (except for the new time which is managed separately).
- $\mathcal{B} = \mathcal{S} \times \mathcal{A} \times \mathcal{O}^* \times \mathcal{T} \times \mathcal{T}$: a set of obligations that subjects can incur. Given an obligation $b \in \mathcal{B}$, we use $b.s$ to refer to the subject that is obligated, $b.a$ for the action the subject is obligated to perform, $b.O$ for the finite sequence of zero or more objects that are parameters to the action, and $b.t_s$ and $b.t_e$ to refer to the start and end, respectively, of the period in which the subject is obligated to perform the action.
- $\mathcal{ST} = \mathcal{T} \times \mathcal{FP}(\mathcal{S}) \times \mathcal{FP}(\mathcal{O}) \times \Sigma \times \mathcal{FP}(\mathcal{B})$: the set of system states. Here, 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. We use $st = \langle t, S, O, \sigma, B \rangle$ to denote systems states, where t is the time in the system, B is the set of pending obligations, and σ is a fully abstract representation of all other features

of the system state. Σ is possibly infinite. We use $st_{cur} = \langle t_{cur}, S_{cur}, O_{cur}, \sigma_{cur}, B_{cur} \rangle$ to denote the current state of the system.

- $\mathcal{P}$: a set of policy rules. Each policy rule specifies an action that can be taken, under what circumstances it may be taken, and what obligations (if any) results from that action. We denote policy rules of this type by using the notation $a(st, s, O) \leftarrow cond : F_{obl}$, where $a \in \mathcal{A}$ and *cond* is a predicate in $\mathcal{S} \times \mathcal{T} \times \Sigma \times \mathcal{O} \rightarrow \{true, false\}$, indicating that subject s is authorized to perform action a on objects O at time t with the system in state σ if $cond(s, t, \sigma, O)$ is true. F_{obl} is an *obligation function*, which takes the current state of the system σ, the current time, the subject s, and the arguments O as its input and outputs a finite set $B \subset \mathcal{B}$ of obligations caused by the action. Note that obligations in B may not necessarily be incurred by the same subject. If the action a is taken, then all the obligations in B should be fulfilled.

  We allow multiple action rules to have the same action as their heads. As long as the condition of one rule is true, the action is allowed, causing the obligations resulting from that rule to be incurred.

  As such, action rules can be used to express obligation requirements where a subject has a choice to take one of several actions to fulfill its obligation. We can simply have several rules with the same head and condition but with different obligations.

  The conditions in action rules essentially model access control policies of a system. Here we adopt a closed world assumption. If none of the conditions of the action rules for an action is true, then a subject is not allowed to take that action.

An obligation $obl(s, a, O, [t_s, t_e])$ may be in one of four states: *invalid*, *pending*, *fulfilled*, or *violated*. If it is the case that the t_e is already passed when it is assigned, then the obligation is *invalid*, as it, on its face, can clearly not be carried out. If an obligation has been assigned and its action has been carried out during the time window $[t_s, t_e]$, then it has been *fulfilled*. If it has been assigned, has not been *fulfilled*, and is not *invalid*, but t_e has passed, then it is *violated*. If an obligation is not *invalid* but has not yet become *fulfilled* or *violated*, then it is *pending*.

3.1 State Transition

As indicated above, we assume system time is discrete, and model it as a non-negative integer representing the number of clock ticks from some predetermined start time. For simplicity, we assume that each action can be finished in a single clock tick, and its effect will be reflected in the state of the next clock tick. Suppose the state of a system at time t_0 is st_0, and Alice takes an action at t_0. This action will not change st_0. Instead the state st_1 at time $t_0 + 1$ will be affected by Alice's action. When multiple actions are attempted at the same time , we assume the system uses a function f_{trans} to get the state of the next clock tick. More specifically, f_{trans} takes the current state of the system (which also includes the current time) and a set of actions attempted by subjects, and returns for the next clock tick the state obtained by applying the permitted actions in some unspecified, fixed order which has the property that at the point each action is taken the state satisfies a policy rule which allows that action. In other words, we adopt a deterministic model of state transition; from the current state and the actions taken at present, we can uniquely determine the next state.

In fact, since an action may have multiple action rules associated with it, when a subject takes an action, it may choose different rules to authorize the action. This may result in different obligations assigned . Thus, when describing state transitions, besides specifying what actions subjects take, it is also necessary to indicate the policy rules applied for these actions. Let AP be a set of tuples (s, a, r), where s is a subject, a is an action and r is a policy rule for a. (AP stands for "action plan".) Let st be a state at time t. We use $st \vdash_{AP} st'$ to denote that st has transitioned to state st' at time $t + 1$ after actions in AP have been taken at time t. Legal state transitions must have the property that for each $(s, a, r) \in AP$, the condition of r is true at the state that is current when (s, a, r) is reached in the unspecified, fixed order in which members of AP are performed by f_{trans}. If $AP = \emptyset$, then st' corresponds to the system state when the time is incremented by one clock tick, but no other component of the system state is changed.

Definition 1. *We say $st \vdash_{AP} st'$ is an* obligation-abiding transition*, if (1) there are no two tuples (s_1, a_1, r_1) and (s_2, a_2, r_2) in AP such that $s_1 = s_2$ and $a_1 = a_2$; and (2) for any $(s, a, r) \in AP$, there exists a pending obligation $(s, a, [t_s, t_e])$ in $st.B$. An obligation-abiding transition is* valid *if no pending obligations in st become violated in st'.*

An obligation-abiding transition corresponds to the system evolution where subjects take actions

only to fulfill their obligations. A sequence of valid obligation-abiding transitions corresponds to the situation where subjects are diligent and always fulfill their obligations. Note that although a transition in which no users take any actions will be guaranteed to be obligation-abiding, it will not be guaranteed to be valid.

3.2 An Example Obligation System

We use a simple conference reviewing system as an example to show how a system with obligations can be represented in the above metamodel.

In this system, after collecting submitted papers, the program chair of a conference assigns papers to reviewers. Once the assignment is done, each reviewer is obliged to submit their reviews by a certain deadline. A reviewer can also submit reviews for papers not assigned to them. If a reviewer submits a review for a paper, she is obliged to attend the discussion of the paper, which decides whether the paper should be accepted.

This system can be modeled as follows.

- Subjects s are the registered users in the system.
- Objects o are submitted papers (and the subjects).
- The actions allowed in the system include assigning papers to reviewers, submitting a review and joining discussion of a paper.
- The σ-portion of the system state is no longer fully abstract, but instead represents attributes of subjects and objects. For instance, the set of roles of subject s is given by $\sigma.roles(s)$.

The policy of the system may be the following:

- $assign\_reviewer(st, s_1, \{s_2, o\}) \leftarrow$
  $prog\_chair \in st.\sigma.roles(s_1) \wedge$
  $reviewer \in st.\sigma.roles(s_2) \wedge$
  $st.\sigma.name(s_2) \notin st.\sigma.author\_list(o) :$
  $\{obl(s_2, submit\_review(s_2, o),$
  $[06/01/06, 07/15/06])\}.$
- $submit\_review(st, s, \{o\}) \leftarrow$
  $reviewer \in st.\sigma.roles(s) \wedge$
  $st.\sigma.name(s) \notin st.\sigma.author\_list(o) :$
  $\{obl(s, discuss(s, o), [07/22/06, 07/22/06]),$
  $obl(s, vote(s, o), [07/23/06, 07/23/06])\}.$
  (Once a reviewer submits a review of a paper, he or she is added into the reviewer list of the paper.)
- $discuss(st, s, \{o\}) \leftarrow reviewer \in st.\sigma.roles(s) \wedge$
  $s.name \notin st.\sigma.author\_list(o) : \emptyset.$
- $vote(st, s, \{o\}) \leftarrow$
  $s.name \in st.\sigma.reviewer\_list(o) : \emptyset.$

We do not show the semantics of each action since most of them are straightforward. Suppose on 06/01/06 the program chair assigns Alice to review papers p_1, p_2 and p_3. Then three pending obligations are added into the system: $obl_1 =$ obl(Alice, $submit\_review$(Alice, p_1), [06/01/06, 07/15/06]), $obl_2 =$ obl(Alice, $submit\_review$(Alice, p_2), [06/01/06, 07/15/06]) and $obl_3 =$ obl(Alice, $submit\_review$(Alice, p_3), [06/01/06, 07/15/06]). On 07/10/06, Alice submits her review for paper p_1. Then in the state of the system on 07/11/06, the status of obl_1 becomes *fulfilled*, and a new pending obligation $obl_4 =$ obl(Alice, $discuss$(Alice, p_1), [07/23/06, 07/23/06]) is created. Note that the transition from the state of the system on 07/10/06 to that on 07/11/06 is obligation-abiding, as the action Alice takes is required by one of her obligations. Also, since on 07/11/06 no pending obligations become *violated*, the transition is a valid obligation-abiding one.

On the other hand, suppose *Bob*, who is not assigned as one of the reviewers for p_1, is interested in the paper, and also submits a review for p_1 on 07/10/06. Then the transition is not obligation-abiding, since Bob's action is not required by any obligation.

Further, suppose Alice fails to submit a review for p_3 before 07/15/06. Then the transition from the state on 07/15/06 to that on 07/16/06 is still obligation-abiding according to our definition. However, it is not a valid one since obl_3 becomes *violated* on 07/16/06.

4 Security Goals in Systems with Obligations

Conceptually, a system's security policy divides system states into two disjoint sets: secure states and insecure states. The goal of security is to ensure that a system always stays in secure states and never transits into insecure states. Under the same principle, we study the question of how we should interpret a security policy which includes obligations, i.e., what states are considered secure under policies with obligations.

One straightforward approach is to define secure states as those that have no obligations being violated. Such states are certainly desirable. However, due to the unenforceable nature of obligations, a sys-

tem can never guarantee that an obligation will be fulfilled. Instead, it seems more appropriate for a system to ensure that all obligations *can* be fulfilled, in the sense that the obligated user has the necessary authorizations to perform the obligatory action. However, as we will see, this alone does not provide sufficiently clear guidance. Certainly if the system determines that it is impossible for a user to fulfill the obligation which would be incurred by his performing some requested action, then the system should deny that action. Likewise, if the system is certain that a user will have sufficient privileges to fulfill an obligation before its deadline, then it should allow the requested action. But what is the appropriate thing for the system to do if the ability of the user to perform the obligation depends on whether or not actions are taken to change his privileges?

Suppose there exists one sequence of actions or events which would cause the user to be unable to fulfill his obligations and another which would cause the user to be able to. One highly conservative response would be always to deny requests that would incur obligations that the obligated user might not be able to fulfill. However, in any system with a superuser, this would result in virtually every request being denied. In fact, so long as there existed any remedy which could take rights away from the user in any circumstance, all requests from that user for actions which carry obligations would be denied. One can imagine a scenario in which a CEO is denied the right to edit a file because it is theoretically possible that the board of directors could oust her from her position in the next five minutes.

Similarly, if one took the completely optimistic approach, the opposite scenario rears its head. So long as there exists some possible way that the user might be able to fulfill the incurred obligation, the requested action would be permitted, even if the events necessary for the user to have the required authorizations are highly unlikely. For example, Bob, a mailroom employee, could be allowed an action even though he could only fulfill the associated obligation if the board fired the CEO and hired Bob in his place within the next five minutes. As such, it is clear that neither strategy is acceptable.

In this paper, we offer the concept of *accountability*. Rather than requiring that it be impossible for obligations to be violated, instead we assume that it is possible that obligations go unfulfilled, but when they do, we would like to clearly identify whose fault it is. Obviously, an obligation can go unfulfilled because a subject simply fails to take the required action before the deadline, even if he has sufficient privileges and resources. It is desirable that this is the only reason that an obligation will go unfulfilled.

Intuitively, if it is the case that all users have sufficient privileges and resource to carry out their obligations so long as every other user carries out his or her obligation, then a system is said to be in an *accountable state*, because we can know that whoever first fails to carry out an obligation is responsible for the violation and anything which results from it. In this paper, we describe systems which attempt to maintain accountability, but we reserve for future work issues pertaining to assigning blame in more complex scenarios such as failures after the system has left an accountable state.

Note that when determining whether a state is accountable, we only consider those actions that are required to be taken by obligations. Although a user may actively take some actions which may interfere with another user's obligations, such actions can be controlled by a system. Once the system determines whether the resulting state will be accountable or not, it can take appropriate actions. For example, it may either prevent the user from taking the action, or it may discharge or change the interfered obligations so that the resulting state is still accountable. In this paper, we focus on the definition and checking of accountable states. We will briefly discuss the handling of actions that may lead a system into an unaccountable state.

Before we give a formal definition of accountable states, it is necessary to discuss in more detail what situation it is in which a user is deemed as failing to fulfill her obligation. Intuitively, when an obligation $(s, a, [t_s, t_e])$ is assigned, we can view it as a contract between a subject s and a system. From the subject's perspective, it has promised to take action a during the given time window. On the other hand, from the system's perspective, it also implicitly promises that if everybody else fulfills their obligations, then s should have the needed privileges and resources to take action a. Depending on the interpretation of obligations, we may have different definitions of accountability.

Here we consider two types of interpretations. In the first type, if everybody else fulfills their obligations, then a system guarantees that Alice can take action a at any time point during $[t_s, t_e]$. This is a very strong promise from the system, which means that the condition of the rule for action a is always

true for Alice during the obligation's time window.

In the second type, a system only promises that, if everybody else fulfills their obligations, then Alice can at least take action a at the end point t_e of the obligation's time window. Note that it does not mean that Alice has to take the action at t_e; it only means that in the worst case Alice will still be able to fulfill the obligation at t_e. Clearly, this type of promise is weaker than the first one, since it only requires that the condition of the rule for action a is true at t_e instead of during the whole time window of the obligation.

Generally speaking, the first type of accountability is suitable for systems in which users may have additional restrictions of which the system is not aware. If there are additional constraints on when a user is available to fulfill her obligations, then the system should ensure that the full time is available to the user so that whenever the user has the opportunity to fulfill one of her obligations, she has the necessary authorizations to do so. For instance, a system at a company with flexible work hours would probably prefer strong accountability since it would not know when employees would be available to fulfill their obligations. By contrast, in a more all-encompassing system, the weaker accountability is sufficient: since the system is aware of all scheduling constraints, whenever the user chooses to attempt to fulfill the obligation, either she will be able to fulfill it at that time, or the system will have ensured that she will again be available at a later time prior to the deadline. Hence, a system at a military base, where all users can be expected to be available precisely when the policies say that they should, might be able to use weak accountability, and derive benefit from the fact that it is a weaker requirement and therefore easier to ensure.

There is a third possible type of accountability, in which the system ensures only that there exists some time within the frame when the user will be able to fulfill his obligation. This type, however, is not likely to be generally suitable for ordinary users since it would require that the user discover that time before it passes. It may be suitable for systems with automated agents which could regularly poll the system to see if the obligation can be fulfilled. However, for reasons of space, we do not formalize or otherwise discuss the implications of that third type of accountability in this paper.

Next, we formally define accountable states for each of the first two interpretations. Recall from Definition 1 that a transition is valid and obligation-abiding if all the actions in its action plan are from existing obligations and no obligations are violated.

Definition 2. *Let st be a system state with time t and pending obligations $obl_1, \ldots, obl_n$. We say st is a* type-1 undesirable state *if there exists an obligation $obl_i = obl(s_i, a_i, [t_{si}, t_{ei}])$, $1 \leq i \leq n$, such that (1) the condition cond of each action rule for a_i is false for s_i, i.e., $cond(s_i, t, st.\sigma) = false$; and (2) $t_{si} \leq t \leq t_{ei}$.*

A state is type-1 undesirable if a subject cannot fulfill an obligation although the current time is within the time window of the obligation.

Definition 3. *A state st is* strongly accountable *if there exists no sequence of valid obligation-abiding transitions that lead st to a type-1 undesirable state.*

This definition of accountability corresponds to the first interpretation of obligations. For the second interpretation, we have the following definition.

Definition 4. *Let st be a system state with time t and pending obligations $obl_1, \ldots, obl_n$. We say st is a* type-2 undesirable state *if there exists an obligation $obl_i = obl(s_i, a_i, [t_{si}, t_{ei}])$, $1 \leq i \leq n$, such that (1) the condition cond of each action rule for a_i is false for s_i, i.e., $cond(s_i, t, st.\sigma) = false$; and (2) $t = t_{ei}$.*

A state is type-2 undesirable if the current time is the deadline of a pending obligation, which however cannot be fulfilled at present.

Definition 5. *A state st is* weakly accountable *if there exists no sequence of valid obligation-abiding transitions that lead st to a type-2 undesirable state.*

Obviously, if a state is strongly accountable, it is also weakly accountable. Let us consider the following scenarios. Suppose in state st at time 0 Alice does not have the privilege to read file f, but she has an obligation to read f between time 10 and time 20. Meanwhile, Bob, whose is the owner of f, has an obligation to grant the read privilege of f to Alice between time 5 and time 15. According to our definition, st is not strongly accountable, as it is possible that Alice decides to fulfill her obligation at time 12, but she cannot do so because she lacks the read privilege. In this case Bob is not to blame, since Bob can decide to fulfill his obligation at time 14 for example. In other words, st may possibly transit into a future state where a subject cannot fulfill its obligation, which however is not due to any subject's negligence.

On the other hand, *st* is weakly accountable. This is because Alice can always fulfill her obligation at time 20. If she still does not have the privilege to read f at time 20, it must be due to Bob's negligence.

5 The Accountability Problem

In this section, we study the *accountability problem*, i.e., given a state in a system, determining whether it is accountable. This problem is naturally faced by a reference monitor when a subject makes a request to take an action which, if allowed, would result in the assignment of obligations.

Note that since the constructs of our metamodel are very abstract, it can accommodate systems with arbitrarily complex internal structures. It is not hard to see that, purely based on the metamodel without any constraints on a system's properties, the accountability problem can easily be undecidable. In the following, we show a reduction of the halting problem to the accountability problem.

Given a Turing machine T, we can construct a system which emulates that Turing machine by specifying that the system state, σ, includes a potentially infinite tape, a position on that tape, a current machine state, and a boolean variable which describes whether or not the system has halted. Then we define a single action "Advance", which changes the state and the tape in accordance with the state transition rules of T and, if T halts, sets the halt variable to true. For purposes of simplicity, we define a single subject, s. To make the Turing machine operate, we define an initial obligation of $(s, Advance, [1, 1])$. Then we define a policy rule for Advance so that the state can be advanced so long as the machine has not halted and causes an obligation to advance the state again. Explicitly, the policy rule is $Advance(s) \leftarrow halt = false : f(\sigma, t, s) = (s, Advance, curr + 1, curr + 1)$, where *curr* denotes the current time of a system. As such, if the machine ever halts, there will be an obligation which cannot be fulfilled. But if it never halts, there will not be one. Therefore, the question of whether or not the state is unaccountable is equivalent to the question of whether or not the Turing machine will halt. As such, in the worst case, the accountability problem is undecidable, when only considering the constructs of the metamodel without any constraints.

To describe such a reduction is, of course, by no means to say that the accountability problem is undecidable in all obligations systems. In specific systems, determining accountability may often be quite easy. In particular, we are interested in identifying the properties of obligation systems which allow us to efficiently solve the accountability problem.

Let us consider obligation systems that satisfy the following conditions:

- No cascading obligations. In the metamodel, the action to fulfill an obligation may also incur further obligations. If a system does not have such cascading obligations, then each obligation only involves actions whose policies do not carry obligations.
- Monotonicity. From a given state, if the condition on a policy is true for a subject, it will remain true in all future states. In other words, the set of rules that a subject can satisfy does not decrease during state transitions. As such, the set of rules is monotonic relative to time.
- Commutative actions. For any two actions, a_1 and a_2 if the conditions of the policy rules of both a_1 and a_2 are met, then taking action a_1 followed by a_2 has the same effect on the system state as taking action a_2 followed by a_1.

These properties might seem a little draconian. But as we will show later, if we remove any one of them, without considering other specifics of a system, the accountability problem is intractable. Again, this does not mean that particular systems which do not have these properties cannot be efficiently solved, since any particular system would have additional properties which we do not assume here. In fact, in section 5 we will present a class of systems which do not conform to all of these assumptions, but do have an efficient algorithm for the accountability problem. With that said, any time we have a system where the above three properties hold, we can efficiently solve the accountability problem, without looking at other specifics of that system.

Theorem 1. *Given a system that satisfies the above three properties, the problem to check whether a given state is weakly accountable is tractable.*

Proof. Sketch. In a monotonic system, once an obligatory action becomes enabled, it remains so in later states. Thus, weak accountability in this context is equivalent to requiring that each obligatory action is enabled at the end of its time window. The scenario that will be most challenging with respect to enabling a given obligatory action is the one in which all other obligatory actions are taken at the latest

possible time, right before their deadlines. Therefore, we can evaluate a particular state by examining what happens when each obligatory action in that state is attempted at the obligation's deadline. If by so doing, every obligation can be fulfilled (presuming that all other obligations are fulfilled), then the state is weakly accountable. If there exists an obligation in the state for whose action there exists no policy rule whose condition is satisfied when the obligation's deadline has arrived, then the state is unaccountable.

Assuming it takes a constant time to check whether a condition is satisfied in a state, the complexity of the above algorithm is $O(nm)$, where n is the number of pending obligations in a state, and m is the number of action rules in the policy. □

We have a similar result for the checking of strong accountability.

Theorem 2. *Given a system that satisfies the above three properties, the problem to check whether a given state is strongly accountable is tractable.*

Proof. Sketch. In strong accountability, it must be the case that an obligation can be fulfilled at any point during its time frame. In a monotonic system, this is equivalent to being able to be fulfilled at the start of its time period. However, if we were to simply schedule all obligations for the start points of their time windows, we would not have a worst case scenario, since all the conditions would be fulfilled as soon as they possibly could.

Instead, we iterate through each obligation individually and consider whether or not it will be able to be fulfilled at the beginning of its time period assuming that the other obligations all happen as late as possible. We can do this by assuming that all obligations whose end time is before the current obligation's start time have occurred and seeing if the state will allow the current obligation to be fulfilled. If all obligations can be fulfilled under these conditions, then the state is accountable, elsewise it is unaccountable. The complexity of this algorithm is $O(n^2m)$, where n is the number of pending obligations in a state, and m is the number of action rules in the policy. □

If we remove one of the above three properties, then the accountability problem becomes intractable, when only given the constructs of the metamodel.

Theorem 3. *Given a system which only satisfies two of above three properties, i.e., the no cascading obligation, the monotonicity and commutative action properties, the problem of determining whether a state of the system is strongly/weakly accountable is intractable.*

Instead of attempting to reduce NP-complete problems to the accountability problem, it is in fact much easier to reduce them to the problem of checking of unaccountability, and as such we prove that under the above conditions, the accountability problem is Co-NP Hard. We present these three reductions in appendix A.

6 A Concrete Model

As we saw in the previous section, in the abstract model, determining the accountability of a system can be done efficiently provided several restrictions are placed on the system. In particular, one requirement was that actions perform state transitions that cause the set of enabled actions in the system to either increase (with respect to $\subset$) or stay the same. In the abstract model, this could not be relaxed without losing tractability. However in practice the restriction is unlikely to be satisfactory, as it means that once a subject is able to perform a given action, they will always be able to perform that action.

By contrast with our meta-model, in practice, permission states are structured objects. It turns out that entirely realistic assumptions about that structure enable us to remove the assumption that actions must increase while preserving tractability.

In this section we present a concrete model based on the HRU access matrix model. Specifically, Σ, the set of abstract states, is instantiated to be $\mathcal{M} = 2^{\mathcal{S}\times\mathcal{O}\times\mathcal{R}}$, the set of permission sets, in which $\mathcal{R}$ is a set of access rights subjects can have on objects. We denote permission sets by M and individual permissions by $m = (s, o, r)$. Each permission is a triple consisting of a subject, an object, and an access right, and signifies that the subject has the right on the object.

Actions are also modified so as to operate on permission sets. Each action $a \in \mathcal{A}$ is now assumed to perform a finite sequence of operations that each either add or remove a single permission from the permission set ($grant(m)$ and $revoke(m)$). Clearly, a subject or an object with no associated permissions has no effect on the system, so we assume that in every state st, an object or a subject exists in $st.O$ and/or $st.S$ if and only if it occurs in some permission in the permission set $st.M$.

As we show below, the following restrictions on the model make the problem of determining whether a state is accountable tractable.

1. Policy rule conditions consist of a Boolean combination of permission tests ($m \in M_{cur}$ or $m \notin M_{cur}$) expressed in conjunctive normal form.
2. Actions are partitioned into two sets—the first consists of actions whose policy rules can impose obligations and the second consists of actions that can occur in those obligations. This means that one cannot become obligated to take actions in the first set, but performing such actions voluntarily can incur obligations to perform actions in the second set. Performing these latter actions does not incur any obligations.

6.1 Algorithm for dealing with the simplified concrete model

Given a current state $st_{cur} = \langle t_{cur}, M_{cur}, B_{cur} \rangle$ that is known to be accountable, we can use the procedure below to determine whether or not adding a new obligation b leaves the system in an accountable state. Given a set of obligations B that need to be added, we can use the procedure by considering the elements of the set one at a time, adding each obligation in turn to B_{cur} unless it would leave the system in an unaccountable state, in which case we stop and return the result that B cannot be added while preserving accountability.

We are given an obligation $b = \langle b.s, b.a, b.t_s, b.t_e \rangle$. Let the policy rule that governs $a = b.a$ be $a \leftarrow cond(s, t, M)$. Under the restrictions identified above, $cond(s, t, M)$ is a Boolean combinations of permission membership tests expressed in conjunctive normal form (*i.e.*, a conjunction of disjunctions). The following steps are used to check each permission membership test in turn to see whether it is guaranteed. The condition is guaranteed to be true if and only if all of its conjuncts is true. Since any one conjunct (/ie, any one disjunction) is guaranteed to be true only if it is true in all possible schedules of obligations, we can know that we can evaluate the truth of each conjunction without regard to any other. That is, we are seeking to answer the question of whether or not there exists a schedule which makes any one of the conjuncts false.

So we examine each of the disjunctions individually, by initially examining all of the tests in a given disjunction individually. If it is guaranteed to be true, then we know that the disjunction is guaranteed to be true, and we can move on to the next disjunction. If we find that it is guaranteed to be false, then we know that we can disregard it. If however, we find that we do not know, we set it aside and later do a more complex test.

This more complex test is not in the conference version of this paper. It is necessary because it is possible that although it may be the case that some pair of rights r_1 and r_2 may each, individually, not be guaranteed to exist in all possible schedules, $r_1 \vee r_2$ could still be guaranteed. For instance, if there were an obligation which both revoked r_1 and granted r_2 and another obligation which both granted r_1 and revoked r_2 such that they had overlapping time intervals, then neither r_1 nor r_2 could be guaranteed, but $r_1 \vee r_2$ would be.

To test an individual right, all we are doing is checking existing obligations to see what state the last obligation to touches that permisison has left it in. If it is unique and it has granted or revoked the right we need, our answer is clear. But, if it is not clear which obligation will be done last due to overlapping, conflicting obligations, then we apply our more complex analysis to see if the condition is guaranteed to be satisfied.

6.1.1 Broad Algorithm

In the following, we assume a function $CheckDisjunction(O, M', t, \sigma, t_0)$, which returns true if there exists a schedule of obligations in O which causes all of the permissions in M' to not exist at time t when starting from state σ at time t_0. The algorithm to compute this function is discussed after the general algorithm.

1. Check condition. We repeat the following for each disjunction in the condition.
   (a) Check if the disjunction contains any two tests, m and $\neg m$ for any permission m. If it does, skip to the next disjunction.
   (b) Define a set of rights, M', and initialize to .
   (c) We do the following for each test which is part of the disjunction. We assume the test is positive (*i.e.*, let it take the form $((s, o, r) \in M)$. If in fact the test is negative $((s, o, r) \notin M)$, then the procedure is obtained from the following by reversing the roles of "grant" and "revoke" and negating permission membership tests. We refer to the permission being tested as m.

i. If there is an overlapping revoke action, *i.e.*, some $br \in B_{cur}$ has $br.a = revoke(b.s, o, r)$ and the intervals $[b.t_s, b.t_e]$ and $[br.t_s, br.t_e]$ intersect, than m cannot be guaranteed, and m should be added to M'.

ii. Otherwise, if the privilege exists in the current state, *i.e.*, $(b.s, o, r) \in M_{cur}$ then

   A. If there is a prior revoke action, *i.e.*, some $br \in B_{cur}$ with $br.a = revoke(b.s, o, r)$ has $br.t_e < \mathbf{b.t_s}$, then pick such a $br \in B_{cur}$ so as to maximize $br.t_e$ (subject to $br.t_e < \mathbf{b.t_s}$). The test can be guaranteed only if there exists an obligation that someone grant the permission (again) after br but before b, *i.e.*, only if some $bg \in B_{cur}$ has $bg.a = grant(b.s, o, r)$, $br.t_e < bg.t_s$ and $bg.t_e < \mathbf{b.t_s}$. If the test can be guaranteed, then proceed to the next disjunction. If it cannot be guaranteed, then add m to M'.

   B. Otherwise, the test can be guaranteed, and we proceed to the next disjunction.

iii. Otherwise, if the privilege does not exist in the current state then

   A. If there is some grant obligation for the tested permission $bg \in B_{cur}$ that ends before b **starts**, then pick some bg so as to maximize $bg.t_s$ while satisfying $bg.t_e < \mathbf{b.t_s}$. The test can be guaranteed only if no revoke obligation for the tested permission $br \in B_{cur}$ that overlaps with the interval $[bg.t_s, b.t_e]$. If the test can be guaranteed, then we proceed to the next disjunction. If the test cannot be guaranteed, then we add m to M'.

   B. Otherwise, the test cannot be guaranteed, and we add m to M'.

iv. Next we need to evaluate whether or not there exists any point t' between $b.t_s$ and $b.t_e$ at which the condition is false. If such a t' exists, then there exists a such t' which is equal to either the start or end of b or the start or end point of some other obligation, such that that start or end point is in between $b.t_s$ and $b.t_e$. As such, we define a set T' to be all such start and end points (including $b.t_s$ and $b.t_e$). Next we evaluate $CheckDisjunction(O, M', t', \sigma, t_0)$ for all $t' \in T'$ where O is the set of all pending obligations other than b, σ is the current state, and t_0 is the current time. If the result is $false$ for all t' then we pass the test and move to the next disjunction. If the result is $true$ for any t' then we are not accountable and we terminate.

(d) If there are no disjunctions remaining, then we are accountable and we terminate.

2. Check effect of b on overlapping and later obligations. The obligation b either grants or revokes some right. Obligations which depend on the presence or absence of this right need to be considered. To check them, we repeat step 1 of this algorithm for each of them.

6.1.2 Helper Function

Next we must describe how to construct the $CheckDisjunction(O, M', t', \sigma, t_0)$ function. First, we remove any irrelevant obligations. $O = O \backslash \{o \in O | o.t_e < t_0 \vee o.t_s > t'\}$. Then we define our base cases. If $M' =$ then we return $true$. If $O =$ then we return $false$ if any member of M' is present in σ and $true$ otherwise.

If neither of these is true then given that O is non-empty, we find a set, L which is the possible obligations which can be the last obligation to occur before t'. In order to do this, we find a set L_0 which is the set of all obligations with the latest start time. If there is a single unique obligation with the latest start time then L_0 will have a single member. If there is a tie, then L_0 will have multiple members. Either way, we will call the start time in question $L_0.t_e$. L is the set of all obligations which overlap $L_0.t_e$. Any obligation which does not overlap $L_0.t_e$ cannot be the last obligation to occur.

Next we define R to be the subset of L which only revokes permissions in M' and does not grant any. If R is empty, we check if any member of L_0 overlaps t'. If no member of L_0 overlaps t' then we return false and terminate. If some member of L_0 overlaps t' then return $CheckDisjunction(O \backslash L_0, M', t', \sigma, t_0)$.

If R is not empty, then assume that all statements in R execute after all other statements in L. Because statements in R only revoke permissions and do not grant any relevant permissions, we can order them arbitrarily. Let M_R be the set of permissions revoked by statements in R. All permissions in M_R can be assumed to be revoked when the obligations in R are executed. Hence, we know that a schedule which denies all of M' exists if we can order $O \backslash R$

such that all of $M' \backslash M_R$ is denied. As such, we return $CheckDisjunction(O \backslash R, M' \backslash M_R, t', \sigma, t_0)$.

6.1.3 Timing Analysis

The first step of the main algorithm will be carried out once for each obligation. Its substeps will be carried out once for each disjunction in the condition. Step 1(c)iv will invoke $CheckDisjunction$ up to twice for each pending obligation (if they all overlap b with both their start and end). As such, $CheckDisjunction$, is called $O(n^2m)$ times where n is the number of obligations and m is the size of the policy. When $CheckDisjunction$ runs, it scans through every obligation and every permission involved in each obligation, so its run time is $O(nm)$, not counting the recursion. In each recursive case, we remove at least one obligation before recursing. As such, the depth of recursion is bounded by the size of O. Since each call to $CheckDisjunction$ can result in at most one recursive call and the depth of recursion is bounded by n, the total running time for $CheckDisjunction$ is $O(n^2m)$. And hence to total running time for the algorithm is $O(n^4m^2)$.

Theorem 4. *Under the restrictions identified above, the problem of determining whether a concrete system is strongly accountable is in* $\boldsymbol{P}$.

6.2 Weak Accountability

Unfortunately, in the case of weak accountability, there is an additional complication. Our above algorithm, simplified to only check $t' = b.t_e$ would be sound and would identify many weakly accountable cases. However, it is not complete and would not incorrectly identify certain accountable states as being unaccountable. Specifically, if a system is strongly accountable, and one obligation must wait to execute until another one has completed, then those two obligations must not overlap. In a weakly accountable system, such overlap is acceptable, so long as the end deadline for the dependent obligation falls after the deadline of the other obligation.

As such, our definition of weak accountability requires that we pay attention to such dependencies. If two obligations overlap, we cannot assume that they can happen in either order. Only some schedules may be possible, and our algorithm above would mark some states as unaccountable despite the fact that the schedule which invalidates a given condition cannot actually occur.

It turns out, in fact, that in this case, unlike the situation with the three general assumptions, that it is more difficult to exactly determine accountability in the weak accountability case. Specifically, the problem of completely identifying weakly accountable states in this concrete model is Co-NP Hard. In the following subsection, we present a reduction of 3-SAT to weak unaccountability checking.

6.2.1 Reduction

Given a 3-SAT expression S with a set of variables $X = x_1, x_2, ..., x_n$, we construct the following system:

Let there be a set of permissions, $m_1, ..., m_n$ and also m^*. To simplify things, rather than write a policy and a list of pending obligations, we will assume that there is a single user carrying out all obligations and express the obligations in the form $(pre = condition, post = effect, [t_s, t_e])$.

The following pending obligations exist:

$o_{start} = (pre = true, post = Grant\ m^*, [1, 2])$

$o_{1,t} = (pre = true, post = Grant\ m_1, [1, 2])$
$o_{1,f} = (pre = true, post = Revoke\ m_1, [1, 2])$
...
$o_{n,t} = (pre = true, post = Grant\ m_1, [1, 2])$
$o_{n,f} = (pre = true, post = Revoke\ m_1, [1, 2])$

$o_r = (pre = true, post = Revoke\ m^*, [3, 9])$
$o_g = (pre = S', post = Grant\ m^*, [6, 12])$
$o_u = (pre = m*, post = nothing, [15, 20])$

Where S' is constructed as follows: Given a clause $(x_i \vee \neg x_j \vee x_k)$ from S, we replace it with $(m_i \vee \neg m_j \vee m_k \vee \neg m^*)$ in S'.

Then we ask whether or not this system is weakly accountable. If it is not, then there exists a satisfying assignment of variables for S.

The logic is as follows. Obviously, for $1 \leq i \leq n$, $o_{i,t}$ and $o_{i,f}$ will always be able to execute at their deadline. Likewise, o_{start} and o_r will. And, o_g will definitely be able to execute at its deadline, since the execution of o_r guarantees that S' will be true. However, whether or not o_u will be able to execute at its deadline depends on how o_r and o_g are scheduled.

If o_r happens after o_g, then m^* will not exist, and, as such, o_u will be unable to execute. As such, o_u will result in the state being unaccountable only when o_r cannot happen before o_g. Clearly the time limits do not prevent such a schedule. So the question is

whether or not a dependency does. If there is no satisfying assignment for S, then S' cannot be satisfied until m^* is revoked, in which case, o_g depends on o_r and therefore cannot execute until after it. In that case, the system is accountable. If there is a satisfying assignment, then it is not guaranteed that o_g occur after o_r and, as such, o_u is not guaranteed to be able to execute, and the system is unaccountable.

As such, the problem of checking accountability in a concrete system as described is Co-NP Hard. However, the model we have described is actually a superset of the original HRU access matrix model. The original model does not support either negative tests for permissions or the use of logical or to connect tests. We are still searching for an algorithm which might solve the weak accountability question in the more traditional version of the HRU access control matrix.

7 Related Work

Several policy languages have been proposed recently that support the specification of obligations in security policies. XACML [27] and KAoS [28] both have a limited model of obligations. Specifically, they model obligations assigned to a system and cannot describe user obligations, i.e., obligations assigned to ordinary users who are not always trusted to fulfill obligations. Ponder [9] and Rei [18] both support the specification of user obligations. However, in the basic constructs of both languages, time constraints of obligations, e.g., deadlines, cannot be directly expressed.

Heimdall [12] is a prototype obligation monitoring platform which keeps track of pending obligations. It detects when obligations are fulfilled or violated. This requires the modeling of time constraints in obligations, which are explicitly supported in its policy language xSPL. Sailer and Morciniec [23] propose a means of using a third party to monitor obligation compliance in contracts in web services settings.

Bettini et al. [4] studied the problem of choosing appropriate policy rules to minimize the provisions and obligations that a user receives in order to take certain actions. In their policy model, each privilege inference rule is associated with a set of obligations and provisions, i.e., actions that have to be taken *before* a request can be granted. Different from the policy model used in this paper, they assume actions in provisions and obligations are disjoint from those requiring privileges. In other words, they can always be fulfilled. Clearly, with such policies, a system state is always accountable. Bettini et al. [5, 6] further extended their policy model to express the handling of obligation violations.

While the above works focus on the specification and monitoring of obligations, this paper formally defines secure states of a system with the presence of obligations, and studies the complexity of checking whether a state is secure. Therefore, this research is complementary to the above mentioned works.

There have been other attempts to analyze systems with obligations to determine whether or not parties have sufficient rights to carry out their obligations. Firozabadi et al. [11] describe a system for reasoning about obligations and policies in virtual organizations. However, their policies and obligations are both just static allotments of resources at specified time periods, making the comparison quite simple at the cost of being very specific to their model. Kamoda et al. [19] attempt something a little more comprehensive in their model of policies in a web services setting. They model obligations and privileges which combine subjects, actions, and roles including a model of role hierarchies. However, their model is still limited because they model privileges and obligations as being triggered only by events which are independent of the actions in the system. As such they are unable to model any situation in which user actions can change the state of the system.

A large amount of work has been done on access control policies. A variety of policy languages and models have been proposed. Some of them are generic (e.g., [10, 16, 17, 24, 29]) while others are for specific applications (e.g., [1, 8, 22, 26]) or data models (e.g., [2, 3, 13, 20]). A common problem in access control is compliance checking, i.e., whether an access should be allowed according to an access control policy. Depending on access control models, compliance checking may be very simple (e.g., checking an access control list), or quite complex (e.g., in distributed trust management [7]). The problem of determining whether a state is accountable is analogous to compliance checking in access control, in that it must be performed to determine whether to allow a requested operation. However, since it needs to consider the fulfillment of obligations in future states, the determining accountability is inherently more complicated.

Another important class of problems in access control is static policy analysis, e.g., safety analysis [14, 25] and availability analysis [21]. It is interesting to investigate what types of policy analysis can

be performed in obligation policies, but existing work does not address obligations. The analysis presented in this paper is dynamic, but accountability can be used in static analysis of systems.

8 Conclusion

Obligations can be important to the correct operation and availability of systems and applications. The specification of obligations thus has increasingly been integrated into security policies. In this paper, we have proposed a useful means of modeling this combination of security policies with obligation policies. This has lead us to the concept of accountability, which we have formally defined. Beyond that, we proven several results concerning the complexity of determining whether or not a system state is accountable. We have also described a reasonable, more concrete system in our meta-model and outlined an algorithm for checking the accountability of states in that system. In short, we have investigated, formally, the relationship between obligations and security policies.

However, many interesting open problems still exist in the management and analysis of obligations. In particular, we would like to investigate the following problems in the future.

- When a reference monitor determines that an action will cause a system to become unaccountable, there are several possible actions it could take. Simple denial may have a negative impact on the availability of services. Another possible approach is to allow the action, but meanwhile adjust the obligations in the system to make sure the resulting state is still accountable. For example, if Bob would like to delete Alice from a system, then the system may either transfer Alice's obligations to Bob or discharge them. What decisions the reference monitor makes should also be part of obligation management policies.
- In some systems it may be complex to check a state's accountability dynamically. However, there may be static accountability analysis which can be done. In particular, we would like to investigate those properties of obligation policies which ensure that when a system enforces its policies it will never become unaccountable. For example, if the conditions of the rules of those actions involved in obligations are always true, then all the system states are trivially accountable. We would like to identify more such properties in the future.
- We present a concrete model for authorization systems with obligations in section 5. It would be interesting to make it more specific to support commonly available features in today's access control systems such as roles and cascading delegations.
- Obligations may also be assigned due to the occurence of unexpected events instead of explicit action requests [9]. For example, a policy may specify that a system administrator is obliged to restore the file server within 24 hours after a system crash. We plan to extend our metamodel to support event triggered obligations, and investigate how the concept of accountability may be affected after introducing them.

Acknowledgments

This research was sponsored by the NSF through IIS CyberTrust grant 0430166 (NCSU), NSF ITR award CCR-0325951 (through a sub-award from Brigham Young University), and NSF award CCF-0524010. We also thank our anonymous reviewers for their helpful comments.

References

[1] R. J. Anderson. A security policy model for clinical information systems. In *Proc. IEEE Symposium on Security and Privacy*, pages 30–43, 1996.

[2] E. Bertino, F. Buccafurri, E. Ferrari, and P. Rullo. A logical framework for reasoning on data access control policies. In *Proc. 12th IEEE Computer Security Foundations Workshop*, pages 175–189, 1999.

[3] E. Bertino, S. Castano, and E. Ferrari. On specifying security policies for web documents with an XML-based language. In *Proc. 6th ACM Symposium on Access Control Models and Technologies*, Chantilly, VA, May 2001.

[4] C. Bettini, S. Jajodia, X. S. Wang, and D. Wijesekera. Provisions and obligations in policy management and security applications. In *VLDB*, Hong Kong, China, Aug. 2002.

[5] C. Bettini, S. Jajodia, X. S. Wang, and D. Wijesekera. Obligation monitoring in policy management. In *IEEE International Workshop on*

Policies for Distributed Systems and Networks (POLICY 2003), Lake Como, Italy, June 2003.

[6] C. Bettini, S. Jajodia, X. S. Wang, and D. Wijesekera. Provisions and obligations in policy rule management. *J. Network Syst. Manage.*, 11(3), 2003.

[7] M. Blaze, J. Feigenbaum, and M. Strauss. Compliance Checking in the PolicyMaker Trust Management System. In *Financial Cryptography*, British West Indies, Feb. 1998.

[8] C. Bussler and S. Jablonski. Policy resolution for workflow management systems. In *Proc. Hawaii International Conference on System Science*, Maui, Hawaii, January 1995.

[9] 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.

[10] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The ponder policy specification language. In *Proc. International Workshop on Policies for Distributed Systems and Networks*, pages 18–38, 2001.

[11] B. S. Firozabadi, M. Sergot, A. Squicciarini, and E. Bertino. A framework for contractual resource sharing in coalitions. In *5th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2004*, Yorktown Heights, New York, June 2004.

[12] 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.

[13] P. Griffiths and B. Wade. An authorization mechanism for a relational database systems. *ACM Transactions on Database Systems*, 1(3), 1976.

[14] M. A. Harrison, W. L. Ruzzo, and J. D. Ullman. Protection in operating systems. *Communications of the ACM*, 19(8):461–471, Aug. 1976.

[15] IBM. *Enterprise Privacy Authorization Language (EPAL 1.1) Specific ation.* `http://www.zurich.ibm.com/security/enterprise-privacy/epal/`.

[16] S. Jajodia, P. Samarati, and V. S. Subrahmanian. A logical language for expressing authorizations. In *Proc. 1997 IEEE Symposium on Security and Privacy*, pages 31–42, 1997.

[17] S. Jajodia, P. Samarati, V. S. Subrahmanian, and E. Bertino. A unified framework for enforcing multiple access control policies. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 474–485, 1997.

[18] L. Kagal, T. W. Finin, and A. Joshi. A policy language for a pervasive computing environment. In *IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2003)*, Lake Como, Italy, June 2003.

[19] H. Kamoda, M. Yamaoka, S. Matsuda, K. Broda, and M. Sloman. Policy conflict analysis using free variable tableaux for access control in web services environments. In *Policy Management for the Web Workshop*, Chiba, Japan, May 2005.

[20] M. Kudo and S. Hada. XML document security based on provisional authorization. In *Proc. ACM Conference on Computer and Communication Security*, Athens, Greece, November 2000.

[21] N. Li, W. H. Winsborough, and J. C. Mitchell. Beyond proof-of-compliance: Safety and availability analysis in trust management. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 123–139. IEEE Computer Society Press, May 2003.

[22] T. Ryutov and C. Neuman. Representation and evaluation of security policies for distributed system services. In *Proc. DARPA Information Survivability Conference and Exposition*, January 2000.

[23] M. Sailer and M. Morciniec. Monitoring and execution for contract compliance. Technical Report TR 2001-261, HP Labs, 2001.

[24] R. 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.

[25] R. S. Sandhu. The Schematic Protection Model: Its definition and analysis for acyclic attenuating systems. *Journal of ACM*, 35(2):404–432, 1988.

[26] E. Sirer and K. Wang. An access control language for web services. In *Proc. 7th ACM Symposium on Access Control Models and Technologies*, Monterey, CA, June 2002.

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

[28] A. Uszok, J. M. Bradshaw, R. Jeffers, N. Suri, P. J. Hayes, M. R. 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 *IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2003)*, Lake Como, Italy, June 2003.

[29] OASIS eXtensible Access Control Markup Language (XACML). http://www.oasis-open.org/committees/xacml/, 2005.

A Reductions

In section 5, we describe three system properties under which we can solve the problem of accountability efficiently. Here we show that if we remove any one of these properties, the problem of unaccountability become NP-hard, and hence the problem of accountability is CoNP-hard.

A.1 Without the Property of No Cascading Obligations

First we show that if we allow cascading obligations, we can reduce SAT to unaccountability. In SAT, there is some set of boolean variables $\{x_1, \ldots, x_n\}$, and some boolean expression of these variables $X(x_1, \ldots, x_n)$, and we are attempting to determine if there is an assignment of values to variables such that X is true.

To describe our reduction, we first define the state of our system as being composed of a set of boolean variables $\{x_1, \ldots, x_n\}$ and an integer counter m. We also define our system having a single subject s.

There are $3n + 1$ different actions available. The first $2n$ of them are used for setting the variables to either 0 or 1. However, simple assignment would not meet our condition that different actions be commutative. As such, we define them to be assignments which cannot decrease the value. Specifically, for $i = 1, \ldots, n$ we define actions set_i^0 which sets x_i to 0 only if it was already 0 and also increments the counter m by 1 and set_i^1 which sets x_i to 1 in all circumstances and also increments m by 1. As such, if both set_i^0 and set_i^1 occur in either order, the result will be that x_i is set to 1.

The next n actions we name $start_i$ for $i = 1, \ldots, n$. These actions have no effect. The final action we name *check*, and it also has no effect.

Then we have a set of policies for $i = 1, \ldots, n$:

$$set_i^0(s) \leftarrow true : f(\sigma, t, s) = \emptyset$$

$$set_i^1(s) \leftarrow true : f(\sigma, t, s) = \emptyset$$

$$start_i(s) \leftarrow true : f(\sigma, t, s) = \{(s, set_i^0, 2n + 2i, 2n + 2i + 1)\}$$

$$start_i(s) \leftarrow true : f(\sigma, t, s) = \{(s, set_i^1, 2n + 2i, 2n + 2i + 1)\}$$

$$check(s) \leftarrow m \geq n \wedge \neg X(x_1, \ldots, x_n) : f(\sigma, t, s) = \emptyset$$

Lastly, we define our initial state in which, $t = 0$, $x_i = 0$ for $i = 1, \ldots, n$, and we have a set of $n + 1$ initial obligations, $\{(s, start_i, 2i, 2i + 1) | i = 1, \ldots, n\} \bigcup \{(s, check, 3n, 3n + 1)\}$. The question of whether or not this initial state is unaccountable is equivalent to the question of whether or not X is satisfiable. In short, the obligations are going to require that each x_i be set to either 1 or 0. Because the final obligation has that X be unsatisfied as its condition, the initial state will only be accountable if there is no assignment of values which will cause X to be satisfied. As such, if the initial state is unaccountable, then X is satisfiable.

With our construction complete, we take a moment to note that it is consistent with our other two assumptions. As we described earlier, the operators are all commutative. Also, the system is monotonic. Only the policy for *check* has any chance of changing at all. However, it is specifically structured so that it will remain false until all variables are assigned, and at that point it may either remain false, or become true. It is not the case that it is monotonic with respect to all possible actions, but it is monotonic with respect to possible actions consistent with obligations moving forward from our current state, which is all that the property requires.

A.2 Without the Property of Monotonicity

In this section, we assume that we can have systems which are not monotonic, and show that in that case, the set covering problem can be reduced to the unaccountability problem.

In set covering problem, there is a set, S, and a set of n subsets of S, $\{S_1, \ldots, S_n\}$. There is also a parameter, k, and the question is whether or not there is a set of k of the subsets such that the union of those sets is equal to S. That is, is every member of S "covered" by one of the k subsets.

In this case we are not going to have cascading options. So we cannot have a system in which the obligations force us to make choices between different options.

As such, our state, σ is represented by a set S' and an integer m. We have a single subject, s. We have $n+1$ actions. For $i = 1, \ldots, n$ we define an action $remove_i$ which removes S_i from S' and increments the value of m by one. We also define an action *check* which has no effect.

Next we define the policies for the actions. For $i = 1, \ldots, n$ we define policies $remove_i(s) \leftarrow true : f(\sigma, t, s) = \emptyset$ and $check(s) \leftarrow \neg(m = k \wedge S' = \emptyset) : f(\sigma, t, s) = \emptyset$.

Our initial state is the state such that $m = 0$, $S' = S$ and we have a set of obligations $\{(s, remove_i, 0, 2n) | i = 0, \ldots, n\} \bigcup \{s, check, 0, 2n\}$. All of the subset actions can be fulfilled. But only those which are fulfilled before the check obligation matter to the check of accountability. So, the question of accountability is the question of whether or not it's possible to schedule k of the subset obligations before the check obligation which together cover S. If it is possible, then the initial state is unaccountable. If it is impossible, then the initial state is accountable, since that means that there is no schedule which will result in an obligation failure. As such, we have reduced Set Cover to unaccountability.

Again we finish by verifying that our other two assumptions hold in our system. Clearly, there are no cascading obligations as no policy contains any new obligations. And it is the case that all actions are commutative since for any two actions, $remove_i$ and $remove_j$, the result of apply them is that S' takes on the value of S'
$(S_i \bigcup S_j)$ and m takes on the value $m + 2$.

A.3 Without the Property of Commutative Actions

For assuming that non-commutative actions are allowed, we once again use SAT in our reduction. Again we assume the existence of a set of n boolean variables $\{x_1, \ldots, x_n\}$ and a boolean expression $X(x_1, \ldots, x_n)$.

Same as the previous construction, we define our state to have a single user s. And we define our system state, σ as a set of boolean variables, $\{x_1, \ldots, x_n\}$ and a counter, m. But this time, we only define $2n+1$ actions. The first $2n$ are used for setting the variables to either 0 or 1. However, unlike the previous case, we do not need them to be commutative, and, in fact, this construction would not work if they were. So they are instead the more traditional assignment. For $i = 1, \ldots, n$, we define actions set_i^0 which sets x_i to be 0 and increments m by 1 and set_i^1 which sets x_i to 1 and increments m by 1. Our last action is *check*, and it has no effect.

Our policies are quite similar to the earlier reduction. We define, for $i = 1, \ldots, n$:

$$set_i^0(s) \leftarrow true : f(\sigma, t, s) = \emptyset$$

$$set_i^1(s) \leftarrow true : f(\sigma, t, s) = \emptyset$$

$$check(s) \leftarrow m \geq 2n \wedge \neg X(x_1, \ldots, x_n) : f(\sigma, t, s) = \emptyset$$

Lastly, we define our initial state. In our initial state, $t = 0$, $x_i = 0$ for $i = 1, \ldots, n$, and we have a set of $2n$ initial obligations, $\{(s, set_i^0, 3i, 3i+2) | i = 1, \ldots, n\} \bigcup \{(s, set_i^1, 3i, 3i+2) | i = 1, \ldots, n\} \bigcup \{(s, check, 3n, 3n+5)\}$. The question of whether or not this initial state is unaccountable is equivalent to the question of whether or not X is satisfiable. In short, the obligations are going to require that each x_i be set to either 1 or 0. In fact, at some point it will be set to both 0 and 1, but only whichever assignment appears last will be relevant to the condition of *check*. Because the final obligation has that X be unsatisfied as its condition, the initial state will only be accountable if there is no assignment of values which will cause X to be satisfied. As such, if the initial state is unaccountable, then X is satisfiable.

With our construction complete, we once more take a moment to note that it is consistent with our other two assumptions. The system is monotonic for the same reasons that the system in the first reduction is monotonic, namely that there is no valid sequence of obligation actions going forward from our particular

initial state which can cause any condition to go from true to false. It's also clearly the case that there are no cascading obligations as no policy carries any obligation.