

Enforcing Security Properties in Task-based Systems

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

Though a user's privileges are often granted based on the tasks that the user is expected to fulfill, the concept of tasks is usually not explicitly modeled in access control. We propose a system where tasks are the central concept that associates users to privileges. Ideally a user should be able to utilize these privileges and fulfill his tasks, but not to take harmful actions. To ensure this, a system often specifies a high-level security property to restrict the sequence of actions that a user can perform. In this paper, we propose a general model of access control in task-based system. This model considers the permissions a user as well as their temporal availability. Based on this model, we investigate the problem of enforcing security properties both statically (i.e., when tasks are assigned) and dynamically (i.e., when actions are performed). We study the complexity of static enforcement, and design efficient dynamic enforcement algorithms that avoiding unnecessary history tracking.

Categories and Subject Descriptors

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

General Terms

Security, Theory

Keywords

Policy, Task-based Access Control, Security Properties

1. INTRODUCTION

In an information system, a user's privileges are fundamentally determined by the tasks that she is expected to perform. For example, if a user is assigned a task to write a report that analyzes her company's sales performance, then the user will be granted the privileges to query the company's sales database, as this privilege is indispensable to finish the report. Meanwhile, she would usually not need to modify the database, and as such should not be given the privilege to do so.

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

SACMAT'08, June 11–13, 2008, Estes Park, Colorado, USA.
Copyright 2008 ACM 978-1-60558-129-3/08/06 ...\$5.00.

Tasks often serve to handle situations that arise dynamically, and users are asked to perform a variety of different tasks that need different permissions. As such, any *static assignment* of privileges or roles to users are likely to give users more privileges than they strictly need the majority of the time. Instead, it would be wise to give users only those privileges that are required for the tasks on which they are currently working. This could be accomplished in traditional discretionary or role-based access control systems by having system administrators who grant and revoke privileges as needed, but that level of manual administration is usually work-intensive and error-prone.

In this paper, we introduce the concept of tasks into access control, and use tasks as a central and explicit means to constraining privileges of users. There are several advantages of doing so. First, tasks are a natural way to think about user actions and their contexts. They are conceptually simple, but also allow for fine-grained control of privileges. In particular, a task is often associated with a time period for its fulfillment (otherwise, there is little motivation for a user to work on it). For example, a report may have to be finished by the end of the month. The temporal property of a task allows us to naturally restrict not only the set of granted privileges, but also when and for how long they are available.

Second, many privacy policies require identifying the purposes of actions. It is much easier to categorize the purpose of a task than that of an individual action. While it is unclear how to ensure that a privilege is being used solely for an approved purpose, ensuring that the privilege is only available when a task that requires it is allocated to the user seems a substantial improvement over, for instance, relying on the user to identify the purpose.

In a task-based system, it is important to analyze what else a user can do with her privileges besides fulfilling her tasks, and in particular, whether it is possible to abuse her privileges and do something harmful. Harmful activities are usually not defined by any single action, but rather by a sequence of actions which taken together can cause harm. As such, low-level policies which describe what individual actions a user may take do not address what constitutes harmful activities. Instead, one way to discriminate between normal action sequences and harmful ones is to define high-level security properties which explicitly specify the sequences of actions to be proscribed. Common examples include simple safety properties and static and dynamic separation of duty policies.

Given an assignment of privileges to tasks and tasks to users, and a high level security property, there are two questions which should be examined. First, whether it is possible for a user to abuse the granted privileges and violate a high-level security property. If the answer is no, the system simply needs to check a user's privileges when she attempts an action. Otherwise, the situation is trickier. The system has to keep track of the operations that a principal has

conducted thus far, and dynamically enforce the high-level security properties, which may be quite expensive. Thus the second question is how to reduce the complexity of the dynamic enforcement of a security property.

In this paper, we investigate the above problems based on a general task-based model of access control. Specifically, we make the following contributions:

- We propose a general model in which users' privileges are determined by their task assignments. We identify two types of security properties. *Intra-task* properties proscribe action sequences that must not be performed by using the privileges granted for a single task. *Inter-task* properties must not be violated by using privileges obtained from any assigned tasks.
- Given a set of privileges along with their temporal constraints, we present an efficient algorithm to statically check whether they may be abused to violate security properties. We further show the intractability of static analysis when considering collusion among multiple users.
- We study the problem of minimizing the change to one's privileges while enforcing security properties, and show its intractability in the general case.
- In general, the complexity of dynamic enforcement of a security property is proportional to the size of the finite state machine modeling it. To improve the efficiency of dynamic enforcement, we develop efficient algorithms to simplify a security property by identifying and removing redundant portion of property, given a user's privileges.
- We show the intractability of optimal allocation of tasks to users while preserving security properties, in the context of the proposed task-based system.

The remaining of the paper is organized as follows. We give an overview of different techniques to enforce security properties in section 2. In section 3, we formally present the models for system access states and security properties. The enforcement of intra-task properties is discussed in section 4. Section 5 discusses issues in enforcing inter-task properties. In particular, we formally propose the task-allocation problem, and investigate its tractability. Closely related work is reported in section 6. And we conclude this paper in section 7.

2. ENFORCING SECURITY PROPERTIES

One way to frame security objectives is in terms of undesirable sequences of actions (behaviors) that should be prevented. For instance, action sequences that corrupt data integrity, misappropriate resources, or disclose confidential data can all be viewed in these terms. A user's copying to the clipboard, followed (possibly after several other operations) by a paste into an email message can signal a breach of confidentiality. A manager approving a travel reimbursement request that he submitted himself is usually viewed as a security violation because it may signal a misappropriation of resources.

It is thus natural to treat security properties as being negative in nature, in the sense that they describe action sequences that are undesirable, and that should not happen during system operation. In general, such security properties can be enforced at three different levels: access control-policy level, access-state level, and action-execution level.

Access control policy level. Access control policies concern when and under what circumstances privileges may be granted to or revoked from principals. We distinguish this from the current access control state which indicates what privileges each user holds

at present. The access control state may change as part of the normal operation of a system, as long as the change is allowed by the system's access control policy.

To enforce a security property, one can design an access control policy that ensures no user will ever obtain privileges that enable her to violate that security property. For example, consider a security property that states Alice should never be allowed to perform action a . To enforce it at the access control-policy level, we need to show that, from an initial state, no matter how the access control state changes, as long as each change conforms to the policy, Alice will never get the privilege to perform a . This is the *safety problem* that has been extensively studied in the literature [10, 14].

The advantage of this approach is that the runtime execution of the system is quite simple. A user's privileges are granted according to the access control policy. And regardless of how they are used, the security properties are guaranteed not to be violated. However, the problem to determine whether an access control policy enforces a security property can be challenging. If the system and policy models are reasonably general, the problem is often undecidable or intractable [10]. Models with restrictions have been proposed to make some of the policy analysis problems tractable [9, 11, 12, 14]. However, many of these models are rather restrictive, and thus are not widely adopted.

Another approach is to design policies conservatively so they are clearly sufficient to enforce security properties. However, this approach often force policies to be too conservative to be useful.

Access control state level. A system's access control state describes the privileges each user has at a given time. To enforce security properties at the access control state level, when an action is performed that would change the privileges of a user (especially when new privileges are about to be granted), the reference monitor could first determine whether the new access control state would enable the user to perform actions that violate security properties. If so, the attempt to change the privileges can be denied even if it would have otherwise been allowed by the access control policy. In the example above, if someone tries to grant Alice the privilege to perform a , the reference monitor would reject the grant action.

Enforcing security properties at the access control state level can be conducted much more simply than can policy analysis, though it must be done more frequently. However, it may still be overly restrictive in some situations. Possessing the privileges to violate a security property does not necessarily mean the user would perform a proscribed sequence of actions. Also, the rejected privileges may be needed to enable the user to fulfill her duties. Consider the simple separation of duty example. A manager may need to have privileges to file a reimbursement request for her own travel and also privileges to approve requests from those in her department. Simply preventing her from obtaining both privileges could prevent her from fulfilling her duties, or force those duties to be unnaturally assigned to others. A security violation occurs only if the manager performs both actions within the same reimbursement task.

Action-execution level. Instead of concentrating on the privileges a user has, we may monitor what actions are actually performed. A user's action request will be denied if the action, taken in the context of the user's previous actions, would violate a security property. In our previous example, if we instead enforce separation of duties at the action-execution level, it is perfectly acceptable for the manager to have both privileges at the same time. When the manager attempts to submit a reimbursement request, the system allows it to be executed, but also records the action. If the manager later attempts to approve the same reimbursement, because the sys-

tem records the prior action, it would determine that the approval would violate the security property, and rejects it.

Action execution-level enforcement offers the maximum flexibility in terms of how users may fulfill their tasks. However, the history information for each user might be quite large. Also, execution-level enforcement has to perform analysis every time a user attempts to take an action. This is much more frequently than enforcement at the access control policy level (which only must consider things when the access control policy is changed) or enforcement at the access control state level (which only needs to mind actions that change users' privileges). Thus, efficient analysis is far more important in this approach.

In practice, it would be desirable to combine the above three types of enforcement mechanisms to secure an information system. There has been a great deal of work on determining whether an access policy permits the system to evolve into a state that would permit some security property to be violated [10, 14]. In this paper, we focus on security property enforcement at the access control state level and the action execution level.

At the access-state level, we use the concept of tasks to bridge access control policies and access control states. Specifically, we observe that, one's privilege changes are often associated with task changes, which do not happen frequently. In other words, once a user is assigned a task, she obtains the privileges that the system deems to be necessary for fulfilling the task. This set of privileges usually does not change until the task is finished. Therefore, by examining the privileges and their time constraints, we can form a more precise analysis of what the user could do.

To facilitate action execution-level enforcement, we also consider the privileges assigned to a user and the time interval during which they are assigned. In particular, knowing what privileges are not available, we may identify history information that is extraneous to enforcing particular security properties. This allows us to improve the efficiency of execution-level enforcement.

3. SYSTEM SETUP

3.1 Integrating Tasks into Access Control

We propose explicitly modeling tasks in access control and using them as a basis for granting and revoking privileges. Similar to role-based access control, privileges are not directly granted to principals. Instead, an information system associates the necessary privileges with a task. And a user obtains privileges through tasks. That is, only when the user is assigned to a task can she use the privileges associated with that task.

A distinction between roles and tasks is that tasks are more temporal in nature. When a task is assigned, it should be specified when it should be finished, at which point the user loses the associated privileges. The time period of the task can be further subdivided to grant different privileges during different phases of the task, which allows even finer grained access control and constrains the sequences of actions that the user can perform.

In our model, a task is assigned to an individual user. If there are processes which should be carried out by multiple users, we model this as a set of tasks which are assigned to a set of users. Since most systems express permissions as belonging to individual users anyway, this should be equivalent in most circumstances.

Note that it is not our intention to have tasks replace roles or other access control means for granting privileges. We view tasks as a useful concept that can be naturally integrated into existing models, enriching their capability. For example, we may introduce tasks on top of role-based access control. Rather than associating privileges to a task directly, when she is assigned to fulfill a task,

we may specify what roles a user can activate, when, and for how long. In this scenario, tasks serve as a simple means to controlling role activation, an important aspect of role-based access control.

The extra layer of tasks facilitates security analysis, particularly with respect to undesirable action sequences. From the tasks assigned to a user, we can determine what privileges she will have.

For simplicity, in the rest of our discussion, we model a task as an association between a user and a set of privileges at certain times. In practice, these privileges may come indirectly from some other underlying access control system such as RBAC.

3.2 Access Control State

Conceptually, the access control policy of a task-based system decides when and to whom a task is assigned. A task may be assigned to a single user or to a set of users who collaboratively fulfill it. For instance, a travel reimbursement task normally needs to be assigned to two users, one who submits her expenses and receipts, and one who approves them. An access control policy further decides the privileges each user should be granted to enable their participation in task completion.

As argued above, a task needs to be associated with a time constraint, indicating when it should be fulfilled. As a consequence, the privileges for the task should also have time constraints, stating when the privileges are available.

Formally, a task-based system consists of these components:

- \mathcal{U} : a set of users.
- \mathcal{K} : a set of tasks which users may carry on.
- \mathcal{A} : a set of actions that can be taken by users.
- t : a clock t , which we represent as a real number. At any time an action is attempted, the clock will have some value in the domain $\mathcal{T} = \mathbb{R}^+$, with 0 indicating the system start time. We assume that any two action attempts can be ordered. This can either be accomplished by assuming that no two action attempts happen at precisely the same time or by assuming that there is an ordering on the actions such that if two are attempted at precisely the same time, then they are still processed sequentially. Although the set of reals is uncountable, we will only be dealing with a finite subset in practice since only time points at which actions occur or permissions change are relevant.
- $\mathcal{P} = \mathcal{A} \times \mathcal{T} \times \mathcal{T}$: a set of privileges. Each privilege $(a, s, e) \in \mathcal{P}$ identifies a time window during which the permission for action a is available. That is to say, if $s < t < e$, then action a can be performed at time t . We require $s < e$ for all $(a, s, e) \in \mathcal{P}$.

For simplicity, our results conservatively do not assume that the number of times an action can occur in a given time window is bounded, though in practice it certainly would be.

- \mathcal{F} : a privilege-granting function which takes as input a task k and a time point t , and returns a set of privileges.

A privilege granting function abstracts the access control policy of a system. It is intended to be flexible in several respects so that it can represent various policies for granting privileges to users. First, the returned privileges of a privilege granting function are not restricted to a single user. This enables the modeling of privilege assignments for collaborative tasks. Second, the returned privileges are not required to have the same time period. In other words, it is possible for a system to require that one privilege is used before another one. Third, we allow different privileges to be returned if the same task is assigned at different times. This gives a system

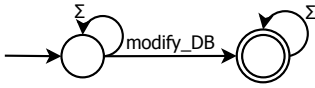


Figure 1: An NFA that represents a simple Safety property

the flexibility to decide the necessary privileges for a task by considering the current state of a system.

Observe that if two privileges are about the same permission and their time periods overlap, we can always merge them into a single privilege by merging their time periods. Clearly, this will neither increase nor decrease what the user can do. Therefore, in the rest of the paper, we assume that the time periods of privileges about the same permission are disjoint.

The access control state of a system is a subset of $\mathcal{U} \times \mathcal{K} \times \mathcal{P}$. It contains one entry for each permission granted to each user and for each task assigned to the user that requires the permission. In other words, if a given user has a given permission that she needs for each of two tasks, there will be two entries in the access control state, one for each task.

3.3 Security Properties

As argued by Alpern and Schneider [2, 17], most practical security properties should be amenable to modeling as a finite state automaton to be effectively enforced in an information system. We follow this convention in this paper as well. Specifically, the alphabet of the automaton is $\Sigma = \mathcal{A} \cup T_H$ in which \mathcal{A} is the set of actions of the system and T_H is a set of times. Here, T_H consists of times that are significant to defining undesirable behaviors. For instance, certain actions should not happen at certain times, but may be acceptable at other times. Additionally, T_H may include times at which certain special system actions are scheduled to occur, actions that can perhaps have compensatory effect, interrupting an otherwise illegal behavior. In our model, undesirable sequences of user actions and times are those that are *accepted* by the automaton, which are said to *violate* the modeled security property.

We follow Alpern and Schneider's model in the respect that if an action sequence q violates a security property, all the sequences that have q as a prefix also violate that property. This implies that from a final state, the automaton should not transit to a non-final state under any input.

We work with non-deterministic finite automata (NFAs). It is natural to specify multiple undesirable action sequences by using an NFA. Converting an NFA to a deterministic finite automaton can result in an explosion of states, whereas the cost of emulating an NFA is normally not prohibitive.

Most typical security properties can be expressed as NFAs. For example, we can express simple safety by having a machine which simply accepts whatever action we wish to prohibit.

EXAMPLE 1 (SIMPLE SAFETY). *It is a security violation if a summer intern modifies the database of a company. (See figure 1.)*

EXAMPLE 2 (SEPARATION OF DUTY). *The clerk initiating a legal draft should not be the principal who decides whether reviews by stakeholders are required [16]. (See figure 2.)*

It is also possible to build machines that ensure more complicated properties. In particular, we observe that many security properties naturally include compensatory actions that safeguard against insecure action sequences.

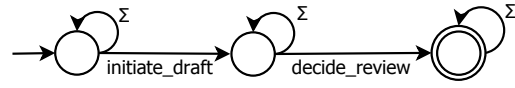


Figure 2: An NFA that represents a security property that concerns a sequence of actions

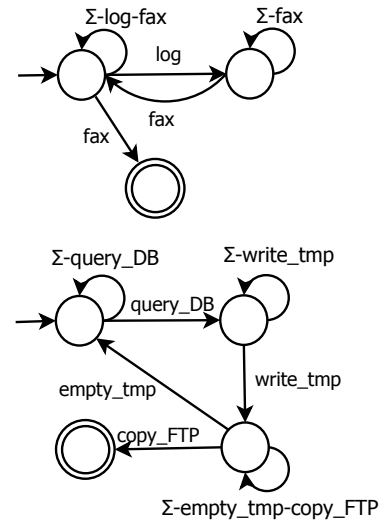


Figure 3: The finite state machines that represent the security properties in example 3 and 4 respectively. Note that Σ -log-fax represents any action other than log or fax

EXAMPLE 3 (COMPENSATORY ACTIONS). *In order to use a company fax machine, an employee is required to log his usage first, e.g., where the fax is sent to and with how many pages. In other words, it is a violation of the company's policy to send a fax unless it is logged, which is a compensatory action.*

In the above example, the compensatory action is performed by a user herself. In many situations, a system will perform such compensatory actions.

EXAMPLE 4 (SYSTEM COMPENSATORY ACTIONS). *A computer system has a temporary directory used as a scratch space. To prevent information leakage, if a user queries a sensitive database and then writes to the temporary directory, then she must not copy anything from the temporary directory to a public ftp server, unless the system first empties the temporary directory.*

Figure 3 shows the finite state machines that represent the security properties in the above two examples.

Because it is trivial to construct an NFA that accepts the union of those sets accepted by a collection of NFA, in this paper we assume there is only one security property that needs to be enforced.

We focus on enforcing a security property of the behavior of a single user. As we show below, when considering collusion among multiple users, the problem becomes intractable. Further, many security properties indeed concern a single user's abuse of her privileges (e.g., the "two-man" rule), as collusion often significantly increases the cost of attacks and the risk of being detected.

We distinguish two categories of security properties, based on the extent of the action sequences that they characterize. We call them intra-task properties and inter-task properties. An intra-task property characterizes sequences of actions that are authorized by the privilege assignments of a single task to the user. In other words, an intra-task property is enforced if a user cannot violate it by using only the privileges she obtained from a single task.

An inter-task property, on the other hand, characterizes the entire sequence of actions conducted by a given user, no matter from which tasks the user obtains a privilege. Generally, inter-task properties can be thought of as working on a larger scale than intra-task properties. In a system which has both an intra-task property and an inter-task property in effect, any action sequences which violates the intra-task property should also violate the inter-task property.

The division of these two categories is natural. Some behaviors are undesirable only when they are undertaken as a part of a single task. A manager can certainly submit travel reimbursement requests and also approve reimbursement requests. He just is not supposed to both submit and approve the same request. Assuming a given reimbursement task includes both submission and approval actions on the same request, it is natural to disallow both actions being performed by any one user for the same reimbursement task.

On the other hand, the undesirability of some behaviors transcends tasks. We saw a news story about financial traders whose company gave a bonus based on the value of trades pending at the end of the day. Apparently several traders were found to be posting large trades shortly before the end of the day, and then canceling them shortly thereafter. In general, insider threats can often be detected by looking for patterns of behaviors over many tasks.

Conceptually, for each task assigned to her, each user has an associated run of the NFA defining each intra-task property. In addition, each user has a single run of each NFA defining an inter-task property. The latter NFAs receive as input each action requested by the user. For the definition and enforcement of intra-task properties, we assume that each authorization is associated with a given task. In the event that two tasks require the same authorization, this is viewed as two authorizations, one for each task. When an action is performed by a user, it becomes input to the intra-task NFAs associated with each task that authorizes the action.

4. ENFORCING INTRA-TASK PROPERTY

4.1 Access State Level Enforcement

Recall that the access state of a user is the set of privileges she has, and that the privileges each indicate the time period in which they are in effect. We now examine the reasoning we can do given the access state of a user. Formally, we ask, given a user's access state, whether it is possible for her to violate a security property. This question is particularly relevant to intra-task properties, since a user's relevant privileges would not change during the fulfillment of a task. If the answer is that she cannot violate the property, then we only need enforce access control according to the user's access state (i.e., whether she has a permission at the current time), and need not maintain any further information to check for violation of intra-task properties.

For example, suppose an intra-task property characterizes as a violation all action sequences in which action a is done and subsequently action b is done. If a user's access state has only $(b, 0, 4)$ and $(a, 5, 10)$, then we can be certain that she will not violate the property in this task.

DEFINITION 4.1. *Given a sequence of actions $q = \{a_1, \dots, a_n\}$ and a set of privileges $P = \{(b_1, s_1, e_1), \dots, (b_m, s_m, e_m)\}$, we say q is compatible with P if there exists a sequence of time points (t_1, \dots, t_n) that satisfies the following conditions:*

- $t_i < t_j$ for all $i < j$; and
- for each a_i in q , there exists a privilege (b_j, s_j, e_j) in P such that $a_i = b_j$ and $s_j < t_i < e_j$.

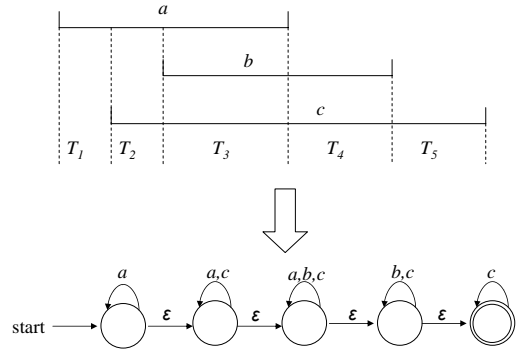


Figure 4: An example to construct an NFA from a set of privileges

Let P be a user's privileges obtained from a task k , and H be an intra-task property. If no action sequences compatible with P violates H , we say P enforces H .

It turns out to be rather straightforward to check whether P enforces H . Basically, we convert P to a finite state machine M which accepts exactly those sequences compatible with P . P enforces H if and only if the intersection of M and H is empty, which can be efficiently checked. This general technique has also been previously employed by Sekar *et al.* [18] to evaluate whether an FSM representing a given piece of code is compatible with a security property. Although we employ a similar technique, we are looking at a very different scenario than theirs, and we must still consider how an FSM representing P can be created.

Given a set of privileges $P = \{(b_1, s_1, e_1), \dots, (b_m, s_m, e_m)\}$, we order the time points in P and get a sequence of time points (t_1, \dots, t_k) where $t_i < t_j$ for $i < j$. We consider the time segments $T_1 = (t_1, t_2), T_2 = (t_2, t_3), \dots, T_{k-1} = (t_{k-1}, t_k)$. We construct an NFA with ϵ -transitions. For each segment T_i , we have a corresponding state S_i . S_i transits to itself for any permission that the user has during the time segment T_i . And there is an ϵ -transition from S_i to S_{i+1} . The start state is S_1 and all the states are final states. Figure 4 shows an example that constructs such an NFA from a set of privileges.

It is easy to see that the cost of the above construction is polynomial to the size of the user's privilege set. The intersection of two NFAs and determining whether an NFA accepts no strings can both be done in polynomial time.

Besides user actions, there are also actions conducted by the system or by trusted users. Some of these actions are particularly designed to thwart potential malicious attempts. For example, the scratch space of a system may be periodically cleaned to limit its availability as a means for information exchange between different data repositories. Suppose a user has the permission to query the sales database and store temporary query results in the scratch space. Meanwhile, she also has the permission to access the scratch space to draft a public document which will be copied to a web server and made available to the public. Then there is a potential risk that information from the sales database may be passed to the public through the scratch space. On the other hand, if the time periods for these two privileges are not overlapping and the system is scheduled to clean the scratch space between them, then the vulnerability is removed.

As such, when enforcing security properties at the access state level, it is necessary to take such scheduled compensatory actions into consideration.

For this purpose, we enhance the basic system setup to further

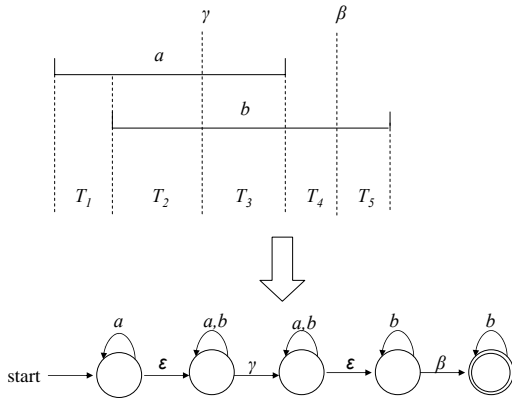


Figure 5: An example to construct an NFA with system compensatory actions

model scheduled system actions. Specifically, let Γ be a set of system actions such that $\Gamma \cap \mathcal{A} = \emptyset$. For each action $\gamma \in \Gamma$, we have a schedule in the form $(\gamma : z_1, \dots, z_n)$, where each z_i is a time point. Essentially, a schedule tells when a system action γ will be performed. Without loss of generality, we assume at a single time point there is at most one system action scheduled.

Note that since γ is to be performed by the system or trust users, we assume that at time z_i , γ will definitely happen. As a consequence, the access state of a user includes her own privileges as well as the schedules of system actions.

Γ is also added to the alphabet of security properties. The finite state automaton of a property may include transitions on symbols from Γ , which reflect the effects of system compensatory actions.

With a similar technique, we can efficiently determine whether it is possible for a user to violate a security property, given her privileges as well as scheduled compensatory actions. Specifically, we order all the time points appearing in the user's privileges as well as in the schedules of compensatory actions, and obtain a set of time segments. The transitions between states are constructed almost the same way as above with one modification. Let S_i and S_{i+1} be two states corresponding to time segments $T_i = (t_i, t_{i+1})$ and $T_{i+1} = (t_{i+1}, t_{i+2})$. If t_{i+1} appears in the schedule of a compensatory action γ (as assumed above, there is at most one compensatory action scheduled at t_{i+1}), then S_i transits to S_{i+1} only on the input γ . This is to reflect that γ is going to happen at t_{i+1} . Otherwise, S_i ϵ -transits to S_{i+1} as before. Figure 5 shows an example construction when considering system compensatory actions.

It is easy to see that a user's privileges along the schedules of compensatory actions enforce a security property if and only if the intersection between the NFA constructed as described above and the security property accepts \emptyset .

The above technique can be easily extended to handle other types of compensatory actions. For example, a system may monitor a user's action closely so that a system compensatory action γ happens immediately after a user takes an action a . In this case, we can replace a with $a\gamma$ in the NFA that represents all the possible action sequences of the user. As another variation, if a task gives permission a to user for a period (s, e) , the system will perform γ at $e + t$, where t is a predefined duration by the system. It is easy to see that, given a set of tasks, we can derive a schedule for system compensatory actions, and apply the above technique.

4.2 Multiple Users

So far we answer whether a single user may abuse her privileges in a task to violate an intra-task property. In many situations a task

is assigned to a group of users to be finish collaboratively, and each user in the group may be assigned with different privileges as they are taking different roles. A natural question would be whether these users may collude and together violate a security property. Formally, given a set of users along with their privileges and a security property H , do there exist fewer than k users whose privileges put together enable those users to violate H ? In other words, can we determine whether harmful sequences of actions can be carried out by any group of k users out of a total of n users.

This problem turns out to be NP complete in general. The proof has been omitted for reasons of space. In the rest of the paper, we focus on the enforcement of security properties that concern with a single user. Identifying non-trivial sufficient conditions to analyze security properties concerning multiple users is left for future work.

4.3 Reduce Privileges to Enforce Property

As mentioned above, if a user's privileges enforce an intra-task property, then the system can safely allow the user to perform any action at any time as long as she has the right permission at that time. Otherwise, we see two possible approaches to enforcing the property. The first is a dynamic monitoring approach, where the system keeps track of the user's past actions, and determines whether a requested action, if allowed to happen, would violate a security property. We will study this approach in the next section. The second approach is to modify the access state and reduce the user's privileges so that security properties are enforced. We focus on this approach in this section.

Though good for security, reducing a user's privilege risks the possibility that we could prevent the user from carrying out legitimate actions which are needed to complete the task. As such, if we are to change the access state, we must logically do it in a minimal way to reduce its impact on task fulfillment.

The natural choice to reduce a user's privileges is to reduce the time periods during which permissions are available to her. Removing a permission a completely is a special case as it is equivalent to reduce the privilege from (a, s_a, e_a) to (a, s_a, s_a) ¹. Note that one way to reduce the time during which a permission is available is to introduce a gap in the period of its availability. For example, given privilege $(a, 1, 10)$, we may reduce it, and obtain two privileges $(a, 1, 4)$ and $(a, 8, 10)$.

DEFINITION 4.2. *Given two sets P_1 and P_2 of privileges, we say P_2 is covered by P_1 if for every $p = (a, s, e)$ in P_2 there exists a $p' = (a', s', e')$ in P_1 such that $a' = a$, $s' \leq s$ and $e' \geq e$.*

DEFINITION 4.3. *Given a set of privileges $P = \{(a_1, s_1, e_1), \dots, (a_n, s_n, e_n)\}$, the measure of P is defined as $\sum_{1 \leq i \leq n} (e_i - s_i)$.*

Intuitively, the measure of a set of privileges shows the availability of permissions to a user. We may further enhance the above definition by considering the relative importance of each privilege. For example, we may associate with each privilege a weight when defining the measure of privileges. As we will show next, even with this simple measurement, the problem to minimize the reduction of a user's privileges to enforce a security property is intractable.

PROBLEM 4.1 (PRIVILEGE REDUCTION PROBLEM). *Given a set of privileges, P , and a security property, H , find a set of privileges P' such that P' enforces H , P' is covered by P , and such that $measure(P) - measure(P')$ is minimized.*

THEOREM 1. *The privilege reduction problem is NP-hard.*

Proof of this theorem has been omitted due to space constraints.

¹Remember that, given a privilege (a, s_a, e_a) , a user only has the permission at t for $s_a < t < e_a$.

4.4 Action Execution Level Enforcement

As mentioned above, when a user's privileges do not enforce a security property, another approach is to keep track of a user's past actions, and dynamically determine whether a requested action, if allowed, would result in violation of a security property.

Much work has been done that examines whether an access control policy ensures a security property is preserved in all or some possible future access states. For example, the safety problem centers on the question of whether or not given some policy and a start state of a system, a particular user can obtain a particular permission. For many practical policy and access state models, this problem is often hard or even undecidable. However, if one were to use a dynamic enforcement mechanism, the underlying goal, preventing a particular user from being able to exercise a particular right becomes much simpler. All a dynamic monitor needs to do would be to reject any action which would grant the given right to the given user or, alternately, to disallow any action which exercises that permission, effectively nullifying it during runtime.

As such, we assert that there are at least some security properties such that they are difficult to be maintained at the access control policy level, but easy to dynamically enforce. Conceptually, given an NFA H that defines a security property, the system maintains the set of states that a user is in. Whenever a user submits a request to take an action at time t , the system would first check whether the user has the right permission at t . If she does, the system would further consult H and determine, if the action is taken, what states the user would be in, and whether those states include a final state. If the answer is yes, the requested action would make the user conduct a sequence of actions that violates the security property. Therefore, this action request should be denied.

Though conceptually simple, the dynamic enforcement approach requires consulting the security property NFA every time a user requests to take an action. As a system's security property tries to characterize all the malicious action sequences, it is often fairly complex, and thus results in a complicated NFA. It would be expensive to consult the NFA, especially when we have to do so to handle every action request.

We observe that a security property is applied to users with all possible privileges. For a user with a particular set of privileges, however, very often only a small portion of the NFA is relevant. Not only some of the states in the NFA cannot be reached by the user, even for those that can be reached, they may not lead to the final state as the user may not possess necessary privileges anyway. By not keeping track of these irrelevant states, a system would be able to reduce the cost of dynamic enforcement. Thus, we investigate how to develop a "personalized" NFA to enforce a security property for a user given her privileges. This personalized NFA will remove all the irrelevant transitions and states, and thus accelerate dynamic enforcement.

Formally, given a string q and an NFA M , we denote $path(q, M)$ the set of paths in M from the start state to one of the final states such that the transition symbols on the path is q .

DEFINITION 4.4. *Given a set of privileges P and a security property H , we say a transition e (state s) in H is redundant regarding P if for every compatible action sequence q of P that are accepted by H , $e(s)$ does not appear in any path in $path(q, H)$.*

Clearly, if a state s is redundant, then all transitions to s and from s are also redundant. As a special case, if P enforces H , then all the states in H are redundant. This matches our intuition. If P enforces H , then we do not need any dynamic enforcement.

Next, we present an algorithm to identify all the redundant transitions and states given a set of privileges and a security property

Algorithm 1 ForwardSegments(H, T)

```

{ $H$  is the security property;  $T = (T_1, \dots, T_n)$  is the sequence
of time segments}
for all transitions  $\alpha$  in  $H$  do
   $forward(\alpha, H, T) := \emptyset$ 
end for
 $C := \{s_0\}$  where  $s_0$  is the start state of  $H$ 
for  $i = 1$  to  $n$  do
   $C' = \emptyset$ 
  while  $C' \neq C$  do
     $C' := C$ 
    for all  $c \in C'$  do
      for all transitions  $\alpha = (s, a, s')$  do
        if  $s = c$  and  $a \in A(T_i) \cup \{e\}$  then
          add  $T_i$  into  $forward(\alpha, H, T)$ 
          add  $s'$  into  $C'$ 
        end if
      end for
    end while
  end for
end for

```

H . Without loss of generality, we assume there is only one final state. Otherwise, we may introduce a new state s' and have an ϵ -transition from all the final states to s' . We then let s' be the only final state.

As introduced in section 4.1, given a set of privileges P , consider the time segments $T = (T_1, \dots, T_n)$ formed from the time points in P . We use $A(T_i)$ to denote the set of permissions available at T_i . The basic idea of our algorithm is as follows.

Let $\alpha = (s_1, a, s_2)$ be a transition in H , which means that at state s_1 we can transit to s_2 given input a . Note that a may be ϵ . We get all the time segments T_i during which α may happen, when starting from the start state, given T_1, \dots, T_i and the the permissions in each segment. Let $forward(\alpha, H, T)$ denote this set of segments.

Similarly, in a reverse direction, we get all the time segments during which this transition may happen, such that with permissions in segments T_i, \dots, T_n , it will be further lead to the final state. Let $backward(\alpha, H, T)$ denote this set of segments. Given an algorithm to compute $forward(\alpha)$, we can also easily compute $backward(\alpha, H, T)$. In more detail, we reverse H , i.e., we switch the start state and the final state, and reverse the direction of each transition. Let the reversed NFA be H' . We then also reverse the order of the time segments to have a segment sequence $T' = (T_n, T_{n-1}, \dots, T_1)$. It is easy to see that $backward(\alpha, H, T) = forward(\alpha, H', T')$.

THEOREM 2. *A transition $\alpha = (s_1, a, s_2)$ is not redundant if and only if $forward(\alpha, H, T) \cap backward(\alpha, H, T) \neq \emptyset$.*

The proof for this theorem has been omitted due to space concerns. Because we can efficiently compute $forward(\alpha, H, T)$ and $backward(\alpha, H, T)$ this provides us with a reasonable way of determining the set of redundant edges.

Once all the redundant transitions are identified, the redundant states are just those whose in and out transitions are all redundant. After removing all the redundant transitions and states from H , we got a simplified security property which can be more efficiently checked to enforce security properties.

The following pseudocode shows algorithms to identify redundant transitions.

Algorithm 2 RedundantTransitions(H, T)

{ H is the security property; $T = (T_1, \dots, T_n)$ is the sequence of time segments}
ForwardSegments(H, T)
let H' be the reverse of H
let $T' = (T_n, T_n - 1, \dots, T_1)$
ForwardSegments(H', T')
for each transition α in H **do**
 if $forward(\alpha, H, T) \cap forward(\alpha, H', T') \neq \emptyset$ **then**
 Return α
 end if
end for

In Algorithm 1, we begin with the start state, and compute the transitive closure with the permissions available in time segment T_1 . During this process, if a transition α is used, we add T_1 into $forward(\alpha, H, T)$. Given the final set of states reachable during T_1 , we continue with the next time segment T_2 . We keep doing so until the last time segment is processed. We can easily see that this algorithm is $O(nsa)$ where n is the number of time segments (which is linear in the size of the task), s is the number of transitions in H , and a is the number of states in H .

5. ENFORCING INTER-TASK PROPERTY

An inter-task property characterizes malicious action sequences when the user may abuse her privileges from different tasks. The enforcement of inter-task properties may seem the same as enforcing intra-task properties. Indeed, in both cases, we are given a set of privileges and a property expressed as an NFA, and tries to prevent action sequences that are accepted by the NFA.

However, one subtlety of the inter-task property makes the problem quite different. In intra-task property enforcement, once the privileges associated with a task is granted, they will not change during the fulfillment of the task. Therefore, we can analyze whether the granted privileges enforce a security property at the access control level. An inter-task property, however, not only concerns what a user can do with her current privileges but also with her future privileges which often cannot be predicated. That makes access state level enforcement not very useful. Even if a user's current privileges enforce a security property, we still need to keep track of her action sequence so that we can detect malicious action sequences when her current sequence is extended with actions allowed by future privileges.

Let's revisit the example in section 4.1. Suppose an inter-task property characterizes all action sequences where action a is done and subsequently action b is done. Also suppose a user's access state has only $(b, 0, 4)$ and $(a, 5, 10)$. Though the user cannot violate the property with her current privileges, if she takes action a , we have to remember it, because she might later acquire the permission to do b , in which case we will have to become concerned that she might have done a at some point in the past.

As such, we have to rely on action execution level enforcement to guarantee inter-task properties. In the previous section, we describe a technique to reduce the cost of dynamic enforcement by removing redundant transitions and states from H . Unfortunately, this technique cannot be applied to enforce inter-task properties either, due to the same reason. For example, let a state s be a redundant state, given a user's current privileges P . If s is still reachable, then we have to keep it in H . Even though H cannot transit from s to the final state with P , a future privilege may enable so. If we remove s from H , we may risk the possibility to fail to detect a malicious

action sequence in the future. Note that if s is not reachable at all given P , then we can safely remove it from H . However, this will not buy us any benefit in terms of dynamic enforcement efficiency, because it will not get involved when the system consults H .

5.1 Action Execution Level Enforcement

The above observation does not imply, however, that no analysis can be done for inter-task properties. We have previously argued that enforcing inter-task properties requires recording a user's actions. However, keeping record of their actions does not necessarily imply dynamically monitoring those actions at all times. If it is known that a user cannot violate a security property using their current privileges, then we do not need to actively monitor for security property violations, but instead we can simply keep a log of actions.

We can answer the question of whether or not a user can violate a security property given their past actions and their current tasks by again using an intersection of finite state machines as we did in section 4.1. However, the NFA modeling the possible actions that a user can take should begin with a string of transitions that represent the actions that the user has already taken so far. That is, rather than have a set of states representing different time segments where the transitions model what actions the user is allowed to take, for time segments prior to the current time, only the transitions representing the actions which the user actually took should remain.

Then, if we intersect the finite state automaton representing the user actions with the one representing the high level property, we can see whether or not it is possible for the user to violate the high level property. If it is impossible, then we can forgo keeping track of the state and simply record their actions for future analysis. If it is possible, then we must dynamically monitor and approve the user's actions.

5.2 Task Assignment

Another important type of security problems concern the assignment of tasks to users while preserving inter-task security properties. In many systems, tasks will arise in response to dynamic events. Either users may observe events and initiate tasks in response to them, or the system may observe events and have a means of assigning tasks to users. A question that naturally arises is whether or not it is safe to allow a user to assume a given task in light of their prior actions and their current tasks. We can solve this question for inter-task properties by applying the same technique we just used. We simply add the permissions of the new task starting at the current time to the set of permissions that we are modeling before we take the intersection. If the intersection is empty, then the new task can safely be given to the user. If we cannot guarantee that a task can safely be given to a user, we can either give it to another user or dynamically monitor this user's actions to ensure the high-level security properties are not violated.

A related problem is the question "what is the soonest time when a given task can be safely assigned to a particular user?" This question can be solved by repeatedly applying the technique we have just described, except beginning the new task in different possible time segments. Note that only the segment boundaries are going to possibly be the correct answer to the question of what the soonest time is, thus making the running time of this algorithm polynomial relative to the number of privileges currently granted to the user plus the size of the high level property.

These techniques can answer questions about assigning particular tasks to particular users. However, when we examine more complex questions, we find that answering them becomes intractable.

One problem that might logically be considered is the question of

given a set of tasks, what is the largest subset of those tasks which a user can safely assume? Unfortunately, this problem is NP-hard. We have omitted the reduction due to space concerns.

If we examine the multiple-user case, another possible problem which we might wish to examine is whether or not it is possible to assign a series of tasks and their associated privileges to a set of users such that no user will be able to violate the security property. Specifically, if given a set of tasks and each of which has a set of privileges associated with it, given a set of users, and given a security policy, is it possible to create a schedule such that no user is able to use their privileges to violate the security policy?

This problem turns out to also be NP-hard. This can be shown using a reduction which has also been omitted for reasons of space.

6. RELATED WORK

Several papers have examined the process of comparing high level security properties to models of system behavior in order to determine whether or not the security properties are guaranteed to hold. Most of these papers focus on the system level or the program level and assume that a complete model of system or program behavior is available and that this behavior is not restricted by low-level security policies.

The work most similar to ours is Sekar *et al.* [18], which is about model-carrying code. Similarly to our work, they model security properties as automata that describe what should not happen in a system and use an automata to model what actions a given program can do. Then they use intersection of automatas to determine whether or not the security property will hold. However, they assume that there is a no low-level security policy that might prevent certain actions from occurring. Hence, if a property fails to hold for a given program the only options are to loosen the security policy or not to run the program. Our approach focusses on the implications of low-level security policies.

However, we are not unique in trying to reason about the relationship between low-level security policies and high-level security properties. One paper that takes this approach is Schaad *et al.* [15], which examines the interplay between the two in a particular scenario relating to financial controls in a loan-origination system. Our paper answers similar questions, but with much greater generality.

Another paper that addresses whether or not low-level policies are sufficient to enforce high-level properties is Wang and Li [21]. They seek to address a somewhat similar question to ours, but with a very different formulation of high-level security policies. Rather than focusing on undesirable action sequences, their high-level properties consider the number and categories of users that must be involved in task fulfillment. Their analysis seeks to determine whether the low-level assignment of privileges ensures that the candidates for fulling the task will be guaranteed to involve appropriate participants, as characterized by the high-level property.

Another paper which examines specifying high level properties which constrain the assignment of users to roles is Bertino *et al.* [5] which examines security in workflow systems. This paper is similar to ours in that it deals with the interplay between high level properties and low level policies and deals with both static and runtime analysis. However, it is limited to analyzing the assignment of users to roles, and so only deals with enforcement on an access control policy and state level, but not on an action execution level.

Several papers examine questions related to the expressiveness of automata for enforcing security policies. Our work is complementary to theirs because we focus on questions related to specific analysis and to methods of dynamic enforcement rather than on expressiveness in general. Schneider [17] shows that all policies which can be enforced by execution monitoring can be modeled as

finite state automata. Bauer *et al.* [3] expands on Schneider's work by looking at automata models for execution monitors which are more flexible than the ones examined by Schneider with respect to how the system handles violations.

Eckmann *et al.* [7] define a rich language called STATL for specification of undesirable sequences of actions for the purpose of intrusion detection. STATL has been shown to be effective in this context. Viewed in terms used in the current paper, STATL is used to specify high-level properties. There is no analogue of our access state in the STATL system, so there is no consideration of its interaction with high-level properties. However, it supports the significance of our work that specification at a high-level of undesirable behaviors has been so effective in this application.

Martinelli *et al.* [13] introduce a policy language that specifies acceptable sequences of actions. The allowable sequences are given by concatenation of actions indicating the acceptable order in which actions can take place. In addition to specifying actions, the policy language enables the specification and testing of state information in the form of variable bindings. Testing is performed by predicates, interleaved with the specified action sequence, that indicate constraints that must hold, usually on the arguments to the following action. These further constrain the allowable action sequences. In contrast with our's, this work does not consider multiple levels at which security objectives may be expressed, nor the interaction of such objectives.

Thomas and Sandhu [19,20] introduced an access control model based on tasks assigned to users. Their Task-Based Authorization Controls (TBAC) raise the level of abstraction of the granularity with which authorizations are granted. The aim is to enable authorization policy to be defined at the application level, rather than at the level of low-level resources such as files and database records. This work focuses on the design and implementation of authorization models in which the authorization state evolves along with the progress of multi-stage tasks. It does not discuss issues concerning the consistency of security requirements specified at different levels of abstraction, nor tools for managing them. By contrast, we have a much simpler task model, and focus on the enforcement of security objectives articulated at various levels of abstraction.

Bertino *et al.* [4] introduced a Temporal Role-Based Access Control (TRBAC) model in which roles can be associated with periodic schedules indicating periods during which the role is enabled (can be activated) and is disabled. They also provide a trigger feature that can be used to establish dependencies between enabling and disabling actions for different roles. An analysis is presented; however it does not concern whether some high-level security property is satisfied. Rather, it determines whether a given specification contains any ambiguities or inconsistencies.

A number of different papers have used some form of formal model checking to statically verify whether or not access control policies have certain properties, such as safety and reachability. One example is Fisler *et al.* [8] who present a system called Margrave which can analyze XACML policies. Another is Ahmed and Tripathi [1] who use model checking to establish security properties of access control policies in cooperative work systems.

The work related to model-checking which is closest to our own is Dougherty *et al.* [6], which reasons not just about policies but about how policies interact with a dynamic environment over time. However, what all the above works have in common is that they simply determine whether or not a given policy has a given property. If it does not have the desired property, it is not clear how to enforce the property. The presumed methodology is that one would run the static analysis tool and then manually modify the policy to make it have the desired properties, iterating as necessary.

By contrast, we present additional alternatives, such as dynamically enforcing the high level properties, which allows us to simply prevent undesired states from occurring rather than having to modify the policy to make such states unreachable.

7. CONCLUSION

In this paper we have presented efficient algorithms to statically check whether a user's privileges enforces security properties. We also present an efficient algorithm to reduce the size of security properties to improve the efficiency of dynamic enforcement at the action execution level for intra-task properties.

For inter-task properties, we have shown the necessity of carrying-out at least some action execution level monitoring or logging, but we provide an efficient algorithm to determine in which cases monitoring is necessary and in what cases simple logging is sufficient. However, we show that when there are task allocation decisions to be made, NP-Hard problems quickly arise if we are attempting to allocate tasks in a manner which avoids dynamic monitoring.

There are many interesting avenues for future research. In this paper, we have modeled a monolithic system in which all access is controlled by a central reference monitor. One extension would be to consider how a high-level policy could be enforced when it needed to be applied over several distinct but related applications, such as front-end programs which have their own reference monitors, but share a common back-end database. We are also interested in applying the techniques to domain-specific applications. By considering the properties of specific access control policies of an application, it would be possible to develop efficient algorithms to enforce security properties under user collusion.

Acknowledgments

This research was sponsored by the NSF through IIS CyberTrust grant 0430166 (NCSU), CNS-0716210 (NCSU), CNS-0716750 (UTSA), CCF-0524010 (UTSA), and ITR award CCR-0325951 (UTSA, through a sub-award from Brigham Young University). We thank our anonymous reviewers for their helpful comments.

8. REFERENCES

- [1] Tanvir Ahmed and Anand R. Tripathi. Static verification of security requirements in role based cscw systems. In *SACMAT '03: Proceedings of the eighth ACM symposium on Access control models and technologies*, pages 196–203, New York, NY, USA, 2003. ACM Press.
- [2] Bowen Alpern and Fred B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3):117–126, 1987.
- [3] L. Bauer, J. Ligatti, and D. Walker. More enforceable security policies, 2002.
- [4] Elisa Bertino, Piero Andrea Bonatti, and Elena Ferrari. Trbac: A temporal role-based access control model. *ACM Trans. Inf. Syst. Secur.*, 4(3):191–233, 2001.
- [5] Elisa Bertino, Elena Ferrari, and Vijay Atluri. The specification and enforcement of authorization constraints in workflow management systems. *ACM Transactions on Information and System Security*, 2(1):65–104, February 1999.
- [6] Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. Specifying and reasoning about dynamic access-control policies. In *IJCAR*, pages 632–646, 2006.
- [7] Steven T. Eckmann, Giovanni Vigna, and Richard A. Kemmerer. Statl: an attack language for state-based intrusion detection. *J. Comput. Secur.*, 10(1-2):71–103, 2002.
- [8] Kathi Fisler, Shriram Krishnamurthi, Leo A. Meyerovich, and Michael Carl Tschantz. Verification and change-impact analysis of access-control policies. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 196–205, 2005.
- [9] Jeremy Frank and Matt Bishop. Extending the take-grant protection system. Technical Report Technical Report, Department of Computer Science, University of California at Davis, 1996.
- [10] Michael A. Harrison, Walter L. Ruzzo, and Jeffrey D. Ullman. Protection in operating systems. *Communications of the ACM*, 19(8):461–471, August 1976.
- [11] Anita K. Jones, Richard J. Lipton, and Lawrence Snyder. A linear time algorithm for deciding security. In *17th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 33–41, October 1976.
- [12] Ninghui Li, William H. Winsborough, and John 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.
- [13] Fabio Martinelli, Paolo Mori, and Anna Vaccarelli. Towards continuous usage control on grid computational services. *icas-icns*, 0:82, 2005.
- [14] Ravi S. Sandhu. The typed access matrix model. In *SP '92: Proceedings of the 1992 IEEE Symposium on Security and Privacy*, page 122, Washington, DC, USA, 1992. IEEE Computer Society.
- [15] Andreas Schaad, Volkmar Lotz, and Karsten Sohr. A model-checking approach to analysing organisational controls in a loan origination process. In *SACMAT '06: Proceedings of the eleventh ACM symposium on Access control models and technologies*, pages 139–149, New York, NY, USA, 2006. ACM Press.
- [16] Andreas Schaad, Pascal Spadone, and Helmut Weischel. A Case Study of Separation of Duty Properties in The Context of the Austrian “eLaw” Process. In *Proceedings of the 20th Annual ACM Symposium on Applied Computing (SAC'05)*, Santa Fe, Nex Mexico, March 2005.
- [17] Fred B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, 2000.
- [18] R. Sekar, V. Venkatakrishnan, S. Basu, S. Bhatkar, and D. DuVarney. Model-carrying code: A practical approach for safe execution of untrusted applications, 2003.
- [19] R. K. Thomas and R. S. Sandhu. Towards a task-based paradigm for flexible and adaptable access control in distributed applications. In *NSPW '92-93: Proceedings on the 1992-1993 workshop on New security paradigms*, pages 138–142, New York, NY, USA, 1993. ACM Press.
- [20] Roshan K. Thomas and Ravi S. Sandhu. Task-based authorization controls (tbac): A family of models for active and enterprise-oriented authorization management. In *Proceedings of the IFIP TC11 WG11.3 Eleventh International Conference on Database Security XI*, pages 166–181, London, UK, UK, 1998. Chapman & Hall, Ltd.
- [21] Qihua Wang and Ninghui Li. Direct static enforcement of high-level security policies. In *ASIACCS '07: Proceedings of the 2nd ACM symposium on Information, computer and communications security*, pages 214–225, New York, NY, USA, 2007. ACM Press.