Privilege Graph: an Extension to the Typed Access Matrix Model

Marc Dacier, Yves Deswarte

LAAS-CNRS & INRIA
7, avenue du Colonel Roche
31077 Toulouse, France
(dacier@laas.fr, deswarte@laas.fr)

Abstract. In this paper, an extension to the TAM model is proposed to deal efficiently with authorization schemes involving sets of privileges. This new formalism provides a technique to analyse the safety problem for this kind of schemes and can be useful to identify which privilege transfers can lead to unsafe protection states. Further extensions are suggested towards quantitative evaluation of operational security and intrusion detection.

1 Introduction

The problem of controlled sharing of information in multi-user computing systems has been the subject of a large literature for more than 20 years. Many solutions have been advanced. Among them, the various access control based models are the most widely-known. The key abstractions they handle are those of subjects, objects and access rights. They also make use of two other concepts: the *protection state* and the *authorization scheme* [14]. A protection state is defined by the sets of rights held by each individual subject. The authorization scheme is defined by a set of rules that lets the protection state evolve by the autonomous activity of the subjects.

The primary goal of these models is to offer an efficient resolution of the so-called *safety problem* defined by Harrison, Ruzzo and Ullman in [7]. It consists in identifying states[1] that violate the security constraints and that are reachable given a initial state and an authorization scheme. Their model, the *HRU* model, possesses a very broad expressive power but appears to be inefficient in most practical cases. As a consequence, other models have been suggested. Lipton and Snyder, in [11], set a model forth that can solve the safety problem in linear time but at a price of poor expressiveness. To fill the gap, various solutions have been proposed (*SPM* [12], *ESPM* [1], *NMT* [13], ...), the most promising of which are *TAM* [14] and *ATAM* [2]. These models are expressive enough — as claimed by the authors — to model most security policies of practical interest and still offer strong safety properties.

In this paper, we focus on a specific class of authorization schemes where a user can grant a, possibly large, set of rights to other users. Such authorization schemes are quite common in most real-world situation and, therefore, are worth considering. Two

1. In the rest of the paper, we use indifferently "state" for "protection state".

different solutions to the safety problem in that case, using the *TAM* formalism, are discussed. It follows that *TAM* can effectively solve that specific class of safety problems but in a non optimal way regarding ease of use and algorithmic complexity. Hence, we propose to enhance *TAM* with a complementary formalism based on a graph of sets of privileges[2] . This approach is presented as well as its main advantages and limitations. Furthermore, we indicate two other possible applications of our formalism, namely in the context of intrusion detection techniques and in the context of quantitative evaluation of computing systems security.

The paper's organization is as follows. Section 2 briefly summarizes previous works, highlighting the connections between them. Section 3 presents a specific authorization scheme using *TAM* formalism and, taking a simple example, gives three different solutions to the safety problem. Section 4 suggests a more efficient approach to solve that specific problem, based on a privilege graph. Section 5 justifies the authorization scheme used in Section 2 and 3 by giving real-world examples of such a scheme. Section 6 describes three possible applications of our approach and Section 7 gives a conclusion.

2 Background

Access controls models originate in Lampson's famous access matrix model [9] but Harrison, Ruzzo and Ullman were the first in [7] to formalize the *safety problem* in their *HRU* model. Their results are deceptive in the sense that they prove that, using *HRU*, the general safety problem is not decidable. In response to these negative results, other approaches have been suggested. One of them is the *take-grant* model [3, 10, 11, 17, 18]. In [11], Lypton and Snyder described an algorithm that can solve the safety problem, for this model, in linear time. Sadly, efficiency is acquired at the price of expressiveness. Indeed, in this model, it is not possible to restrict the granting of rights. One is allowed to grant all the rights one holds to someone else or none of them. Therefore, as noted in [17], the model appears to be disappointingly weak when applied to typical protection problems. More recently, work has been carried out to relax some of the assumptions of this model [4, 6, 19, 20] but, still, it remains ill-adapted for practical applications.

In [12], Sandhu defined the *Schematic Protection Model* (*SPM*) whose intent is to fill the gap in expressive power between the *take-grant* and the *HRU* models. However, attempts to prove the equivalence of *SPM* to monotonic *HRU* have remained unsuccessful and another model *ESPM* (*Extended SPM*) had to be designed for that purpose [1].

---

2. In order to be consistent with previous work we use the same convention as in [14]: "We view *privilege* as a primitive undefined concept. For the most part, privileges can be treated as synonymous with access rights. However, there are privileges such as security level, type or rôle, which are usually represented as attributes of subjects and objects rather than as access rights".

Later, Sandhu proposed the *Typed Access Matrix* model (*TAM*), a refinement of the *HRU* model in which he introduces strong typing [14]. A *TAM* model is characterized by a finite set $\mathcal{R}$ of rights, a finite set $\mathcal{T}$ of objects types, and a set $\mathcal{T}_s$ of subjects types ($\mathcal{T}_s \subseteq \mathcal{T}$). These sets are used to define the protection state by means of a typed access matrix. The authorization scheme consists of $\mathcal{R}$, $\mathcal{T}$ and a finite collection of commands.

command $\alpha(X_1 : t_1, X_2 : t_2, ..., X_k : t_k)$
if $r_1 \in [X_{s_1}, X_{o_1}] \wedge r_2 \in [X_{s_2}, X_{o_2}] \wedge r_m \in [X_{s_m}, X_{o_m}]$
then $op_1; op_2; ... ; op_n$

Table 1. Format of a TAM command

A *TAM* command has the format shown in Table 1 where α is the *name* of the command; $X_1, X_2, ..., X_k$ are *formal parameters* whose types are respectively $t_1, t_2, ..., t_k$; $r_1, r_2, ..., r_m$ are rights; and $s_1, s_2, ..., s_m$ and $o_1, o_2, ..., o_m$ are integers between 1 and k. Each op_i is one of the *primitive operations* shown in Table 2, in which $z \in \mathcal{R}$ and s and o are integers between 1 and k.

enter z **into** $[X_s, X_o]$	**create subject** X_s of type t_s	**create object** X_o of type t_o
delete z **from** $[X_s, X_o]$	**destroy subject** X_s of type t_s	**destroy object** X_o of type t_o

Table 2. The six primitive operations of TAM.

In the same paper [14], Sandhu demonstrates that it is possible in many practical cases to make safety tractable without loss of expressive power. An algorithm is described to compute the *maximal state*, i.e., a state where no rule can be applied any more. In [2], an augmented version of *TAM* has been proposed which allows to test for absence of rights. The aim of *ATAM* is to easily allow separation of duties in the definition of the authorization scheme. It has been proved in [15] that both models are formally equivalent in their expressive power.

This result highlights two points: i) TAM is a general model and, therefore, can be used as a reference, ii) ATAM existence shows the usefulness of enriching TAM when dealing with some specific authorization schemes.

To conclude with our historical review, it is worth mentioning that, in [13], the problem of non-monotonic transfer of rights has been considered. The model proposed (*NMT*) exhibits some promising results, though no formal proof of its expressiveness is given. Moreover, even if safety is shown to be decidable for *NMT*, yet the decision procedure has exponential complexity.

3 The problem of granting sets of rights

3.1 A simple example

As indicated in Section 2, the expressive power of the *take-grant* model is very restricted. In this model, a grant action can focus neither on a given object, nor on a given right. One is only allowed to grant a given subject every rights in one's possession on every object. On the contrary, *TAM* commands are such that they allow the granting of one right on one object to one subject. In real-world situations, the problem of granting sets of privileges appears to be quite common and seems to require an

expressiveness virtually located between the expressive power of these two models. This is the reason why it can be interesting to consider the ability of *TAM* to deal efficiently with such authorization schemes.

In the following, we will focus on some of these schemes which are characterized by empty intersections between the granted privileges and those checked in the conditional parts of the commands. Such schemes present interesting properties to solve the safety problem, namely the absence of cycles[3].

Throughout this paper, we use a simple example where the set $\mathcal{T}$ of types is defined as $\mathcal{T}$= {*user*, *file*$_1$, *file*$_2$, *file*$_3$} and the set $\mathcal{R}$ of rights as $\mathcal{R}$ = {*e*,*o*,*r*,*w*} (where the letters stand for the mnemonics of "*execute*", "*own*", "*read*" and "*write*" respectively). Table 3 gives the initial protection state for this example[4].

	f : *file*$_1$	*g* : *file*$_2$	*h* : *file*$_3$	*i* : *file*$_3$
a : *user*	*e*,*o*,*r*,*w*			
b : *user*	*e*	*r*,*w*	*r*	
c : *user*		*e*,*o*,*r*,*w*	*o*,*r*,*w*	*r*

Table 3. A simple Typed Access Matrix

The authorization scheme we consider consists of two rules:

Rule 1: If a user U_1 owns a file F of type *file*$_1$ and if a user U_2 can execute that file F, then U_2 can grant U_1 all the *read* rights that U_2 holds on files of type *file*$_3$.

Rule 2: if a user U_1 can write into a file F of type *file*$_2$ and if a user U_2 owns that file F, then U_2 can grant U_1 all *read* and *write* rights that U_2 holds on files of type *file*$_3$

Section 5 explains this choice by showing that these rules are representative of real-world problems. Based on these protection state and authorization scheme, we consider in the next subsections the two following basic safety problems:

Q1: Can the system reach a state where the user *a* can gain the *r* right on *i* ?
Q2: Can the system reach a state where the user *a* can gain the *w* right on *h* ?

3.2 First Solution: Direct Application of TAM

The most obvious way to model this authorization scheme, using TAM formalism, is by defining the three following commands:

command $R_1(U_1$: *user*, U_2: *user*, F_1: *file*$_1$, F_2: *file*$_3)$
if $o \in [U_1,F_1] \wedge e \in [U_2,F_1] \wedge r \in [U_2,F_2]$
then enter *r* **into** $[U_1, F_2]$

This first command expresses the first rule of our authorization scheme.

---

3. As discussed in Section 4.4, performance reasons can impose another constraint to the authorization scheme: there should be small mutual intersections between granted privilege sets.
4. Empty rows and empty columns are not represented for the sake of conciseness.

command $R_{2read}(U_1$: *user*, U_2: *user*, F_1: *file*$_2$, F_2: *file*$_3$)
if $w \in [U_1,F_1] \wedge o \in [U_2,F_1] \wedge r \in [U_2,F_2]$
then enter r **into** $[U_1, F_2]$

command $R_{2write}(U_1$: *user*, U_2: *user*, F_1: *file*$_2$, F_2: *file*$_3$)
if $w \in [U_1,F_1] \wedge o \in [U_2,F_1] \wedge w \in [U_2,F_2]$
then enter w **into** $[U_1, F_2]$

These last two commands express the second rule of our authorization scheme. Table 4 represents the maximal state of this system. Obtaining this maximal state can be achieved in different ways. Here is an example of one sequence of command applications that reaches it: $R_1(a,b,f,h)$ - $R_{2\text{read}}(b,c,g,i)$ - $R_{2\text{write}}(b,c,g,h)$ - $R_1(a,b,f,i)$. Four command applications are required. Resolving the safety problem is straightforward with this method: a simple inspection of the matrix leads us to answer the question Q1 positively and Q2 negatively.

	f : *file*$_1$	*g* : *file*$_2$	*h* : *file*$_3$	*i* : *file*$_3$
a : *user*	*e,o,r,w*		*r*	*r*
b : *user*	*e*	*r,w*	*r,w*	*r*
c : *user*		*e,o,r,w*	*o,r,w*	*r*

Table 4. Maximal state for the first solution

It is important to note that the number of command applications is directly proportional to the number of files of type *file*$_3$. This can be highlighted by a rough generalization of our example. Consider an authorization scheme defined by the only rule R_1; consider a protection state with n users (U_1, U_2,..., U_n), each of them having the r right on m different files of type *file*$_3$. Suppose also the existence of n-1 files of type *file*$_1$ (F_1,F_2,...,F_{n-1}) such that: $\forall j, (1 \le j \le n-1) \quad o \in [U_j,F_j] \wedge e \in [U_{j+1},F_j]$. In this case, the maximal state can be reached by applying m times the command R_1 with U_n and U_{n-1} as parameters, $2m$ times with U_{n-1} and U_{n-2},..., $(n-1)m$ times with U_1 and U_2. Thus, the amount of command applications required to build the maximal state in this case is equal to:

$$m + (m+m) + \ldots + (m + \ldots + m) = m \times (1 + \ldots + n-1) = \frac{m \times n \times (n-1)}{2}$$

Hence, in this case, we need $O(mn^2)$ command applications to reach the maximal state. It is shown in the next Subsection, that one could easily take profit of the richness of TAM to find a much more efficient modelization.

3.3 Second solution: Introducing ad-hoc privileges

The number of applications could remain constant, whatever is the number of *file*$_3$ present in the system, thanks to the definition of two *ad-hoc* privileges *tr* and *tw*, where *tr* (resp. *tw*) stands for the mnemonic of "*take read*" (resp. "*take write*"). Hence, we can express the same authorization scheme with only two rules:

command $R'_1(U_1$: *user*, U_2: *user*, F_1: *file*$_1$)
if $o \in [U_1,F_1] \wedge e \in [U_2,F_1]$
then enter *tr* **into** $[U_1, U_2]$

command $R'_2(U_1$: *user*, U_2: *user*, F_1: *file*$_2)$
if $w \in [U_1,F_1] \wedge o \in [U_2,F_1]$
then enter *tr* **into** $[U_1, U_2]$; **enter** *tw* **into** $[U_1, U_2]$

Table 5 shows the maximal state obtained by applying $R'_1(a,b,f)$ and $R'_2(b,c,g)$ to the protection state defined in Table 3.

	b	*c*	*f* : *file*$_1$	*g* : *file2*	*h* : *file*$_3$	*i* : *file*$_3$
a : *user*	*tr*		*e,o,r,w*			
b : *user*		*tr,tw*	*e*	*r,w*	*r*	
c : *user*				*e,o,r,w*	*r,w*	*r*

Table 5. Maximal State with the *tr* and *tw* privileges.

It was shown in the previous section that the rough generalization of our example required $O(mn^2)$ to build the maximal state. Now, with this solution, it would require only n-1 command applications but the answer cannot be found directly in the matrix as before. The privileges *tr* and *tw* have semantics that must be considered to answer that question.

Actually, these privileges act like pointers. We note that introducing special rights which have such rôle is nothing new in a *TAM* model. This is the trick used by Sandhu and Ganta in [15] to show that *TAM* and *ATAM* are formally equivalent in their expressive power. In our case, a recursive function *get_answer* will be used to solve the safety problem. Its algorithm[5] is given below:

function *get_answer*(U_1: *user*, *F*: *object*, *R*: *right*)
if $R \in [U_1, F]$
  then **answer is** *YES*
  elsif $(R = r)$
    then foreach (*U*: *user* **such that** $tr \in [U_1, U]$) {*get_answer*(*U*, *F*, *R*)};
  end if;
  elsif $(R = w)$
    then foreach (*U*: *user* **such that** $tw \in [U_1, U]$) {*get_answer*(*U*, *F*, *R*)};
  end if;
end if

Hence, *get_answer*(*a,i,r*) will first search in the matrix if *a* has the right *r* on *i*. If not, it looks for a pointer to another subject who would hold this right or who would have another pointer to a third subject, etc. In this particular case, getting the answer requires five inspections[6] of the matrix for the question Q1 and three[7] for Q2. It is important to note that these results are independent of the number of files of type *file*$_3$ and linear with respect to the number of users (to compare with $O(mn^2)$ of the rough generalization).

---

5. The function *get_answer* returns *YES* if the answer is positive. It returns nothing if the answer is negative. This is a simplified version of the algorithm. A complete one should take care of the existence of possible cycles.
6. **Q1** : $r \notin [a,i] \Rightarrow tr \in [a,b] \Rightarrow r \notin [b,i] \Rightarrow tr \in [b,c] \Rightarrow r \in [c,i] \Rightarrow$ *YES*
7. **Q2**: $w \notin [a,h] \Rightarrow tw \notin [a,b] \Rightarrow tw \notin [a,c] \Rightarrow$ *NO*

Of course, the astute reader has noticed that this example has been designed on purpose. In fact, this solution is directly proportional to the *number of pointers.* If we consider a protection state with many users, none of them having the right to read *"i"*, this solution is clearly worse than the first one. Indeed, it will probably impose us to follow a long list of pointers to eventually reach a negative conclusion that could have been immediately derived with the first method! However, it will be shown in Section 5 that our example is representative of many common, yet specific, real-world situations. In these cases, the second solution is better than the first one.

3.4 Discussion

As can be seen, the introduction of ad-hoc privileges in a given TAM model can improve dramatically its efficiency in some given situations. Unfortunately, this requires at least a new right for each class of set of privileges that can be granted. Each new right definition induces the rewriting of the *get_answer* algorithm in order to take the new pointer's semantics into account. Such task could rapidly become cumbersome with the growth of the set of commands.This solution looks promising but it suffers from its lack of modularity.

As a result of this comment, one could be tempted to define a new approach using only one kind of pointer towards new virtual users created on purpose. For instance, to represent that the user b grants to a all his r rights on objects of type $file_3$, one could create a new user β, give him all the r rights that b possesses on objects of type $file_3$ and introduce in $[a,\beta]$ a pointer called, for example, *take_set*. However, such solution is not easy to implement with TAM. The two following scenarios highlight this point:

1) Once β created, suppose that b acquires a new right r on an object of type $file_3$, then β's privileges must be updated ! Thus, we must define rules to take care of every change in b's privileges.

2) Once β created, b acquires all the r and w rights on all objects of type $file_3$ that c has. Therefore, a new user γ is created (with all the r and w rights on all objects of type $file_3$ that c has) and the right *take_set* has to be inserted in $[b,\gamma]$. How should this update be taken in consideration for the update of β ? If we put a *take_set* right into $[\beta,\gamma]$ then a, by transitivity, will gain the w rights of c, which is not correct ! It is clear, therefore, that the required update rules are not easy to define. Actually, this approach suffers from the same lack of modularity that the one explained hereabove.

If we could *define* β rather than *create* it, as we do by putting access rights in matrix cells, then the problem, explained in the first scenario, would disappear because no update would be necessary any more. Indeed, its definition would be independent of the evolution of the privileges of b.

Furthermore, if we have such formal definition of β, then we can also get rid of the second problem by integrating the update commands into the usual commands. This is explained into the next section where we propose an extension to TAM that offers such formal definitions of sets of privileges.

4 The Privilege Graph.

4.1 Definitions

It is important to note that our formalism is not aimed at replacing *TAM* which has proved to be very powerful in many situations but rather as an efficient complementary notation. Our solution is based on a directed graph, the nodes of which are sets of triples $(U, O, \Sigma_{\mathcal{R}})$ where U is a subject, O an object and $\Sigma_{\mathcal{R}}$ a set of rights $(\Sigma_{\mathcal{R}} \subseteq \mathcal{R})$. For each type θ ($\theta \in \mathcal{T}$), we define Σ_θ as the set of all objects of type θ. We define $\Sigma_{\mathcal{T}}$ as a union of sets: $\Sigma_{\mathcal{T}} = \bigcup_{\theta \in \mathcal{T}} \Sigma_\theta$.

Nodes do represent sets of privileges on sets of objects. A node is not supposed to correspond to any row in any access matrix. It defines a set of privileges that can be granted to other users. For instance, suppose that a rule specifies that the user b can grant all the read rights he has on every file of type $file_3$. The application of this rule will create a node defined as: $N = \{(b,O,r) | O \in \Sigma_{file_3} \wedge r \in [b,O]\}$. This node represents a subset of the privileges that b effectively holds in the access matrix when the rule is applied but this subset is not "frozen". Indeed, such a definition could take "new" rights into account, i. e. rights entered into the matrix for b by some rule application after the creation of the node. This is possible because the content of the node is characterized by a formal definition rather than by the enumeration of its contents.

For each user U, we define $\mathcal{M}_U$ as the maximal set of privileges that U could get. This set can be identified with the row corresponding to U in the classical maximal state, defined in Section 3.2. Formally[8]:

$$\forall U \qquad \mathcal{M}_U = \{(U,O,\Sigma_{\mathcal{R}}) \,|\, O \in \Sigma_{\mathcal{T}} \wedge \left(\Sigma_{\mathcal{R}} = \left(\mathcal{R} \cap [U,O]^*\right)\right) \wedge \Sigma_{\mathcal{R}} \neq \varnothing\} .$$

It is important to note that we do not define any node that corresponds to that definition. Actually, as explained below, we never have to compute this maximal state.

The existence of a directed edge in the graph from a node N_1 to a node N_2 implies that $\forall U, U \in \Sigma_{users}, \mathcal{M}_U \supseteq \mathcal{N}_1 \Rightarrow \mathcal{M}_U \supseteq \mathcal{N}_2$. Roughly speaking, the existence of an edge from a node N_1 to a node N_2 means that, every user who can acquire N_1, can acquire N_2. The formal definition is not used in practice because, as already mentioned, we do not want to compute $\mathcal{M}_U$.

Edges and nodes are created by successive applications of the rules that compose the authorization scheme. Therefore, we add two primitive operations to *TAM: make_node* and *make_edge*. The operation *make_node* (resp. *make_edge*) will create a node in the graph (resp. an edge) only if this node (resp. edge) does not already exist.

4.2 Construction

We have already mentioned that we do not have to compute $\mathcal{M}_U$ at any time. Indeed, with this method, solving the safety problem is reduced to finding a path in a digraph. This is highlighted by the following protocol:

1) For each subject U in the initial protection state, create a node defined as follows $\mathcal{N}_U = \{(U,O,\Sigma_{\mathcal{R}}) \,|\, O \in \Sigma_{\mathcal{T}} \wedge (\Sigma_{\mathcal{R}} = (\mathcal{R} \cap [U,O])) \wedge \Sigma_{\mathcal{R}} \neq \varnothing\}$.

8. The notation $[U,O]^*$, instead of $[U,O]$, indicates that we do refer to the matrix representing the maximal state.

At any time, this definition represents the privileges present in the matrix for this user[9].

2) Apply the commands up to reach a maximal state[10]. The maximal state, in this case, is characterized by the matrix and by the graph. Both are needed to define the final protection state.

3) Reformulate the safety problem in terms of two conflicting sets of nodes and find if a path exists between these two sets.

Each step of this process is better understood by showing how it is achieved in our running example. Therefore, we need to define new commands to characterize our authorization scheme:

command $R''_1(U_1$: *user*, U_2: *user*, F_1: *file*$_1$, $\mathcal{N}_1$: *node*, $\mathcal{N}_2$: *node*)
if $o \in [U_1,F_1] \wedge e \in [U_2,F_1] \wedge \{(U_1,O,r) | O \in \Sigma_{file_3} \wedge r \in [U_1,O]\} \subseteq \mathcal{N}_1$
then **make_node** $\mathcal{N}_2 = \{(U_2,O,r) | O \in \Sigma_{file_3} \wedge r \in [U_2,O]\}$
make_edge from $\mathcal{N}_1$ **to** $\mathcal{N}_2$
end if

Compared to command R'$_1$, we see that R''$_1$ contains a third test in its conditional part:

1) $o \in [U_1,F_1] \wedge e \in [U_2,F_1]$ deals with the authorization scheme itself (identical to R'$_1$).

2) $\{(U_1,O,r) | O \in \Sigma_{file_3} \wedge r \in [U_1,O]\} \subseteq \mathcal{N}_1$ identifies in the graph the node to which the primitive operation *make_edge* will be applied. The rule will be applied for each node that satisfies this definition. As a result, for a given triple (U_1,U_2,F_1), this rule will create one node N_2 but several links could be created, originating from various nodes N_1 to N_2.

Keeping the same principles in mind, we define two new rules R''$_{2read}$ and R''$_{2write}$ to implement the second rule. Namely:

command $R''_{2read}(U_1$: *user*, U_2: *user*, F_1: *file*$_2$, $\mathcal{N}_1$: *node*, $\mathcal{N}_2$: *node*)
if $w \in [U_1,F_1] \wedge o \in [U_2,F_1] \wedge \{(U_1,O,r) | O \in \Sigma_{file_3} \wedge r \in [U_1,O]\} \subseteq \mathcal{N}_1$
then
make_node $\mathcal{N}_2 = \{(U_2,O,r) | O \in \Sigma_{file_3} \wedge r \in [U_2,O]\}$
make_edge from $\mathcal{N}_1$ **to** $\mathcal{N}_2$
end if

command $R''_{2write}(U_1$: *user*, U_2: *user*, F_1: *file*$_2$, $\mathcal{N}_1$: *node*, $\mathcal{N}_2$: *node*)
if $w \in [U_1,F_1] \wedge o \in [U_2,F_1] \wedge \{(U_1,O,w) | O \in \Sigma_{file_3} \wedge w \in [U_2,O]\} \subseteq \mathcal{N}_1$
then
make_node $\mathcal{N}_2 = \{(U_2,O,w) | O \in \Sigma_{file_3} \wedge w \in [U_2,O]\}$
make_edge from $\mathcal{N}_1$ **to** $\mathcal{N}_2$
end if

---

9. $\mathcal{N}_U$ is identical to $\mathcal{M}_U$ if and only if there is no edge originating from $\mathcal{N}_U$. In general, this is not true and we have $\mathcal{N}_U \subseteq \mathcal{M}_U$.
10. The task of constructing the graph is finite because the number of nodes is at most a linear combination of the number of cells in the maximal state matrix of Section 3.2 — the size of which is finite [14].

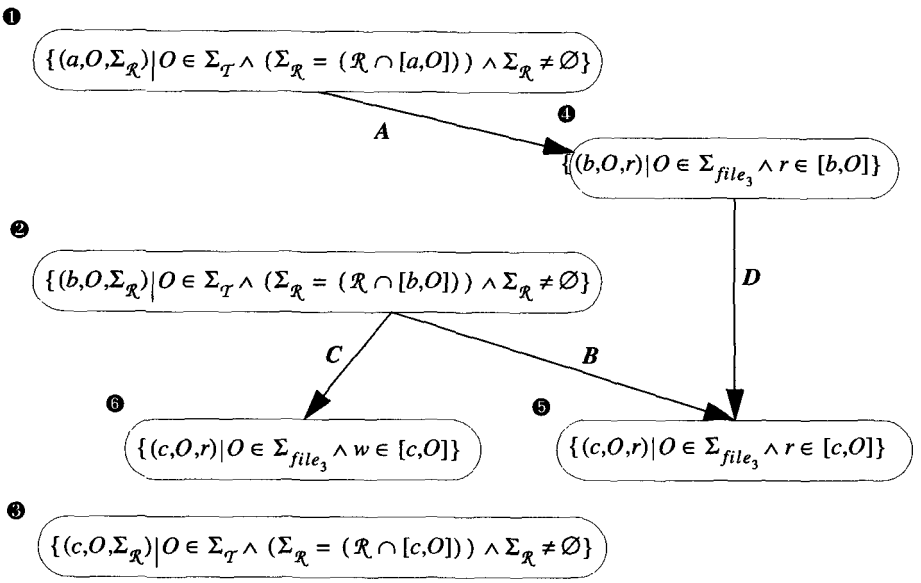

Fig 1. Example of a Privilege Graph

In the first step, we create the nodes ❶, ❷, and ❸. They refer to the protection state of Table 3. In the second step, we can apply, for instance, the following sequence of command applications that leads to a maximal state: $R''_1(a,b,f,❶,❹)$ - $R''_{2read}(b,c,g,❷,❺)$ - $R''_{2write}(b,c,g,❷,❻)$, $R''_{2read}(b,c,g,❹,❺)$. This is represented in the graph of Fig. 1 which could have been obtained with any other sequence. The third step is detailed in the next subsection.

4.3 Resolution of the safety problem

To solve the question Q1 of our safety problem, we run the following steps:

1) Check in the matrix if *a* has *read* access to *i*; if the answer is no, go to the next step.
2) Identify in the matrix the subjects who have *read* access to *i*; the only subject with that right in our example is *c*.
3) Identify in the graph every node which contains the triple (*c*,*i*,*r*); both nodes ❸ and ❺ have this property in our example.
4) If a path can be found between the node representing the set of privileges of *a* (❶) and one of the nodes identified in step 3, then, we know that *a* can read *i*; in this case the existence of the path between ❶ and ❺ (arcs *A* and *D*) implies a positive answer to the question.

For the question Q2, the same scheme leads us to deduce a negative answer because no path exists, neither between the nodes ❶ and ❻, nor between ❶ and ❸.

4.4 Discussion

We have already mentioned that this method was only efficient for specific authorization schemes. The conditions under which our method is worth being used are recalled hereafter:

1) Sets of granted privileges should not contain access rights on objects checked in the conditional parts of the commands. This ensures that the conditional part can be evaluated by the sole inspection of the matrix, without looking at the graph (of course, the node identification still requires inspection of the graph). In our example, for instance, we grant privileges on objects of type $file_3$ but the conditions always checks files of type $file_1$ or $file_2$. If this was not the case, the expression of the condition, though possible, would be cumbersome and require to check for the existence of some well-defined node in a specific path. It is clear that the complexity of this process would impede the usefulness of this solution.

2) Sets of granted privileges should have small mutual intersections. Clearly, the best situation consists of distinct sets of privileges. This will minimize the number of edges created.

3) If the safety problem is made of a conjunction of n questions, such as $(r \in [a,i]) \wedge (w \in [a,i])$, then its solution is found as the conjunction of the n answers to each individual question. In this case, $(true \wedge false)$ returns *false*.

The following example highlights the algorithmic complexity of the method when these two requirements are satisfied. Consider the initial access matrix represented in Table 8 and the command R_e.

	$x : file_1$	$y : file_1$	$z : file_1$	$u : file_2$	$v : file_2$	$w : file_2$
a: user	*e,o,r,w*			*e,o,r,w*		
b: user		*e,o,r,w*			*e,o,r,w*	
c: user			*e,o,r,w*	*o,r,w*		*e,o,r,w*

Table 6. A simple Typed Access Matrix

command $R_e(U_1$: *user*, U_2: *user*, F_1: $file_1$, F_2: $file_2)$
if $o \in [U_1,F_1] \wedge e \in [U_1,F_2]$
then enter e **into** $[U_2, F_2]$

Suppose also the existence of a rule R_r (resp. R_w) equivalent to R_e where the right e has been replaced by the right r (resp. w).

In terms of privileges sets, we can rewrite[11] this authorization scheme as:

11. Actually, one such rule must be written for each subset of the set $\{e,r,w\}$. They determine the definition of N_1 but are equivalent for the primitive operations involved.

command $R'_e(U_1$: *user*, U_2: *user*, F_1: *file*$_1$, $\mathcal{N}_1$: *node*, $\mathcal{N}_2$: *node*)
if $o \in [U_1,F_1] \wedge \{(U_2,O,\Sigma_{\mathcal{R}}) | O \in \Sigma_{file_2} \wedge (\Sigma_{\mathcal{R}} = \{e\} \cap [U_2,O]) \wedge \Sigma_{\mathcal{R}} \neq \emptyset\} \subseteq \mathcal{N}_1$
then
make_node $\mathcal{N}_2 = \{(U_1,O,\Sigma_{\mathcal{R}}) | O \in \Sigma_{file_2} \wedge (\Sigma_{\mathcal{R}} = (\{e,r,w\} \cap [U_1,O])) \wedge \Sigma_{\mathcal{R}} \neq \emptyset\}$
make_edge from $\mathcal{N}_1$ **to** $\mathcal{N}_2$
end if

In this case, the two above requirements are satisfied: i) objects involved in the condition are distinct from those granted and ii) the sets of privileges granted are distinct.

Hence, suppose that we have n users. Each user has the o right on at least one object of type *file*$_1$. Furthermore, each user has r rights that can be granted on m objects of type *file*$_2$. Then, in this case, TAM requires $n \times r \times m \times (n-1)$ rule applications to reach a maximal state. With the privilege graph, it requires $n \times r \times (n-1+n-1)$. Thus, when the requirements are satisfied, the complexity is reduced by a factor approximately equal to $m/2$. Furthermore, in general, we will have $m>>n$. Of course, if the requirements are not satisfied, the best trade-off must be found between pure TAM and pure privilege graph, based on complexity evaluation. This evaluation can only be made with full knowledge of the access matrix and of the rules.

In order to show the usefulness of the method, we show hereafter real-world examples where the requirements are satisfied, and where, therefore, the use of privilege graph must be preferred to TAM.

5 Real World Examples

In this section, we wish to stress that the authorization scheme presented in Section 3 is not artificial; it is representative of privilege transfer features that can be found in most real life systems. To show this, we consider three examples based on Unix™: a `.xinitrc` file, a `.rhosts` file and `setuid` files.

5.1 The `.xinitrc` File

When running, the X Window System initializer looks for a specific file in the user's home directory, called `.xinitrc`, to run as a shell script to start up client programs. Daily practice shows that novice users can encounter some difficulty to configure correctly this file. If a novice user trusts another user, more expert in X than himself, he may prefer use the expert's configuration file rather than bother to understand all the commands and options. To do so, an easy solution is to establish a symbolic link between his own `.xinitrc` file and the expert's file[12]. Then, if the so-called expert enhances his set-up file, the novice will enjoy the result as well.

From a security point of view, this can also be a good solution. Indeed, if the novice chooses inappropriate options or commands, this file will turn out as a trapdoor, letting his data unprotected. Using the expert's file — who should be aware of the vulnerabilities — his data security is enhanced. Of course, he is at the mercy of this

12. `ln -s ~expert/.xinitrc ~novice/.xinitrc`

expert who can introduce a Trojan horse in his configuration file, and then acquire most of the novice's privileges[13] . This is exactly what the first rule of our authorization scheme wanted to characterize in its conditional part: the expert owns the `.xinitrc` file executed by the novice.

5.2 The `.rhosts` File

To log in a Unix system, a password is required. However, there is a mechanism in Unix that allows remote trusted users to access local system without supplying a password. This functionality is implemented by the `.rhosts` file which enables to bypass the standard password-based authentication: if in John's home directory there is a `.rhosts` file which contains Fred's username, then Fred, when logged in another machine, can establish a remote connection to John's machine and be identified as John on this machine, without typing John's password. Once again, this allows John to grant Fred almost all his privileges. This feature is frequently used, for instance if John wishes Fred to take care of any urgent work during his vacations, without giving him his own password. Another advantage of this feature is to enable remote login without transmitting a password on the network where it would be vulnerable to wire tapping.

If such a file exists, any user who has *write* access to John's `.rhosts` can get this set of privileges. This is an example of the second rule of our authorization scheme: a user who can write in John's `.rhosts` can read and write the same files as John[14] .

5.3 Setuid Files

In Unix, every process holds the same privileges as the user for whom the process is run. However, it is possible to let a process get the privileges of the owner of the program rather than the privileges of the user initiating the process. This is particularly useful when an operation needs more privileges than held by the user. An example of this is the program `/bin/passwd` that changes user passwords: every user must be able to change his own password but this operation requires to write in a protected file, usually the `/etc/passwd` file, to which no user has write access except the superuser; to do so, `/bin/passwd` uses the setuid facility to run with superuser privileges on behalf of less privileged users. This functionality has many other applications, all of them being examples of grants of sets of privileges by the owner of the program to the user of the program. As long as these setuid programs are correct and no low privileged user can create or modify such programs, the security is satisfactory. Indeed, this feature strengthens security since, without this feature, users should be granted more privileges constantly. But if a setuid-program owner trusts another user and gives him write access to his program, he is at the mercy of this user. Such behaviour is another example of the second rule of the authorization scheme given in Section 3.1.

---

13. Actually, the expert cannot acquire all the novice's privileges since, for instance, without knowing the novice's password he will not be able to change it. Other specific privileges could be denied to the expert due to the fact that, for instance, he is not physically located at the same place than the novice, etc.
14. It is clearly a very bad idea to grant another user write access to your `.rhosts` file but this is another problem! Preventive and/or corrective actions are beyond the scope of this paper.

6 Potential Applications of the Privilege Graph

6.1 Practical Solutions to the Safety Problem

It has already been explained at length how the *privilege graph* formalism could be used to analyse in an efficient way the safety problem. But to know whether an unsafe state is reachable is not enough: we wish to know what can be done to prevent to reach this state, i.e., which modification of the protection state can solve the problem. The graph enables to identify which paths are conflicting with the security constraints. In our experiments, this feature has proved to be helpful to solve conflicts.

6.2 Quantitative Evaluation of Security

The safety problem accepts only a boolean answer: either an unsafe state is reachable or not. There is no information on how easily or how fast the unsafe state can be reached. Yet, in most practical systems, attacks and intrusions are more or less easy and fast according to the configuration of the system. For instance, it can be more or less difficult to guess a user's password. In the safety problem, either you consider that passwords are guessable and then the system is unsafe, or that no password can be guessed and then ignore that indeed some of them can be guessed by chance or by brute force[15].

With the privilege graph, it can be envisaged to assign a weight to each edge corresponding to the likelihood associated to this privilege transfer; for instance, if an edge represents the possibility to guess user A's password, the corresponding weight can be lower if A's password is in a dictionary than if it had been carefully chosen. Moreover, it is possible to consider that successful attacks are represented in the graph as paths between potential attackers' privileges (e.g., non-users, or ftp users) and potential targets' privileges (e.g., superuser). The system security can then be assessed not only by the existence or absence of such a path, but also by the length of this path and the weights on the traversed edges. This approach could lead to a quantitative evaluation of the operational security but, firstly, open theoretical problems have to be solved, as discussed in [5].

6.3 Intrusion Detection

Intrusion detection is another potential application of the privilege graph: if it is possible to correlate the user's behaviour observed by an intrusion detection system with a progress in the privilege graph towards a potential target, alarms of different levels can be triggered according to the likelihood to reach the target. This approach is similar to the pattern-oriented model proposed by [16]. It is probably possible to integrate the privilege graph analysis in sophisticated intrusion detection tools such as NIDES [8], e. g., in the resolver module, to help in detecting malicious activities carried on by a hacker impersonating other users by using their privileges. The graph could be used to correlate various suspicious activities that, if considered separately, would not

15. Of course, intermediate considerations could be that low privileged users' passwords are guessable and superuser's password is not, but this does not change the problem.

bring enough evidence to detect an intruder. Also, their correlation could highlight on the graph that some possible attack is progressing along a path leading to a target.

7 Conclusions

In this paper, a graphical extension to the TAM model has been proposed to represent authorization schemes based on privilege transfers. This formalism provides an efficient technique to analyse the safety problem and can be useful to identify which privilege transfers can lead to an unsafe state. Further extensions are suggested towards quantitative evaluation of operational security and intrusion detection.

It is our claim that this formalism is flexible enough to represent real world systems such as Unix systems. Indeed, it is possible to build a privilege graph by means of an automatic tool analysing the permissions in the Unix file system. In this case, nodes are privileges held by users or groups and edges are elementary privilege transfers corresponding to Unix operations on permissions. A prototype of such a tool has been implemented and experimented successfully [5].

8 Acknowledgments

Thanks are due to Mohamed Kaâniche and Jean-Claude Laprie for useful discussions that have led to the writing of this paper. The authors also want to thank the anonymous referees for their valuable comments. Finally, the authors acknowledge several insightful comments from Catherine Meadows and Gerard Eizenberg which enabled significant improvements of this paper.

This work has been partially supported by ESPRIT Basic Research Action Project n°6362: Predictable Dependable Computing Systems (*PDCS2*) and by the ESPRIT Basic Research Network of Excellence in Distributed Computing Systems Architectures - (*CaberNet*).

References

1. Amman, P. E. and Sandhu, R. S. "Extending the Creation Operation in the Schematic Protection Model," *Proc. Sixth Annual Computer Security Applications Conference*, 1990, pp. 340-348.
2. Amman, P. E. and Sandhu, R. S. "Implementing Transaction Control Expressions by Checking for Absence of Access Rights," *Proc. Eighth Annual Computer Security Applications Conference*, San Antonio (Texas, USA), December 1992, pp. 131-140.
3. Bishop, M. and Snyder, L. "The Transfer of Information and Authority in a Protection System," *Proc. of the Seventh Symposium on Operating Systems Principles*, Pacific Grove, California (USA), December 10-12, 1979, SIGOPS (ACM), pp. 45-54.
4. Biskup, J. "Some Variants of the Take-Grant Protection System", *Information Processing Letters*, 19, 1984, pp. 151-156.

5. Dacier, M., Deswarte, Y. and Kaâniche, M. "A Framework for Security Assessment of Insecure Systems," Predictably Dependable Computing Systems (PDCS-2), *First Year Report,* ESPRIT Project 6362, September 1993, pp. 561-578.

6. Dacier, M. "A Petri Net Representation of the Take-Grant Model," *Proc. of the 6th. Computer Security Foundations Workshop*, Franconia (USA), June 15-17, 1993, pp. 99-108.

7. Harrison, M. A., Ruzzo, W. L. and Ullman, J. D. "Protection in Operating Systems," *Communications of the ACM*, 19(8), August 1976, pp. 461-470.

8. Jagannathan, R., Lunt, T., Gilham, F., Tamaru, A., Jalali, C., Neumann, P., Anderson, D., Garvey, T. and Lowrance, J., Requirements Specification: Next-Generation Intrusion Detection Expert System (NIDES), SRI Project 3131 - Requirement Specifications (A001, A002, A003, A004, A006), September 3, 1992.

9. Lampson, B. W. "Protection", *ACM Operating Systems Review*, 8(1), 1974, pp. 18-24.

10. Landwehr, C. E. "Formal Models for Computer Security", *ACM Computing Surveys*, 13(3), 1981, pp. 247-278.

11. Lypton, R. J. and Snyder, L. "A Linear Time Algorithm for Deciding Subject Security," *Communications of the ACM*, ACM, 24(3), July 1977, pp. 455-464.

12. Sandhu, R.S. "The Schematic Protection Model: Its Definition and Analysis of Acyclic Attenuation Schemes," *Journal of the ACM*, No. 2, 1988, pp. 404-432.

13. Sandhu, R. S. and Suri, G. S. "Non-monotonic Transformation of Access Rights," *Proc. 1992 IEEE Symposium on Research in Security and Privacy*, May 4-6, 1992, pp. 148-161.

14. Sandhu, R. S. "The Typed Access Matrix Model," *Proc. 1992 IEEE Symposium on Research in Security and Privacy*, May 4-6, 1992, pp. 122-136.

15. Sandhu, R. S. and Ganta, S. "On Testing for Absence of Rights in Access Control Models," *Proc. of the Computer Security Foundations Workshop VI*, IEEE Computer Society Press, Franconia (NH,USA), June 15-17, 1993, pp. 109-118.

16. Shieh, S. W. and Gligor, V. D. "A Pattern-Oriented Intrusion-Detection Model and Its Application", *Proc. 1991 IEEE Symposium on Research in Security and Privacy*, Oakland (USA), May 20-22, 1991, pp. 327-342.

17. Snyder, L. "On the Synthesis and Analysis of Protection Systems," *Proc. of the Sixth Symposium on Operating Systems Principles*, Purdue University (USA), November 16-18, 1977, SIGOPS (ACM), 11(5), pp 141-150.

18. Snyder, L. "Formal Models of Capability-Based Protection Systems", *IEEE Transactions on Computers*, C-30(3), 1981, pp.172-181.

19. Snyder, L. "Theft and Conspiracy in the Take-Grant Protection Model", *Journal of Computer and System Sciences*, 23, 1981, pp. 333-347.

20. von Solms, S. H. and de Villiers, D. P. "Protection Graph Rewriting Grammars and the Take-Grant Security Model", *Quæstiones Informaticæ*, 6(1), 1988, pp.15-18.