Creating Objects in the Flexible Authorization Framework*

Nicola Zannone[1,2], Sushil Jajodia[2], and Duminda Wijesekera[2]

[1] Dep. of Information and Communication Technology
University of Trento
zannone@dit.unitn.it
[2] Center for Secure Information Systems
George Mason University
{jajodia,dwijesek}@gmu.edu

Abstract. Access control is a crucial concern to build secure IT systems and, more specifically, to protect the confidentiality of information. However, access control is necessary, but not sufficient. Actually, IT systems can manipulate data to provide services to users. The results of a data processing may disclose information concerning the objects used in the data processing itself. Therefore, the control of information flow results fundamental to guarantee data protection. In the last years many information flow control models have been proposed. However, these frameworks mainly focus on the detection and prevention of improper information leaks and do not provide support for the dynamical creation of new objects.
In this paper we extend our previous work to automatically support the dynamical creation of objects by verifying the conditions under which objects can be created and automatically associating an access control policy to them. Moreover, our proposal includes mechanisms tailored to control the usage of information once it has been accessed.

1 Introduction

Access control is one of the main challenges in IT systems and has received significant attention in the last years. These efforts have matched with the development of many frameworks dealing with access control issues [1–6]. However, many of these proposals focus on the restriction on the release of information but not its propagation [7].

Actually, IT systems are developed not only to merely store data, but also to provide a number of functionalities designed to process data. Thereby, they may release information as part of their functionalities [8]. Yet, a malicious user can embed in some

* This material is based upon work supported by the National Science Foundation under grants IIS-0242237 and IIS-0430402. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. This work was partly supported by the projects RBNE0195K5 FIRB-ASTRO, 016004 IST-FP6-FET-IP-SENSORIA, 27587 IST-FP6-IP-SERENITY, 2003-S116-00018 PAT-MOSTRO.

application provided by the IT system, a Trojan horse that, once the application is executed, copies sensitive information in a file accessible by the malicious user [9]. In this setting, information flow control plays a key role in ensuring that derived objects do not disclose sensitive information to unauthorized users.

This issue has spurred the research and development of frameworks that improve authorization frameworks with some form of flow control. Sammarati et al. [10] proposed to detect unauthorized information flow by checking if the set of authorizations associated with a derived object are a subset of the intersection of the sets of authorizations associated with the objects used to derive it. Similar approaches [11, 12] have associated with each object an access control list that is propagated together with the information in the object. However, in these approaches the creation of objects is implicit. Essentially, they attempt to identify leaking information, but do not deal with the creation of new objects.

Moreover, this approach is to rigid to implement real access control policies. Actually, it is not flexible enough to support information declassification [8]. For instance, the US Privacy Act allows an agency to disclose information to those officers and employees of the agency who need it to perform their duties without the consent of the data subject. Furthermore, the Act does not impose any constraint to data that do not disclose personal identifying information.

In this paper, we extend our previous work [13] in order to automatically enforce access control policies on objects dynamically created in Flexible Authorization Framework (FAF) [14]. This requires to deal with some issues:

- deciding if an object can be created;
- associating authorizations with the new object;
- verifying if the derived object does not disclose sensitive information to unauthorized users.

The first issue is addressed by introducing conditions under which a data processing can be performed and enforcing the system to verify them before creating new objects. To cope with the second issue, we allow system administrators to define the policies governing access to derived objects, based on the authorizations associated with the objects used to derive them.

However, this is not sufficient to fully guarantee data protection. Actually, if a user is authorized to execute an application in which a Trojan horse is embedded, such a malicious application is considered as legitimate by the authorization framework. To this end, we propose an approach based on [10–12] to block non safe information flow. However, it is up to system administrators to decide whether or not an information flow is safe. Thereby, we only provide support for detecting flows of information that may be harmful to data subjects.

Other issues come up when the proposed approach is integrated in FAF. Actually, its current architecture does not support the dynamical creation of objects. To this intent, we need to improve it together with its underlying logic-based framework.

The remainder of the paper is structured as follows. Next (§2) we provide a brief overview of FAF. Then, we illustrate our approach for dealing with the dynamical creation of objects (§3) and for automatically deriving their access control policy (§4).

Next, we propose a mechanism to control information flow and show how such a mechanism copes with the Trojan horse problem (§5). Finally, we discuss related work (§6) and conclude the paper (§7).

2 Flexible Authorization Framework

Flexible Authorization Framework (FAF) [14] is a logic-based framework developed to manage access to data by users. It consists of four stages that are applied in sequence. The first stage takes in input the extensional description of the system, as subject and object hierarchies and a set of authorizations, and propagates authorizations through the organizational structure of the system. However, in this stage it is possible to derive contradictory authorizations, that is, a subject could be authorized and denied to execute an action on an object at the same time. The second stage aims to resolve this problem by applying conflict resolution policies. Once authorizations are propagated and conflicts resolved, there is the possibility that some access is neither authorized nor denied. In the third stage, decision policies are used to ensure the completeness of authorizations. In the last stage, specific domain properties are verified using integrity constraints, and all authorizations that violate them are removed.

FAF provides a logic-based language, called authorization specification language (**ASL**), tailored for encoding security needs. Before defining the language, we introduce the logic programming terminology needed to understand the framework. Let p be a predicate with arity n, and $t_1, \ldots, t_n$ be its appropriate terms. $p(t_1, \ldots, t_n)$ is called *atom*. Then, the term *literal* denotes an atom or its negation. **ASL** syntax includes the following predicates:

- A ternary predicate cando. Literal $\mathsf{cando}(o, s, a)$ is used to represent authorizations directly defined by the system administrator where o is an object, s is a subject, and a is a signed action terms. Depending on the sign, authorizations are permissions or prohibitions.
- A ternary predicate dercando that has the same arguments of predicate cando and is used to represent authorizations derived through propagation policies.
- A ternary predicate do that has the same arguments of predicate cando and represents effective permissions derived by applying conflicts resolution and decision policies.
- A 5-ary predicate done that is used to describe the actions executed by users. Intuitively, $\mathsf{done}(o, s, r, a, t)$ holds if subject s playing role r has executed action a on object o at time t.
- A propositional symbol error. Its occurrence in the model corresponds to a violation of some integrity constraints.
- A set of hie-predicates. In particular, the ternary predicate $\mathsf{in}(x, y, \mathsf{H})$ is used to denote that $x \leq y$ in hierarchy H.
- A set of rel-predicates. They are specific domain predicates.

Based on the architecture previously presented, every authorization specification **AS** is a locally stratified program where stratification is implemented by assigning levels to

Table 1. Strata in FAF specification

Stratum	Predicate	Rules defining predicate
$\mathbf{AS}_0$	hie-predicates	base relations.
	rel-predicates	base relations.
	done	base relation.
$\mathbf{AS}_1$	cando	the body may contain done, hie- and rel-literals.
$\mathbf{AS}_2$	dercando	the body may contain cando, dercando, done, hie- and rel-literals. Occurrences of dercando literals must be positive.
$\mathbf{AS}_3$	do	the head must be of the form $\mathsf{do}(\_, \_, +a)$ and the body may contain cando, dercando, done, hie- and rel-literals.
$\mathbf{AS}_4$	do	the head must be of the form $\mathsf{do}(o, s, -a)$ and the body contains the literal $\neg\mathsf{do}(o, s, +a)$.
$\mathbf{AS}_5$	error	the body may contain cando, dercando, do, done, hie- and rel-literals.

predicates (Table 1 [14]). For any specification **AS**, $\mathbf{AS}_i$ denotes the rules belonging to the i-th level.

For optimizing the access control process, Jajodia et al. [14] proposed a materialized view architecture, where instances of predicates corresponding to views are maintained. Because predicates belong to strata, the materialization structure results (locally) stratified. This guarantees that the specification has a unique stable model and well-founded semantics [15, 16]. Following [14], we use the notation $\mathcal{M}(\mathbf{AS})$ to refer to the unique stable model of specification **AS**.

3 Creating objects

When a user requires to perform a data processing, the IT system should verify whether or not such a user has all necessary authorizations. In the remainder of this section, we address this issue.

Let O be the name space of all possible objects that may occur in the specification. We assume that they are organized into a hierarchical structure. This means that all possible objects are fully classified with respect to their type. Further, we assume that objects do not exist until they are created. This means that objects (together with their classification) may be not in the scope of the specification, although they are defined in O. Essentially, we assume that a possible object is considered only if some event demands its existence, that is, it is created.

Following [17], we introduce predicate exists, where $\mathsf{exists}(o)$ holds if object o exists, that is, it is already created. We define the *state of the system* as the set of existing objects and their relationships. To deal with the creation of objects, Liskov et al. [18] introduced two kinds of functions: *constructors* and *creators*. Constructors of a certain type return new objects of the corresponding type and creators initialize them. Essentially, constructors add object identifiers (i.e., names) to the state of the system and creators assign a value to such names. However, this approach distinguishes the identifier of an object from the values the object can assume. We merge this pair of functions into a single function, called *initiator*. Essentially, when an object is created,

it exists together with its value. This allows us to be consistent with the semantics of FAF. Further, we assume that objects are never destroyed. From these assumptions, we can deduce that the set of objects belonging to a state of the system is a subset of the set of objects belonging to the next state.

IT systems process data as part of their functionalities by providing automatic procedures used to organize and manipulate data. As done in [13], we represent data processing through initiators and make explicit the objects used by data processing and the users who performs them. Thus, we introduce an initiator for each procedure supported by the IT system. For instance, we write

$$f(s, o_1, \ldots, o_m) = o$$

to indicate that object o is the result of data processing f when this is performed by subject s and objects $o_1, \ldots, o_m$ are passed as input.[3] We assume that when an object is created (i.e., it enters in the scope of the specification), also its classification belongs to the specification. Notice that initiators do not belong to the specification language. We use them only to emphasize the objects used in the procedure and the subject that executes it.

Subjects may need to access exiting objects in order to create new objects. Moreover, only users that play a certain role or belong to a certain group may be entitled to perform a certain data processing. This means that an authorization framework should verify whether the subject has enough rights to access all objects needed to create the new one and whether he can execute the procedure.

Our idea is to enforce the system to verify the capabilities of the subject before an object is created. Based on this intuition, we redefine initiator f as

$$f(s, o_1, \ldots, o_m) = \begin{cases} o & \text{if } \mathcal{C} \text{ is true} \\ \perp & \text{otherwise} \end{cases}$$

where $\mathcal{C}$ represents the condition that must be satisfied and $\perp$ means that object o cannot be created since s does not have sufficient rights to execute the procedure.

Initiators are implemented in our framework through rules, called *creating rules*. These rules enforce the system to verify the conditions under which a user can create the object.

Definition 1. *Let f be an initiator, s be the subject executing f, $o_1, \ldots, o_m$ be the objects required by f, and $o = f(s, o_1, \ldots, o_m)$ be the derived object. A* creating rule *has the form*

$$\mathsf{exists}(o) \leftarrow L_1 \ \& \ \ldots \ \& \ L_n \ \& \ \mathsf{exists}(o_1) \ \& \ \ldots \ \& \ \mathsf{exists}(o_m).$$

where $L_1, \ldots, L_n$ are cando, dercando, do, done, hie-, *or* rel-*literals.* cando, dercando, do *literals may refer only to $o_1, \ldots, o_m$.*

Essentially, the conjunction of literals $L_1, \ldots, L_n$ represents the condition that a subject must satisfy in order to create object o. Last part of the body of the rule ensures that all objects necessary to create the new object already exist.

[3] Notice that initiators are not total functions since if one combines personal data of different users for creating an account, such account is not a valid object.

Example 1. A bank needs customer personal information, namely name, shipping address, and phone number, for creating an account. The bank IT system provides the procedure $openA$ for creating new accounts. Suppose a customer discloses his name (n), shipping address (sa), and phone number (p) to the bank. A bank employee s will be able to create $account$ ($= openA(s, n, sa, p)$) only if it is authorized to read customer data and he works in the Customer Services Division (CSD). In symbol,

$$\mathsf{exists}(account) \leftarrow \mathsf{do}(n, s, +\mathsf{read}) \;\&\; \mathsf{do}(sa, s, +\mathsf{read}) \;\&\; \mathsf{do}(p, s, +\mathsf{read}) \;\&\; \mathsf{in}(s, \mathit{CSD\text{-}employee}, \mathsf{ASH}) \;\&\; \mathsf{exists}(n) \;\&\; \mathsf{exists}(sa) \;\&\; \mathsf{exists}(p).$$

The outcome of a data processing may then be used to derive further objects. We represent the process to create an object as a tree, called *creation tree*, where the root is the "final" object and the leaves are *primitive objects* (i.e., objects that are directly stored in the system by users). In order to rebuild the creation tree, the system should keep trace of the process used to create the object. To this end, we introduce the binary predicate derivedFrom where $\mathsf{derivedFrom}(o_1, o_2)$ is used to indicate that object o_2 is used to derive object o_1. As for classification literals, derivedFrom literals referring an object are added to the model only when the object is created.

Example 2. Back to Example 1, the bank IT system stores the following set of literals:

$$\{\mathsf{derivedFrom}(account, n), \mathsf{derivedFrom}(account, sa), \mathsf{derivedFrom}(account, p)\}$$

4 Associating authorizations with new objects

Once an object has been created, authorizations should be associated with it. Since the object is not independent from the objects used to derive it, the policy associated with it should take into account the authorizations associated with them. Some proposals [11, 12] associate with each object an access control list (ACL) that is propagated together with the information in the object. Essentially, the ACL associated with the new object is given by the intersection of all ACLs associated with the objects used to create it. However, when a system administrator specifies an access control policy for derived objects, he should consider that not all data processing disclose individually identifiable information [8]. For example, the sum of all account balances at a bank branch does not disclose data that allows to recover information associating a user with his own account balance.

We propose a flexible framework in order to allow system administrators to determine how authorizations are propagated to new objects. The idea is that authorizations associated with the objects used to derive the new one can be used to determine the authorizations associated with it. However, this approach cannot be directly implemented in FAF since the specification results no more stratified [13]. Next, we propose how FAF can be modified in order to support access control on derived objects maintaining the locally stratified structure.

4.1 Redefining Rules

To maintain the locally stratified structure, we need to redefine *creating rules*, *authorization rules* [14], *derivation rules* [14], and *positive decision rules* [14] by enforcing

some syntactic constraints to predicates occurring in the body of rules. Before doing this, we have also to redefine the predicates defined in FAF. Essentially, we introduce a new parameter representing the depth of the creation tree of the object into predicates exists, cando, dercando and do. Further, we enforce rules to be applied only to existing objects.

Definition 2. *Let f be an initiator, s be the subject executing f, $o_1, \ldots, o_m$ be the objects required by data processing f, and $o = f(s, o_1, \ldots, o_m)$ be the derived object. A* creating rule *is a rule of the form*

$$\mathsf{exists}(i, o) \leftarrow L_1 \ \& \ \ldots \ \& \ L_n \ \& \ \mathsf{exists}(j_1, o_1) \ \& \ \ldots \ \& \ \mathsf{exists}(j_m, o_m).$$

where o is an object, i represents the current iteration, and $L_1, \ldots, L_n$ are cando*,* dercando*,* do*,* done*,* hie-*, or* rel-*literals.* cando*,* dercando*,* do *literals refer only to $o_1, \ldots, o_m$ and $0 \leq j_1, \ldots, j_m < i$.*

Once an object has been introduced in the scope of the specification, its access control policy is inferred by the system through authorization rules.

Definition 3. *An* authorization rule *is a rule of the form*

$$\mathsf{cando}(i, o, s, a) \leftarrow L_1 \ \& \ \ldots \ \& \ L_n \ \& \ \mathsf{exists}(i, o).$$

where o, s and a are respectively an object, a subject and a signed action, i represents the current iteration, and $L_1, \ldots, L_n$ are cando*,* dercando*,* do*,* done*,* derivedFrom*, hie-, or rel-literals. Every* cando*,* dercando *and* do *literal must be inferred at an iteration j such that $0 \leq j < i$.*

Example 3. A customer may prefer to not receive advertising on new services offered by the bank. Therefore, he specifies that his information (i.e., name (n), shipping address (sa), and phone number (p)) cannot be accessed by the Marketing Division (MD).

$$\begin{aligned}
\mathsf{cando}(0, n, x_s, -\mathsf{read}) &\leftarrow \mathsf{in}(x_s, \mathit{MD\text{-}employee}, \mathsf{ASH}) \ \& \ \mathsf{exists}(i, n).\\
\mathsf{cando}(0, sa, x_s, -\mathsf{read}) &\leftarrow \mathsf{in}(x_s, \mathit{MD\text{-}employee}, \mathsf{ASH}) \ \& \ \mathsf{exists}(i, sa).\\
\mathsf{cando}(0, p, x_s, -\mathsf{read}) &\leftarrow \mathsf{in}(x_s, \mathit{MD\text{-}employee}, \mathsf{ASH}) \ \& \ \mathsf{exists}(i, p).
\end{aligned}$$

It is possible that no authorization is explicitly defined for the user with respect to a request access. Thereby, the framework allows system administrators to specify policies to propagate authorizations through the organizational structure of the system.

Definition 4. *A* derivation rule *is a rule of the form*

$$\mathsf{dercando}(i, o, s, a) \leftarrow L_1 \ \& \ \ldots \ \& \ L_n \ \& \ \mathsf{exists}(i, o).$$

where o, s and a are respectively an object, a subject and a signed action, i represents the current iteration, and $L_1, \ldots, L_n$ are cando*,* dercando*,* do*,* done*,* derivedFrom*, hie-, or rel-literals. Every* cando*,* over *and positive* dercando *literal must be inferred at an iteration j such that $0 \leq j \leq i$, and every* do *and negative* dercando *literal at an iteration k such that $0 \leq k < i$.*

Example 4. Employees of the Customer Services Division are authorized to manage bank accounts. However, the actions that they can perform depend on the actions that they are authorized to perform on the customer information used to create such an account.

$$\begin{aligned}\mathsf{dercando}(i, x_{o_1}, x_s, x_a) \leftarrow\ & \mathsf{in}(x_{o_1}, \mathit{Account}, \mathsf{AOH})\ \&\ \mathsf{derivedFrom}(x_{o_1}, x_{o_2})\ \&\\ & \mathsf{do}(j, x_{o_2}, x_s, x_a)\ \&\ \mathsf{in}(x_s, \mathit{CSD\text{-}employee}, \mathsf{ASH})\ \&\\ & \mathsf{exists}(i, x_{o_1})\ \&\ j < i.\end{aligned}$$

Using derivation rules, a system administrator can specify very flexible policies for propagating authorizations. However, such a propagation may lead conflicting authorizations. Decision rules are introduced to cope with this issue.

Definition 5. *A* positive decision rule *is a rule of the form*

$$\mathsf{do}(i, o, s, +a) \leftarrow L_1\ \&\ \ldots\ \&\ L_n\ \&\ \mathsf{exists}(i, o).$$

where o, s and a are respectively an object, a subject and an action, i represents the current iteration, and $L_1, \ldots, L_n$ are cando, dercando, do, done, derivedFrom, *hie-, or rel-literals. Every* cando *and* dercando *literal must be inferred at an iteration j such that $0 \leq j \leq i$, and every* do *literal at an iteration k such that $0 \leq k < i$.*

Example 5. Information on accounts is also required by employees of other divisions of the bank in order to perform their duties. Thus, bank employees are entitled to access an account only if they are not explicitly denied to access the account and the information of its owner.

$$\begin{aligned}\mathsf{do}(i, x_{o_1}, x_s, +\mathsf{read}) \leftarrow\ & \neg\ \mathsf{dercando}(i, x_{o_1}, x_s, -\mathsf{read})\ \&\ \mathsf{in}(x_{o_1}, \mathit{Account}, \mathsf{AOH})\ \&\\ & \mathsf{derivedFrom}(x_{o_1}, x_{o_2})\ \&\ \neg\ \mathsf{do}(j, x_{o_2}, x_s, -\mathsf{read})\ \&\\ & \mathsf{in}(x_s, \mathit{employee}, \mathsf{ASH})\ \&\ \mathsf{exists}(i, x_{o_1})\ \&\ j < i.\end{aligned}$$

4.2 Materialized Views

The architecture proposed in [14] works properly when authorizations refer to primitive objects, but it is not able to completely enforce access control policies when objects are dynamically created. The main problem is when objects are "introduced" in the state of the system. If derived objects are introduced before applying propagation policies, they could not be created since required authorizations might be not yet computed. Otherwise, if they are introduced after applying propagation policies, authorizations on derived objects are not propagated.

Authorizations on new objects could depend on the authorizations associated with those objects used to create them. To maintain the flexibility provided by FAF, we permit any authorization predicate to occur in the body of rules. However, this affects the process for enforcing access control policies. In particular, the locally stratified structure of specifications is not preserved. Next, we present the process to enforce access control policies when objects are dynamically created.

The idea is to iterate the access control process proposed in [14] for $n + 1$ times where n is the greatest depth of creation trees. At each step i, we compute the stable

model of $\mathbf{AS}^i \cup \mathcal{M}(\mathbf{AS}^{i-1})$, where $\mathbf{AS}^i$ is the set of authorization specifications applied at the i-th iteration and $\mathcal{M}(\mathbf{AS}^{i-1})$ is the unique stable model of specification $\mathbf{AS}^{i-1}$. Next, we describe the process for computing this materialization.

The first step corresponds to the "standard" FAF process where only primitive objects are considered. Essentially, creating rules add to the state of the system objects that occur as leaves in some creation tree. Then, authorizations on these objects are propagated, possible conflicts are resolved, and decision policies are applied. If authorizations comply with integrity constraints, the output of the first iteration, $\mathcal{M}(\mathbf{AS}^0)$, is used as input for the second iteration where objects derived by one derivation step are considered. Repeatedly, the process proceeds until all derived objects are considered where the i-th iteration takes in input the output of the previous iteration, $\mathcal{M}(\mathbf{AS}^{i-1})$, and creating rules add to the state of the system objects whose creation tree has depth equal to i.

We now analyze the computation of the unique stable model of an authorization specification $\mathbf{AS}$ during one step of the previous process. This process is, in turn, an iterative process that, at each step i, computes the model of $\mathbf{AS}^i_j \cup \mathcal{M}(\mathbf{AS}^i_{j-1})$, where $\mathcal{M}(\mathbf{AS}^i_{j-1})$ is the unique stable model of stratum $\mathbf{AS}^i_{j-1}$. Next, we describe the different steps of this materialization computation process at the i-th iteration.

Step (0): $\mathbf{AS}^i_0$ represents the lowest stratum. This stratum contains facts derived at the $i-1$-th iteration, $\mathcal{M}(\mathbf{AS}^{i-1})$, and creation rules used to derive objects that are the root of a creation tree having depth equal to i. Creating rules are recursive, but, in agreement with Definition 2, exists literals occurring in the body of such rules must be derived in one of previous iterations so that they belong to $\mathcal{M}(\mathbf{AS}^{i-1})$. Moreover, Definition 2 allows only cando, dercando and do derived in previous iterations to occur in the body of creating rules. This guarantees that such literals belong to $\mathcal{M}(\mathbf{AS}^{i-1})$.

Step (1): $\mathbf{AS}^i_1$ contains facts derived at the previous stratum, $\mathcal{M}(\mathbf{AS}^i_0)$, and authorization rules. Differently from [14], here authorization rules are recursive. However, according to Definition 3, cando literals occurring in the body of such rules must be derived in previous iterations so that they belong to $\mathcal{M}(\mathbf{AS}^{i-1})$. This holds also for dercando and do literals. Moreover, Definition 3 allows only exists literals derived in the previous step to occur in the body of authorizations rules. Therefore, we can conclude that if a cando literal is added to the model, every literal that can occur in the body of the authorization rule belongs to $\mathcal{M}(\mathbf{AS}^i_0)$.

Step (2): $\mathbf{AS}^i_2$ contains facts derived at the previous stratum, $\mathcal{M}(\mathbf{AS}^i_1)$, and derivation rules. As in [14], derivation rules permit a "real" (positive) recursion. In particular, positive dercando literals having iteration parameter equal to i can occur in the body of the rule. It is possible to prove, along the same lines as done in [14], the correctness of the materialized view. Essentially, the body of derivation rules is split into two parts: positive dercando literals having iteration parameter equal to i and the rest. By Definition 4, we can easily verify that the literals belonging to the last set either are in $\mathcal{M}(\mathbf{AS}^{i-1})$ or are derived in one of previous steps. Thereby, they belong to $\mathcal{M}(\mathbf{AS}^i_1)$. On the other side, we refer to [14] for the fixpoint evaluation procedure that proves the correctness of $\mathcal{M}(\mathbf{AS}^i_2)$.

Step (3): $\mathbf{AS}_3^i$ contains facts derived at the previous stratum, $\mathcal{M}(\mathbf{AS}_2^i)$, and positive decision rules. By Definition 5, we can easily verify that literals occurring in the body either are in $\mathcal{M}(\mathbf{AS}^{i-1})$ or are derived in one of previous steps. Thus, do literals are added to the specification only if every literal that can occur in the body of positive decision rules belongs to $\mathcal{M}(\mathbf{AS}_2^i)$.

Step (4) *and* **Step (5)** are analogous to the ones presented in [14].

The above process ensures that the stable model computed during one iteration is a superset of the stable model computed in previous iterations. Further, it guarantees that every literal derived during an iteration refers to objects created in that iteration.

Theorem 1. *Let* $\mathbf{AS}^{i-1}$ *and* $\mathbf{AS}^i$ *be authorization specifications at* $i-1$*-th and* i*-th iterations, respectively. The following statements hold.*

1. $\mathcal{M}(\mathbf{AS}^{i-1}) \subseteq \mathcal{M}(\mathbf{AS}^i)$
2. *Every literal in* $\mathcal{M}(\mathbf{AS}^i) \setminus \mathcal{M}(\mathbf{AS}^{i-1})$ *refers to objects created at the* i*-th iteration.*

In [14], authors have proved the locally stratified structure by assigning a level to each type of predicates. In our setting, this is not sufficient since the level of the head predicate could be strictly lower than that of predicates occurring in its body. However, the locally stratified structure is maintained by distinguishing the iteration in which facts are deduced and limiting the application of rules to existing objects. Essentially, strata are ordered with respect to a lexicographic order: the first component is the iteration, and the second corresponds to the level as defined in [14].

Theorem 2. *Every authorization specification is a locally stratified logic program.*

This result ensures that the specification has a unique stable model [16]. Baral et al. [15] proved that well-founded semantics coincide with stable model for locally stratified logic programs. This guarantees that the stable model of authorization specifications can be computed in quadratic time on data complexity [19].

5 Information Flow Control

Data subjects want that their information is not misused once it has been accessed. However, FAF does not provide any form of control on the usage of information. This lack makes this authorization framework vulnerable to Trojan horses embedded in applications. If a user executing a tampered application has the required authorizations, the Trojan horse will copy sensible information into a file accessible by a malicious and unauthorized user. Therefore, it is necessary to characterize the flow of information within the system.

FAF supports an integrity constraints phase in order to verify the consistency of the system with respect to specific domain properties. Our idea is to use integrity constraints also to verify the presence of leaks in the information flow. Similarly to [10–12, 20], we propose to verify that the set of authorizations associated with a derived object is a subset of the intersection of the authorizations associated with the objects used to create it. Essentially, we want to ensure that a derived object does not disclose more information that the objects used to derive it does.

To this end, we define the 4-ary predicate warning. The intuition is that literal $\mathsf{warning}(o_1, o_2, s, a)$ holds if subject s can perform action a on object o_1, but he cannot perform a on object o_2 where o_2 is used to derive o_1. Notice that warnings are different from errors: they are failure of integrity constraints, like errors, but the system administrator may be perfectly happy with a system that does not satisfy them since information should be disclosed for complying with availability requirements. Thus, if the system reports a warning, the system administrator has to establish whether a leak complies with system requirements or corresponds to a system vulnerability.

Definition 6. *An* information flow constraint *is a rule of the form*

$$\mathsf{warning}(o_1, o_2, s, a) \leftarrow \mathsf{do}(i, o_1, s, a) \& \ \mathsf{derivedFrom}(o_1, o_2) \& \ \neg \mathsf{do}(j, o_2, s, a).$$

where s and a are respectively a subject and an action, i and j are iterations such that $0 \leq j < i$, and object o_2 is used to derive object o_1.

Essentially, the presence of warning literals in the model corresponds to the presence of covert channels [21] in the system. Therefore, we can detect possible illegal information flow by checking the occurrence of warning in the model.

Theorem 3. *If* warning *does not occur in the model, all information flows are safe.*

We remark that it is up to the system administration decides if an "unauthorized" flow is permitted or not. Every time a warning literal occurs in the model, he has to decide if it corresponds to an unauthorized leakage and, in this case, fix it.

Example 6. Suppose a malicious user, Mallory, has tampered the procedure $openA$ provided by the bank IT system for creating new accounts. In this setting, the modified procedure copies customer information in a file (foo) accessible by the malicious user himself along with its legal functionalities. Such information will then sell to bank competitors by Mallory.

Accordingly, the malicious user defines the following rules

$$\begin{aligned}
\mathsf{exists}(i, foo) \leftarrow\ & \mathsf{do}(j, n, s, +\mathsf{read}) \ \& \ \mathsf{do}(j, sa, s, +\mathsf{read}) \ \& \\
& \mathsf{do}(j, p, s, +\mathsf{read}) \ \& \ \mathsf{in}(s, \mathit{CSD\text{-}employee}, \mathsf{ASH}) \ \& \\
& \mathsf{exists}(j, n) \ \& \ \mathsf{exists}(j, sa) \ \& \ \mathsf{exists}(j, p). \\
\mathsf{cando}(i, foo, \mathit{Mallory}, +\mathsf{read}) \leftarrow\ & \mathsf{exists}(i, foo). \\
\mathsf{dercando}(i, foo, \mathit{Mallory}, +\mathsf{read}) \leftarrow\ & \mathsf{cando}(i, foo, \mathit{Mallory}, +\mathsf{read}) \ \& \ \mathsf{exists}(i, foo). \\
\mathsf{do}(i, foo, \mathit{Mallory}, +\mathsf{read}) \leftarrow\ & \mathsf{dercando}(i, foo, \mathit{Mallory}, +\mathsf{read}) \ \& \ \mathsf{exists}(i, foo).
\end{aligned}$$

The first rule sets the permissions necessary to create file foo. The other rules are needed by Mallory to access such a file.

Once an employee of Customer Services Division has run the procedure $openA$, the bank account is created together with file foo. The authorization framework then infers that Mallory is entitled to read foo. However, the bank IT system keeps trace that foo is derived by customer information by storing the following literals:

$$\{\mathsf{derivedFrom}(foo, n), \mathsf{derivedFrom}(foo, sa), \mathsf{derivedFrom}(foo, p)\}$$

Applying the information flow constraint to this scenario, the system spots a harmful situation for the data subject since his information can be accessed by unauthorized users.

Notice that we have proposed to verify only step-by-step flow, that is, we compare authorizations associated with an object only with those associated with the objects directly used to derive it. We adopt this solution since we claim that, if a system administrator has already allowed an "unauthorized" flow, an additional warning on the same flow is unnecessary. However, we can easily verify information flow with respect to primitive information by making relation derivedFrom transitive.

6 Related Work

Proposals for enforcing access control policies can be classified under three main classes: discretionary access control (DAC) [4, 5], mandatory access control (MAC) [1–3], and role based access control (RBAC) [6]. DAC allows users to specify which subjects can access their objects by directly defining an access control policy for each of their own objects. In MAC approaches, users cannot fully control the access to their objects, but it is the system that entirely determine the access that is to be granted. RBAC improves DAC and MAC proposals by integrating access control policies into the organizational structure of the system.

DAC models restrict access to objects on the basis of the identity of the invoking user and authorizations defining the actions he can execute on such objects. However, DAC models do not provide any form of support to control the usage of information once it is has been accessed [9]. Thereby, they are vulnerable to Trojan horse attacks [22]. This awareness has been matched by a number of research proposals on incorporating information flow control into access control models. Some proposals [11, 12] associate with each object an ACL and propagate it together with the information in the object. In particular, the access control list associated with a new object is given by the intersection of all ACLs associated with the objects used to create it. Similarly, in [20, 10] an information flow is defined to be safe if the ACL associated with the new object is a subset of the intersection of the sets of authorizations associated with the objects used to derive it. However, contrarily to our work, these proposals do not deal with the dynamic creation of objects.

Among MAC models, the model proposed by Bell and LaPadula [1] is a milestone for later work. Essentially, the model categorizes the security levels of objects and subjects, and enforces information flow to comply with "no read up" and "no write down" rules. Then, this model was generalized into the lattice model [2, 3]. The above rules are very robust but have some disadvantages. The main drawback is covert channels [21]. A covert channel represents an implicit information flow that cannot be controlled by the security policy. Several proposals have been presented to cope with this problem [23–25]. However, their focus is on the detection of improper information leaks rather than on the the dynamic creation of objects.

Osborn [26] proposed to verify information flow in the RBAC model through a MAC approach. Essentially, they propose to map a role graph [27] (i.e., a graphical notation for representing RBAC hierarchies) into an information flow graphs which shows the information flow among roles. However, in this work information flow refers to the propagation of primitive information with respect to hierarchies of roles, rather than to derived information.

Yusuda et al. [28] propose a purpose-oriented access control model. Essentially, purpose-oriented access rules identify which operations associated with an object can invoke operations associated with other objects modifying the objects themselves. These operations are classified with respect to the type of information flow. Based on this classification, they build an invocation graph that, together with a MAC model, is used to detect information leakages. Izaki et al. [29] integrate the RBAC model into the purpose-oriented model. Essentially, the purpose-oriented model is enhanced by introducing the concept of role. The idea underlying this approach is to classify object methods and derive a flow graph from method invocations. From such a graph, non-secure information flows can be identified.

7 Conclusion

The main contribution of this paper is a procedure for dynamically creating objects and automatically deriving access control policies to be associated with them. First, we have introduced creating rules in order to verify the conditions under which objects can be created and add "legal" objects to the state of the system. Then, we have defined a flexible framework for associating with a new object an access control policy based on the authorizations associated with the objects used to create it. However, the architecture of FAF does not support the dynamical creation of objects. Thus, we have improved it together with its underlying logic-based framework in order to preserve the locally stratified structure. This ensures the validity of advantage gained by FAF over its predecessors in specifying and enforcing access control policies. Finally, we have provided a mechanism in order to detect information leakages in the specification.

References

1. Bell, D.E., LaPadula, L.J.: Secure Computer System: Unified Exposition and MULTICS Interpretation. Technical Report MTR-2997 Rev. 1, The MITRE Corporation, Bedford, MA (1976)
2. Brewer, D.F.C., Nash, M.J.: The chinese wall security policy. In: Proc. of Symp. on Sec. and Privacy, IEEE Press (1989) 206–214
3. Denning, D.E., Denning, P.J.: Certification of programs for secure information flow. CACM **20**(7) (1977) 504–513
4. Downs, D., Rub, J., Kung, K., Jordan, C.: Issues in Discretionary Access Control. In: Proc. of Symp. on Sec. and Privacy, IEEE Press (1985) 208–218
5. Griffiths, P.P., Wade, B.W.: An authorization mechanism for a relational database system. TODS **1**(3) (1976) 242–255
6. Sandhu, R.S., Coyne, E.J., Feinstein, H.L., Youman, C.E.: Role-based access control models. IEEE Comp. **29**(2) (1996) 38–47
7. Sabelfeld, A., Myers, A.C.: Language-Based Information-Flow Security. IEEE J. on Selected Areas in Comm. **21**(1) (2003) 5–19
8. Chong, S., Myers, A.C.: Security Policies for Downgrading. In: Proc. of CCS'04, ACM Press (2004) 198–209
9. Bertino, E., Samarati, P., Jajodia, S.: High assurance discretionary access control for object bases. In: Proc. of CCS'93, ACM Press (1993) 140–150

10. Samarati, P., Bertino, E., Ciampichetti, A., Jajodia, S.: Information flow control in object-oriented systems. TKDE **9**(4) (1997) 524–538
11. McCollum, C.D., Messing, J.R., Notargiacomo, L.: Beyond the pale of MAC and DAC-defining new forms of access control. In: Proc. of Symp. on Sec. and Privacy, IEEE Press (1990) 190–200
12. Stoughton, A.: Access flow: A protection model which integrates access control and information flow. In: Proc. of Symp. on Sec. and Privacy, IEEE Press (1981) 9–18
13. Zannone, N., Jajodia, S., Massacci, F., Wijesekera, D.: Maintaining Privacy on Derived Objects. In: Proc. of WPES'05, ACM Press (2005) 10–19
14. Jajodia, S., Samarati, P., Sapino, M.L., Subrahmanian, V.S.: Flexible support for multiple access control policies. TODS **26**(2) (2001) 214–260
15. Baral, C.R., Subrahmanian, V.S.: Stable and extension class theory for logic programs and default logics. J. of Autom. Reas. **8**(3) (1992) 345–366
16. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Proc. of ICLP'88, MIT Press (1988) 1070–1080
17. Scott, D.S.: Identity and existence in intuitionistic logic. In: Application of Sheaves. Volume 753 of Lecture Notes in Mathematics. Springer Verlag (1979) 660–696
18. Liskov, B.H., Wing, J.M.: A Behavioral Notion of Subtyping. TOPLAS **16**(6) (1994) 1811–1841
19. van Gelder, A.: The alternating fixpoint of logic programs with negation. In: Proc. of PODS'89, ACM Press (1989) 1–10
20. Ferrari, E., Samarati, P., Bertino, E., Jajodia, S.: Providing flexibility in information flow control for object oriented systems. In: Proc. of Symp. on Sec. and Privacy, IEEE Press (1997) 130–140
21. Focardi, R., Gorrieri, R.: The Compositional Security Checker: A Tool for the Verification of Information Flow Security Properties. TSE **23**(9) (1997) 550–571
22. Samarati, P., di Vimercati, S.D.C.: Access Control: Policies, Models, and Mechanisms. In: FOSAD 2001/2002. Volume 2946 of LNCS. Springer (2001) 137–196
23. He, J., Gligor, V.D.: Information-Flow Analysis for Covert-Channel Identification in Multilevel Secure Operating Systems. In: Proc. of the 3rd IEEE Comp. Sec. Found. Workshop, IEEE Press (1990) 139–149
24. National Computer Security Center: A Guide to Understanding Covert Channel Analysis of Trusted Systems. Technical Report NCSC-TG-030, Library No. S-240,572, National Security Agency (1993)
25. Pernul, G.: Database Security. Advances in Computers **38** (1994) 1–72
26. Osborn, S.L.: Information flow analysis of an RBAC system. In: Proc. of SACMAT'02, ACM Press (2002) 163–168
27. Nyanchama, M., Osborn, S.: The role graph model and conflict of interest. TISSEC **2**(1) (1999) 3–33
28. Yasuda, M., Tachikawa, T., Takizawa, M.: Information Flow in a Purpose-Oriented Access Control Model. In: Proc. of ICPADS'97, IEEE Press (1997) 244–249
29. Izaki, K., Tanaka, K., Takizawa, M.: Information flow control in role-based model for distributed objects. In: Proc. of ICPADS'01, IEEE Press (2001) 363–370