Maintaining Privacy on Derived Objects

Nicola Zannone
University of Trento
Via Sommarive 14, 38050
Povo, Trento, Italy
zannone@dit.unitn.it

Sushil Jajodia
Center for Secure Information Systems
George Mason University
Fairfax, VA 22030
jajodia@gmu.edu

Fabio Massacci
University of Trento
Via Sommarive 14, 38050
Povo, Trento, Italy
massacci@dit.unitn.it

Duminda Wijesekera
Center for Secure Information Systems
George Mason University
Fairfax, VA 22030
dwijesek@gmu.edu

ABSTRACT

Protecting privacy means to ensure users that access to their personal data complies with their preferences. However, information can be manipulated in order to derive new objects that may disclose part of the original information. Therefore, control of information flow is necessary for guaranteeing privacy protection since users should know and control not only who access their personal data, but also who access information derived from their data. Actually, current approaches for access control do not provide support for managing propagation of information and for representing user preferences.

This paper proposes to extend the Flexible Authorization Framework (FAF) in order to automatically verify whether a subject is entitled to process personal data and derive the authorizations associated with the outcome of data processing. In order to control information flow, users may specify the range of authorizations that can be associated with objects derived from their data. The framework guarantees that every "valid" derived object does not disclose more information than users want and preserves the permissions that users want to maintain. To make the discussion more concrete, we illustrate the proposal with a bank case study.

Categories and Subject Descriptors: H.2.7 [**Database Administration**]: Security, integrity, and protection

General Terms: Security, Languages.

Keywords: Data protection, Access control, Information flow.

1. INTRODUCTION

Privacy protection is becoming important because enterprises are becoming conscious of losing market share if they do not implement proper privacy practices. Many countries have promulgated privacy legislation to enforce data protection. The US privacy legislation is essentially based on Privacy Act of 1974 [1]. The Privacy Act defines a number of privacy guidelines on disclosure of data, accounting of certain disclosures, access to data, and agency requirements and rules in order to guarantee data protection. One of the central principles is the principle of transparency, i.e., when enterprises store data about customers they should disclose to customers which data is collected and how it is used. Therefore, an enterprise should declare who can access customers' personal data and how data is processed by the enterprise itself.

Data protection is a critical issue since systems may release information as part of their functionalities [7]. The outcome of a data processing can be seen as a new object, for example a report, containing information derived from those objects used to create it. If data owners do not maintain the control on such new objects, they could lose control on their own information, and so their privacy is not guaranteed. In order to avoid unauthorized propagation of information, data owners may directly specify their preferences on who can access their information and how it can be used, and then the system matches them with the authorizations it has computed before disclosing information. However, current access control proposals [5, 9, 11, 21] and privacy-aware technologies [3, 4, 8, 13] do neither automatically capture authorizations on derived information nor compare them with user preferences in order to enforce privacy protection.

Control of information flow [17] is essential to ensure that information flow does not disclose sensitive information to unauthorized entities, that is, entities not authorized by data owners. Some proposals have tried to overcome this weakness of access control systems by complementing them with some form of flow control [10, 16, 18, 24]. Some approaches [16, 24] associate with each object an access control list that is propagated together with the information in the object. Other approaches [10, 18] mainly protect against unauthorized outflow of information. However, these approaches are not flexible enough to take into account user preferences, specially, they do not guarantee the data owner right to control that its data is not misused.

In this paper, we extend the Flexible Authorization Framework (FAF) [12], a logic-based framework developed to specify and enforce access control policies, in order to deal with propagation of information with respect to user preferences. When a bank employee processes our personal data, we want to ensure that the employee is entitled to process it and, at the same time, we want to know and control who can access the object derived from our data and how it is used. On one side, we propose to verify if a subject has enough rights to create the object. On the other side, we define the notion of *access control policy* associated with an object as the set of authorizations involving that object and propose an approach to automatically derive the policy associated with objects created by using it. To represent user preferences, we introduce the notions of *at least policy* and *at most policy* that define the range in which authorizations can be granted. We also provide a taxonomy of functions that can be used to create objects. Based on the type of function, we define how policies for new objects can be automatically derived from those associated with the objects used to create it. Further, we discuss the conditions under which the authorizations associated with derived objects comply with user preferences. Defining the policies associated with derived objects, we will take into account the guidelines proposed by the US privacy legislation [1]. This ensures that the framework can be applied to real scenarios.

The remainder of the paper is structured as follows. Next (§2) we introduce a Bank Information System used as running example throughout the paper. Then, we provide a brief overview of FAF and describe some concepts we have introduced in order to specify policies (§3). We illustrate our approach for ensuring that objects are created only by authorized users and associating policies with such objects (§4). Next, we discuss the properties that are guaranteed by the framework (§5). Finally, we discuss related work (§6) and conclude with some directions for future work (§7).

2. RUNNING EXAMPLE

This section presents a sketch of the structure and access control policies for a bank branch that will be used throughout the paper as a running example.

A bank branch is directed by a general manager who has the task to guarantee correct data processing and ensure law enforcement. The bank branch is composed of several divisions. For sake of simplicity, we only consider Customer Service (CS) Division and Investment and Financial (IF) Division. Each division has its director, respectively CS-Director and IF-Director, and many divisional managers – respectively CS-Manager and IF-Manager – who have the task to help their directors to guarantee correct data processing. Finally, cashiers and analysts are basic employees, respectively, in CS Division and in IF Division. Further, CS-Managers can also act as cashiers.

Only CS-Staff can access personal and account data of bank customers, while IF Division should provide a report stating the financial status of the bank branch monthly to the general manager. The general manager has also a secretary who helps him to interact with the central bank management and branch divisions.

The Bank Information System manages clients information. According to the Privacy Act [1], the system should provide client account information if and only if consent is obtained from the client in question, unless disclosure of information would be to those officers and employees of the bank who need the data to perform their duties. Clients may refuse to share their data if they do not trust the system or feel they do not have sufficient control over the use of their own data. The Bank Information System also provides some automatic procedures in order to help employees to manage clients data and perform data processing.

3. FLEXIBLE AUTHORIZATION FRAMEWORK

The Flexible Authorization Framework (FAF) [12] is a logic-based framework developed to manage access to data by users. This framework is enough flexible to allow system administrators to specify multiple access control policies that can be enforced within a single system.

FAF provides a Data System (**DS**) that consists of users (U), groups (G), the objects (Obj) and sets of objects (types) (T) they are accessing, together with the roles (R) they may play, and access modes (A) they may use. Classification hierarchies on the various components are defined by meaning of partial orderings, namely $\leq_{\mathsf{OT}}$ for object-type, $\leq_{\mathsf{UG}}$ for user-group and $\leq_{\mathsf{R}}$ for role. **DS** is formally defined as a 5-tuple $(\mathsf{OTH}, \mathsf{UGH}, \mathsf{RH}, \mathsf{A}, \mathsf{Rel})$ where

- $\mathsf{OTH} = (\mathsf{Obj}, \mathsf{T}, \leq_{\mathsf{OT}})$ is an object-type hierarchy where Obj is a set of identifiers of objects, T is a set of type, and $\leq_{\mathsf{OT}}$ is a partial ordering such that $\forall o \in \mathsf{Obj}$ and $t \in \mathsf{T}$, $o \leq_{\mathsf{OT}} t$ iff o is of type t.
- $\mathsf{UGH} = (\mathsf{U}, \mathsf{G}, \leq_{\mathsf{UG}})$ is a user-group hierarchy where U is a set of identifiers of users, G is a set of identifiers of collection of users, and $\leq_{\mathsf{UG}}$ is a partial ordering such that $\forall u \in \mathsf{U}$ and $g \in \mathsf{G}$, $u \leq_{\mathsf{UG}} g$ iff user u is in group g.
- $\mathsf{RH} = (\emptyset, \mathsf{R}, \leq_{\mathsf{R}})$ is a role hierarchy where R is a set of roles, and $\leq_{\mathsf{R}}$ is a partial ordering such that $x \leq_{\mathsf{UG}} y$ iff x is a specialization of y. Notice that there is not the concept of primitive role and that users do not appear in the role hierarchy.
- A is a set of access modes;
- Rel is a set of (first order) predicates used to specify the access control policies.

According to [12], the expression *authorization subject* (AS) denotes those entities for which accesses are authorized (users, groups and roles), and the expression *authorization object* (AO) denotes those entities on which accesses are authorized (objects, types and roles). An authorization is a triple of the form $(o, s, \langle sign \rangle a)$ where $o \in \mathsf{AO}$, $s \in \mathsf{AS}$, “sign” is either “$+$” or “$-$”, and $a \in \mathsf{A}$. Essentially, the triple $(o, s, +a)$ means that subject s is authorized to execute action a on object o. Similarly, the triple $(o, s, -a)$ means that subject s cannot execute action a on object o. The set of authorizations is denoted by AUTH, while AUTH^+ and AUTH^- denote, respectively, the sets of positive and negative authorizations.

The architecture of FAF is based on four modules: a propagation module, a conflict resolution module, a decision module, and an integrity enforcement module. Each module corresponds to a phase for managing authorizations. The first module provides some basic facts, such as component hierarchies and a set of authorizations along with rules to derive additional authorizations. The second module uses conflict resolution policies to eliminate contradictory authorizations. In the third module, decision policies are applies to ensure the completeness of authorizations. The last module is used to check integrity constraints.

Based on this architecture, FAF uses a locally stratified logic programming language, Authorization Specification Language (**ASL**), in order to specify authorizations. Next, we provide a brief overview of its syntax. A summary of the syntax is given in Table 1 [12].

- A ternary predicate symbol cando representing authorizations directly defined by the system administrator. The first argument is a subject, the second is an object, and the third is a signed action terms. Depending on the sign, authorizations are permissions or prohibitions.

Predicate	Rules defining predicate
hie-predicates	base relations.
rel-predicates	base relations.
done	base relation.
cando	body may contain done, hie- and rel-literals.
over	body may contain cando, done, hie- and rel-literals.
dercando	body may contain cando, over, dercando, done, hie-, and rel- literals. Occurrences of dercando literals must be positive.
do	in the case when head is of the form do$(o, s, +a)$, body may contain cando, dercando, done, hie- and rel- literals.
do	in the case when head is of the form do$(o, s, -a)$, body contains just one literal $\neg$ do$(o, s, +a)$.
error	body may contain do, dercando, cando, done, hie- and rel-literals.

Table 1: Syntax of the component of ASL

- A ternary predicate symbol dercando that has the same arguments of predicate cando and is used to represent authorizations derived through propagation policies.
- A ternary predicate symbol 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 symbol done. In particular, $\mathsf{done}(o, s, r, a, t)$ that holds if subject s playing role r has executed action a on object o at time t.
- Two 4-ary predicate symbols $\mathsf{over}_{\mathsf{AO}}$ and $\mathsf{over}_{\mathsf{AS}}$ where $\mathsf{over}_{\mathsf{AO}}$ takes as arguments two object terms, a subject term, and a signed action term, and $\mathsf{over}_{\mathsf{AS}}$ takes as arguments a subject term, two object terms, and a signed action term. They are used to express overriding policies.
- A propositional symbol error used to represent violation of integrity constraints.
- A set of hie-predicate symbols. In particular, the ternary predicate $\mathsf{in}(x, y, \mathsf{H})$ denotes that $x \leq y$ in hierarchy H.
- A set of rel-predicate symbols. In particular, the unary predicate symbols isuser, isgroup, isrole, isobject and istype which are true if their argument is a user, a group, a role, an object, or a type, respectively.

Let p be a predicate symbol with arity n, and $t_1, \ldots, t_n$ be its appropriate terms. Then, $p(t_1, \ldots, t_n)$ is called *atom*. Further, the expression *literal* denotes an atom or its negation.

In this paper we mainly focus on the first module of the architecture. Therefore, we define only the rules used by the system administrator for deriving authorizations. Essentially, authorizations are specified by using authorization rules [12] that have the form

$$\mathsf{cando}(o, s, \langle sign \rangle a) \leftarrow L_1 \ \& \ \cdots \ \& \ L_n.$$

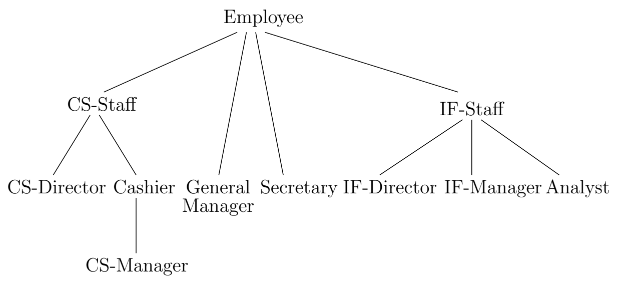
where o, s and a are, respectively, elements of AO, AS and A, $n \geq 0$, sign is either $+$ or $-$, and $L_1, \ldots, L_n$ are done, hie-, or rel-literals. Then, the collection of derived cando literals represents the set of authorizations.

3.1 Role Domination Hierarchy

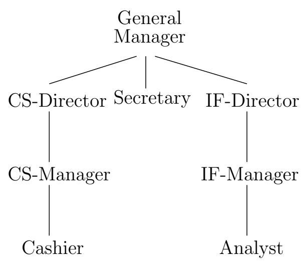
FAF supports a notion of role classification hierarchy based on the concept of specialization where a role is a specialization of another role if it refers to more specialized activities. Roles also may be organized with respect to the concept of domination. We define *role domination hierarchy* as follows.

DEFINITION 1. *Let* R *be the set of roles. A* role domination hierarchy RDH *is a triple* $(\emptyset, \mathsf{R}', \leq_{\mathsf{RD}})$ *where:*

1. $\mathsf{R}' \subseteq \mathsf{R}$,



(a) Classification Hierarchy



(b) Domination Hierarchy

Figure 1: Role Hierarchies

2. $\leq_{\mathsf{RD}}$ *is a partial order on* R'*. We say that* $x \leq_{\mathsf{RD}} y$ *if* y dominates x *(or, equivalently,* x is dominated by y*).*

EXAMPLE 1. *Figure 1(a) and Figure 1(b) show, respectively, the role classification hierarchy and role domination hierarchy related to our running example. Since the secretary should explain and justify her work to the general manager, this dominates the secretary. In symbols,*

$$Secretary \leq_{\mathsf{RD}} General\_Manager$$

In **ASL***, it is represented as*

$$\mathsf{in}(Secretary, General\_Manager, \mathsf{RDH})$$

FAF does not support the concept of primitive role and users do not appear in role classification and role domination hierarchies. In order to associate users to roles, FAF uses an explicit activation expressed in the form of authorizations.

EXAMPLE 2. *To express that Bob is the user playing the role of director in CS Division, the following authorization is used*

$$\mathsf{cando}(CS\text{-}Director, Bob, +\mathsf{activate})$$

3.2 Constraints

Entities can be seen as pairs of attributes and values. This allows policy writers to formulate system requirements in terms of constraints on the values of attributes. Since FAF is based on a logic programming language, we base our notion of constraint on that presented in Constraint Logic Programming [15].

The constraint domain $\mathcal{D}$ is composed of elements, together with a language for dealing with them. The language is defined by a set of function symbols, and relation symbols. The constraint domain defines the rules for building constraints. In our constraint domain, the set of variables $\mathcal{V}$ takes values over the sets Obj, T, U, G, R, A, and natural numbers $\mathbb{N}$. The set of predicate symbols Γ contains predicates for comparing natural numbers and members of Obj, T, U, G, R, A. The set of function symbols $\mathcal{F}$ contains constants, that is, every member of Obj, T, U, G, R, A and $\mathbb{N}$.

Given a constraint domain $\mathcal{D}$, the basic form of constraint is the *primitive constraint*. A primitive constraint consists of a relation symbol from $\mathcal{D}$ together with the appropriate number of well-typed[1] arguments. These are constants and variables.

DEFINITION 2. *Let Γ be the set of predicates, $\mathcal{F}$ be the set of functions and $\mathcal{V}$ be the set of variables. A* primitive constraint *is a literal built on $\langle \Gamma, \mathcal{F}, \mathcal{V} \rangle$. We assume that* true *and* false *are primitive constraints where* true *always holds, and* false *never holds.*

By replacing variables by all possible values, a primitive constraint becomes a variable-free constraint which may be either true or false. Therefore, we can determine the values of variables for which the primitive constraint holds, that is, a ground instantiation of the primitive constraint that is true in the constraint domain.

Next, we redefine *authorization rules*.

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

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

where $L_1, \ldots, L_n$ are done*,* do*, hie-, or rel-literals, and primitive constraints.*

The use of do literals in the body of authorization rules will be clear in Section 4. However, the architecture of FAF should be modified for integrating these "new" authorization rules into the entire process used to determine whether an access is granted. We refer to Appendix A for a brief overview of this process.

3.3 Access Control Policies and User Preferences

Authorizations are derived through authorization rules and are used to determine which entities are entitled to access an object and which actions they can perform on it. Based on this intuition, we define the access control policy of an object as the set of positive authorizations referring to the object.

DEFINITION 4. *An* access control policy *associated with an object o is a set of positive authorizations where o occurs as authorization object.*

Notice that we only focus on positive authorizations. Surely, specifying also negative authorizations in the access control policy allows us to express more sophisticate policies, but requires to solve problems, such as conflict resolution, that go well beyond the aims of this paper. So here we assume that only positive authorizations can be specified in an access control policy, and leave the possibility of specifying negative authorizations to future work.

[1]Essentially, well-typed means that arguments have the same type of those required by the definition of the relation.

More complicated access control policies can be built from authorization rules by using union, intersection, set difference, and complement. To be able to incrementally compute policies upon changes to the specification, we maintain a materialized view [12] that stores the policy terms derived from authorization rules.

DEFINITION 5. *The* materialization structure *for an authorization specification is a set of pair (A, S), where A is a ground atom in the authorization specification language and S is a set of (index of) rules whose head unifies with A.*

Given a materialization structure $\mathcal{MS}$ of an authorization rule $\mathcal{A}$, the model $\mathcal{M}$ of $\mathcal{A}$ is the projection over the first element of the pair, that is, $\mathcal{M}(\mathcal{A}) = \Pi_1(\mathcal{MS}(\mathcal{A}))$.

DEFINITION 6. *Access control policies are constructed using the following rules:*

1. *If $\mathcal{A}$ is an authorization rule, then $\mathcal{P} = \mathcal{M}(\mathcal{A})$ is an access control policy;*
2. *If $\mathcal{P}_1$ and $\mathcal{P}_2$ are access control policies, then so are $\mathcal{P}_1 \cup \mathcal{P}_2$, $\mathcal{P}_1 \cap \mathcal{P}_2$, $\mathcal{P}_1 \setminus \mathcal{P}_2$, and $\overline{\mathcal{P}_1}$.*

Notice that, if $\mathcal{A}$ and $\mathcal{B}$ are authorization rules, the union of their models, $\mathcal{M}(\mathcal{A}) \cup \mathcal{M}(\mathcal{B})$, is equal to the model of their union, $\mathcal{M}(\mathcal{A} \cup \mathcal{B})$, since authorization rules are not recursive. In the remainder, we refer to "set of authorization rules" for indicating the union of authorization rules and represent the union of their models as the model of their union. Further, intersection, set difference and complement could be directly implemented in the specification [25], but this requires changing the specification itself.

Since a policy refers to a specific object, for sake of simplicity we omit the object whenever it is fixed and clear from the context. So we represent a policy as a set of elements of the form $(s, +a)$ where s is a subject and a is an action.

DEFINITION 7. *Let* AO *be the set of authorization objects and* AUTH^+ *be the set of positive authorizations. The function policy* : $\mathsf{AO} \mapsto \wp(\mathsf{AUTH}^+)$ *associates a set of authorization pairs with an object o such that*

$$\mathit{policy}(o) = \{(s, a) | (o, s, +a) \in \mathsf{AUTH}^+\}$$

Essentially, function policy returns the access control list associated with a given object.

A data owner may want to maintain some permissions on objects created by using his data for checking that they are not misused, and, at the same time, he may want to restrict access to them. This represents user preferences that can be modeled through two sets of authorizations. The first set is compounded by the authorizations that at least an object should have associated with.

DEFINITION 8. *Let* AO *be the set of authorization objects and* AUTH^+ *be the set of positive authorizations. The function $\mathit{policy}_{\geq}$* : $\mathsf{AO} \mapsto \wp(\mathsf{AUTH}^+)$ *associates with an object the set of authorizations that it should at least have. Given an object o, $\mathit{policy}_{\geq}(o)$ is called* at least policy *of o.*

To restrict access, we introduce the set of authorizations that at most an object should have associated with.

DEFINITION 9. *Let* AO *be the set of authorization objects and* AUTH^+ *be the set of positive authorizations. The function* $policy_{\leq} : \mathsf{AO} \mapsto \wp(\mathsf{AUTH}^+)$ *associates with an object the set of authorizations that it should at most have. Given an object o,* $policy_{\leq}(o)$ *is called* at most policy *of o.*

In this paper, we assume that users can express their preferences only by defining positive authorizations. Then, negative authorizations can be easily computed from the positive ones: when a positive authorization is missing, this means that there is the corresponding negative authorization.

At least and at most policies are defined by users with respect to their preferences and together represent the range in which authorizations can be granted. Obviously, conflicts can arise between enterprise policies and user preferences.

DEFINITION 10. *For every authorization object o the following relation must hold*

$$policy_{\geq}(o) \subseteq policy(o) \subseteq policy_{\leq}(o)$$

If the policies associated with an object do not respect this relation, we call it zombie object *and assume that every access to it is blocked until conflicts are resolved.*

Zombie objects are objects that either disclose more information that data owners want or do not guarantees data owners to maintain the right to verify that their information is not misused. Therefore, they are objects for which privacy requirements are not satisfied.

To model the concepts we have introduced in this section, we extend **ASL** by introducing the ternary predicates $\mathsf{cando}_{\geq}(o, s, a)$ and $\mathsf{cando}_{\leq}(o, s, a)$, and the unary predicate $\mathsf{zombie}(o)$. The first two predicates are used to represent, respectively, at least and at most policy, and their behavior is similar to predicate cando. The predicate $\mathsf{zombie}(o)$ holds if authorization object o is a zombie object. To derive zombie objects, we use the following rules:

$$\mathsf{zombie}(o) \leftarrow \mathsf{cando}_{\geq}(o, s, a) \,\&\, \neg\mathsf{cando}(o, s, a).$$
$$\mathsf{zombie}(o) \leftarrow \mathsf{cando}(o, s, a) \,\&\, \neg\mathsf{cando}_{\leq}(o, s, a).$$

These rules can be seen as integrity constraints used to verify the validity of subset relations.

4. CREATING OBJECTS

Enterprises process data in order to provide services to their customers, where data processing refers to a class of procedures used to organize and manipulate data. The outcome of a data processing can be seen as a new object that could contain information belonging to those objects used to create it. Current approaches for access control and privacy-aware technologies enforce control on the disclosure of information but not its propagation [17]. In other words, they deal with data directly stored in the system (in the remainder of the paper, we call such kind of data *primitive objects*) but not with information derived by the system itself from such data. Controlling information flow is a critical issue since derived information may improperly release to unauthorized users information about the primitive objects used to derived it.

Without loss of generality we assume that functions are used to derive objects and that such objects do not belong to the system until they are created. Indeed procedures that manipulate more than one object can be seen as function on the object "collection of objects". Further, objects created without making use of other objects can be describe by functions with no arguments. In order to avoid unauthorized propagation of information, we start looking at what happen when a subject wants to derive an object. For example, suppose that the function f is used to create the object o.

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

where id is a unique identifier that univocally identifies the object, s is the subject that wants to create o, and $o_1, \ldots, o_m$ are the objects needed to create o and are called *source objects*. In order to enforce data protection, we should be able to answer two questions:

1. Is the subject s entitled to create the derived object o?
2. Who is authorized to access the derived object o?

In agreement with the transparency principle, this information should be provided by an enterprise to its customers before they disclose their data. For example, if a customer thinks that bank policies are not "reasonable", he could decide to change the bank.

This principle is also at the basis of the assumption that the system exactly knows the objects used by data processing. Suppose to execute a function f_1 that invokes a hidden function f_2 that accesses objects o_1 and o_2. If the system does not advise the owner of o_1 and o_2 that these objects are used by function f_1, his privacy could be not preserved. Further, there may be cases where a subject cannot execute f_1 since he is not authorized to access o_1 and/or o_2.

Notice that this assumption is absolutely realistic for all business IT systems. Indeed whereas the actual users (and even administrators) of the bank IT system may not know that the hidden function is called when they open the "checkLoan" form, the developers of the system surely knew it as they actually put the hidden call! So we simply need to have this information made available to the privacy subsystem and then it is immediate that the system itself would know exactly which objects are necessary for creating a new one.

4.1 Authorizations for creating objects

To create an object, subjects may need to use exiting objects (either primitive objects or objects already created). Further, only subjects that play a certain role or belong to a certain group may be entitled to create the object. The Privacy Act declares that agencies that maintain personal data should adopt appropriate administrative and technical measures to enforce the security and confidentiality of the data, and so to protect it against any threats which could result in harm to any individual's data. In other words, data access should be granted only to authorized subjects. This means that authorization systems must verify whether the subject has enough rights to access all objects used to create the new one and whether the subject can create the object.

The idea is to make explicit the conditions under which a subject can create an object in order to verify his responsibilities and capabilities. Based on this intuition, we redefine the function f used to create the object o as

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

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 function. In **ASL**, this can be implemented with rules of the form

$$\mathsf{isobject}(f(id, s, o_1, \ldots, o_m)) \leftarrow L_1 \,\&\, \ldots \,\&\, L_n.$$

where $L_1, \ldots, L_n$ are done, do, hie-, or rel-literals, and primitive constraints such that their conjunction is equal to $\mathcal{C}$, and do literals refer only to $o_1, \ldots, o_m$ (see Appendix A for more details). To create objects we need to verify whether subjects have enough right on the source objects. Thus, we propose to use do literals since they represent effective authorizations[2] on objects. Notice also that we have used $f(id, s, o_1, \ldots, o_m)$ as argument of predicate isobject instead of o only to point out that it is a derived object and emphasize which objects are used to derive it. Notice that function symbols can be introduced into the specification language without loosing decidability of the specification provided all terms in a rules are range restricted by (non-recursive) unary domain predicates. In the previous example we would have had to add the literals $\mathsf{name}(n)$, $\mathsf{shipping\_addr}(sa)$, $\mathsf{bank\_account}(openBA(\ldots))$ etc. in the body of the rule. We don't add these domain predicates because this will hamper the readability of the rules in the examples.

From a practical standpoint this is not a limitation because even in sophisticated ERP systems the user (once again in contrast to the developer) can create objects only using predefined functions and these are never recursive and thus domain predicates can always be defined. For example, to create the loan form one needs to have a bank account, and bank accounts require an address, and addresses may be broken down in street and city names. Though we can complicate the picture down to floor in the storey, each time we use different types and at the end the name of the city is essentially a primitive object. Figure 5 gives another example of non-recursive structured information that is perfectly decidable with domain predicates.

EXAMPLE 3. *Suppose that a customer wants to open a bank account. A bank policy states that the officer responsible for opening bank accounts must have the permission to access customers' personal data, namely name (n), shipping address (sa), and phone number (p). Further, only a manager of the Customer Service Division is able to open a bank account, and the manager's name is associated with the account for security reason.*[3] *The Bank Information System provides the function openBA for creating a bank account. This function is also used to insert the balance of the account (the first deposit (d)) into the Bank Information System. The bank also requires that the deposit is greater than* 0.

$$\begin{aligned}\mathsf{isobject}(openBA(id, s, n, sa, p, d)) \leftarrow\ & \mathsf{isuser}(s)\ \&\ \mathsf{do}(n, s, +\mathsf{read})\\ & \&\ \mathsf{do}(sa, s, +\mathsf{read})\ \&\ \mathsf{do}(p, s, +\mathsf{read})\\ & \&\ \mathsf{do}(d, s, +\mathsf{read})\ \&\ d > 0\ \&\\ & \mathsf{do}(CS\text{-}manager, s, +\mathsf{activate}).\end{aligned}$$

EXAMPLE 4. *IF Division has the task to monthly provide a financial status report to the general manager. Let* Account *be the type of objects $openBA(id, s, n, sa, p, d)$ stored into Bank Information System and createFSR be the function to create the report.*

$$\begin{aligned}\mathsf{isobject}(createFSR(id, s, sum(\mathsf{Account}))) \leftarrow\ & \mathsf{in}(s, \mathit{IF\text{-}Staff}, \mathsf{RH})\\ & \&\mathsf{do}(sum(\mathsf{Account}), s, +\mathsf{read}).\end{aligned}$$

4.2 Authorizations on derived objects

Once an object is created, the owners of the primitive objects used to create it want to know who can access it and how it is used. To this end, a policy should be associated with such object. Since the new object is not independent from the objects used to derived it, the policy associated with it should take into account the effective authorizations associated with the objects used to derive it.

However, when we define the policy for a derived object we should take into account that not all data processing disclose individually identifiable information. For example, the sum of account balances at a bank branch does not disclose data that allows to recover information associating a user with his own account balance. Therefore, we need a taxonomy of functions used to create objects. Here, we have identified three main types of functions: *non parametric functions*, *disclosure functions*, and *non disclosure functions*. Next, we present such functions and show the policies associated with derived objects for each function.

4.2.1 Non Parametric Function (NPF)

The basic case is when an object is created by using no additional objects. Thus, we introduce non parametric functions for creating fresh objects such as new files. In this case, arbitrary policies can be used to express policies for derived objects.

$$\begin{aligned}policy(f_{NPF}(id, s)) &= \mathcal{P}\\ policy_{\geq}(f_{NPF}(id, s)) &= \mathcal{P}_{\geq}\\ policy_{\leq}(f_{NPF}(id, s)) &= \mathcal{P}_{\leq}\end{aligned}$$

where $\mathcal{P}$, $\mathcal{P}_{\geq}$ and $\mathcal{P}_{\leq}$ are policies such that $\mathcal{P}_{\geq} \subseteq \mathcal{P} \subseteq \mathcal{P}_{\leq}$.

EXAMPLE 5. *Let* R *be the set of roles and* BulletinBoard *be the type of objects bulletin. Let writeB be the function to write a bulletin. The access control policy associated with the bulletin should allow all the roles that dominate the one which has written the bulletin to read and modify it. The policy associated with the bulletin is the model of the following authorization rules*

$$\begin{aligned}\mathsf{cando}(writeB(id, t), S, A) &\leftarrow \mathsf{in}(t, S, \mathsf{RDH})\ \&\ A = +\mathsf{modify}.\\ \mathsf{cando}(writeB(id, t), S, A) &\leftarrow \mathsf{in}(t, S, \mathsf{RDH})\ \&\ A = +\mathsf{read}.\end{aligned}$$

Suppose that the general manager delegates to his secretary the task to write the notice concerning the closing dates for holidays to the branch staff, the policy associated with the notice allows the general manager to read and modify the notice.

4.2.2 Disclosure Function (DF)

When a subject creates an object, this may disclose information about the objects used to create it. To protect such information, the policy associated with the object should be the intersection of the policies associated with the source objects [16]. However, some information must be disclosed for satisfying availability requirements. The Privacy Act allows an agency to disclose data without the consent of the data owner to those officers and employees of the agency who have a need for the data to perform their duties. Therefore, an employee that should provide a certain task for which the data is necessary should be entitled to access to the data. Further, some accesses could be restricted. For example, a bank does not consider "reasonable" that a client modifies his account balance by himself. Based on these observations, we define the policy associated with an object created by using a disclosure function as

$$policy(f_{DF}(id, s, o_1, \ldots, o_m)) = \left(\left(\bigcap_{i \in [1,\ldots,m]} policy(o_i)\right) \cup \mathcal{P}_1\right) \setminus \mathcal{P}_2$$

where policy $\mathcal{P}_1$ is used to grant access for guaranteeing availability requirements and policy $\mathcal{P}_2$ to limit the access to the object.

EXAMPLE 6. *Referring to Example 3, the sets of authorization rules associated with a new bank account are defined in Figure 2. The first represents the policies associated with the objects used to derive it, while the second refers to the bank policy stating that only employees working in the CS Division can read and modify information on customer bank account. The last refers to the capability of a client to modify information about his account by himself. The policy associated with the bank account is given by*

$$policy(openBA(id, s, n, sa, p, d)) = \big(\mathcal{M}(\mathcal{A}_1) \cup \mathcal{M}(\mathcal{A}_2)\big) \setminus \mathcal{M}(\mathcal{A}_3)$$

[2]We remark that do literals are derived after conflicts resolution and decision policies.

[3]This impersonation is closer to reality than one may think: the law requires to assign the responsibility of each entity to human beings.

$\mathcal{A}_1$	$\mathsf{cando}(\mathit{openBA}(id,t,n,sa,p,d),S,A) \leftarrow \mathsf{do}(n,S,A).$
	$\mathsf{cando}(\mathit{openBA}(id,t,n,sa,p,d),S,A) \leftarrow \mathsf{do}(sa,S,A).$
	$\mathsf{cando}(\mathit{openBA}(id,t,n,sa,p,d),S,A) \leftarrow \mathsf{do}(p,S,A).$
	$\mathsf{cando}(\mathit{openBA}(id,t,n,sa,p,d),S,A) \leftarrow \mathsf{do}(d,S,A).$
$\mathcal{A}_2$	$\mathsf{cando}(\mathit{openBA}(id,t,n,sa,p,d),S,A) \leftarrow \mathsf{in}(S, \mathit{CS\text{-}Staff}, \mathsf{RH})\ \&\ A = +\mathsf{read}.$
	$\mathsf{cando}(\mathit{openBA}(id,t,n,sa,p,d),S,A) \leftarrow \mathsf{in}(S, \mathit{CS\text{-}Staff}, \mathsf{RH})\ \&\ A = +\mathsf{modify}.$
$\mathcal{A}_3$	$\mathsf{cando}(\mathit{openBA}(id,t,n,sa,p,d),S,A) \leftarrow \mathsf{owner}(n,S)\ \&\ A = +\mathsf{modify}.$

Figure 2: Policy associated with bank account

$\mathcal{A}_1$	$\mathsf{cando}(\mathit{createFSR}(id,t,\mathit{sum}(\mathsf{Account})),S,A) \leftarrow \mathsf{do}(\mathit{sum}(\mathsf{Account}),S,A).$
$\mathcal{A}_2$	$\mathsf{cando}(\mathit{createFSR}(id,t,\mathit{sum}(\mathsf{Account})),S,A) \leftarrow \mathsf{in}(S, \mathit{IF\text{-}Staff}, \mathsf{RH})\ \&\ A = +\mathsf{modify}.$
$\mathcal{A}_3$	$\mathsf{cando}(\mathit{createFSR}(id,t,\mathit{sum}(\mathsf{Account})),S,A) \leftarrow S = \mathit{General\_Manager}\ \&\ A = +\mathsf{read}.$
	$\mathsf{cando}(\mathit{createFSR}(id,t,\mathit{sum}(\mathsf{Account})),S,A) \leftarrow S = \mathit{General\_Manager}\ \&\ A = +\mathsf{modify}.$
$\mathcal{A}_4$	$\mathsf{cando}(\mathit{createFSR}(id,t,\mathit{sum}(\mathsf{Account})),S,A) \leftarrow \mathsf{in}(S, \mathit{CS\text{-}Staff}, \mathsf{RH})\ \&\ A = +\mathsf{modify}.$

Figure 3: Policy associated with financial status report

EXAMPLE 7. *The sets of authorization rules associated with the financial status report (Example 4) are defined in Figure 3. The first set represents the policies associated with the objects used to derive it. The second refers to the capability of employees of IF Division to modify the report, and the third to the capability of the general manager to read and modify the report. The last refers to the capability of CS-Staff to modify the report. The policy associated with the financial status report is given by*

$$\mathit{policy}(\mathit{createFSR}(id,s,\mathit{sum}(\mathsf{Account}))) = \big(\mathcal{M}(\mathcal{A}_1) \cup \big(\mathcal{M}(\mathcal{A}_2) \cup \mathcal{M}(\mathcal{A}_3)\big)\big) \setminus \mathcal{M}(\mathcal{A}_4)$$

The first part establishes that CS-Staff is authorized to read the report for verifying whether the value of the sum is correct (Example 6). The second policy grants the permission to employees of IF Division to modify the report and authorize the general manager to read and modify the report. Notice that the explicit permission for the general manager is not necessary if authorization propagation is applied. The last policy denials CS-Staff to modify the report.

At least and at most policies associated with the derived object should be in agreement with the range defined by the user preferences associated with the object used to derive it. Thus, the derived at least policy should contain all at least policies associated with origin objects, and the derived at most policy should not allow to disclose more information than each at most policy associated with origin objects does. In symbols,

$$\mathit{policy}_{\geq}(f_{DF}(id,s,o_1,\ldots,o_m)) = \bigcup_{i\in[1,\ldots,m]} \mathit{policy}_{\geq}(o_i)$$
$$\mathit{policy}_{\leq}(f_{DF}(id,s,o_1,\ldots,o_m)) = \bigcap_{i\in[1,\ldots,m]} \mathit{policy}_{\leq}(o_i)$$

It is up to policy developers to define appropriate policies for implementing access restriction and availability requirements. These policies should respect basic privacy properties. Therefore, policy $\mathcal{P}_1$ should be not too "restrictive" since any individual should control his data [1]. Further, policy $\mathcal{P}_2$ should not provide more access than ones are needed since agencies should manage only such data about an individual as is relevant and necessary to accomplish the purpose for which data is maintained [1], or simply more than that data owner wants to grant. Therefore, these policies have to be compared with user preferences to ensure privacy protection.

PROPOSITION 1. *Let o be an object derived by using a disclosure function and $o_1,\ldots,o_m$ be the non zombie objects used to derive it. Let $\mathcal{P}_1$ be the policy used to grant additional rights and $\mathcal{P}_2$ be the policy used to limit the access to o. Object o is not a zombie object if and only if*

1. $\forall i \in [1,\ldots,m]\ \ \mathit{policy}_{\geq}(o_i) \cap \mathcal{P}_2 = \emptyset$;
2. $\mathcal{P}_1 \setminus \mathcal{P}_2 \subseteq \bigcap_{i\in[1,\ldots,m]} \mathit{policy}_{\leq}(o_i)$.
3. $\forall j,i \in [1,\ldots,m]\ \ \mathit{policy}(o_j) \subseteq \mathit{policy}_{\leq}(o_i)$
4. $\forall j,i \in [1,\ldots,m]\ \ \mathit{policy}_{\geq}(o_j) \subseteq \mathit{policy}(o_i)$

Intuitively the first conditions says that the restrictions should only affect authorizations beyond the basic ones, which should be granted anyhow. The third and the fourth conditions imply that the only essential differences between the objects is the way permissions are classified among what one should at least have, actually has or should at most have. Loosely speaking, they belong to very similar security domains. This is what is intuitively expected for *disclosure functions*: trying to compose objects with radically different authorizations into an object whose components can be reconstructed would result in zombie objects. The second condition is trickier as it states that the additional privileges granted to actually use the compound object should not exceed the total set of privileges assigned to the overall collections of component objects. We remark that zombie objects cannot be used to derive new objects since any access to them is denied until conflicts are resolved.

EXAMPLE 8. *Alice opening a bank account declares in her user preferences that she wants to have the permission not only to read but also to modify all the objects derived from her data. These user preferences are shown in Figure 4. In this case, the account $\mathit{openBA}(id,s,n,f,sa,p,d)$ is a zombie object since her preferences are in conflict with the bank policy that denials clients to modify their account by them selves ($\mathcal{M}(\mathcal{A}_3)$ in Example 6).*

4.2.3 Non Disclosure Function (NDF)

Functions such as statistical operations (sum, average and so on) do not disclose sensitive information, that is, the disclosure of information is not sufficient to trace the origin of the information itself. In this case policies can be "relaxed". This is also in agreement with the Privacy Act that does not impose any conditions on aggregate statistical data without any personal identifiers. From these considerations, we define the policy associated with an object derived by using a non disclosure function as the union of all the policies associated with the source objects. This is also compatible with the notion of declassification presented in [7]. As for disclosure functions, this is not sufficient and additional policies may be defined to denial or grant accesses. Thus, we define the policies associated with an object created from non disclosure functions as

$$\mathit{policy}(f_{NDF}(id,s,o_1,\ldots,o_m)) = \left(\left(\bigcup_{i\in[1,\ldots,m]} \mathit{policy}(o_i)\right) \cup \mathcal{P}_1\right) \setminus \mathcal{P}_2$$

$$\begin{array}{|l|}\hline \mathsf{cando}_{\geq}(n,S,A) \leftarrow \mathsf{owner}(n,S)\ \&\ a=+\mathsf{read}.\\ \mathsf{cando}_{\geq}(sa,S,A) \leftarrow \mathsf{owner}(sa,S)\ \&\ a=+\mathsf{read}.\\ \mathsf{cando}_{\geq}(p,S,A) \leftarrow \mathsf{owner}(p,S)\ \&\ a=+\mathsf{read}.\\ \mathsf{cando}_{\geq}(n,S,A) \leftarrow \mathsf{owner}(n,S)\ \&\ a=+\mathsf{modify}.\\ \mathsf{cando}_{\geq}(sa,S,A) \leftarrow \mathsf{owner}(sa,S)\ \&\ a=+\mathsf{modify}.\\ \mathsf{cando}_{\geq}(p,S,A) \leftarrow \mathsf{owner}(p,S)\ \&\ a=+\mathsf{modify}.\\ \hline \end{array}$$

Figure 4: User Preferences for at least policy

where policy $\mathcal{P}_1$ is used to grant access for guaranteeing availability requirements and policy $\mathcal{P}_2$ to limit the access to the object. As for the access control policy, also at least and at most policies should be relaxed since the derived object does not disclose individually identifiable information. In particular, the derived at least policy should contain only authorizations that the at least policies associated with source objects have in common, and the derived at most policy should allow to grant all the authorizations that each at most policy associated with source objects does. In symbols,

$$policy_{\geq}(f_{NDF}(id,s,o_1,\ldots,o_m)) = \bigcap_{i\in[1,\ldots,m]} policy_{\geq}(o_i)$$
$$policy_{\leq}(f_{NDF}(id,s,o_1,\ldots,o_m)) = \bigcup_{i\in[1,\ldots,m]} policy_{\leq}(o_i)$$

Notice that at least and at most policies for non disclosure functions are dual to those defined for disclosure functions. Further, we have proposed to derive the access control, at least and at most policies from source objects instead of defining arbitrary policies. This choice is due to the fact that we believe the the owners of source objects are in turn the owners of the derived object. These owners may want to maintain the control of their objects even if the object does not disclose sensitive information. Otherwise, if we assume that the owner is the entity who creates the object, for example the system administrator, the policies should be defined differently.

Enterprise policies and user preferences could be in conflict, and so the derived object could be a zombie objects. Next, we provide some conditions under which the policy associated with the derived object respects user preferences.

PROPOSITION 2. *Let o be an object derived by using a non disclosure function and $o_1,\ldots,o_m$ be the non zombie objects used to derive it. Let $\mathcal{P}_1$ be the policy used to grant additional rights and $\mathcal{P}_2$ be the policy used to limit the access to o. Object o is not a zombie object if and only if*

1. $\bigcap_{i\in[1,\ldots,m]} policy_{\geq}(o_i) \cap \mathcal{P}_2 = \emptyset$;
2. $\mathcal{P}_1 \setminus \mathcal{P}_2 \subseteq \bigcup_{i\in[1,\ldots,m]} policy_{\leq}(o_i)$.

EXAMPLE 9. *Let* Account *be the type of objects created by using function $openBA$ and sum_{bal} be the function that returns the sum of all account balance at branch. The derived policy is defined as the model of the following authorization rules.*

$$\begin{aligned}\mathsf{cando}(sum_{bal}(id,t,\mathsf{Account}),S,A) \leftarrow\ & \mathsf{do}(O,S,A)\ \&\\ & \mathsf{in}(O,\mathsf{Account},\mathsf{OGH}).\\ \mathsf{cando}(sum_{bal}(id,t,\mathsf{Account}),S,A) \leftarrow\ & \mathsf{in}(S,\textit{IF-Staff},\mathsf{RH})\ \&\\ & A=+\mathsf{read}.\end{aligned}$$

Essentially, the policy associated with $sum_{bal}(id,t,\mathsf{Account})$ is the union of all the policies associated with each account. The last rule is introduced since IF-Staff is not authorized to access to clients data (Example 6) and they need that information to create the financial status report (Example 4).

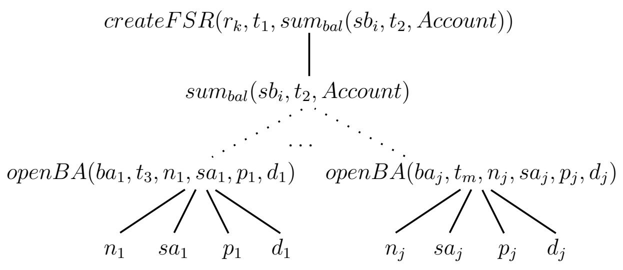

$createFSR(r_k,t_1,sum_{bal}(sb_i,t_2,Account))$
$sum_{bal}(sb_i,t_2,Account)$
$\cdots$
$openBA(ba_1,t_3,n_1,sa_1,p_1,d_1)$
$openBA(ba_j,t_m,n_j,sa_j,p_j,d_j)$
n_1
sa_1
p_1
d_1
n_j
sa_j
p_j
d_j

Figure 5: Derivation Tree

5. GUARANTEEING DATA PROTECTION

The outcome of data processing may be used as input for other data processing. The process to derive an object can be seen as a tree, *derivation tree*, where the root is the "final" object and the leaves are either primitive objects or objects derived from non parametric functions.

EXAMPLE 10. *Figure 5 shows the derivation tree of the financial status report created by IF-Staff. In the picture, full edges are used to indicate that the up level object is derived by using a disclosure function and dotted edges that the up level object is derived by using a non disclosure function. We call them disclosure step and non disclosure step, respectively.*

Next, we present some properties that are guaranteed by our framework. Their proofs are given in Appendix B.

The following result establishes that valid (i.e., no zombie) objects whose derivation tree contains only disclosure steps, do not propagate information to unauthorized subject and that data owners maintain the control on objects derived from their data.

THEOREM 1. *Let o be an object whose derivation tree contains only disclosure steps, and $o_1,\ldots,o_n$ be the primitive objects used to derive it. If zombie objects do not occur in the tree, the access control policy associated with the derived object is such that*

$$\forall i \in [1,\ldots,n] \quad policy_{\geq}(o_i) \subseteq policy(o) \subseteq policy_{\leq}(o_i)$$

Notice that the assumption for which zombie objects do not occur in the derivation tree is not too strong since the system cannot access such objects.

Additional properties can be guaranteed considering non disclosure functions. For any object whose derivation tree contains only non disclosure steps and no restrictions are introduced, the access control policy associated with each primitive object is a subset of the policy associated with the object derived by using them.

THEOREM 2. *Let o be an object whose derivation tree contains only non disclosure steps, and $o_1,\ldots,o_n$ be the primitive objects used to derive it. If no restriction policies are introduced, the access control policy associated with the derived object is such that*

$$\forall i \in [1,\ldots,n] \quad policy(o_i) \subseteq policy(o)$$

Further, under some conditions objects derived only by non disclosure functions are always valid objects.

THEOREM 3. *Let Δ be a derivation tree contains only non disclosure steps. If no restriction or additional policies are introduced, every object occurring in Δ is not a zombie object.*

Finally, we argue that the conditions given in Proposition 1 and Proposition 2 are enough expressive to guarantee data protection.

THEOREM 4. *Let Δ be a derivation tree where every disclosure step complies with the conditions in Proposition 1 and every non disclosure step complies with the condition in Proposition 2. Then, every object occurring in Δ is not a zombie object.*

6. RELATED WORK

The last decades have seen an increasing awareness that privacy plays a key role in organizations. One of the first approach addressing to privacy concerns was statistical databases [2] that allow only requests of statistical information through aggregate queries. In the cryptography field, the work on anonymizing has a long history since Chaum's first proposals [6]. One could even say that trust negotiation [22, 23] was inspired by privacy concerns of non-disclosure of sensitive credentials to unknown subjects. We argue that these approaches are not sufficient to guarantee privacy since privacy means giving the data owners the right to say what can be done with their data [1].

Among the privacy-aware technologies, Agrawal et al. propose Hippocratic databases [3] that use purpose as central concept around which privacy protection is built. Their aim is to negotiate the personal data between customers and enterprises and to enforce the enterprise to look up to its privacy policies. The P3P standard [8] provides mechanisms that allow users to check web site privacy policies before they discloses their personal data to the site. Karjoth et al. [13] improve P3P by proposing mechanisms for enforcing sites to act according to their stated policies. The Enterprise Privacy Authorization Language (EPAL) [4] enables an enterprise to exactly formalize the privacy policies that shall be enforced within the enterprise itself. However, these technologies are not able to control the information flow. Further, purpose-based approaches require the presence of an external authority that monitors the behavior of data recipients. Actually, once a subject has gotten access to sensitive information (even fairly), there is no way to enforce him to use data correctly.

Access control is fundamental for building secure information systems and, more specifically, for protecting the confidentiality of information manipulated by such systems [19, 20]. However, current approaches for access control do not provide complete support for managing propagation of information and representing users preferences. Ponder [9], Cassandra [5] and dRBAC [11] are developed to specify and manage access control policies in large-scale distributed systems. Essentially, they specify policies as rules that govern access control decisions and define the behavior of a system. However, these frameworks deal only with delegation of primitive information, and do not support propagation of information. Other access control frameworks address to these issues only partially. Some approaches [16, 24] propagate the access control list associated with an object, while others [10, 18] mainly protect against unauthorized outflow of information. However, these approaches are not expressive enough to model and enforce user preferences.

7. CONCLUSION

The main contribution of this paper is a procedure for automatically verifying permissions to create objects and deriving their access control policies which are enforced by the authorization framework with respect to user preferences. Although our approach has been implemented in FAF, it is quit general and can be applied to other authorization frameworks. Further, the approach is based on the the US privacy legislation. This allows us to apply the framework to real scenarios.

Future works will involve to extend the notion of access control policy and user preferences in order to take into account negative authorizations. This entails defining some mechanisms to solve possible conflicts that arise when objects used to create another object have associated incompatible policies. Further, the entire process for enforcing access control policies concerning derived objects should be formalized. Once the framework is fully formalized we plan to implement it into DLV system [14].

Acknowledgments. This work has been partially supported by the National Science Foundation under the grants CCR-0113515, IIS-0430402, and IIS-0242237, by the IST programme of the EU Commission, through a FET under the IST-2001-37004 WASP project, by the FIRB programme of MIUR under the RBNE0195K5 and RBAU01P5SS projects, by MOSTRO and SMTPPs projects of PAT.

8. REFERENCES

[1] Privacy Act of 1974. 5 USC, Section 552A. Available at `http://www.usdoj.gov/foia/privstat.htm` "Privacy of Consumer Financial Information; Final Rule." 16 CFR Part 313. Federal Register 65, No. 101.

[2] N. R. Adam and J. C. Worthmann. Security-control methods for statistical databases: a comparative study. *CSUR*, 21(4):515–556, 1989.

[3] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Hippocratic Databases. In *Proc. of VLDB'02*, pages 143–154. Morgan Kaufmann, 2002.

[4] M. Backes, B. Pfitzmann, and M. Schunter. A Toolkit for Managing Enterprise Privacy Policies. In *Proc. of ESORICS'03*, *LNCS 2808*, pages 162–180. Springer, 2003.

[5] M. Y. Becker and P. Sewell. Cassandra: distributed access control policies with tunable expressiveness. In *Proc. of POLICY'04*, pages 159–168. IEEE Press, 2004.

[6] D. L. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *CACM*, 24(2):84–90, 1981.

[7] S. Chong and A. C. Myers. Security Policies for Downgrading. In *Proc. of CCS'04*, pages 198–209. ACM Press, 2004.

[8] L. Cranor, M. Langheinrich, M. Marchiori, and J. Reagle. The Platform for Privacy Preferences 1.0 (P3P1.0) Specification. W3C Recommendation, Apr. 2002.

[9] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The Ponder Policy Specification Language. In *Proc. of POLICY'01*, *LNCS 1995*, pages 18–39. Springer, 2001.

[10] E. Ferrari, P. Samarati, E. Bertino, and S. Jajodia. Providing flexibility in information flow control for object oriented systems. In *Proc. of Symp. on Sec. and Privacy*, pages 130–140. IEEE Press, 1997.

[11] E. Freudenthal, T. Pesin, L. Port, E. Keenan, and V. Karamcheti. dRBAC: distributed role-based access control for dynamic coalition environments. In *Proc. of ICDCS'02*, pages 411–420. IEEE Press, 2002.

[12] S. Jajodia, P. Samarati, M. L. Sapino, and V. S. Subrahmanian. Flexible support for multiple access control policies. *TODS*, 26(2):214–260, 2001.

[13] G. Karjoth, M. Schunter, and M. Waidner. Platform for Enterprise Privacy Practices: Privacy-enabled Management of Customer Data. In *Proc. of PET'02*, *LNCS 2482*, pages 69–84. Springer, 2002.

[14] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV System for Knowledge Representation and Reasoning. *TOCL*, 2005. To appear.

[15] K. Marriott and P. J. Stuckey. *Programming with constraints: an introduction*. MIT Press, 1998.

[16] C. D. McCollum, J. R. Messing, and L. Notargiacomo. Beyond the pale of MAC and DAC-defining new forms of access control. In *Proc. of Symp. on Sec. and Privacy*, pages 190–200. IEEE Press, 1990.

[17] A. Sabelfeld and A. C. Myers. Language-Based Information-Flow Security. *IEEE J. on Selected Areas in Comm.*, 21(1):5–19, 2003.

[18] P. Samarati, E. Bertino, A. Ciampichetti, and S. Jajodia. Information flow control in object-oriented systems. *TKDE*, 9(4):524–538, 1997.

[19] P. Samarati and S. D. C. di Vimercati. Access Control: Policies, Models, and Mechanisms. In *FOSAD 2001/2002*, *LNCS 2946*, pages 137–196. Springer, 2001.

[20] R. Sandhu and P. Samarati. Authentication, access control, and audit. *CSUR*, 28(1):241–243, 1996.

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

[22] K. Seamons, M. Winslett, and T. Yu. Limiting the Disclosure of Access Control Policies during Automated Trust Negotiation. In *Proc. of NDSS'01*, pages 109–125. IEEE Press, 2001.

[23] K. E. Seamons, M. Winslett, T. Yu, L. Yu, and R. Jarvis. Protecting Privacy during On-line Trust Negotiation. In *Proc. of PET'02*, *LNCS 2482*, pages 129–143. Springer, 2002.

[24] A. Stoughton. Access flow: A protection model which integrates access control and information flow. In *Proc. of Symp. on Sec. and Privacy*, pages 9–18. IEEE Press, 1981.

[25] T. Syrjänen. *Lparse 1.0: User's Manual*. Helsinki University of Technology, 2000.

APPENDIX

A. CREATING OBJECTS IN FAF

The architecture of FAF consists of some components: a *history table*, an *authorization table*, *propagation policies*, *conflict resolution policies*, *decision policies*, and a set of *integrity constraints*. Each of these module is mapped into a stratum in the stratified logic program that implements the authorization system. The materialized view of each module is the input for the next module. We refer to [12] for full details.

This solution works correctly when authorizations refer only to primitive objects, that is, objects directly stored from users, but it is not able to fully enforce access control policies when derived objects are dynamically introduced into the system. One of the problem is when derived objects should be introduced and then when authorizations on these objects should be determined. Currently, if we introduce derived objects before applying propagation policies, they could not be created since some authorizations required by the information system in order to allow subjects to process data are not yet computed. Otherwise, if we introduce them after applying propagation policies, we cannot propagate authorizations on derived objects. In the remainder, we give a brief overview of the process to enforce access control policies when derived object are considered, while a complete description and proofs are left for future work.

The basic idea is to iterate FAF approach for $n+1$ times where n is the greatest depth of derivation trees. In the first step, only primitive objects and objects derived by using non parametric functions are considered. Authorizations on these objects are propagated, possible conflicts are resolved, and decisions are taken. If authorizations complies with integrity constraints, the output of the first step is used as input for the second iteration where objects derived by one derivation step are considered. Then, the process is reiterated until all derived objects are considered.

Rules to create derived objects (Section 4.1) are applied in stratum 0, and authorization rules (Section 3.2) in stratum 1 (see [12] for a description of FAF strata). In particular, rules for creating object have the form

$$\mathsf{isobject}(o) \leftarrow L_1 \ \& \ \cdots \ \& \ L_n.$$

where o is an elements of AO, and $L_1, \ldots, L_n$ are done, do, hie-, or rel-literals, and primitive constraints. do literals are permitted only when o is a derived object and they refer only to the objects used to derive it. Rules for deriving access control policies (authorization rules) have the form

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

where o, s and a are elements of AO, AS and A respectively, and $L_1, \ldots, L_n$ are done, do, hie-, or rel-literals, and primitive constraints. do literals are permitted only when o is a derived object and they refer only to the objects used to derive it. Notice that we use do literals since they represent effective authorizations on object. Actually, they are computed after applying conflicts resolution and decision policies. In stratum 1, user preferences are also determined for derived objects. Rules for user preferences are similar to authorization rules. Then, access control policies are compared with user preferences, and so zombie objects are identified.

B. PROOFS OF CLAIMS IN SECTION 5

LEMMA 1. *Let o be an object whose derivation tree contains only disclosure steps and $o_1, \ldots, o_m$ be the primitive objects used to derive it. Then, for each primitive object o_i, $policy_{\geq}(o_i) \subseteq policy_{\geq}(o)$ and $policy_{\leq}(o) \subseteq policy_{\leq}(o_i)$.*

Proof: Let Δ be the derivation tree of object o with depth p. The proof is by induction on derivation tree depth.

base case: If Δ has depth equal to 1, it is immediate to verify that $policy_{\geq}(o_i) \subseteq policy_{\geq}(o)$ by the definition of at least policy for disclosure function and $policy_{\leq}(o) \subseteq policy_{\leq}(o_i)$ by the definition of at most policy for disclosure function.

ind. case: Suppose that derivation tree Δ has depth $p > 1$. Thus, Δ is constituted by subtrees which have smaller depth. By induction, the at least (at most) policy associated with each object o_j that is root of a subtree is such that $policy_{\geq}(o_{p_i}) \subseteq policy_{\geq}(o_j)$ ($policy_{\leq}(o_j) \subseteq policy_{\leq}(o_{p_i})$) where $i \in [1, \ldots, m]$ and $o_{p_1}, \ldots, o_{p_m}$ are the primitive objects used to derive o_j. The at least (at most) policy associated with object o is such that $policy_{\geq}(o_{p_i}) \subseteq policy_{\geq}(o)$ ($policy_{\leq}(o) \subseteq policy_{\leq}(o_{p_i})$) since, for each object o_{j_k} used to derived o, $policy_{\geq}(o) = \bigcup_{k\in[1,\ldots,n]} policy_{\geq}(o_{j_k}) \supseteq policy_{\geq}(o_{j_k})$ ($policy_{\leq}(o) = \bigcap_{k\in[1,\ldots,n]} policy_{\leq}(o_{j_k}) \subseteq policy_{\leq}(o_{j_k})$). □

Proof of Theorem 1: It is immediate by Def. 10 and Lemma 1. □

Proof of Theorem 2: Let Δ be the derivation tree of object o with depth p. The proof is by induction on derivation tree depth.

base case: If Δ has depth equal to 1, it is immediate that $policy(o_i) \subseteq policy(o)$ by the definition of access control policy.

ind. case: Suppose that Δ has depth $p > 1$. Thus, Δ is constituted by subtrees which have smaller depth. By induction, the policy associated with each object o_j that is root of a subtree is such that $policy(o_{p_i}) \subseteq policy(o_j)$ with $i \in [1, \ldots, m]$ and $o_{p_1}, \ldots, o_{p_m}$ the primitive objects used to derive o_j. The policy associated with object o is such that $policy(o_{p_i}) \subseteq policy(o)$ since, for each object o_{j_k} used to derived o, $policy(o) = \left(\bigcup_{k\in[1,\ldots,n]} policy(o_{j_k})\right) \cup \mathcal{P} \supseteq policy(o_{j_k})$ where $\mathcal{P}$ is the policy used to grant additional right. □

Proof of Theorem 3: Let Δ be the derivation tree of object o with depth p. The proof is by induction on derivation tree depth.

base case: If Δ has depth equal to 1, it is immediate to verify that o is a valid object by Def. 10, the definitions of access control, at least and at most policy.

ind. case: Suppose that Δ has depth $p > 1$. Thus, Δ is constituted by subtrees which have smaller depth. By induction, each object o_j that is root of a subtree is a valid object, that is, $policy_{\geq}(o_j) \subseteq policy(o_j) \subseteq policy_{\leq}(o_j)$. Further, $policy_{\geq}(o) \subseteq policy_{\geq}(o_j)$ by the definitions of at least policy, and $policy(o_j) \subseteq policy(o)$ by the definition of access control policy. Then, $policy_{\geq}(o) \subseteq policy(o)$. On the other side, $policy(o) = \bigcup_{k\in[1,\ldots,n]} policy(o_{j_k})$ and $policy_{\leq}(o) = \bigcup_{k\in[1,\ldots,n]} policy_{\leq}(o_{j_k})$. By induction, for every object o_{j_k} the relation $policy(o_{j_k}) \subseteq policy_{\leq}(o_{j_k})$ holds, then $policy(o) \subseteq policy_{\leq}(o)$. □