On Properties of Policy-Based Specifications*

Andrea Margheri
Università degli Studi di Firenze
andrea.margheri@unifi.it
Università di Pisa
margheri@di.unipi.it

Rosario Pugliese
Università degli Studi di Firenze
rosario.pugliese@unifi.it

Francesco Tiezzi
Università di Camerino
francesco.tiezzi@unicam.it

The advent of large-scale, complex computing systems has dramatically increased the difficulties of securing accesses to systems' resources. To ensure confidentiality and integrity, the exploitation of access control mechanisms has thus become a crucial issue in the design of modern computing systems. Among the different access control approaches proposed in the last decades, the policy-based one permits to capture, by resorting to the concept of attribute, all systems' security-relevant information and to be, at the same time, sufficiently flexible and expressive to represent the other approaches. In this paper, we move a step further to understand the effectiveness of policy-based specifications by studying how they permit to enforce traditional security properties. To support system designers in developing and maintaining policy-based specifications, we formalise also some relevant properties regarding the structure of policies. By means of some concrete examples, we present real instances of such properties and outline an approach towards their automatised verification.

1 Introduction

The ever increasing diffusion of the Internet and the Web has fostered the development of large-scale, complex computing systems. These modern distributed systems, that are pervading our everyday life, produce and exploit a huge amount of data that are readily available through the underlying network platforms. Given their importance and societal impact, it is of paramount importance to ensure that data is accessed in a controlled way and that these systems behave in a secure way, e.g. not to compromise sensitive data. For achieving this objective, some major challenges come from the fact that the operating environment is highly dynamic and open, the involved entities are heterogeneous and possibly untrusted, the interactions are complex and unpredictable, and the control is distributed.

In this setting, we believe that policy-based specifications can be used to regulate the behaviour of entities relatively to the access to shared resources, thus ensuring systems *security*. *Policies*, that is sets of declarative rules expressing what can(not) be done in a system, are indeed high-level abstractions that can be used to define various aspects of systems behaviour. In particular, a *security policy* is a statement that defines in which states a system is considered secure. A system is *secure* if starting from a secure state it cannot enter a nonsecure one while computation progresses. The security of a state depends on the behaviours the system exposes and, hence, on which guarantees a security policy managing and controlling the system ensures. The enforcement of such a policy relies on a combination of various approaches, ranging, e.g., from cryptography to access control, according to features and specificity of the controlled system.

We focus on *access control*, usually considered the first line of defence in protection of computer systems, networks, and information. Access control is a broad field that covers several different approaches

*This work has been partially sponsored by the Italian MIUR PRIN project CINA (2010LHT4KM).

Submitted to:
WWV 2015

that enable the protection of systems by restricting physical and logical access rights of (authenticated) subjects to shared resources. In practice, these approaches establish if a subject's request to access a resource should be permitted or denied according to some given access control rules.

Since their original introduction in the context of operating systems, to the more recently conceived ones for modern distributed applications, many approaches for access control have been proposed in the literature. Traditional approaches are based on the identity of the subject, either directly – e.g., Access Control Matrix [18, 13] and its variants Capability Lists and Access Control Lists – or through predefined attributes, such as roles or groups assigned to that subject – e.g., Role-Based Access Control (RBAC [10]). In our frame of reference, these approaches are cumbersome to manage and not sufficiently expressive, given the need to associate access rights to the requester qualifiers of identity, groups, and roles that can change frequently and could not be known in advance. To overcome scalability problems of these traditional access control approaches, an alternative is to use Attribute-Based Access Control (ABAC [23]). Here, the authorisation decision is based on *attributes*, which represent arbitrary information exposed by the system, subject, action, object, or the authorisation context itself that is relevant to the rules at hand. Thus, ABAC permits defining fine-grained, flexible and context-aware access control policies, and fosters systems integration, as attributes can be retrieved from different information systems. Attribute-based access control rules are typically hierarchically structured and paired with strategies for automatic treatment of conflicting decisions and errors. These structured specifications are called *policies*; from this name derives the terminology Policy-Based Access Control (PBAC), sometimes used in the literature in place of ABAC.

Approaches to access control can be classified also with respect to other features. For example, if we consider resource ownership then we can distinguish between *discretionary* access control (DAC), where subjects may decide who can access their own resources, i.e. the access control is at the discretion of the owner, and *mandatory* access control (MAC), where the system decides who is allowed to access any resource. In this respect, the access control matrix better fits the DAC approach, while RBAC and ABAC can be used both for the MAC and DAC approaches. To conclude this overview of the relevant access control approaches, on the base of the results in [16], we can say that ABAC is sufficiently expressive to represent in an uniform way all the other approaches.

Controlling accesses to system resources concerns the three main security principles of *confidentiality*, *integrity*, and *availability*. Specifically, confidentiality refers to the assurance on non-disclosure of sensitive resources to unauthorised subjects; integrity to the protection of resources from being altered by unauthorised subjects; and availability to the enablement of the effective use of resources by authorised subjects. As instantiations of these general principles, many security properties have been introduced and studied (e.g., the Bell-LaPadula [3] and Biba [5] models). However, enforcing such properties by means of access control policies is a tricky task. In fact, the hierarchical structure of policies, the presence of conflict resolution strategies and the intricacies deriving from the many involved controls do not permit to easily check whether a given security property is properly enforced. Therefore, in our work we consider a general instance of the ABAC approach, i.e. the FACPL language [20], and study in details a set of relevant security properties, presenting how they can be rendered in terms of policy-based specifications.

Policy-based specifications are formed by multiple rules and policies, and to characterise the relationships with the behaviours they enforce, various properties on the structure of policies have been proposed in the literature (e.g., change-impact analysis [11] and redundancy minimisation [14]). The approaches used for defining and verifying these properties are different and cannot be uniformly represented. Therefore, in our work we focus on a set of relevant structural properties and propose a uniform formalisation in terms of the FACPL semantics.

Furthermore, for providing a concrete support to the verification of both security and structural prop-

Table 1: Syntax of a light version of FACPL

Policy Decision Points	PDP ::=	$\{Alg$ policies : $Policy^+\}$
Combining algorithms	Alg ::=	permit-overrides | deny-overrides | deny-unless-permit | permit-unless-deny | first-applicable | only-one-applicable | weak-consensus | strong-consensus
Policies	$Policy$ ::=	$(Effect$ target : $Expr)$ | $\{Alg$ target : $Expr$ policies : $Policy^+\}$
Effects	$Effect$ ::=	permit | deny
Expressions	$Expr$ ::=	$Name$ | $Value$ | and$(Expr, Expr)$ | or$(Expr, Expr)$ | not$(Expr)$ | equal$(Expr, Expr)$ | in$(Expr, Expr)$ | greater-than$(Expr, Expr)$ | add$(Expr, Expr)$ | subtract$(Expr, Expr)$ | divide$(Expr, Expr)$ | multiply$(Expr, Expr)$
Attribute Names	$Name$ ::=	$Identifier/Identifier$
Literal Values	$Value$ ::=	true | false | $Double$ | $String$ | $Date$
Access Requests	$Request$ ::=	$(Name, Value)^+$

erties, we outline a constraint-based approach enabling automated verification by means of constraint solver software tools. At the time of writing, this constraint-based analysis of policies is under development; thus, in this paper, we just present main features and strengths of the approach.

The rest of the paper is organised as follows. Section 2 briefly reports main features of policy-based languages and introduces the FACPL policy language. Section 3 presents the representation in terms of policy-based specifications and the formalisation of a set of security properties, while Section 4 addresses policies' structural properties. The verification approach, together with our proposal towards an automated tool support, is sketched in Section 5. Finally, Section 6 reviews more strictly related work and Section 7 concludes by touching upon directions for future work. Background definitions and concepts on computer security used in rest of the paper are based on the well-known text books [6, 12].

2 A Policy Language

Policy languages for access control provide high-level abstractions for the specifications of declarative sets of access control rules. Specifically, these languages allow systems' designers to express structured sets of attribute-based positive (resp. negative) rules granting (resp. forbidding) the access to systems' resources. In this section, by informally introducing (a light version of) the policy language FACPL [20], we detail all the typical features of access control specifications. The authorisation process that is pursued to authorise or forbid an access request is outlined by means of a simple example.

2.1 Syntax and Informal Semantics of FACPL

The syntax of a light version of FACPL is reported in Table 1. It is given trough EBNF-like grammars, where as usual the symbol ? indicates optional items and + indicates non-empty sequences of items.

The top-level term is a *PDP*, which is defined by a sequence of policies $Policy^+$ and an algorithm *Alg* for combining the results of the evaluation of these policies.

A *policy* can be a basic authorization *rule* (*Effect* `target` : *Expr*) or a *policy set* {*Alg* `target` : *Expr* `policies` : *Policy*$^+$} collecting rules and (other) policy sets, so that it is possible to define hierarchical policies. A rule specifies an *Effect*, i.e. permit or deny, indicating the rule-writer's intended consequence of a successful evaluation for the rule, and a *target*, i.e. an expression *Expr* defining the applicability of the rule to a request. A *policy* instead specifies a target, a sequence of contained elements, i.e. rules or policies themselves, and an algorithm *Alg* for combining the results of the evaluation of these contained elements.

A *combining algorithm* implements a strategy for resolving conflicts among the decisions resulting from the evaluation of a collection of rules/policies, e.g. whenever both decisions permit and deny are returned. We report below the strategies implemented by some of these algorithms.

- permit-overrides: if the processing of an element returns permit, then the result is permit, i.e., permit takes precedence over any other decision. Instead, if at least one element returns deny and all others return not-applicable or deny, then the result is deny. If all elements return not-applicable, then the result is not-applicable. In the remaining cases, the result is indeterminate.
- deny-unless-permit: similarly to permit-overrides, permit takes precedence over deny, but it never returns not-applicable or indeterminate, which are instead evaluated as deny.
- strong-consensus: it returns permit (resp., deny) only if all elements return permit (resp., deny). If all elements return not-applicable then the result is not-applicable. Otherwise, it returns indeterminate.

A *target* is an expression indicating the access requests to which a policy applies. *Expressions* are built from attribute names and *literal* values, i.e. booleans, doubles, strings, and dates, by using standard operators. As usual, string values are written as sequences of characters delimited by double quotes. For simplicity sake, the expressions syntax does not take types explicitly into account; however, the semantics of expressions returns an error if the arguments of operations have incorrect types. The latter can be anyway managed by policies by resorting to appropriate combining algorithms.

An *attribute name* indicates the value of an attribute within an access request to authorise. Attributes are expressed in terms of pairs name-value, where names are structured in the form cat/att, with cat standing for a category name (as, e.g., subject, resource, action) and att for a specific name (as, e.g., id and role). For example, the structured name subject/role represents the value of the attribute role within the category subject.

An *access request* represents a subject willing to execute an action that has to be authorised. This request holds all the attributes relevant for taking the authorisation decision, such as the information of the subject originating the request and that of the requested action. A request sample of a subject *sub*, which is assigned the role *role*1, that wants to *read* a resource *res* follows:

$$(\mathsf{subject/id}, sub)\ \ (\mathsf{subject/role}, role1)\ \ (\mathsf{resource/id}, res)\ \ (\mathsf{action/id}, read)$$

The evaluation of a request with respect to a policy results in one decision among permit, deny, not-applicable, and indeterminate. The meaning of the first two decisions is obvious (i.e., granting and forbidding the access, respectively), while the third means that there is no policy that applies to the request and the fourth means that some errors occur in the evaluation.

By way of example, to regulate accesses to a resource *res*, we might use the following policy

{deny-unless-permit
 `target` : equal(resource/id, *res*)
 `policies` : (permit `target` : equal(action/id, *read*) and equal(subject/role, *role*1))}

The evaluation of the previous request with respect to the policy above starts by evaluating the policy's target, i.e. the boolean expression after the first keyword `target`. Since the request satisfies the equal comparison function, the evaluation carries on with the enclosed rule. The rule's target is satisfied as well and the decision permit is returned. Then, the combining algorithm applies to the resulting set of decisions, which in this case only contains the permit one, and returns the final decision for the policy, i.e. permit. Notice that the policy does not authorize requests not exposing the value *role*1 as a role and that the remaining requests are granted only if they ask for *read* operations. Also notice that should the policy's target not apply to a request, then not-applicable would be immediately returned without evaluating the enclosed rule and applying the combining algorithm.

2.2 A glimpse of the FACPL Formal Semantics

In this section, we briefly outline the formal semantics of FACPL (we refer the interested reader to [20] for a full account). The semantics is defined by following a denotational approach, which means that

- we introduce some semantic functions mapping each FACPL syntactic construct to an appropriate *denotation*, that is an element of a semantic domain representing the meaning of the construct;
- the semantic functions are defined in a *compositional* way, so that the semantic of each construct is formulated as a function of the semantics of its immediate sub-constructs.

To this purpose, for each FACPL syntactic category, we specify the semantic domain into which the syntactic constructs map and define the semantic function $[\![\ ]\!]$ by giving its domain and codomain, and by using semantic clauses to specify, inductively on the syntactic constructs, how the function acts on each construct. Thus, if P stands for a FACPL policy, $[\![P]\!]r$ corresponds to the decision resulting from the application of the semantic function to (the syntactic object) P and (the semantic object) r representing an access request.

A FACPL request, in order to be evaluated, is represented in its functional form. This is a function r belonging to the set $R \triangleq \mathit{Name} \to (\mathit{Value} \cup 2^{\mathit{Value}} \cup \{\bot\})$ containing all those total functions mapping attribute names (i.e., the structured names in the syntactic domain *Name*) to either values, or set of values, or the special value $\bot$ (modelling the fact that an attribute name is missing).

The semantics of a policy is then a function that, given a request, returns an authorisation decision. Formally, it is a function of the form $R \to \mathit{Decision}$, where *Decision* corresponds to the semantic (and syntactic) domain of authorisation decisions. To define the semantics of policies we use two clauses, one deals with rules, the other one with policies. For a generic rule $(e\ \texttt{target}:\mathit{expr})$, its semantics is given by the following clause:

$$[\![(e\ \texttt{target}:\mathit{expr})]\!]r = \begin{cases} e & \texttt{if } [\![\mathit{expr}]\!]r = \mathsf{true} \\ \mathsf{not\text{-}applicable} & \texttt{if } [\![\mathit{expr}]\!]r = \mathsf{false} \ \vee\ [\![\mathit{expr}]\!]r = \bot \\ \mathsf{indeterminate} & \texttt{otherwise} \end{cases}$$

where $[\![\mathit{expr}]\!]r$ is the value returned by evaluating the target expression *expr* with respect to the request r. Thus, the rule's decision is returned when the target evaluates to true, which means that the rule applies to the request. Otherwise, it could be the case that the rule does not apply to the request, i.e. when the target evaluates to false or to $\bot$ (which means that the target is an attribute name missing in the request), or that an error has occurred while evaluating the target.

Since the clause for policies relies on the semantics of combining algorithms, we first introduce it. For each combining algorithm, we use a two-dimensional matrix that, given two decisions, calculates

Table 2: The two-dimensional matrix for the permit-overrides combining algorithm

$d_1 \backslash^{d_2}$	permit	deny	not-applicable	indeterminate
permit	permit	permit	permit	permit
deny	permit	deny	deny	indeterminate
not-applicable	permit	deny	not-applicable	indeterminate
indeterminate	permit	indeterminate	indeterminate	indeterminate

the resulting combined one; then, by means of an iterative application of this matrix, we can define the decision returned by the algorithm when given as input a sequence of decisions (each resulting from the evaluation of a policy or a rule). For example, Table 2 reports the matrix for permit-overrides. Notably, when a matrix takes into account the order of policy decisions (see, e.g., the matrix for first-applicable in [20]), the combination is not associative.

Finally, for a generic policy $\{a \ \ \mathsf{target}: expr \ \ \mathsf{policies}: P^+\}$, where P^+ stands for a non-empty sequence of policies or rules, its semantic clause is

$$[\![\{a \ \mathsf{target}: expr \ \mathsf{policies}: P^+\}]\!]r = \begin{cases} e & \texttt{if } [\![expr]\!]r = \mathsf{true} \ \wedge \ [\![a(P^+)]\!]r = e \\ \mathsf{not\text{-}applicable} & \texttt{if } [\![expr]\!]r = \mathsf{false} \ \vee \ [\![expr]\!]r = \bot \\ & \quad \vee ([\![expr]\!]r = \mathsf{true} \wedge [\![a(P^+)]\!]r = \mathsf{not\text{-}applicable}) \\ \mathsf{indeterminate} & \texttt{otherwise} \end{cases}$$

where $[\![a(P^+)]\!]r$ is the decision returned by evaluating the combining algorithm a on the sequence of (decisions resulting from the evaluation of) policies or rules P^+. Thus, the policy applies to the request when the target evaluates to true and the semantic of the combining algorithm a (which is applied to the enclosed sequence of policies and the request) returns a decision e, i.e. permit or deny. In this case, the resulting decision of the policy is e. Instead, if the target evaluates to false or to $\bot$, or the combining algorithm states that the contained sequence of policies is not applicable, the policy does not apply to the request. In the remaining cases, an error has occurred and the decision is indeterminate.

3 Security Properties for Polices

Policy-based specifications are sufficiently flexible and expressive to permit addressing, even in a mixed-up way, different security aspects. As stated in the Introduction, verifying whether a policy enforces a given security property is not straightforward. Therefore, in this section, we first present the attribute-based controls that the policy-based specifications must contain for ensuring various security properties. Then, we exploit the semantics of policies to formalise under which conditions a policy properly enforces such properties.

We start by providing a more precise definition of the three general security principles mentioned in the Introduction. Given a controlled system, we let $res \in Res$, $Sub' \subseteq Sub$ and $Act' \subseteq Act$, where Res, Sub and Act are respectively the set of resources, subjects and actions involved in the system's operation. Then, the three principles can be defined as follows:

- *confidentiality*: the resource res has the property of *confidentiality* with respect to subjects Sub' and actions Act' if none of the subjects in Sub' can execute actions in Act' on res;

- *integrity*: the resource *res* has the property of *integrity* with respect to subjects Sub' and actions Act' if actions in Act' executed by subjects in Sub' cannot alter the trustworthiness of *res*;
- *availability*: the resource *res* has the property of *availability* with respect to subjects Sub' and actions Act' if all subjects in Sub' can execute all actions in Act' on *res*.

It is worth noticing that the above principles could be naively instantiated by resorting to checks on the identity of subjects. For example, when a subject whose identifier is s tries to access the resource *res*, confidentiality could be achieved by denying the access to s if $s \in Sub'$. However, this requires to know the identity of the requestor, as well as of all the other forbidden subjects, in advance. To overcome this limitation, different instantiations of these principles have been proposed, which rely on the features of subjects and resources for characterising the set Sub' of (dis)allowed subjects. We present some of these instantiations below, by focussing on the attribute-based controls necessary for expressing the wanted features and checking specific security aspects.

Notably, from the access control point of view, the availability principle implies that the policy-based specifications have to grant the access to a subject that exhibits all the required credentials. This goal is achieved "by construction" in the proposed instantiations of the confidentiality and integrity principles, hence we do not further insist on the availability principle.

3.1 Attribute-based Characterisation

We use attribute names of the form $\mathsf{subject}/*$, $\mathsf{actions}/*$ and $\mathsf{resource}/*$ to identify the characteristics of a *subject* willing to perform a given *action* on a *resource*. For example, for a given access tentative, $\mathsf{action}/\mathsf{id}$ returns the identifier of the requested action, like e.g. *read* or *write*.

In the following attribute-based characterisation of the security properties, we rely on the commonly used *close-world* assumption [27] of access control systems, which forbids all behaviours that are not explicitly granted. We show in Section 5 how this assumption can be enforced using FACPL policies.

Confidentiality: multi-level security. The security policies commonly referred to as *multi-level security* [3, 25] represent typical instantiations of the confidentiality principle and are also the formal basis of the MAC approach. The goal of these kinds of policies is to prevent that a resource with a certain confidentiality level be disclosed to a subject with a lower level. To this aim, each subject and resource is assigned, through a function f_L, a confidentiality level from a given partially ordered set $< L, \leq_L >$ of levels.

The Bell-LaPadula model [3] formalises these security policies in terms of some security properties that must hold with respect to *read* and *write* actions. These properties are defined as follows:

- *no read-up*: a subject s can read a resource *res* only if the the security level of the subject dominates the one of the object, i.e. $f_L(res) \leq_L f_L(s)$;
- *no write-down*: a subject s can write a resource *res* only if the level of the subject s is dominated by the level of the object, i.e. $f_L(s) \leq_L f_L(res)$.

If we let the attributes $\mathsf{subject}/\mathsf{level}$ and $\mathsf{resource}/\mathsf{level}$ denote the confidentiality level assigned by function f_L to subjects and resources, respectively, then the previous properties can be characterised in terms of policy-based specifications by the following rules:

$$\begin{array}{l}(\mathsf{permit\ \ target} : \mathsf{equal}(\mathsf{action}/\mathsf{id}, \mathit{read})\ \mathsf{and}\ \mathsf{leq}(\mathsf{resource}/\mathsf{level}, \mathsf{subject}/\mathsf{level}))\\ (\mathsf{permit\ \ target} : \mathsf{equal}(\mathsf{action}/\mathsf{id}, \mathit{write})\ \mathsf{and}\ \mathsf{leq}(\mathsf{subject}/\mathsf{level}, \mathsf{resource}/\mathsf{level}))\end{array} \quad (1)$$

where function leq corresponds to the partial order relation $\leq_L$.

The Bell-LaPadula model is usually extended to also consider DAC controls. For instance, if we use access control lists as a DAC approach, these controls could be rendered by the following rule:

$$(\mathsf{permit\ target} : \mathsf{equal}(\mathsf{action/id}, \mathit{read})\ \mathsf{and}\ \mathsf{in}(\mathsf{subject/id}, \mathsf{resource/read.ids})) \tag{2}$$

where we assume that the attribute resource/read.ids returns the set of all subjects allowed to execute the read action on the resource.

Integrity: separation of duty. The integrity principle regards various system aspects in addition to accesses authorisation, like e.g. the trustworthiness of conveyance and storage means used by the system to keep resources. Since we only focus on access control, we instantiate the principle in terms of the Biba model [5] and the property of separation of duty.

The Biba model formalises integrity with respect to execution of read and write actions in terms of integrity levels associated to subjects and resources. Assuming that the integrity levels are defined in the same way as the confidentiality ones, the Biba model is the 'dual' of the Bell-LaPadula one in that it relies on the *no read-down* and *no write-up* properties, which can be characterised as before.

An additional property that instantiates the integrity principle is *separation of duty* (SoD), which was introduced in the Clark-Wilson model [8] and since then has been largely adopted to define secure systems. In general, this property ensures that if two or more actions are required to perform a critical transaction, then these actions must be performed by at least two different subjects. SoD is valuable in deterring fraudulent behaviours, since no single subject has the possibility to perform complex actions, but only well-defined, elementary actions.

A basic example of SoD is to prevent that an action be executed when a subject is assigned two roles that are conflicting, i.e. there is no separation of duties among the actions that these roles permit. For instance, if we assume roles $\mathit{role}1$ and $\mathit{role}2$ to be in conflict, we can define a rule that permits a read action only when a subject exposes the first role but not the second one; the rule is as follows

$$\begin{aligned}(\mathsf{permit\ target} : \mathsf{equal}(\mathsf{action/id}, \mathit{read})\ \mathsf{and}\ \\ \mathsf{in}(\mathit{role}1, \mathsf{subject/role})\ \mathsf{and}\ \mathsf{not}(\mathsf{in}(\mathit{role}2, \mathsf{subject/role})))\end{aligned} \tag{3}$$

Indeed, the rule checks that the roles assigned to the subject, that are obtained through the attribute subject/role, include $\mathit{role}1$, which is required for executing the read action, and not $\mathit{role}2$.

This last property is an example of *static* SoD, i.e. an integrity requirement that can be fulfilled by evaluating a single access request. However, if we define SoD in terms of conflicting actions, rather than in terms of conflicting roles as done before, checking a single access request is not adequate anymore to enforce the intended property. Indeed, SoD could be easily circumvented by executing conflicting actions in two or more attempts, as it is the case of a subject that is assigned, in two different instants of time, different roles granting conflicting actions. To avoid that the subject be authorised to execute both actions, we must resort to considering the previous actions it has performed, which is an example of *dynamic* SoD. This kind of properties can be still addressed by using policy-based specifications, but we need to use attributes for storing the history of the accesses previously performed by a subject. We will provide further details on this aspect when discussing future work.

Role-based design: hybrid properties and least-privilege. The role-based design is a high level approach that permits to enforce confidentiality and integrity properties on the controlled resources at the same time. It consists in assigning different roles to subjects within the system and using policies stating what accesses are allowed to subjects depending on the roles they have. Although it is not an instantiation of one of the three general security principles, we consider this approach explicitly since it is largely used due to its better scalability with respect to other models, like e.g. the Bell-LaPadula and Biba ones.

The basic characterisation of role-based controls in terms of policy-based specifications is straightforward: the attribute subject/role permits to define controls on the subject's roles and Rule (3) is a concrete example of this. However, the role-based approach takes also different, more complicated forms [26], that exploit role hierarchies, i.e. a role inherits the privileges of the roles that are higher up in the hierarchy, or constraints on role assignments, i.e. two conflicting roles cannot be assigned at the same time. The former case can be rendered by exploiting the hierarchy of polices or an appropriate ordering function, while the latter one by using an approach similar to that used for SoD properties. The characterisation of role-based controls thus formalises a sort of 'hybrid' property, consisting of both confidentiality and integrity aspects.

Let us consider an hybrid property stating that an action *write* can be executed by all the subjects with assigned role *role*3, or by any other subject in the underlying role hierarchy which is not assigned role *role*4 at the same time. Its characterisation in terms of attribute-based specifications can be defined as follows:

$$\begin{array}{l}(\mathsf{permit\ \ target} : \mathsf{equal}(\mathsf{action/id}, \mathit{write})\ \mathsf{and} \\ \qquad\qquad \mathsf{sub\text{-}role}(\mathsf{subject/role}, \mathit{role}3)\ \mathsf{and}\ \mathsf{not}(\mathsf{in}(\mathit{role}4, \mathsf{subject/role})))\end{array} \tag{4}$$

where we use the ad-hoc function sub-role to check if the subject's role is a sub-role of (or coincides with) *role*3 and the additional control on role *role*4 to encode the integrity check.

A guideline commonly used in role-based design is *least privilege*. It means that each subject should not expose more privileges than those necessary to perform the requested action. Differently from the properties we have previously considered, least privilege is not implemented through specific rules. Rather, it affects the design choices pursued for defining access control policies. We will present more details in the semantic-based formalisation presented in the next section.

3.2 Semantic-Based Formalisation

A policy-based specification, as e.g. a FACPL policy, in addition to the rules previously presented, contains many other elements, such as e.g. other rules implementing additional controls and conflict resolution strategies. We now formalise under which conditions a policy enforces a given security property.

The formal representation of a security property is obtained by exploiting the fact that an access control request is an assignment of values to a collection of attributes. We can then use sets of requests to represent the (non)secure system behaviours with respect to a given property. Formally, given a security property *pr*, we let R_{pr} (resp., $\overline{R}_{pr}$) be the *permit* (resp. *deny*) set, i.e. the set of requests that represent the secure (resp., nonsecure) behaviours with respect to *pr*, and Sub_{pr} (resp., Res_{pr}) be the subset of subjects (resp., resources) for which the property *pr* is defined. A policy *P* containing the rules characterising *pr* correctly enforces such property if the following conditions hold

$$\begin{array}{l}\forall\, r \in R_{pr} :\ r(\mathit{resource/id}) \in \mathit{Res}_{pr}\,,\, r(\mathit{subject/id}) \in \mathit{Sub}_{pr} \Rightarrow\ [\![P]\!]r = \mathsf{permit} \\ \forall\, r \in \overline{R}_{pr} :\ r(\mathit{resource/id}) \in \mathit{Res}_{pr}\,,\, r(\mathit{subject/id}) \in \mathit{Sub}_{pr} \ \Rightarrow\ [\![P]\!]r = \mathsf{deny}\end{array}$$

where notation $r(\mathsf{attr\_name})$ indicates the value assigned to the attribute named attr_name by the request *r*. Hence, we require that all the secure behaviours are allowed (i.e., all the requests in R_{pr} evaluate to permit) and all the nonsecure ones are forbidden (i.e., all the requests in $\overline{R}_{pr}$ evaluate to deny). Notably, we consider the (non)secure behaviours that only refer to the subset of subjects and resources that the property *pr* takes into account. This means that the set $\overline{R}_{pr}$ is not the complementary set of R_{pr} with respect to the universe of all the possible behaviours of the system; rather it represents those behaviours

that are considered nonsecure by the property pr. In the sequel we report the definition of the (non)secure sets of requests for each property we presented before.

Confidentiality: multi-level security. The secure behaviours identified by the *no read-up* property corresponds to the set of requests R_{nru} whose elements r must satisfy the following conditions

$$r(\mathsf{action/id}) = read\ ,\ r(\mathsf{resource/level}) = l_1\ ,\ r(\mathsf{subject/level}) = l_2 \quad : \quad l_1, l_2 \in L\ ,\ \ l_1 \leq_L l_2$$

The set $\overline{R}_{nru}$ instead contains those requests satisfying the following conditions

$$r(\mathsf{action/id}) = read\ ,\ r(\mathsf{resource/level}) = l_1'\ ,\ r(\mathsf{subject/level}) = l_2' \quad : \quad l_1', l_2' \in L\ ,\ \ l_1' \not\leq_L l_2'$$

The permit and deny sets for the *no write-down* property are similarly defined. In case of DAC properties as e.g. that defined by Rule (2), the requests of the set R_{dac} are characterised by the following conditions

$$r(\mathsf{action/id}) = read\ ,\ r(\mathsf{resource/read.ids}) = Sub_{res}\ ,\ r(\mathsf{subject/id}) = s \quad : \quad s \in Sub_{res}$$

where Sub_{res} is set of all subjects allowed to execute the read action on the resource res. Instead, the elements of the deny set $\overline{R}_{dac}$ must satisfy the following conditions

$$r(\mathsf{action/id}) = read\ ,\ r(\mathsf{resource/read.ids}) = Sub'_{res}\ ,\ r(\mathsf{subject/id}) = s \quad : \quad s \notin Sub'_{res}$$

Indeed, the set of granted subjects Sub'_{res} does not contain the subject s.

Integrity: separation of duty. The *no read-down* and the *no write-up* properties, representing the Biba model, are formalised like the confidentiality ones.

Let us consider the SoD property for a read action expressed by Rule (3). Thus, if Rol is the set of authorised non conflicting sets of roles, i.e. all the sets contain $role1$ and not $role2$, the secure behaviours R_{sod} are defined as follows

$$r(\mathsf{action/id}) = read\ ,\ r(\mathsf{subject/role}) = rol \quad : \quad rol \in Rol$$

The non secure behaviours R'_{sod} are instead defined as follows

$$r(\mathsf{action/id}) = read\ ,\ r(\mathsf{subject/role}) = rol' \quad : \quad rol' \in Rol_{all} \backslash Rol$$

where the set Rol_{all} represents the set of all sets of roles that a subject can play in the system. Thus, a request is non secure when the set of subject's roles does not contain $role1$, i.e. the subject has not the right to execute the *read* action, or it contains $role1$ and $role2$ at the same time, i.e. the exposed roles are in conflict.

Role-based design: hybrid properties and least-privilege. The secure and nonsecure behaviours identified by hybrid properties are just a combination of the previous examples. The formalisation of the least privilege requires instead additional comments.

Let us consider a security property pr and the set of request R_{pr} representing the secure behaviours with respect to such property. The sets of secure and nonsecure behaviours for the least privilege, with respect to pr, are defined as follows

$$R_{lp} = R_{pr} \qquad \overline{R}_{lp} = R_{all} \backslash R_{pr}$$

where R_{all} indicates all the possible requests. Therefore, in order to enforce the least privilege, a policy has to authorise all those behaviours of the system deemed as secure by the property pr and to forbid all the other behaviours, not only those violating pr as in the previous cases. All the behaviours that are not defined secure by pr are considered as nonsecure. Hence, forbidding them ensures that possibly granted accesses cannot be used to circumvent, in a malicious way, other policies in the system.

4 Structural Properties for Policies

We now address some of the properties proposed in the literature which refer to the structure of policies. We start considering completeness of a single policy, after which we will consider redundancy, disjointness and coverage of one policy with respect to other ones. The properties dealing with multiple policies capture the relationships among the different sets of system behaviours they enforce. In this section, we report a uniform characterisation of these properties by means of the semantic-based approach used before.

By referring to FACPL, we use P to range over policies, alg to range over combining algorithms and d to range over authorisation decisions. Moreover, we use R_{all} to denote the set of all possible requests.

Completeness. A policy P is *complete* if there is no access request for which there is an absence of decision. Formally, this property can be rendered through the following condition

$$\forall\, r \in R_{all} : [\![P]\!]r \neq \mathsf{not\text{-}applicable}$$

In fact, we require that the policy applies to any request, i.e. it always returns a decision different from $\mathsf{not\text{-}applicable}$. Notably, in this formulation $\mathsf{indeterminate}$ is considered as an admissible decision; a more restrictive formulation could be defined that only accepts decisions permit and deny.

Redundancy. Redundancy among policies means that to enforce the same set of system behaviours some policies are not needed. Therefore, if we eliminate redundant policies, we can improve performance of policy evaluation while leaving unchanged the enforced behaviours. Although the concept seems natural and quite simple, different formalisations, that often lack of precision, have been proposed in the literature. We follow an approach similar to [14].

Formally, if we let the FACPL policy S be defined as $S = \mathsf{alg}(P_1, \ldots, P_i, P, P_{i+1}, \ldots, P_n)$, then the policy P is *redundant* with respect to S if the following condition holds

$$\forall\, r \in R_{all} : [\![\mathsf{alg}(P_1, \ldots, P_i, P, P_{i+1}, \ldots, P_n)]\!]r \;=\; [\![\mathsf{alg}(P_1, \ldots, P_i, P_{i+1}, \ldots, P_n)]\!]r$$

In fact, we require that, for any request, the decision returned by S is not affected by the presence of P. Notably, this property generalises in the obvious way to the case S contains rules instead of policies (thus P would be a redundant rule) and to the case a target is present in S.

Disjointness. Disjointness among policies means that such policies apply to disjoint sets of behaviours. Thus, two policies are *disjoint* if there is no request for which both policies evaluate to permit or deny. Formally, policies P and P' are *disjoint* if the following condition holds

$$\forall\, r \in R_{all} \;\; : \;\; \{\, [\![P]\!]r,\; [\![P']\!]r \,\} \not\subseteq \{\mathsf{permit}, \mathsf{deny}\}$$

It is worth noticing that disjoint polices can be combined with the assurance that the allowed or forbidden behaviours enforced by each of them are not in conflict, which simplifies the choice of the combining algorithm to be used.

Coverage. Coverage among policies means that one of such policies enforce the same decisions as the other ones for a set of requests of interest. Formally, if R_{cov} is a set of requests, we say that the policy P *covers* the policy P' if, for each request $r \in R_{cov}$ to which P' applies, i.e. $[\![P']\!]_r \in \{\mathsf{permit}, \mathsf{deny}\}$, P applies too and returns the same decision. Formally, it is expressed by the following condition

$$\forall\, r \in R_{cov} \;\; : \;\; [\![P']\!]r \in \{\mathsf{permit}, \mathsf{deny}\} \;\; \Rightarrow \;\; [\![P]\!]r = [\![P']\!]r$$

Thus, relatively to the set of requests of interest, P enforces at least the same allowed and forbidden behaviours as P'. Consequently, if P' also covers P, then the two policies enforce exactly the same behaviours relatively to the set of requests of interest.

These structural properties permit to statically reason on the relationships among policies and provide useful support to system's designers in developing and maintaining policy-based specifications. One technique they support is the *change-impact analysis* [11]. This analysis examines the effect of policy modifications for discovering unintended consequences of such changes. To be practically effective it requires that the verification of the previous properties be supported by automatic tools. We further deal with this issue in the next section.

5 Verification of Properties

The formalisation of security and structural properties presented in Sections 3 and 4 determines the conditions on attributes stating when a policy enjoys a certain property. To verify such conditions, we need to take into account the various elements composing a policy. Specifically, the hierarchical structure of policies and the various elements originating the decisions make this verification cumbersome and error-prone if not supported by an automatised technique. As an example of the difficulties to address, we consider the case of verifying two security properties: the *no read-up* and DAC ones for a read action by a set of subjects Sub' on a resource *res*. Thus, we define various combination approaches for creating a policy containing Rules (1) and (2), and we study for each approach if the two properties are properly enforced.

The first combination we propose for the two rules is defined as follows

$$
\begin{array}{l}
\{\mathsf{permit\text{-}overrides} \\
\quad \texttt{target} : \mathsf{equal}(\mathsf{resource/id}, \mathit{res}) \text{ and } \mathsf{in}(\mathsf{subject/id}, \mathit{Sub}') \\
\quad \texttt{policies} : \\
\qquad (\mathsf{permit}\ \ \texttt{target} : \mathsf{equal}(\mathsf{action/id}, \mathit{read}) \text{ and } \mathsf{leq}(\mathsf{resource/level}, \mathsf{subject/level})) \\
\qquad (\mathsf{permit}\ \ \texttt{target} : \mathsf{equal}(\mathsf{action/id}, \mathit{read}) \text{ and } \mathsf{in}(\mathsf{subject/id}, \mathsf{resource/read.ids}))\}
\end{array}
$$

The chosen combination algorithm is permit-overrides, which seems the natural choice since each allowed behaviour is explicitly authorised. Notably, the policy's target ensures that the policy exclusively applies to the considered resource *res* and to the subset of system's subjects Sub'.

To verify that this policy enforces the intended properties, we show that all the secure behaviours are authorised, while the nonsecure ones are forbidden. We consider first the *no read-up* property. As formalised in Section 3.2, the secure behaviours correspond to all the requests containing the resource and subject levels that respect the partial ordering relation. These ones clearly match the target of the first rule, hence this rule, as well as the permit-overrides algorithm, return permit. The nonsecure behaviours are instead represented by all the requests containing resource's and subject's levels not properly ordered. In this case, both internal rules do not apply and the permit-overrides algorithm returns not-applicable, because neither permit nor deny are returned by the rules. However, the nonsecure behaviours should be evaluated as deny, hence we can conclude that the policy does not properly enforce the *no read-up* property. The same also holds for the DAC property.

To fix this first policy, we can replace the permit-overrides algorithm by the deny-unless-permit one, which ensures that deny is taken as the default decision whenever no rule evaluates to permit. In this case all the nonsecure behaviours of both properties are properly forbidden. However, as we are addressing two properties, the secure behaviours are all those ones that are secure, at the same time, for

both properties. This means that permit must be returned only when the two rules apply at the same time as well, but this does not happen in the presented policies. In fact, the combining algorithm does not enforce any form of consensus between the two rules. As a matter of fact, a subject can circumvent the access control system reading a resource, e.g., only having the correct confidentiality level and not the discretionary access.

This additional issue can be addressed by adding a new policy layer and requesting a strong consensus between the rules. The extended policy is thus as follows

{deny-unless-permit
 policies :
  {strong-consensus
    target : equal(resource/id, *res*) and in(subject/id, *Sub*′)
    policies :
      (permit target : equal(action/id, *read*) and leq(resource/level, subject/level))
      (permit target : equal(action/id, *read*) and in(subject/id, resource/read.ids)) }}

deny-unless-permit is used at top level to ensure that the resulting decisions of the overall policy will be only permit or deny. In the inner policy, strong-consensus ensures that permit is returned only when both internal rules apply at the same time. In this case, all secure and nonsecure behaviours of the two intended properties are properly enforced. Notably, we can achieve the same result by merging the two rules and avoiding the additional policy layer; however, the modelling approach we present permits to achieve separation of concerns among rules, which are thus easier to maintain and possibly change.

Verifying that a policy properly enforces a set of properties is not straightforward. This example, which seems easy enough for being manually checked, shows us that also in case of simple policies we need an automated verification approach. Specifically, this approach must be capable to take into account all the aspects of a policy specification, e.g. policy stratification and combining algorithms, and to exhaustively check all the significant requests representing the possible behaviours. A viable approach towards an automated verification of security and structural properties is outlined in the next subsection.

5.1 Towards an Automated Verification Approach

Automatising the verification of properties permits to facilitate the analysis of policy-based specifications. To enable such analysis, we need a formalism that, on the one hand, permits to collapse hierarchical policies into a single-layered representation and to uniformly represent all policy elements and, on the other hand, is sufficiently flexible to deal with multiple domain values for attribute assignments. To this aim, we propose a constraint-based formalism.

Constraints permit to specify satisfaction problems based both on boolean formulae and on formulae dealing with different theories as, e.g. linear arithmetics. Such kind of formulae are called *satisfiability modulo theories* (SMT) formulae. Choosing a SMT-based formalism is advocated also by the relevant progress made in the development of automatic SMT solvers (e.g., Z3 [21]), which make SMT formulae to be extensively employed in diverse analysis applications [22]. Of course, the feasibility of the approach crucially depends on decidability of the satisfiability checks; in other words, the used constrains must be represented by decidable theories, as e.g. uninterpreted functions and array theories.

To achieve a single-layered representation of policies, we have to provide a translation function from the language used for writing policies to the constraint-based formalism that preserves the semantics of the original language. Indeed, since FACPL is equipped with a formal semantics, it has to be exploited

for defining a rigorous encoding. Notably, as the evaluation of a policy can return four possible decisions, we have to define a different constraint for each of them.

A constraint-based representation of policy-based specifications enables the verifications of both security and structural properties. Specifically, in the case of security properties, the attribute values identifying the class of (non)secure requests correspond to assignment assertions in the constraint of interest (i.e. the one modelling the decision to which the requests should evaluate) and then, by means of an SMT-solver, it is checked if such constraint is satisfiable. If this happens, it means that the requests of the class can evaluate, under the assignment model returned by the solver, to the decision modelled by the constraint. In case of structural properties, we can instead define boolean combinations among the single constraints of each policy, and then check the satisfiability of the resulting constraint to understand if a certain property holds. For instance, the disjointness between two policies holds if the constraint resulting from the implication of the permit (resp., deny) constraints of both policies is not satisfiable.

6 Related Works

Policy-based specifications have recently been the subject of extensive research, both by industry and academia, in many application areas. In fact, policy languages have been adopted for managing different aspects of systems' behaviour, not only access control but also adaptation enforcement and network management. A large variety of languages for defining access controls has been proposed, and the more significant ones follow two main specification approaches: *rule-based*, as e.g. XACML [24] and Ponder [9], and *logic-based*, as e.g. ASL [15] and the logical framework presented in [1]. We present the relevant features of these languages, showing the effectiveness of choosing FACPL as the target language for studying policies' properties. Notably, the uniform approach based on attributes presented in [16] does not provide any evaluable property characterisation, but only an high-level access control model.

XACML is the most widely-used instantiation of the ABAC approach. It relies on an XML-based syntax and permits to write policies and access requests. However, XML does not permit compact specifications and, due to the lack of a formal semantics, an explicit unambiguous formalisation of request's evaluation. The use of FACPL permits thus to avoid verbose examples, and to rely on a rigours formal semantics to formalise properties.

Ponder is instead a strongly-typed language defined in terms of Event-Condition-Action rules. Differently from XACML and FACPL, it does not provide any explicit combination strategy to resolve conflicts. Thus, the presence of conflicts or inconsistency is statically analysed by means of abductive reasoning techniques [2]. This reasoning generates a refinement for the considered policy. Ponder, on the one hand, permits to avoid policy hierarchies, but, on the other hand, it does not provide any modularity and compositionality in the specification of policies. The FACPL-based specification approach consists instead in basic building rules, that can be appropriately combined to enforce different security properties, ensuring separation of concerns in the enforced behaviours.

The increasing spread of policy-based specifications has prompted the development of multiple verification techniques like, e.g., property checking and behavioural characterisations. Such techniques have been implemented by means of different formalisms, varying from multi-terminal binary decision diagrams (MTBDD) to different kinds of logics. We review the more relevant techniques and formalisms.

The change-impact analysis of XACML policies presented in [11] permits to study the consequences of policy's modifications. In particular, to verify structural properties among policies by means of automatic tools, this approach relies on a MTBDD-based representation of policies. However, it cannot deal

with many of the classical combining algorithms, e.g. all the XACML's ones, and, as outlined in [1], an SMT-based approach (i.e. the one we are exploring), scales significantly better than the MTBDD one.

The ASL language [15] is a logical framework for the formalisation of access control policies. Specifically, it enables hierarchisation, conflict resolution, and role- and group-based definitions of access rights. Furthermore, by means of additional predicates representing a posteriori checks on authorisation decisions, it permits to easily express various history-dependent properties, e.g. dynamic separation of duty. Similarly, the framework in [1] permits a logic-based specification of control policies. A policy is thus a list of constraint assertions that are evaluated by a SMT-based tool, and various structural properties can be encoded in terms of additional, low-level assertions. The FACPL-based approach permits instead to abstract from the underlying logical means (that are still used to in the FACPL formal semantics and for the automatised analysis we foster), allowing a better usability for system's designers of the properties formalisation.

An additional logic-based analysis is the one presented in [17], which aims at verifying structural properties of XACML policies. Specifically, it defines a partial encoding of XACML into description logics and a set of supporting analysis services. However, this approach does not take into account many combining algorithms and, also, the decisions not-applicable and indeterminate, which are instead useful in the definition of structural properties. Furthermore, the used reasoning tool suffers the same scalability issues as the one based on MTBDD.

Finally, the redundancy property has been object of specific intensive studies. In fact, the identification of redundant policies and their 'safe' elimination increases the evaluation performance of access control systems. A rigorous formalisation of redundancy is proposed in [14], where an algorithmic approach for minimising access control policies is proposed and its computational complexity studied.

7 Conclusion

Policy-based specifications are widely used to regulate the behaviour of system's entities relatively to the access to shared resources. The policy-based access control, by resorting to the concept of attribute, is sufficiently expressive to represent in an uniform way all classical access control approaches, varying from access control list and role-based to discretionary and mandatory ones. Policies permit indeed to define fine-grained, flexible and context-aware access controls, fostering systems integration, as attributes can be retrieved from different information systems. To ensure confidentiality and integrity principles, such policies need to take into account multiple security aspects, e.g., the ones studied by well-known security models, such as the Bell-LaPadula and Biba ones. However, enforcing in terms of policy-based specifications the security properties characterising such models is a tricky task. In fact, the hierarchical structure of policies, the presence of conflict resolution strategies and the intricacies deriving from the many controls involved do not permit to easily check whether a given security property is properly enforced. By means of the FACPL policy language, we have provided some specification examples of a significant set of security properties, and showed under which conditions such properties are properly enforced. To characterise the relationships with the behaviours that different polices enforce, we have also formalised, in a uniform way, various properties on the structure of policies. Furthermore, to effectively support system's designers in developing and maintaining policy-based specifications, we outlined a constraint-based approach enabling automated verification of security and structural properties by means of constraint solver tools.

We conclude by reviewing some additional properties we plan to study in the next future. On the one hand, to take into account dynamic behaviours of systems, we want to address history-dependent

security properties, and provide specialised formal analysis techniques. On the other hand, access control policies can also be used to produce, together with the authorisation decision, additional actions, named *obligations*, that can adapt the computing system's configuration. To reason on obligations, we want to formalise properties on conflicts and dependencies among them. Further details follow.

History-Dependent Properties. Classical examples of history-dependent properties are dynamic SoD and Chinese Wall [7]. Dynamic SoD properties correspond to enforcing separation of duty by evaluating not only the current subject's request, but also the history of actions the subject has previously performed. Chinese Wall properties correspond instead to an hybrid instantiation of the confidentiality and integrity principles, where history is used to adapt the access rights granted by the confidentiality controls. Specifically, it means that a subject is only allowed to access resources which are not in conflict with any other resource that the subject has already accessed.

Enforcing these properties within policy-based specifications means checking the history of system's authorisations. This could be done, e.g., by means of attributes representing the history. These attributes should in fact collect all the information needed for properly enforcing a considered history-dependent property, e.g., in case of Chinese Wall, which resources have been already accessed. In order to formally verify that such properties are enforced, we need to enhance our semantic-based formalisation with an explicit representation of history. Possible approaches to pursue for achieving this formalisation are those used in Usage Control [19], i.e. a novel access control model for ensuring continuous authorisation when an access is in progress.

Obligations. Obligations have been introduced in access control for modeling the need of fulfilling additional actions in order to gain access. For instance, XACML supports the definition of obligations and, to allow an access, it requires that all obligations possibly generated by the policy evaluation are correctly fulfilled. Obligations can be thus used to adapt the computing system's configuration. However, these obligations may have conditional requirements on their execution, e.g. conflicts and dependencies, that have to be taken into account. For instance, an obligation can require to be executed only if another one has not been already executed. To formalise and analyse properties on obligations, we plan to start from the representation model of obligation's features outlined in [4], and instantiate such model with respect to the FACPL policy language.

References

[1] Konstantine Arkoudas, Ritu Chadha & Cho-Yu Jason Chiang (2014): *Sophisticated Access Control via SMT and Logical Frameworks*. *ACM Trans. Inf. Syst. Secur.* 16(4), p. 17.

[2] Arosha K. Bandara, Emil Lupu & Alessandra Russo (2003): *Using Event Calculus to Formalise Policy Specification and Analysis*. In: *POLICY*, IEEE Computer Society, p. 26.

[3] David E. Bell & Leonard J. Lapadula (1976): *Secure Computer System: Unified Exposition and MULTICS Interpretation*. Technical Report, The MITRE Corporation.

[4] Elisa Bertino, Carolyn Brodie, Seraphin B. Calo, Lorrie Faith Cranor, Clare-Marie Karat, John Karat, Ninghui Li, Dan Lin, Jorge Lobo, Qun Ni, Prathima Rao & Xiping Wang (2009): *Analysis of privacy and security policies*. *IBM Journal of Research and Development* 53(2).

[5] K. J. Biba (1977): *Integrity Considerations for Secure Computer Systems*. Technical Report, The MITRE Corporation.

[6] Matthew A. Bishop (2002): *The Art and Science of Computer Security*. Addison-Wesley.

[7] D. F. C. Brewer & M. J. Nash (1989): *The Chinese Wall Security Policy*. In: *Security and Privacy*, IEEE Computer Society, pp. 206–214.

[8] D. D. Clark & D. R. Wilson (1987): *A Comparison of Commercial and Military Computer Security Policies*. In: *Security and Privacy*, IEEE Computer Society, pp. 184–195.

[9] Nicodemos Damianou, Naranker Dulay, Emil Lupu & Morris Sloman (2001): *The Ponder Policy Specification Language*. In: *POLICY*, LNCS 1995, Springer, pp. 18–38.

[10] David Ferraiolo & Richard Kuhn (1992): *Role-Based Access Control*. In: *NIST-NCSC*, pp. 554–563.

[11] Kathi Fisler, Shriram Krishnamurthi, Leo A. Meyerovich & Michael Carl Tschantz (2005): *Verification and change-impact analysis of access-control policies*. In: *ICSE*, ACM, pp. 196–205.

[12] Dieter Gollmann (2011): *Computer Security (3. ed.)*. Wiley.

[13] G. Scott Graham & Peter J. Denning (1972): *Protection: Principles and Practice*. In: *AFIPS*, ACM, pp. 417–429.

[14] Marco Guarnieri, Mario Arrigoni Neri, Eros Magri & Simone Mutti (2013): *On the notion of redundancy in access control policies*. In: *SACMAT*, ACM, pp. 161–172.

[15] Sushil Jajodia, Pierangela Samarati & V. S. Subrahmanian (1997): *A Logical Language for Expressing Authorizations*. In: *Security and Privacy*, IEEE Computer Society, pp. 31–42.

[16] Xin Jin, Ram Krishnan & Ravi S. Sandhu (2012): *A Unified Attribute-Based Access Control Model Covering DAC, MAC and RBAC*. In: *DBSec*, Springer, pp. 41–55.

[17] Vladimir Kolovski, James A. Hendler & Bijan Parsia (2007): *Analyzing web access control policies*. In: *WWW*, ACM, pp. 677–686.

[18] Butler W. Lampson (1974): *Protection*. *Operating Systems Review* 8(1), pp. 18–24.

[19] Aliaksandr Lazouski, Fabio Martinelli & Paolo Mori (2010): *Usage control in computer security: A survey*. *Computer Science Review* 4(2), pp. 81–99.

[20] Andrea Margheri, Rosario Pugliese & Francesco Tiezzi (2015): *A Light Version of the FACPL Policy Language*. Technical Report. Available at `http://facpl.sourceforge.net/research/lightFACPLTR.pdf`.

[21] Leonardo Mendonça de Moura & Nikolaj Bjørner (2008): *Z3: An Efficient SMT Solver*. In: *TACAS 2008*, Springer, pp. 337–340.

[22] Leonardo Mendonça de Moura & Nikolaj Bjørner (2011): *Satisfiability modulo theories: introduction and applications*. *Commun. ACM* 54(9), pp. 69–77.

[23] NIST (2009): *A survey of access control models*. `http://csrc.nist.gov/news_events/privilege-management-workshop/PvM-Model-Survey-Aug26-2009.pdf`.

[24] OASIS XACML TC (2013): *eXtensible Access Control Markup Language (XACML) version 3.0* . `https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml`.

[25] Ravi S. Sandhu (1993): *Lattice-Based Access Control Models*. *IEEE Computer* 26(11), pp. 9–19.

[26] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein & Charles E. Youman (1996): *Role-Based Access Control Models*. *IEEE Computer* 29(2), pp. 38–47.

[27] Sabrina De Capitani di Vimercati, Sara Foresti & Pierangela Samarati (2008): *Recent Advances in Access Control*. In Michael Gertz & Sushil Jajodia, editors: *Handbook of Database Security*, Springer, pp. 1–26.